

The L^AT_EX3 Sources

The L^AT_EX Project*

Released 2022-02-24

Abstract

This is the reference documentation for the `expl3` programming environment. The `expl3` modules set up an experimental naming scheme for L^AT_EX commands, which allow the L^AT_EX programmer to systematically name functions and variables, and specify the argument types of functions.

The T_EX and ϵ -T_EX primitives are all given a new name according to these conventions. However, in the main direct use of the primitives is not required or encouraged: the `expl3` modules define an independent low-level L^AT_EX3 programming language.

The `expl3` modules are designed to be loaded on top of L^AT_EX 2 ϵ . With an up-to-date L^AT_EX 2 ϵ kernel, this material is loaded as part of the format. The fundamental programming code can also be loaded with other T_EX formats, subject to restrictions on the full range of functionality.

*E-mail: latex-team@latex-project.org

Contents

I	Introduction	1
1	Introduction to <code>expl3</code> and this document	2
1.1	Naming functions and variables	2
1.1.1	Scratch variables	5
1.1.2	Terminological inexactitude	5
1.2	Documentation conventions	5
1.3	Formal language conventions which apply generally	7
1.4	<code>TeX</code> concepts not supported by <code>L^AT_EX3</code>	7
II	Bootstrapping	8
2	The <code>l3bootstrap</code> package: Bootstrap code	9
2.1	Using the <code>L^AT_EX3</code> modules	9
3	The <code>l3names</code> package: Namespace for primitives	11
3.1	Setting up the <code>L^AT_EX3</code> programming language	11
III	Programming Flow	12
4	The <code>l3basics</code> package: Basic definitions	13
4.1	No operation functions	13
4.2	Grouping material	13
4.3	Control sequences and functions	14
4.3.1	Defining functions	14
4.3.2	Defining new functions using parameter text	15
4.3.3	Defining new functions using the signature	17
4.3.4	Copying control sequences	19
4.3.5	Deleting control sequences	20
4.3.6	Showing control sequences	20
4.3.7	Converting to and from control sequences	20
4.4	Analysing control sequences	22
4.5	Using or removing tokens and arguments	23
4.5.1	Selecting tokens from delimited arguments	25
4.6	Predicates and conditionals	25
4.6.1	Tests on control sequences	27
4.6.2	Primitive conditionals	27
4.7	Starting a paragraph	28
4.8	Debugging support	29

5	The <code>l3expan</code> package: Argument expansion	30
5.1	Defining new variants	30
5.2	Methods for defining variants	31
5.3	Introducing the variants	32
5.4	Manipulating the first argument	34
5.5	Manipulating two arguments	36
5.6	Manipulating three arguments	37
5.7	Unbraced expansion	38
5.8	Preventing expansion	38
5.9	Controlled expansion	40
5.10	Internal functions	42
6	The <code>l3sort</code> package: Sorting functions	43
6.1	Controlling sorting	43
7	The <code>l3tl-analysis</code> package: Analysing token lists	45
8	The <code>l3regex</code> package: Regular expressions in <code>T_EX</code>	46
8.1	Syntax of regular expressions	47
8.1.1	Regular expression examples	47
8.1.2	Characters in regular expressions	48
8.1.3	Characters classes	48
8.1.4	Structure: alternatives, groups, repetitions	49
8.1.5	Matching exact tokens	50
8.1.6	Miscellaneous	52
8.2	Syntax of the replacement text	52
8.3	Pre-compiling regular expressions	54
8.4	Matching	55
8.5	Submatch extraction	56
8.6	Replacement	57
8.7	Scratch regular expressions	59
8.8	Bugs, misfeatures, future work, and other possibilities	59
9	The <code>l3prg</code> package: Control structures	62
9.1	Defining a set of conditional functions	62
9.2	The boolean data type	64
9.2.1	Scratch booleans	66
9.3	Boolean expressions	66
9.4	Logical loops	68
9.5	Producing multiple copies	69
9.6	Detecting <code>T_EX</code> 's mode	70
9.7	Primitive conditionals	70
9.8	Nestable recursions and mappings	70
9.8.1	Simple mappings	71
9.9	Internal programming functions	71

10 The <code>l3sys</code> package: System/runtime functions	72
10.1 The name of the job	72
10.2 Date and time	72
10.3 Engine	73
10.4 Output format	73
10.5 Platform	74
10.6 Random numbers	74
10.7 Access to the shell	74
10.8 Loading configuration data	75
10.8.1 Final settings	76
11 The <code>l3msg</code> package: Messages	77
11.1 Creating new messages	77
11.2 Customizable information for message modules	78
11.3 Contextual information for messages	78
11.4 Issuing messages	80
11.4.1 Messages for showing material	83
11.4.2 Expandable error messages	83
11.5 Redirecting messages	83
12 The <code>l3file</code> package: File and I/O operations	85
12.1 Input–output stream management	85
12.1.1 Reading from files	87
12.1.2 Writing to files	90
12.1.3 Wrapping lines in output	92
12.1.4 Constant input–output streams, and variables	93
12.1.5 Primitive conditionals	93
12.2 File operation functions	93
13 The <code>l3luatex</code> package: Lua_{TeX}-specific functions	98
13.1 Breaking out to Lua	98
13.2 Lua interfaces	99
14 The <code>l3legacy</code> package: Interfaces to legacy concepts	100
 IV Data types	 101
15 The <code>l3tl</code> package: Token lists	102
15.1 Creating and initialising token list variables	102
15.2 Adding data to token list variables	103
15.3 Token list conditionals	104
15.3.1 Testing the first token	106
15.4 Working with token lists as a whole	107
15.4.1 Using token lists	107
15.4.2 Counting and reversing token lists	108
15.4.3 Viewing token lists	109
15.5 Manipulating items in token lists	110
15.5.1 Mapping over token lists	110
15.5.2 Head and tail of token lists	112

15.5.3	Items and ranges in token lists	113
15.5.4	Sorting token lists	116
15.6	Manipulating tokens in token lists	116
15.6.1	Replacing tokens	116
15.6.2	Reassigning category codes	117
15.7	Constant token lists	118
15.8	Scratch token lists	118
16	The <code>l3str</code> package: Strings	120
16.1	Creating and initialising string variables	121
16.2	Adding data to string variables	122
16.3	String conditionals	122
16.4	Mapping over strings	124
16.5	Working with the content of strings	126
16.6	Modifying string variables	129
16.7	String manipulation	130
16.8	Viewing strings	131
16.9	Constant strings	132
16.10	Scratch strings	132
17	The <code>l3str-convert</code> package: string encoding conversions	133
17.1	Encoding and escaping schemes	133
17.2	Conversion functions	135
17.3	Conversion by expansion (for PDF contexts)	135
17.4	Possibilities, and things to do	135
18	The <code>l3quark</code> package: Quarks	137
18.1	Quarks	137
18.2	Defining quarks	138
18.3	Quark tests	138
18.4	Recursion	139
18.4.1	An example of recursion with quarks	140
18.5	Scan marks	141
19	The <code>l3seq</code> package: Sequences and stacks	142
19.1	Creating and initialising sequences	142
19.2	Appending data to sequences	144
19.3	Recovering items from sequences	144
19.4	Recovering values from sequences with branching	145
19.5	Modifying sequences	147
19.6	Sequence conditionals	147
19.7	Mapping over sequences	148
19.8	Using the content of sequences directly	150
19.9	Sequences as stacks	151
19.10	Sequences as sets	152
19.11	Constant and scratch sequences	153
19.12	Viewing sequences	154

20 The <code>l3int</code> package: Integers	155
20.1 Integer expressions	155
20.2 Creating and initialising integers	157
20.3 Setting and incrementing integers	158
20.4 Using integers	158
20.5 Integer expression conditionals	159
20.6 Integer expression loops	160
20.7 Integer step functions	162
20.8 Formatting integers	163
20.9 Converting from other formats to integers	164
20.10 Random integers	165
20.11 Viewing integers	166
20.12 Constant integers	166
20.13 Scratch integers	166
20.14 Direct number expansion	167
20.15 Primitive conditionals	167
21 The <code>l3flag</code> package: Expandable flags	169
21.1 Setting up flags	169
21.2 Expandable flag commands	170
22 The <code>l3clist</code> package: Comma separated lists	171
22.1 Creating and initialising comma lists	172
22.2 Adding data to comma lists	173
22.3 Modifying comma lists	174
22.4 Comma list conditionals	175
22.5 Mapping over comma lists	175
22.6 Using the content of comma lists directly	177
22.7 Comma lists as stacks	178
22.8 Using a single item	179
22.9 Viewing comma lists	180
22.10 Constant and scratch comma lists	180
23 The <code>l3token</code> package: Token manipulation	181
23.1 Creating character tokens	182
23.2 Manipulating and interrogating character tokens	183
23.3 Generic tokens	186
23.4 Converting tokens	187
23.5 Token conditionals	187
23.6 Peeking ahead at the next token	191
23.7 Description of all possible tokens	195

24 The <code>l3prop</code> package: Property lists	198
24.1 Creating and initialising property lists	198
24.2 Adding and updating property list entries	199
24.3 Recovering values from property lists	200
24.4 Modifying property lists	201
24.5 Property list conditionals	201
24.6 Recovering values from property lists with branching	202
24.7 Mapping over property lists	203
24.8 Viewing property lists	204
24.9 Scratch property lists	205
24.10 Constants	205
25 The <code>l3skip</code> package: Dimensions and skips	206
25.1 Creating and initialising <code>dim</code> variables	206
25.2 Setting <code>dim</code> variables	207
25.3 Utilities for dimension calculations	207
25.4 Dimension expression conditionals	208
25.5 Dimension expression loops	210
25.6 Dimension step functions	211
25.7 Using <code>dim</code> expressions and variables	212
25.8 Viewing <code>dim</code> variables	213
25.9 Constant dimensions	214
25.10 Scratch dimensions	214
25.11 Creating and initialising <code>skip</code> variables	214
25.12 Setting <code>skip</code> variables	215
25.13 Skip expression conditionals	216
25.14 Using <code>skip</code> expressions and variables	216
25.15 Viewing <code>skip</code> variables	216
25.16 Constant skips	217
25.17 Scratch skips	217
25.18 Inserting skips into the output	217
25.19 Creating and initialising <code>muskip</code> variables	218
25.20 Setting <code>muskip</code> variables	218
25.21 Using <code>muskip</code> expressions and variables	219
25.22 Viewing <code>muskip</code> variables	219
25.23 Constant muskips	220
25.24 Scratch muskips	220
25.25 Primitive conditional	220
26 The <code>l3keys</code> package: Key–value interfaces	221
26.1 Creating keys	222
26.2 Sub-dividing keys	227
26.3 Choice and multiple choice keys	227
26.4 Key usage scope	230
26.5 Setting keys	230
26.6 Handling of unknown keys	231
26.7 Selective key setting	231
26.8 Utility functions for keys	233
26.9 Low-level interface for parsing key–val lists	233

27 The <code>l3intarray</code> package: fast global integer arrays	236
27.1 <code>l3intarray</code> documentation	236
27.1.1 Implementation notes	237
28 The <code>l3fp</code> package: Floating points	238
28.1 Creating and initialising floating point variables	240
28.2 Setting floating point variables	240
28.3 Using floating points	241
28.4 Floating point conditionals	242
28.5 Floating point expression loops	244
28.6 Some useful constants, and scratch variables	246
28.7 Floating point exceptions	247
28.8 Viewing floating points	248
28.9 Floating point expressions	249
28.9.1 Input of floating point numbers	249
28.9.2 Precedence of operators	250
28.9.3 Operations	250
28.10 Disclaimer and roadmap	257
29 The <code>l3fparray</code> package: fast global floating point arrays	260
29.1 <code>l3fparray</code> documentation	260
30 The <code>l3cctab</code> package: Category code tables	261
30.1 Creating and initialising category code tables	261
30.2 Using category code tables	261
30.3 Category code table conditionals	262
30.4 Constant category code tables	262
 V Text manipulation	 263
31 The <code>l3unicode</code> package: Unicode support functions	264
32 The <code>l3text</code> package: text processing	265
32.1 Expanding text	265
32.2 Case changing	266
32.3 Removing formatting from text	267
32.4 Control variables	267
 VI Typesetting	 269

33 The l3box package: Boxes	270
33.1 Creating and initialising boxes	270
33.2 Using boxes	271
33.3 Measuring and setting box dimensions	272
33.4 Box conditionals	273
33.5 The last box inserted	273
33.6 Constant boxes	273
33.7 Scratch boxes	273
33.8 Viewing box contents	274
33.9 Boxes and color	274
33.10 Horizontal mode boxes	274
33.11 Vertical mode boxes	275
33.12 Using boxes efficiently	277
33.13 Affine transformations	278
33.14 Primitive box conditionals	281
34 The l3coffins package: Coffin code layer	282
34.1 Creating and initialising coffins	282
34.2 Setting coffin content and poles	283
34.3 Coffin affine transformations	284
34.4 Joining and using coffins	284
34.5 Measuring coffins	285
34.6 Coffin diagnostics	285
34.7 Constants and variables	286
35 The l3color package: Color support	288
35.1 Color in boxes	288
35.2 Color models	288
35.3 Color expressions	289
35.4 Named colors	290
35.5 Selecting colors	291
35.6 Colors for fills and strokes	291
35.6.1 Coloring math mode material	292
35.7 Multiple color models	292
35.8 Exporting color specifications	293
35.9 Creating new color models	293
35.9.1 Color profiles	294
36 The l3pdf package: Core PDF support	295
36.1 Objects	295
36.2 Version	296
36.3 Compression	297
36.4 Destinations	297
VII Additions and removals	299

37 The l3candidates package: Experimental additions to l3kernel	300
37.1 Important notice	300
37.2 Additions to l3box	301
37.3 Additions to l3expan	301
37.4 Additions to l3fp	301
37.5 Additions to l3file	302
37.6 Additions to l3flag	302
37.7 Additions to l3intarray	302
37.8 Additions to l3msg	303
37.9 Additions to l3prg	303
37.10 Additions to l3prop	304
37.11 Additions to l3seq	305
37.12 Additions to l3sys	306
37.13 Additions to l3tl	307
37.14 Additions to l3token	308
 VIII Implementation	 310
38 l3bootstrap implementation	311
38.1 LuaTeX-specific code	311
38.2 The \pdfstrcmp primitive in XeTeX	311
38.3 Loading support Lua code	312
38.4 Engine requirements	312
38.5 Extending allocators	313
38.6 The L ^A T _E X3 code environment	314
 39 l3names implementation	 317
 40 l3kernel-functions: kernel-reserved functions	 343
40.1 Internal kernel functions	343
40.2 Kernel backend functions	349
 41 l3basics implementation	 351
41.1 Renaming some T _E X primitives (again)	351
41.2 Defining some constants	353
41.3 Defining functions	353
41.4 Selecting tokens	354
41.5 Gobbling tokens from input	356
41.6 Debugging and patching later definitions	356
41.7 Conditional processing and definitions	357
41.8 Dissecting a control sequence	363
41.9 Exist or free	365
41.10 Preliminaries for new functions	367
41.11 Defining new functions	368
41.12 Copying definitions	370
41.13 Undefining functions	370
41.14 Generating parameter text from argument count	371
41.15 Defining functions from a given number of arguments	372
41.16 Using the signature to define functions	373

41.17	Checking control sequence equality	375
41.18	Diagnostic functions	375
41.19	Decomposing a macro definition	377
41.20	Doing nothing functions	378
41.21	Breaking out of mapping functions	378
41.22	Starting a paragraph	379
42	l3expan implementation	380
42.1	General expansion	380
42.2	Hand-tuned definitions	384
42.3	Last-unbraced versions	387
42.4	Preventing expansion	389
42.5	Controlled expansion	390
42.6	Emulating e-type expansion	391
42.7	Defining function variants	398
42.8	Definitions with the automated technique	408
43	l3sort implementation	410
43.1	Variables	410
43.2	Finding available \toks registers	411
43.3	Protected user commands	413
43.4	Merge sort	415
43.5	Expandable sorting	418
43.6	Messages	423
44	l3tl-analysis implementation	426
44.1	Internal functions	426
44.2	Internal format	426
44.3	Variables and helper functions	427
44.4	Plan of attack	429
44.5	Disabling active characters	430
44.6	First pass	431
44.7	Second pass	436
44.8	Mapping through the analysis	439
44.9	Showing the results	439
44.10	Peeking ahead	442
44.11	Messages	447
45	l3regex implementation	449
45.1	Plan of attack	449
45.2	Helpers	450
45.2.1	Constants and variables	452
45.2.2	Testing characters	453
45.2.3	Internal auxiliaries	454
45.2.4	Character property tests	457
45.2.5	Simple character escape	459
45.3	Compiling	464
45.3.1	Variables used when compiling	465
45.3.2	Generic helpers used when compiling	467
45.3.3	Mode	468

45.3.4	Framework	470
45.3.5	Quantifiers	473
45.3.6	Raw characters	476
45.3.7	Character properties	478
45.3.8	Anchoring and simple assertions	479
45.3.9	Character classes	479
45.3.10	Groups and alternations	483
45.3.11	Catcodes and csnames	485
45.3.12	Raw token lists with \u	489
45.3.13	Other	493
45.3.14	Showing regexes	493
45.4	Building	499
45.4.1	Variables used while building	499
45.4.2	Framework	500
45.4.3	Helpers for building an NFA	503
45.4.4	Building classes	504
45.4.5	Building groups	506
45.4.6	Others	511
45.5	Matching	512
45.5.1	Variables used when matching	513
45.5.2	Matching: framework	515
45.5.3	Using states of the NFA	518
45.5.4	Actions when matching	519
45.6	Replacement	522
45.6.1	Variables and helpers used in replacement	522
45.6.2	Query and brace balance	523
45.6.3	Framework	524
45.6.4	Submatches	528
45.6.5	Csnames in replacement	529
45.6.6	Characters in replacement	531
45.6.7	An error	534
45.7	User functions	534
45.7.1	Variables and helpers for user functions	538
45.7.2	Matching	540
45.7.3	Extracting submatches	541
45.7.4	Replacement	546
45.7.5	Peeking ahead	549
45.8	Messages	554
45.9	Code for tracing	561
46	l3prg implementation	562
46.1	Primitive conditionals	562
46.2	Defining a set of conditional functions	562
46.3	The boolean data type	562
46.4	Internal auxiliaries	564
46.5	Boolean expressions	565
46.6	Logical loops	570
46.7	Producing multiple copies	571
46.8	Detecting T _E X's mode	573
46.9	Internal programming functions	573

47 l3sys implementation	575
47.1 Kernel code	575
47.1.1 Detecting the engine	575
47.1.2 Randomness	577
47.1.3 Platform	577
47.1.4 Configurations	577
47.1.5 Access to the shell	579
47.2 Dynamic (every job) code	582
47.2.1 The name of the job	582
47.2.2 Time and date	582
47.2.3 Random numbers	583
47.2.4 Access to the shell	584
47.2.5 Held over from l3file	585
47.3 Last-minute code	585
47.3.1 Detecting the output	586
47.3.2 Configurations	586
48 l3msg implementation	588
48.1 Internal auxiliaries	588
48.2 Creating messages	588
48.3 Messages: support functions and text	590
48.4 Showing messages: low level mechanism	591
48.5 Displaying messages	593
48.6 Kernel-specific functions	602
48.7 Internal messages	603
48.8 Expandable errors	609
48.9 Message formatting	610
49 l3file implementation	611
49.1 Input operations	611
49.1.1 Variables and constants	611
49.1.2 Stream management	612
49.1.3 Reading input	615
49.2 Output operations	618
49.2.1 Variables and constants	618
49.2.2 Internal auxiliaries	619
49.3 Stream management	619
49.3.1 Deferred writing	621
49.3.2 Immediate writing	621
49.3.3 Special characters for writing	622
49.3.4 Hard-wrapping lines to a character count	623
49.4 File operations	632
49.4.1 Internal auxiliaries	634
49.5 GetIdInfo	649
49.6 Checking the version of kernel dependencies	650
49.7 Messages	652
49.8 Functions delayed from earlier modules	653

50 l3luatex implementation	655
50.1 Breaking out to Lua	655
50.2 Messages	656
50.3 Lua functions for internal use	656
50.4 Preserving iniTeX Lua data for runs	661
51 l3legacy Implementation	663
52 l3tl implementation	665
52.1 Functions	665
52.2 Constant token lists	667
52.3 Adding to token list variables	667
52.4 Internal quarks and quark-query functions	670
52.5 Reassigning token list category codes	670
52.6 Modifying token list variables	673
52.7 Token list conditionals	677
52.8 Mapping over token lists	682
52.9 Using token lists	684
52.10 Working with the contents of token lists	685
52.11 The first token from a token list	688
52.12 Token by token changes	693
52.13 Using a single item	695
52.14 Viewing token lists	698
52.15 Internal scan marks	700
52.16 Scratch token lists	700
53 l3str implementation	701
53.1 Internal auxiliaries	701
53.2 Creating and setting string variables	702
53.3 Modifying string variables	703
53.4 String comparisons	704
53.5 Mapping over strings	707
53.6 Accessing specific characters in a string	709
53.7 Counting characters	714
53.8 The first character in a string	715
53.9 String manipulation	716
53.10 Viewing strings	718
54 l3str-convert implementation	719
54.1 Helpers	719
54.1.1 Variables and constants	719
54.2 String conditionals	721
54.3 Conversions	722
54.3.1 Producing one byte or character	722
54.3.2 Mapping functions for conversions	723
54.3.3 Error-reporting during conversion	724
54.3.4 Framework for conversions	725
54.3.5 Byte unescape and escape	729
54.3.6 Native strings	730
54.3.7 <code>clist</code>	731

54.3.8	8-bit encodings	731
54.4	Messages	734
54.5	Escaping definitions	735
54.5.1	Unescape methods	736
54.5.2	Escape methods	740
54.6	Encoding definitions	742
54.6.1	UTF-8 support	742
54.6.2	UTF-16 support	747
54.6.3	UTF-32 support	752
54.7	PDF names and strings by expansion	755
54.7.1	ISO 8859 support	756
55	l3quark implementation	773
55.1	Quarks	773
55.2	Scan marks	781
56	l3seq implementation	783
56.1	Allocation and initialisation	784
56.2	Appending data to either end	787
56.3	Modifying sequences	788
56.4	Sequence conditionals	790
56.5	Recovering data from sequences	792
56.6	Mapping over sequences	796
56.7	Using sequences	800
56.8	Sequence stacks	801
56.9	Viewing sequences	802
56.10	Scratch sequences	803
57	l3int implementation	804
57.1	Integer expressions	805
57.2	Creating and initialising integers	807
57.3	Setting and incrementing integers	809
57.4	Using integers	810
57.5	Integer expression conditionals	810
57.6	Integer expression loops	814
57.7	Integer step functions	815
57.8	Formatting integers	817
57.9	Converting from other formats to integers	822
57.10	Viewing integer	825
57.11	Random integers	826
57.12	Constant integers	826
57.13	Scratch integers	826
57.14	Integers for earlier modules	827
58	l3flag implementation	828
58.1	Non-expandable flag commands	828
58.2	Expandable flag commands	829

59	l3clist implementation	831
59.1	Removing spaces around items	832
59.2	Allocation and initialisation	833
59.3	Adding data to comma lists	835
59.4	Comma lists as stacks	836
59.5	Modifying comma lists	838
59.6	Comma list conditionals	841
59.7	Mapping over comma lists	842
59.8	Using comma lists	846
59.9	Using a single item	848
59.10	Viewing comma lists	850
59.11	Scratch comma lists	851
60	l3token implementation	852
60.1	Internal auxiliaries	852
60.2	Manipulating and interrogating character tokens	852
60.3	Creating character tokens	855
60.4	Generic tokens	863
60.5	Token conditionals	864
60.6	Peeking ahead at the next token	874
61	l3prop implementation	881
61.1	Internal auxiliaries	882
61.2	Allocation and initialisation	883
61.3	Accessing data in property lists	885
61.4	Property list conditionals	890
61.5	Recovering values from property lists with branching	891
61.6	Mapping over property lists	892
61.7	Viewing property lists	893
62	l3skip implementation	895
62.1	Length primitives renamed	895
62.2	Internal auxiliaries	895
62.3	Creating and initialising <code>dim</code> variables	895
62.4	Setting <code>dim</code> variables	896
62.5	Utilities for dimension calculations	897
62.6	Dimension expression conditionals	898
62.7	Dimension expression loops	900
62.8	Dimension step functions	901
62.9	Using <code>dim</code> expressions and variables	903
62.10	Viewing <code>dim</code> variables	905
62.11	Constant dimensions	905
62.12	Scratch dimensions	905
62.13	Creating and initialising <code>skip</code> variables	905
62.14	Setting <code>skip</code> variables	907
62.15	Skip expression conditionals	907
62.16	Using <code>skip</code> expressions and variables	908
62.17	Inserting skips into the output	908
62.18	Viewing <code>skip</code> variables	909
62.19	Constant skips	909

62.20	Scratch skips	909
62.21	Creating and initialising muskip variables	909
62.22	Setting muskip variables	910
62.23	Using muskip expressions and variables	911
62.24	Viewing muskip variables	911
62.25	Constant muskips	912
62.26	Scratch muskips	912
63	l3keys Implementation	913
63.1	Low-level interface	913
63.2	Constants and variables	920
63.2.1	Internal auxiliaries	922
63.3	The key defining mechanism	923
63.4	Turning properties into actions	925
63.5	Creating key properties	932
63.6	Setting keys	937
63.7	Utilities	946
63.8	Messages	948
64	l3intarray implementation	950
64.1	Lua implementation	950
64.1.1	Allocating arrays	950
64.1.2	Array items	953
64.1.3	Working with contents of integer arrays	955
64.2	Font dimension based implementation	956
64.2.1	Allocating arrays	957
64.2.2	Array items	958
64.2.3	Working with contents of integer arrays	960
64.3	Common parts	962
64.3.1	Random arrays	962
65	l3fp implementation	964
66	l3fp-aux implementation	965
66.1	Access to primitives	965
66.2	Internal representation	965
66.3	Using arguments and semicolons	966
66.4	Constants, and structure of floating points	967
66.5	Overflow, underflow, and exact zero	970
66.6	Expanding after a floating point number	970
66.7	Other floating point types	971
66.8	Packing digits	974
66.9	Decimate (dividing by a power of 10)	977
66.10	Functions for use within primitive conditional branches	979
66.11	Integer floating points	980
66.12	Small integer floating points	981
66.13	Fast string comparison	982
66.14	Name of a function from its l3fp-parse name	982
66.15	Messages	982

67 l3fp-traps Implementation	983
67.1 Flags	983
67.2 Traps	983
67.3 Errors	987
67.4 Messages	987
68 l3fp-round implementation	989
68.1 Rounding tools	989
68.2 The round function	993
69 l3fp-parse implementation	998
69.1 Work plan	998
69.1.1 Storing results	999
69.1.2 Precedence and infix operators	1000
69.1.3 Prefix operators, parentheses, and functions	1003
69.1.4 Numbers and reading tokens one by one	1004
69.2 Main auxiliary functions	1006
69.3 Helpers	1007
69.4 Parsing one number	1008
69.4.1 Numbers: trimming leading zeros	1014
69.4.2 Number: small significand	1015
69.4.3 Number: large significand	1017
69.4.4 Number: beyond 16 digits, rounding	1019
69.4.5 Number: finding the exponent	1022
69.5 Constants, functions and prefix operators	1025
69.5.1 Prefix operators	1025
69.5.2 Constants	1028
69.5.3 Functions	1029
69.6 Main functions	1030
69.7 Infix operators	1032
69.7.1 Closing parentheses and commas	1033
69.7.2 Usual infix operators	1035
69.7.3 Juxtaposition	1036
69.7.4 Multi-character cases	1036
69.7.5 Ternary operator	1037
69.7.6 Comparisons	1037
69.8 Tools for functions	1039
69.9 Messages	1042
70 l3fp-assign implementation	1043
70.1 Assigning values	1043
70.2 Updating values	1044
70.3 Showing values	1044
70.4 Some useful constants and scratch variables	1045

71 l3fp-logic Implementation	1046
71.1 Syntax of internal functions	1046
71.2 Tests	1046
71.3 Comparison	1047
71.4 Floating point expression loops	1050
71.5 Extrema	1053
71.6 Boolean operations	1055
71.7 Ternary operator	1056
72 l3fp-basics Implementation	1058
72.1 Addition and subtraction	1058
72.1.1 Sign, exponent, and special numbers	1059
72.1.2 Absolute addition	1061
72.1.3 Absolute subtraction	1063
72.2 Multiplication	1067
72.2.1 Signs, and special numbers	1067
72.2.2 Absolute multiplication	1069
72.3 Division	1071
72.3.1 Signs, and special numbers	1071
72.3.2 Work plan	1072
72.3.3 Implementing the significand division	1075
72.4 Square root	1080
72.5 About the sign and exponent	1087
72.6 Operations on tuples	1088
73 l3fp-extended implementation	1090
73.1 Description of fixed point numbers	1090
73.2 Helpers for numbers with extended precision	1091
73.3 Multiplying a fixed point number by a short one	1092
73.4 Dividing a fixed point number by a small integer	1092
73.5 Adding and subtracting fixed points	1093
73.6 Multiplying fixed points	1094
73.7 Combining product and sum of fixed points	1095
73.8 Extended-precision floating point numbers	1098
73.9 Dividing extended-precision numbers	1100
73.10 Inverse square root of extended precision numbers	1104
73.11 Converting from fixed point to floating point	1106
74 l3fp-expo implementation	1108
74.1 Logarithm	1108
74.1.1 Work plan	1108
74.1.2 Some constants	1109
74.1.3 Sign, exponent, and special numbers	1109
74.1.4 Absolute ln	1109
74.2 Exponential	1116
74.2.1 Sign, exponent, and special numbers	1116
74.3 Power	1121
74.4 Factorial	1127

75 l3fp-trig Implementation	1130
75.1 Direct trigonometric functions	1131
75.1.1 Filtering special cases	1131
75.1.2 Distinguishing small and large arguments	1134
75.1.3 Small arguments	1135
75.1.4 Argument reduction in degrees	1135
75.1.5 Argument reduction in radians	1136
75.1.6 Computing the power series	1144
75.2 Inverse trigonometric functions	1146
75.2.1 Arctangent and arccotangent	1147
75.2.2 Arcsine and arccosine	1152
75.2.3 Arccosecant and arcsecant	1154
76 l3fp-convert implementation	1156
76.1 Dealing with tuples	1156
76.2 Trimming trailing zeros	1156
76.3 Scientific notation	1157
76.4 Decimal representation	1158
76.5 Token list representation	1160
76.6 Formatting	1161
76.7 Convert to dimension or integer	1161
76.8 Convert from a dimension	1162
76.9 Use and eval	1163
76.10 Convert an array of floating points to a comma list	1164
77 l3fp-random Implementation	1166
77.1 Engine support	1166
77.2 Random floating point	1170
77.3 Random integer	1170
78 l3fparray implementation	1176
78.1 Allocating arrays	1176
78.2 Array items	1177
79 l3cctab implementation	1181
79.1 Variables	1181
79.2 Allocating category code tables	1182
79.3 Saving category code tables	1183
79.4 Using category code tables	1184
79.5 Category code table conditionals	1189
79.6 Constant category code tables	1190
79.7 Messages	1192
80 l3unicode implementation	1194
81 l3text implementation	1198
81.1 Internal auxiliaries	1198
81.2 Utilities	1199
81.3 Configuration variables	1202
81.4 Expansion to formatted text	1203

82 l3text-case implementation	1211
82.1 Case changing	1211
82.2 Case changing data for 8-bit engines	1232
83 l3text-purify implementation	1244
83.1 Purifying text	1244
83.2 Accent and letter-like data for purifying text	1250
84 l3box implementation	1258
84.1 Support code	1258
84.2 Creating and initialising boxes	1258
84.3 Measuring and setting box dimensions	1259
84.4 Using boxes	1260
84.5 Box conditionals	1261
84.6 The last box inserted	1261
84.7 Constant boxes	1261
84.8 Scratch boxes	1262
84.9 Viewing box contents	1262
84.10 Horizontal mode boxes	1263
84.11 Vertical mode boxes	1265
84.12 Affine transformations	1268
85 l3coffins Implementation	1278
85.1 Coffins: data structures and general variables	1278
85.2 Basic coffin functions	1279
85.3 Measuring coffins	1285
85.4 Coffins: handle and pole management	1285
85.5 Coffins: calculation of pole intersections	1289
85.6 Affine transformations	1291
85.7 Aligning and typesetting of coffins	1299
85.8 Coffin diagnostics	1304
85.9 Messages	1310
86 l3color Implementation	1311
86.1 Basics	1311
86.2 Predefined color names	1312
86.3 Setup	1313
86.4 Utility functions	1313
86.5 Model conversion	1314
86.6 Color expressions	1315
86.7 Selecting colors (and color models)	1322
86.8 Math color	1324
86.9 Fill and stroke color	1327
86.10 Defining named colors	1328
86.11 Exporting colors	1330
86.12 Additional color models	1333
86.13 Applying profiles	1347
86.14 Diagnostics	1348
86.15 Messages	1348

87 l3pdf implementation	1352
87.1 Compression	1352
87.2 Objects	1353
87.3 Version	1353
87.4 Destinations	1355
88 l3candidates Implementation	1356
88.1 Additions to l3box	1356
88.1.1 Viewing part of a box	1356
88.2 Additions to l3flag	1358
88.3 Additions to l3msg	1359
88.4 Additions to l3prg	1360
88.5 Additions to l3prop	1361
88.6 Additions to l3seq	1361
88.7 Additions to l3sys	1366
88.8 Additions to l3file	1367
88.9 Additions to l3tl	1367
88.9.1 Building a token list	1367
88.9.2 Other additions to l3tl	1371
88.10 Additions to l3token	1371
89 l3deprecation implementation	1374
89.1 Patching definitions to deprecate	1374
89.2 Removed functions	1376
89.3 Deprecated l3str functions	1380
89.4 Deprecated l3seq functions	1381
89.5 Deprecated l3sys functions	1381
89.6 Deprecated l3tl functions	1381
89.7 Deprecated l3token functions	1382
Index	1384

Part I
Introduction

Chapter 1

Introduction to expl3 and this document

This document is intended to act as a comprehensive reference manual for the `expl3` language. A general guide to the `LATEX3` programming language is found in [expl3.pdf](#).

1.1 Naming functions and variables

`LATEX3` does not use `@` as a “letter” for defining internal macros. Instead, the symbols `_` and `:` are used in internal macro names to provide structure. The name of each *function* is divided into logical units using `_`, while `:` separates the *name* of the function from the *argument specifier* (“arg-spec”). This describes the arguments expected by the function. In most cases, each argument is represented by a single letter. The complete list of arg-spec letters for a function is referred to as the *signature* of the function.

Each function name starts with the *module* to which it belongs. Thus apart from a small number of very basic functions, all `expl3` function names contain at least one underscore to divide the module name from the descriptive name of the function. For example, all functions concerned with comma lists are in module `clist` and begin `\clist_`.

Every function must include an argument specifier. For functions which take no arguments, this will be blank and the function name will end `:`. Most functions take one or more arguments, and use the following argument specifiers:

N and n These mean *no manipulation*, of a single token for `N` and of a set of tokens given in braces for `n`. Both pass the argument through exactly as given. Usually, if you use a single token for an `n` argument, all will be well.

c This means *cname*, and indicates that the argument will be turned into a `cname` before being used. So `\foo:c {ArgumentOne}` will act in the same way as `\foo:N \ArgumentOne`.

V and v These mean *value of variable*. The `V` and `v` specifiers are used to get the content of a variable without needing to worry about the underlying `TEX` structure containing the data. A `V` argument will be a single token (similar to `N`), for example `\foo:V \MyVariable`; on the other hand, using `v` a `cname` is constructed first, and then the value is recovered, for example `\foo:v {MyVariable}`.

- o This means *expansion once*. In general, the **V** and **v** specifiers are favoured over **o** for recovering stored information. However, **o** is useful for correctly processing information with delimited arguments.
- x The **x** specifier stands for *exhaustive expansion*: every token in the argument is fully expanded until only unexpandable ones remain. The \TeX `\edef` primitive carries out this type of expansion. Functions which feature an **x**-type argument are *not* expandable.
- e The **e** specifier is in many respects identical to **x**, but with a very different implementation. Functions which feature an **e**-type argument may be expandable. The drawback is that **e** is extremely slow (often more than 200 times slower) in older engines, more precisely in non- \LaTeX engines older than 2019.
- f The **f** specifier stands for *full expansion*, and in contrast to **x** stops at the first non-expandable token (reading the argument from left to right) without trying to expand it. If this token is a *space token*, it is gobbled, and thus won't be part of the resulting argument. For example, when setting a token list variable (a macro used for storage), the sequence

```
\tl_set:Nn \l_my_a_tl { A }
\tl_set:Nn \l_my_b_tl { B }
\tl_set:Nf \l_my_a_tl { \l_my_a_tl \l_my_b_tl }
```

will leave `\l_my_a_tl` with the content `A\l_my_b_tl`, as `A` cannot be expanded and so terminates expansion before `\l_my_b_tl` is considered.

- T and F** For logic tests, there are the branch specifiers **T** (*true*) and **F** (*false*). Both specifiers treat the input in the same way as **n** (no change), but make the logic much easier to see.
- p The letter **p** indicates \TeX *parameters*. Normally this will be used for delimited functions as `expl3` provides better methods for creating simple sequential arguments.
- w Finally, there is the **w** specifier for *weird* arguments. This covers everything else, but mainly applies to delimited values (where the argument must be terminated by some specified string).
- D The **D** stands for **Do not use**. All of the \TeX primitives are initially `\let` to a **D** name, and some are then given a second name. These functions have no standardized syntax, they are engine dependent and their name can change without warning, thus their use is *strongly discouraged* in package code: programmers should instead use the interfaces documented in [interface3.pdf](#)¹.

Notice that the argument specifier describes how the argument is processed prior to being passed to the underlying function. For example, `\foo:c` will take its argument, convert it to a control sequence and pass it to `\foo:N`.

Variables are named in a similar manner to functions, but begin with a single letter to define the type of variable:

¹If a primitive offers a functionality not yet in the kernel, programmers and users are encouraged to write to the \LaTeX -L mailing list (<mailto:LATEX-L@listserv.uni-heidelberg.de>) describing their use-case and intended behaviour, so that a possible interface can be discussed. Temporarily, while an interface is not provided, programmers may use the procedure described in the [l3styleguide.pdf](#).

c Constant: global parameters whose value should not be changed.

g Parameters whose value should only be set globally.

l Parameters whose value should only be set locally.

Each variable name is then build up in a similar way to that of a function, typically starting with the module² name and then a descriptive part. Variables end with a short identifier to show the variable type:

clist Comma separated list.

dim “Rigid” lengths.

fp Floating-point values;

int Integer-valued count register.

mskip “Rubber” lengths for use in mathematics.

seq “Sequence”: a data-type used to implement lists (with access at both ends) and stacks.

skip “Rubber” lengths.

str String variables: contain character data.

tl Token list variables: placeholder for a token list.

Applying V-type or v-type expansion to variables of one of the above types is supported, while it is not supported for the following variable types:

bool Either true or false.

box Box register.

coffin A “box with handles” — a higher-level data type for carrying out **box** alignment operations.

flag Integer that can be incremented expandably.

fpararray Fixed-size array of floating point values.

intarray Fixed-size array of integers.

ior/iow An input or output stream, for reading from or writing to, respectively.

prop Property list: analogue of dictionary or associative arrays in other languages.

regex Regular expression.

²The module names are not used in case of generic scratch registers defined in the data type modules, e.g., the **int** module contains some scratch variables called `\l_tmpa_int`, `\l_tmpe_int`, and so on. In such a case adding the module name up front to denote the module and in the back to indicate the type, as in `\l_int_tmpe_int` would be very unreadable.

1.1.1 Scratch variables

Modules focussed on variable usage typically provide four scratch variables, two local and two global, with names of the form `\<scope>_tmpa_<type>/\<scope>_tmpb_<type>`. These are never used by the core code. The nature of \TeX grouping means that as with any other scratch variable, these should only be set and used with no intervening third-party code.

1.1.2 Terminological inexactitude

A word of warning. In this document, and others referring to the `expl3` programming modules, we often refer to “variables” and “functions” as if they were actual constructs from a real programming language. In truth, \TeX is a macro processor, and functions are simply macros that may or may not take arguments and expand to their replacement text. Many of the common variables are *also* macros, and if placed into the input stream will simply expand to their definition as well — a “function” with no arguments and a “token list variable” are almost the same.³ On the other hand, some “variables” are actually registers that must be initialised and their values set and retrieved with specific functions.

The conventions of the `expl3` code are designed to clearly separate the ideas of “macros that contain data” and “macros that contain code”, and a consistent wrapper is applied to all forms of “data” whether they be macros or actually registers. This means that sometimes we will use phrases like “the function returns a value”, when actually we just mean “the macro expands to something”. Similarly, the term “execute” might be used in place of “expand” or it might refer to the more specific case of “processing in \TeX ’s stomach” (if you are familiar with the \TeX book parlance).

If in doubt, please ask; chances are we’ve been hasty in writing certain definitions and need to be told to tighten up our terminology.

1.2 Documentation conventions

This document is typeset with the experimental `l3doc` class; several conventions are used to help describe the features of the code. A number of conventions are used here to make the documentation clearer.

Each group of related functions is given in a box. For a function with a “user” name, this might read:

`\ExplSyntaxOn`
`\ExplSyntaxOff`

`\ExplSyntaxOn ... \ExplSyntaxOff`

The textual description of how the function works would appear here. The syntax of the function is shown in mono-spaced text to the right of the box. In this example, the function takes no arguments and so the name of the function is simply reprinted.

For programming functions, which use `_` and `:` in their name there are a few additional conventions: If two related functions are given with identical names but different argument specifiers, these are termed *variants* of each other, and the latter functions are printed in grey to show this more clearly. They will carry out the same function but will take different types of argument:

³ \TeX nically, functions with no arguments are `\long` while token list variables are not.

<code>\seq_new:N</code>	<code>\seq_new:N</code> $\langle sequence \rangle$
<code>\seq_new:c</code>	

When a number of variants are described, the arguments are usually illustrated only for the base function. Here, $\langle sequence \rangle$ indicates that `\seq_new:N` expects the name of a sequence. From the argument specifier, `\seq_new:c` also expects a sequence name, but as a name rather than as a control sequence. Each argument given in the illustration should be described in the following text.

Fully expandable functions Some functions are fully expandable, which allows them to be used within an **x**-type or **e**-type argument (in plain \TeX terms, inside an `\edef` or `\expanded`), as well as within an **f**-type argument. These fully expandable functions are indicated in the documentation by a star:

<code>\cs_to_str:N</code> \star	<code>\cs_to_str:N</code> $\langle cs \rangle$
-----------------------------------	--

As with other functions, some text should follow which explains how the function works. Usually, only the star will indicate that the function is expandable. In this case, the function expects a $\langle cs \rangle$, shorthand for a $\langle control\ sequence \rangle$.

Restricted expandable functions A few functions are fully expandable but cannot be fully expanded within an **f**-type argument. In this case a hollow star is used to indicate this:

<code>\seq_map_function:NN</code> \star	<code>\seq_map_function:NN</code> $\langle seq \rangle$ $\langle function \rangle$
---	--

Conditional functions Conditional (**if**) functions are normally defined in three variants, with **T**, **F** and **TF** argument specifiers. This allows them to be used for different “true”/“false” branches, depending on which outcome the conditional is being used to test. To indicate this without repetition, this information is given in a shortened form:

<code>\sys_if_engine_xetex:TF</code> \star	<code>\sys_if_engine_xetex:TF</code> $\{\langle true\ code \rangle\} \{\langle false\ code \rangle\}$
--	---

The underlining and italic of **TF** indicates that three functions are available:

- `\sys_if_engine_xetex:T`
- `\sys_if_engine_xetex:F`
- `\sys_if_engine_xetex:TF`

Usually, the illustration will use the **TF** variant, and so both $\langle true\ code \rangle$ and $\langle false\ code \rangle$ will be shown. The two variant forms **T** and **F** take only $\langle true\ code \rangle$ and $\langle false\ code \rangle$, respectively. Here, the star also shows that this function is expandable. With some minor exceptions, *all* conditional functions in the `expl3` modules should be defined in this way.

Variables, constants and so on are described in a similar manner:

<code>\l_tmpa_tl</code>

A short piece of text will describe the variable: there is no syntax illustration in this case.

In some cases, the function is similar to one in $\text{\LaTeX 2}_{\epsilon}$ or plain \TeX . In these cases, the text will include an extra “ **\TeX hackers note**” section:

<code>\token_to_str:N</code> *	<code>\token_to_str:N</code> $\langle token \rangle$
--------------------------------	--

The normal description text.

T_EXhackers note: Detail for the experienced T_EX or L^AT_EX 2_ε programmer. In this case, it would point out that this function is the T_EX primitive `\string`.

Changes to behaviour When new functions are added to `expl3`, the date of first inclusion is given in the documentation. Where the documented behaviour of a function changes after it is first introduced, the date of the update will also be given. This means that the programmer can be sure that any release of `expl3` after the date given will contain the function of interest with expected behaviour as described. Note that changes to code internals, including bug fixes, are not recorded in this way *unless* they impact on the expected behaviour.

1.3 Formal language conventions which apply generally

As this is a formal reference guide for L^AT_EX3 programming, the descriptions of functions are intended to be reasonably “complete”. However, there is also a need to avoid repetition. Formal ideas which apply to general classes of function are therefore summarised here.

For tests which have a TF argument specification, the test is evaluated to give a logically TRUE or FALSE result. Depending on this result, either the $\langle true\ code \rangle$ or the $\langle false\ code \rangle$ will be left in the input stream. In the case where the test is expandable, and a predicate (`_p`) variant is available, the logical value determined by the test is left in the input stream: this will typically be part of a larger logical construct.

1.4 T_EX concepts not supported by L^AT_EX3

The T_EX concept of an “`\outer`” macro is *not supported* at all by L^AT_EX3. As such, the functions provided here may break when used on top of L^AT_EX 2_ε if `\outer` tokens are used in the arguments.

Part II

Bootstrapping

Chapter 2

The l3bootstrap package

Bootstrap code

2.1 Using the L^AT_EX3 modules

The modules documented in `source3` are designed to be used on top of L^AT_EX 2_ε and are loaded all as one with the usual `\usepackage{expl3}` or `\RequirePackage{expl3}` instructions.

As the modules use a coding syntax different from standard L^AT_EX 2_ε it provides a few functions for setting it up.

`\ExplSyntaxOn`
`\ExplSyntaxOff`

Updated: 2011-08-13

`\ExplSyntaxOn` *<code>* `\ExplSyntaxOff`

The `\ExplSyntaxOn` function switches to a category code regime in which spaces and new lines are ignored, and in which the colon (:) and underscore (_) are treated as “letters”, thus allowing access to the names of code functions and variables. Within this environment, ~ is used to input a space. The `\ExplSyntaxOff` reverts to the document category code regime.

T_EXhackers note: Spaces introduced by ~ behave much in the same way as normal space characters in the standard category code regime: they are ignored after a control word or at the start of a line, and multiple consecutive ~ are equivalent to a single one. However, ~ is *not* ignored at the end of a line.

`\ProvidesExplPackage`
`\ProvidesExplClass`
`\ProvidesExplFile`

Updated: 2017-03-19

`\RequirePackage{expl3}`
`\ProvidesExplPackage` *{<package>}* *{<date>}* *{<version>}* *{<description>}*

These functions act broadly in the same way as the corresponding L^AT_EX 2_ε kernel functions `\ProvidesPackage`, `\ProvidesClass` and `\ProvidesFile`. However, they also implicitly switch `\ExplSyntaxOn` for the remainder of the code with the file. At the end of the file, `\ExplSyntaxOff` will be called to reverse this. (This is the same concept as L^AT_EX 2_ε provides in turning on `\makeatletter` within package and class code.) The *<date>* should be given in the format *<year>/<month>/<day>* or in the ISO date format *<year>-<month>-<day>*. If the *<version>* is given then it will be prefixed with v in the package identifier line.

\GetIdInfo

Updated: 2012-06-04

\RequirePackage{l3bootstrap}**\GetIdInfo \$Id:** *<SVN info field>* \$ *{<description>}*

Extracts all information from a SVN field. Spaces are not ignored in these fields. The information pieces are stored in separate control sequences with **\ExplFileName** for the part of the file name leading up to the period, **\ExplFileDate** for date, **\ExplFileVersion** for version and **\ExplFileDescription** for the description.

To summarize: Every single package using this syntax should identify itself using one of the above methods. Special care is taken so that every package or class file loaded with **\RequirePackage** or similar are loaded with usual L^AT_EX 2_ε category codes and the L^AT_EX 3 category code scheme is reloaded when needed afterwards. See implementation for details. If you use the **\GetIdInfo** command you can use the information when loading a package with

\ProvidesExplPackage{\ExplFileName}**{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}**

Chapter 3

The l3names package

Namespace for primitives

3.1 Setting up the L^AT_EX3 programming language

This module is at the core of the L^AT_EX3 programming language. It performs the following tasks:

- defines new names for all T_EX primitives;
- emulate required primitives not provided by default in LuaT_EX;
- switches to the category code régime for programming;

This module is entirely dedicated to primitives (and emulations of these), which should not be used directly within L^AT_EX3 code (outside of “kernel-level” code). As such, the primitives are not documented here: *The T_EXbook*, *T_EX by Topic* and the manuals for pdfT_EX, X_YT_EX, LuaT_EX, pT_EX and upT_EX should be consulted for details of the primitives. These are named `\tex_⟨name⟩:D`, typically based on the primitive’s *⟨name⟩* in pdfT_EX and omitting a leading pdf when the primitive is not related to pdf output.

Part III
Programming Flow

Chapter 4

The l3basics package

Basic definitions

As the name suggest this package holds some basic definitions which are needed by most or all other packages in this set.

Here we describe those functions that are used all over the place. With that we mean functions dealing with the construction and testing of control sequences. Furthermore the basic parts of conditional processing are covered; conditional processing dealing with specific data types is described in the modules specific for the respective data types.

4.1 No operation functions

`\prg_do_nothing:` ***`\prg_do_nothing:`**

An expandable function which does nothing at all: leaves nothing in the input stream after a single expansion.

`\scan_stop:`**`\scan_stop:`**

A non-expandable function which does nothing. Does not vanish on expansion but produces no typeset output.

4.2 Grouping material

`\group_begin:`
`\group_end:`**`\group_begin:`**
`\group_end:`

These functions begin and end a group for definition purposes. Assignments are local to groups unless carried out in a global manner. (A small number of exceptions to this rule will be noted as necessary elsewhere in this document.) Each **`\group_begin:`** must be matched by a **`\group_end:`**, although this does not have to occur within the same function. Indeed, it is often necessary to start a group within one function and finish it within another, for example when seeking to use non-standard category codes.

`\group_insert_after:N`

`\group_insert_after:N` $\langle token \rangle$

Adds $\langle token \rangle$ to the list of $\langle tokens \rangle$ to be inserted when the current group level ends. The list of $\langle tokens \rangle$ to be inserted is empty at the beginning of a group: multiple applications of `\group_insert_after:N` may be used to build the inserted list one $\langle token \rangle$ at a time. The current group level may be closed by a `\group_end:` function or by a token with category code 2 (close-group), namely a `}` if standard category codes apply.

`\group_show_list:`
`\group_log_list:`

`\group_show_list:`
`\group_log_list:`

New: 2021-05-11

Display (to the terminal or log file) a list of the groups that are currently opened. This is intended for tracking down problems.

TeXhackers note: This is a wrapper around the `\showgroups` primitive.

4.3 Control sequences and functions

As TeX is a macro language, creating new functions means creating macros. At point of use, a function is replaced by the replacement text (“code”) in which each parameter in the code (`#1`, `#2`, *etc.*) is replaced the appropriate arguments absorbed by the function. In the following, $\langle code \rangle$ is therefore used as a shorthand for “replacement text”.

Functions which are not “protected” are fully expanded inside an `x` expansion. In contrast, “protected” functions are not expanded within `x` expansions.

4.3.1 Defining functions

Functions can be created with no requirement that they are declared first (in contrast to variables, which must always be declared). Declaring a function before setting up the code means that the name chosen is checked and an error raised if it is already in use. The name of a function can be checked at the point of definition using the `\cs_new...` functions: this is recommended for all functions which are defined for the first time.

There are three ways to define new functions. All classes define a function to expand to the substitution text. Within the substitution text the actual parameters are substituted for the formal parameters (`#1`, `#2`, ...).

new Create a new function with the **new** scope, such as `\cs_new:Npn`. The definition is global and results in an error if it is already defined.

set Create a new function with the **set** scope, such as `\cs_set:Npn`. The definition is restricted to the current TeX group and does not result in an error if the function is already defined.

gset Create a new function with the **gset** scope, such as `\cs_gset:Npn`. The definition is global and does not result in an error if the function is already defined.

Within each set of scope there are different ways to define a function. The differences depend on restrictions on the actual parameters and the expandability of the resulting function.

nopar Create a new function with the **nopar** restriction, such as `\cs_set_nopar:Npn`. The parameter may not contain `\par` tokens.

protected Create a new function with the **protected** restriction, such as `\cs_set_protected:Npn`. The parameter may contain `\par` tokens but the function will not expand within an **x**-type or **e**-type expansion.

Finally, the functions in Subsections 4.3.2 and 4.3.3 are primarily meant to define *base functions* only. Base functions can only have the following argument specifiers:

N and **n** No manipulation.

T and **F** Functionally equivalent to **n** (you are actually encouraged to use the family of `\prg_new_conditional:` functions described in Section 9.1).

p and **w** These are special cases.

The `\cs_new:` functions below (and friends) do not stop you from using other argument specifiers in your function names, but they do not handle expansion for you. You should define the base function and then use `\cs_generate_variant:Nn` to generate custom variants as described in Section 5.2.

4.3.2 Defining new functions using parameter text

<code>\cs_new:Npn</code> <code>\cs_new:cpn</code> <code>\cs_new:Npx</code> <code>\cs_new:cpx</code>	<code>\cs_new:Npn <function> <parameters> {<code>}</code> Creates <code><function></code> to expand to <code><code></code> as replacement text. Within the <code><code></code> , the <code><parameters></code> (<code>#1</code> , <code>#2</code> , etc.) will be replaced by those absorbed by the function. The definition is global and an error results if the <code><function></code> is already defined.
--	--

<code>\cs_new_nopar:Npn</code> <code>\cs_new_nopar:cpn</code> <code>\cs_new_nopar:Npx</code> <code>\cs_new_nopar:cpx</code>	<code>\cs_new_nopar:Npn <function> <parameters> {<code>}</code> Creates <code><function></code> to expand to <code><code></code> as replacement text. Within the <code><code></code> , the <code><parameters></code> (<code>#1</code> , <code>#2</code> , etc.) will be replaced by those absorbed by the function. When the <code><function></code> is used the <code><parameters></code> absorbed cannot contain <code>\par</code> tokens. The definition is global and an error results if the <code><function></code> is already defined.
--	---

<code>\cs_new_protected:Npn</code> <code>\cs_new_protected:cpn</code> <code>\cs_new_protected:Npx</code> <code>\cs_new_protected:cpx</code>	<code>\cs_new_protected:Npn <function> <parameters> {<code>}</code> Creates <code><function></code> to expand to <code><code></code> as replacement text. Within the <code><code></code> , the <code><parameters></code> (<code>#1</code> , <code>#2</code> , etc.) will be replaced by those absorbed by the function. The <code><function></code> will not expand within an x -type or e -type argument. The definition is global and an error results if the <code><function></code> is already defined.
--	---

<code>\cs_new_protected_nopar:Npn</code> <code>\cs_new_protected_nopar:cpn</code> <code>\cs_new_protected_nopar:Npx</code> <code>\cs_new_protected_nopar:cpx</code>	<code>\cs_new_protected_nopar:Npn <function> <parameters> {<code>}</code>
--	---

Creates `<function>` to expand to `<code>` as replacement text. Within the `<code>`, the `<parameters>` (`#1`, `#2`, etc.) will be replaced by those absorbed by the function. When the `<function>` is used the `<parameters>` absorbed cannot contain `\par` tokens. The `<function>` will not expand within an **x**-type or **e**-type argument. The definition is global and an error results if the `<function>` is already defined.

<code>\cs_set:Npn</code>	<code>\cs_set:Npn <function> <parameters> {<code>}</code>
<code>\cs_set:cpn</code>	Sets $\langle function \rangle$ to expand to $\langle code \rangle$ as replacement text. Within the $\langle code \rangle$, the
<code>\cs_set:Npx</code>	$\langle parameters \rangle$ (#1, #2, etc.) will be replaced by those absorbed by the function. The
<code>\cs_set:cpx</code>	assignment of a meaning to the $\langle function \rangle$ is restricted to the current T _E X group level.

<code>\cs_set_nopar:Npn</code>	<code>\cs_set_nopar:Npn <function> <parameters> {<code>}</code>
<code>\cs_set_nopar:cpn</code>	Sets $\langle function \rangle$ to expand to $\langle code \rangle$ as replacement text. Within the $\langle code \rangle$, the
<code>\cs_set_nopar:Npx</code>	$\langle parameters \rangle$ (#1, #2, etc.) will be replaced by those absorbed by the function. When the
<code>\cs_set_nopar:cpx</code>	$\langle function \rangle$ is used the $\langle parameters \rangle$ absorbed cannot contain <code>\par</code> tokens. The assignment

of a meaning to the $\langle function \rangle$ is restricted to the current T_EX group level.

<code>\cs_set_protected:Npn</code>	<code>\cs_set_protected:Npn <function> <parameters> {<code>}</code>
<code>\cs_set_protected:cpn</code>	Sets $\langle function \rangle$ to expand to $\langle code \rangle$ as replacement text. Within the $\langle code \rangle$, the
<code>\cs_set_protected:Npx</code>	$\langle parameters \rangle$ (#1, #2, etc.) will be replaced by those absorbed by the function. The
<code>\cs_set_protected:cpx</code>	assignment of a meaning to the $\langle function \rangle$ is restricted to the current T _E X group level.

The $\langle function \rangle$ will not expand within an x-type or e-type argument.

<code>\cs_set_protected_nopar:Npn</code>	<code>\cs_set_protected_nopar:Npn <function> <parameters> {<code>}</code>
<code>\cs_set_protected_nopar:cpn</code>	
<code>\cs_set_protected_nopar:Npx</code>	
<code>\cs_set_protected_nopar:cpx</code>	

Sets $\langle function \rangle$ to expand to $\langle code \rangle$ as replacement text. Within the $\langle code \rangle$, the $\langle parameters \rangle$ (#1, #2, etc.) will be replaced by those absorbed by the function. When the $\langle function \rangle$ is used the $\langle parameters \rangle$ absorbed cannot contain `\par` tokens. The assignment of a meaning to the $\langle function \rangle$ is restricted to the current T_EX group level. The $\langle function \rangle$ will not expand within an x-type or e-type argument.

<code>\cs_gset:Npn</code>	<code>\cs_gset:Npn <function> <parameters> {<code>}</code>
<code>\cs_gset:cpn</code>	Globally sets $\langle function \rangle$ to expand to $\langle code \rangle$ as replacement text. Within the $\langle code \rangle$,
<code>\cs_gset:Npx</code>	the $\langle parameters \rangle$ (#1, #2, etc.) will be replaced by those absorbed by the function. The
<code>\cs_gset:cpx</code>	assignment of a meaning to the $\langle function \rangle$ is <i>not</i> restricted to the current T _E X group level: the assignment is global.

<code>\cs_gset_nopar:Npn</code>	<code>\cs_gset_nopar:Npn <function> <parameters> {<code>}</code>
<code>\cs_gset_nopar:cpn</code>	Globally sets $\langle function \rangle$ to expand to $\langle code \rangle$ as replacement text. Within the $\langle code \rangle$,
<code>\cs_gset_nopar:Npx</code>	the $\langle parameters \rangle$ (#1, #2, etc.) will be replaced by those absorbed by the function.
<code>\cs_gset_nopar:cpx</code>	When the $\langle function \rangle$ is used the $\langle parameters \rangle$ absorbed cannot contain <code>\par</code> tokens. The

assignment of a meaning to the $\langle function \rangle$ is *not* restricted to the current T_EX group level: the assignment is global.

<code>\cs_gset_protected:Npn</code>	<code>\cs_gset_protected:Npn <function> <parameters> {<code>}</code>
<code>\cs_gset_protected:cpn</code>	Globally sets $\langle function \rangle$ to expand to $\langle code \rangle$ as replacement text. Within the $\langle code \rangle$,
<code>\cs_gset_protected:Npx</code>	the $\langle parameters \rangle$ (#1, #2, etc.) will be replaced by those absorbed by the function. The
<code>\cs_gset_protected:cpx</code>	assignment of a meaning to the $\langle function \rangle$ is <i>not</i> restricted to the current T _E X group level: the assignment is global. The $\langle function \rangle$ will not expand within an x-type or

e-type argument.

<code>\cs_gset_protected_nopar:Npn</code>	<code>\cs_gset_protected_nopar:Npn <function> <parameters> {\code}</code>
<code>\cs_gset_protected_nopar:cpn</code>	
<code>\cs_gset_protected_nopar:Npx</code>	
<code>\cs_gset_protected_nopar:cpx</code>	

Globally sets $\langle function \rangle$ to expand to $\langle code \rangle$ as replacement text. Within the $\langle code \rangle$, the $\langle parameters \rangle$ ($\#1$, $\#2$, etc.) will be replaced by those absorbed by the function. When the $\langle function \rangle$ is used the $\langle parameters \rangle$ absorbed cannot contain `\par` tokens. The assignment of a meaning to the $\langle function \rangle$ is *not* restricted to the current \TeX group level: the assignment is global. The $\langle function \rangle$ will not expand within an x -type or e -type argument.

4.3.3 Defining new functions using the signature

<code>\cs_new:Nn</code>	<code>\cs_new:Nn <function> {\code}</code>
<code>\cs_new:(cn Nx cx)</code>	

Creates $\langle function \rangle$ to expand to $\langle code \rangle$ as replacement text. Within the $\langle code \rangle$, the number of $\langle parameters \rangle$ is detected automatically from the function signature. These $\langle parameters \rangle$ ($\#1$, $\#2$, etc.) will be replaced by those absorbed by the function. The definition is global and an error results if the $\langle function \rangle$ is already defined.

<code>\cs_new_nopar:Nn</code>	<code>\cs_new_nopar:Nn <function> {\code}</code>
<code>\cs_new_nopar:(cn Nx cx)</code>	

Creates $\langle function \rangle$ to expand to $\langle code \rangle$ as replacement text. Within the $\langle code \rangle$, the number of $\langle parameters \rangle$ is detected automatically from the function signature. These $\langle parameters \rangle$ ($\#1$, $\#2$, etc.) will be replaced by those absorbed by the function. When the $\langle function \rangle$ is used the $\langle parameters \rangle$ absorbed cannot contain `\par` tokens. The definition is global and an error results if the $\langle function \rangle$ is already defined.

<code>\cs_new_protected:Nn</code>	<code>\cs_new_protected:Nn <function> {\code}</code>
<code>\cs_new_protected:(cn Nx cx)</code>	

Creates $\langle function \rangle$ to expand to $\langle code \rangle$ as replacement text. Within the $\langle code \rangle$, the number of $\langle parameters \rangle$ is detected automatically from the function signature. These $\langle parameters \rangle$ ($\#1$, $\#2$, etc.) will be replaced by those absorbed by the function. The $\langle function \rangle$ will not expand within an x -type or e -type argument. The definition is global and an error results if the $\langle function \rangle$ is already defined.

<code>\cs_new_protected_nopar:Nn</code>	<code>\cs_new_protected_nopar:Nn <function> {\code}</code>
<code>\cs_new_protected_nopar:(cn Nx cx)</code>	

Creates $\langle function \rangle$ to expand to $\langle code \rangle$ as replacement text. Within the $\langle code \rangle$, the number of $\langle parameters \rangle$ is detected automatically from the function signature. These $\langle parameters \rangle$ ($\#1$, $\#2$, etc.) will be replaced by those absorbed by the function. When the $\langle function \rangle$ is used the $\langle parameters \rangle$ absorbed cannot contain `\par` tokens. The $\langle function \rangle$ will not expand within an x -type or e -type argument. The definition is global and an error results if the $\langle function \rangle$ is already defined.

<code>\cs_set:Nn</code>	<code>\cs_set:Nn <function> {\code}</code>
<code>\cs_set:(cn Nx cx)</code>	

Sets $\langle function \rangle$ to expand to $\langle code \rangle$ as replacement text. Within the $\langle code \rangle$, the number of $\langle parameters \rangle$ is detected automatically from the function signature. These $\langle parameters \rangle$ ($\#1$, $\#2$, etc.) will be replaced by those absorbed by the function. The assignment of a meaning to the $\langle function \rangle$ is restricted to the current \TeX group level.

`\cs_set_nopar:Nn`
`\cs_set_nopar:(cn|Nx|cx)`

`\cs_set_nopar:Nn <function> {<code>}`

Sets *<function>* to expand to *<code>* as replacement text. Within the *<code>*, the number of *<parameters>* is detected automatically from the function signature. These *<parameters>* (*#1, #2, etc.*) will be replaced by those absorbed by the function. When the *<function>* is used the *<parameters>* absorbed cannot contain `\par` tokens. The assignment of a meaning to the *<function>* is restricted to the current T_EX group level.

`\cs_set_protected:Nn`
`\cs_set_protected:(cn|Nx|cx)`

`\cs_set_protected:Nn <function> {<code>}`

Sets *<function>* to expand to *<code>* as replacement text. Within the *<code>*, the number of *<parameters>* is detected automatically from the function signature. These *<parameters>* (*#1, #2, etc.*) will be replaced by those absorbed by the function. The *<function>* will not expand within an x-type or e-type argument. The assignment of a meaning to the *<function>* is restricted to the current T_EX group level.

`\cs_set_protected_nopar:Nn`
`\cs_set_protected_nopar:(cn|Nx|cx)`

`\cs_set_protected_nopar:Nn <function> {<code>}`

Sets *<function>* to expand to *<code>* as replacement text. Within the *<code>*, the number of *<parameters>* is detected automatically from the function signature. These *<parameters>* (*#1, #2, etc.*) will be replaced by those absorbed by the function. When the *<function>* is used the *<parameters>* absorbed cannot contain `\par` tokens. The *<function>* will not expand within an x-type or e-type argument. The assignment of a meaning to the *<function>* is restricted to the current T_EX group level.

`\cs_gset:Nn`
`\cs_gset:(cn|Nx|cx)`

`\cs_gset:Nn <function> {<code>}`

Sets *<function>* to expand to *<code>* as replacement text. Within the *<code>*, the number of *<parameters>* is detected automatically from the function signature. These *<parameters>* (*#1, #2, etc.*) will be replaced by those absorbed by the function. The assignment of a meaning to the *<function>* is global.

`\cs_gset_nopar:Nn`
`\cs_gset_nopar:(cn|Nx|cx)`

`\cs_gset_nopar:Nn <function> {<code>}`

Sets *<function>* to expand to *<code>* as replacement text. Within the *<code>*, the number of *<parameters>* is detected automatically from the function signature. These *<parameters>* (*#1, #2, etc.*) will be replaced by those absorbed by the function. When the *<function>* is used the *<parameters>* absorbed cannot contain `\par` tokens. The assignment of a meaning to the *<function>* is global.

`\cs_gset_protected:Nn`
`\cs_gset_protected:(cn|Nx|cx)`

`\cs_gset_protected:Nn <function> {<code>}`

Sets *<function>* to expand to *<code>* as replacement text. Within the *<code>*, the number of *<parameters>* is detected automatically from the function signature. These *<parameters>* (*#1, #2, etc.*) will be replaced by those absorbed by the function. The *<function>* will not expand within an x-type or e-type argument. The assignment of a meaning to the *<function>* is global.

<code>\cs_gset_protected_nopar:Nn</code>	<code>\cs_gset_protected_nopar:Nn <function> {<code>}</code>
<code>\cs_gset_protected_nopar:(cn Nx cx)</code>	

Sets $\langle function \rangle$ to expand to $\langle code \rangle$ as replacement text. Within the $\langle code \rangle$, the number of $\langle parameters \rangle$ is detected automatically from the function signature. These $\langle parameters \rangle$ ($\#1$, $\#2$, etc.) will be replaced by those absorbed by the function. When the $\langle function \rangle$ is used the $\langle parameters \rangle$ absorbed cannot contain `\par` tokens. The $\langle function \rangle$ will not expand within an x -type or e -type argument. The assignment of a meaning to the $\langle function \rangle$ is global.

<code>\cs_generate_from_arg_count:NNnn</code>	<code>\cs_generate_from_arg_count:NNnn <function> <creator> {<number>}</code>
<code>\cs_generate_from_arg_count:(cNnn Ncnn)</code>	<code>{<code>}</code>

Updated: 2012-01-14

Uses the $\langle creator \rangle$ function (which should have signature Npn , for example `\cs_new:Npn`) to define a $\langle function \rangle$ which takes $\langle number \rangle$ arguments and has $\langle code \rangle$ as replacement text. The $\langle number \rangle$ of arguments is an integer expression, evaluated as detailed for `\int_eval:n`.

4.3.4 Copying control sequences

Control sequences (not just functions as defined above) can be set to have the same meaning using the functions described here. Making two control sequences equivalent means that the second control sequence is a *copy* of the first (rather than a pointer to it). Thus the old and new control sequence are not tied together: changes to one are not reflected in the other.

In the following text “cs” is used as an abbreviation for “control sequence”.

<code>\cs_new_eq:NN</code>	<code>\cs_new_eq:NN <cs₁> <cs₂></code>
<code>\cs_new_eq:(Nc cN cc)</code>	<code>\cs_new_eq:NN <cs₁> <token></code>

Globally creates $\langle control\ sequence_1 \rangle$ and sets it to have the same meaning as $\langle control\ sequence_2 \rangle$ or $\langle token \rangle$. The second control sequence may subsequently be altered without affecting the copy.

<code>\cs_set_eq:NN</code>	<code>\cs_set_eq:NN <cs₁> <cs₂></code>
<code>\cs_set_eq:(Nc cN cc)</code>	<code>\cs_set_eq:NN <cs₁> <token></code>

Sets $\langle control\ sequence_1 \rangle$ to have the same meaning as $\langle control\ sequence_2 \rangle$ (or $\langle token \rangle$). The second control sequence may subsequently be altered without affecting the copy. The assignment of a meaning to the $\langle control\ sequence_1 \rangle$ is restricted to the current $\text{T}_{\text{E}}\text{X}$ group level.

<code>\cs_gset_eq:NN</code>	<code>\cs_gset_eq:NN <cs₁> <cs₂></code>
<code>\cs_gset_eq:(Nc cN cc)</code>	<code>\cs_gset_eq:NN <cs₁> <token></code>

Globally sets $\langle control\ sequence_1 \rangle$ to have the same meaning as $\langle control\ sequence_2 \rangle$ (or $\langle token \rangle$). The second control sequence may subsequently be altered without affecting the copy. The assignment of a meaning to the $\langle control\ sequence_1 \rangle$ is *not* restricted to the current $\text{T}_{\text{E}}\text{X}$ group level: the assignment is global.

4.3.5 Deleting control sequences

There are occasions where control sequences need to be deleted. This is handled in a very simple manner.

<code>\cs_undefine:N</code>	<code>\cs_undefine:N <control sequence></code>
<code>\cs_undefine:c</code>	Sets <i><control sequence></i> to be globally undefined.

Updated: 2011-09-15

4.3.6 Showing control sequences

<code>\cs_meaning:N</code> ★	<code>\cs_meaning:N <control sequence></code>
<code>\cs_meaning:c</code> ★	This function expands to the <i>meaning</i> of the <i><control sequence></i> control sequence. For a macro, this includes the <i><replacement text></i> .

Updated: 2011-12-22

TeXhackers note: This is TeX's `\meaning` primitive. For tokens that are not control sequences, it is more logical to use `\token_to_meaning:N`. The `c` variant correctly reports undefined arguments.

<code>\cs_show:N</code>	<code>\cs_show:N <control sequence></code>
<code>\cs_show:c</code>	Displays the definition of the <i><control sequence></i> on the terminal.

Updated: 2017-02-14

TeXhackers note: This is similar to the TeX primitive `\show`, wrapped to a fixed number of characters per line.

<code>\cs_log:N</code>	<code>\cs_log:N <control sequence></code>
<code>\cs_log:c</code>	Writes the definition of the <i><control sequence></i> in the log file. See also <code>\cs_show:N</code> which displays the result in the terminal.

New: 2014-08-22
Updated: 2017-02-14

4.3.7 Converting to and from control sequences

<code>\use:c</code> ★	<code>\use:c {(control sequence name)}</code>
-----------------------	---

Expands the *<control sequence name>* until only characters remain, and then converts this into a control sequence. This process requires two expansions. As in other `c`-type arguments the *<control sequence name>* must, when fully expanded, consist of character tokens, typically a mixture of category code 10 (space), 11 (letter) and 12 (other).

TeXhackers note: Protected macros that appear in a `c`-type argument are expanded despite being protected; `\exp_not:n` also has no effect. An internal error occurs if non-characters or active characters remain after full expansion, as the conversion to a control sequence is not possible.

As an example of the `\use:c` function, both

```
\use:c { a b c }
```

and

```
\tl_new:N \l_my_tl
\tl_set:Nn \l_my_tl { a b c }
\use:c { \tl_use:N \l_my_tl }
```

would be equivalent to

```
\abc
```

after two expansions of `\use:c`.

<code>\cs_if_exist_use:N</code>	★	<code>\cs_if_exist_use:N</code> <i><control sequence></i>
<code>\cs_if_exist_use:c</code>	★	<code>\cs_if_exist_use:NTF</code> <i><control sequence></i> <i>{<true code>}</i> <i>{<false code>}</i>
<code>\cs_if_exist_use:NTF</code>	★	Tests whether the <i><control sequence></i> is currently defined according to the conditional
<code>\cs_if_exist_use:cTF</code>	★	<code>\cs_if_exist:NTF</code> (whether as a function or another control sequence type), and if it is
		inserts the <i><control sequence></i> into the input stream followed by the <i><true code></i> . Otherwise
		the <i><false code></i> is used.

New: 2012-11-10

<code>\cs:w</code>	★	<code>\cs:w</code> <i><control sequence name></i> <code>\cs_end:</code>
<code>\cs_end:</code>	★	Converts the given <i><control sequence name></i> into a single control sequence token. This
		process requires one expansion. The content for <i><control sequence name></i> may be literal
		material or from other expandable functions. The <i><control sequence name></i> must, when
		fully expanded, consist of character tokens which are not active: typically of category
		code 10 (space), 11 (letter) or 12 (other), or a mixture of these.

TeXhackers note: These are the TeX primitives `\csname` and `\endcsname`.

As an example of the `\cs:w` and `\cs_end:` functions, both

```
\cs:w a b c \cs_end:
```

and

```
\tl_new:N \l_my_tl
\tl_set:Nn \l_my_tl { a b c }
\cs:w \tl_use:N \l_my_tl \cs_end:
```

would be equivalent to

```
\abc
```

after one expansion of `\cs:w`.

<code>\cs_to_str:N</code>	★	<code>\cs_to_str:N</code> <i><control sequence></i>
		Converts the given <i><control sequence></i> into a series of characters with category code 12
		(other), except spaces, of category code 10. The result does <i>not</i> include the current
		escape token, contrarily to <code>\token_to_str:N</code> . Full expansion of this function requires
		exactly 2 expansion steps, and so an x-type or e-type expansion, or two o-type expansions
		are required to convert the <i><control sequence></i> to a sequence of characters in the input
		stream. In most cases, an f-expansion is correct as well, but this loses a space at the
		start of the result.

4.4 Analysing control sequences

`\cs_split_function:N` ★

New: 2018-04-06

`\cs_split_function:N` $\langle function \rangle$

Splits the $\langle function \rangle$ into the $\langle name \rangle$ (*i.e.* the part before the colon) and the $\langle signature \rangle$ (*i.e.* after the colon). This information is then placed in the input stream in three parts: the $\langle name \rangle$, the $\langle signature \rangle$ and a logic token indicating if a colon was found (to differentiate variables from function names). The $\langle name \rangle$ does not include the escape character, and both the $\langle name \rangle$ and $\langle signature \rangle$ are made up of tokens with category code 12 (other).

The next three functions decompose \TeX macros into their constituent parts: if the $\langle token \rangle$ passed is not a macro then no decomposition can occur. In the latter case, all three functions leave `\scan_stop:` in the input stream.

`\cs_prefix_spec:N` ★

New: 2019-02-27

`\cs_prefix_spec:N` $\langle token \rangle$

If the $\langle token \rangle$ is a macro, this function leaves the applicable \TeX prefixes in input stream as a string of tokens of category code 12 (with spaces having category code 10). Thus for example

```
\cs_set:Npn \next:nn #1#2 { x #1~y #2 }
\cs_prefix_spec:N \next:nn
```

leaves `\long` in the input stream. If the $\langle token \rangle$ is not a macro then `\scan_stop:` is left in the input stream.

\TeX hackers note: The prefix can be empty, `\long`, `\protected` or `\protected\long` with backslash replaced by the current escape character.

`\cs_argument_spec:N` ★

New: 2019-02-27

`\cs_argument_spec:N` $\langle token \rangle$

If the $\langle token \rangle$ is a macro, this function leaves the primitive \TeX argument specification in input stream as a string of character tokens of category code 12 (with spaces having category code 10). Thus for example

```
\cs_set:Npn \next:nn #1#2 { x #1 y #2 }
\cs_argument_spec:N \next:nn
```

leaves `#1#2` in the input stream. If the $\langle token \rangle$ is not a macro then `\scan_stop:` is left in the input stream.

\TeX hackers note: If the argument specification contains the string `->`, then the function produces incorrect results.

`\cs_replacement_spec:N` ★

New: 2019-02-27

`\cs_replacement_spec:N` $\langle token \rangle$

If the $\langle token \rangle$ is a macro, this function leaves the replacement text in input stream as a string of character tokens of category code 12 (with spaces having category code 10). Thus for example

```
\cs_set:Npn \next:nn #1#2 { x #1~y #2 }
\cs_replacement_spec:N \next:nn
```

leaves `x#1~y#2` in the input stream. If the $\langle token \rangle$ is not a macro then `\scan_stop:` is left in the input stream.

T_EXhackers note: If the argument specification contains the string `->`, then the function produces incorrect results.

4.5 Using or removing tokens and arguments

Tokens in the input can be read and used or read and discarded. If one or more tokens are wrapped in braces then when absorbing them the outer set is removed. At the same time, the category code of each token is set when the token is read by a function (if it is read more than once, the category code is determined by the situation in force when first function absorbs the token).

`\use:n` ★

`\use:n` $\{\langle group_1 \rangle\}$

`\use:nn` ★

`\use:nn` $\{\langle group_1 \rangle\} \{\langle group_2 \rangle\}$

`\use:nnn` ★

`\use:nnn` $\{\langle group_1 \rangle\} \{\langle group_2 \rangle\} \{\langle group_3 \rangle\}$

`\use:nnnn` ★

`\use:nnnn` $\{\langle group_1 \rangle\} \{\langle group_2 \rangle\} \{\langle group_3 \rangle\} \{\langle group_4 \rangle\}$

As illustrated, these functions absorb between one and four arguments, as indicated by the argument specifier. The braces surrounding each argument are removed and the remaining tokens are left in the input stream. The category code of these tokens is also fixed by this process (if it has not already been by some other absorption). All of these functions require only a single expansion to operate, so that one expansion of

```
\use:nn { abc } { { def } }
```

results in the input stream containing

```
abc { def }
```

i.e. only the outer braces are removed.

T_EXhackers note: The `\use:n` function is equivalent to L^AT_EX 2_ε's `\@firstofone`.

<code>\use_i:nn</code>	★	<code>\use_i:nn {⟨arg₁⟩} {⟨arg₂⟩}</code>
------------------------	---	--

<code>\use_ii:nn</code>	★
-------------------------	---

These functions absorb two arguments from the input stream. The function `\use_i:nn` discards the second argument, and leaves the content of the first argument in the input stream. `\use_ii:nn` discards the first argument and leaves the content of the second argument in the input stream. The category code of these tokens is also fixed (if it has not already been by some other absorption). A single expansion is needed for the functions to take effect.

TeXhackers note: These are equivalent to L^AT_EX 2_ε's `\@firstoftwo` and `\@secondoftwo`.

<code>\use_i:nnn</code>	★	<code>\use_i:nnn {⟨arg₁⟩} {⟨arg₂⟩} {⟨arg₃⟩}</code>
-------------------------	---	---

<code>\use_ii:nnn</code>	★
--------------------------	---

<code>\use_iii:nnn</code>	★
---------------------------	---

These functions absorb three arguments from the input stream. The function `\use_i:nnn` discards the second and third arguments, and leaves the content of the first argument in the input stream. `\use_ii:nnn` and `\use_iii:nnn` work similarly, leaving the content of second or third arguments in the input stream, respectively. The category code of these tokens is also fixed (if it has not already been by some other absorption). A single expansion is needed for the functions to take effect.

<code>\use_i:nnnn</code>	★	<code>\use_i:nnnn {⟨arg₁⟩} {⟨arg₂⟩} {⟨arg₃⟩} {⟨arg₄⟩}</code>
--------------------------	---	--

<code>\use_ii:nnnn</code>	★
---------------------------	---

<code>\use_iii:nnnn</code>	★
----------------------------	---

<code>\use_iv:nnnn</code>	★
---------------------------	---

These functions absorb four arguments from the input stream. The function `\use_i:nnnn` discards the second, third and fourth arguments, and leaves the content of the first argument in the input stream. `\use_ii:nnnn`, `\use_iii:nnnn` and `\use_iv:nnnn` work similarly, leaving the content of second, third or fourth arguments in the input stream, respectively. The category code of these tokens is also fixed (if it has not already been by some other absorption). A single expansion is needed for the functions to take effect.

<code>\use_i_ii:nnn</code>	★	<code>\use_i_ii:nnn {⟨arg₁⟩} {⟨arg₂⟩} {⟨arg₃⟩}</code>
----------------------------	---	--

This function absorbs three arguments and leaves the content of the first and second in the input stream. The category code of these tokens is also fixed (if it has not already been by some other absorption). A single expansion is needed for the function to take effect. An example:

`\use_i_ii:nnn { abc } { { def } } { ghi }`

results in the input stream containing

`abc { def }`

i.e. the outer braces are removed and the third group is removed.

<code>\use_ii_i:nn</code>	★	<code>\use_ii_i:nn {⟨arg₁⟩} {⟨arg₂⟩}</code>
---------------------------	---	---

New: 2019-06-02

This function absorbs two arguments and leaves the content of the second and first in the input stream. The category code of these tokens is also fixed (if it has not already been by some other absorption). A single expansion is needed for the function to take effect.

<hr/>		
<code>\use_none:n</code>	*	<code>\use_none:n {⟨group₁⟩}</code>
<code>\use_none:nn</code>	*	These functions absorb between one and nine groups from the input stream, leaving nothing on the resulting input stream. These functions work after a single expansion. One or more of the <code>n</code> arguments may be an unbraced single token (<i>i.e.</i> an <code>N</code> argument).
<code>\use_none:nnn</code>	*	
<code>\use_none:nnnn</code>	*	
<code>\use_none:nnnnn</code>	*	
<code>\use_none:nnnnnn</code>	*	
<code>\use_none:nnnnnnn</code>	*	TeXhackers note: These are equivalent to L ^A T _E X 2 _ε 's <code>\@gobble</code> , <code>\@gobbletwo</code> , <i>etc.</i>
<code>\use_none:nnnnnnnn</code>	*	
<code>\use_none:nnnnnnnnn</code>	*	
<code>\use_none:nnnnnnnnnn</code>	*	
<hr/>		

<hr/>		
<code>\use:e</code>	*	<code>\use:e {⟨expandable tokens⟩}</code>
<hr/>		
New: 2018-06-18		Fully expands the <i>⟨token list⟩</i> in an <code>x</code> -type manner, <i>but</i> the function remains fully expandable, and parameter character (usually <code>#</code>) need not be doubled.
<hr/>		

TeXhackers note: `\use:e` is a wrapper around the primitive `\expanded` where it is available: it requires two expansions to complete its action. When `\expanded` is not available this function is very slow.

<hr/>		
<code>\use:x</code>		<code>\use:x {⟨expandable tokens⟩}</code>
<hr/>		
Updated: 2011-12-31		Fully expands the <i>⟨expandable tokens⟩</i> and inserts the result into the input stream at the current location. Any hash characters (<code>#</code>) in the argument must be doubled.
<hr/>		

4.5.1 Selecting tokens from delimited arguments

A different kind of function for selecting tokens from the token stream are those that use delimited arguments.

<hr/>		
<code>\use_none_delimit_by_q_nil:w</code>	*	<code>\use_none_delimit_by_q_nil:w {⟨balanced text⟩} \q_nil</code>
<code>\use_none_delimit_by_q_stop:w</code>	*	<code>\use_none_delimit_by_q_stop:w {⟨balanced text⟩} \q_stop</code>
<code>\use_none_delimit_by_q_recursion_stop:w</code>	*	<code>\use_none_delimit_by_q_recursion_stop:w {⟨balanced text⟩} \q_recursion_stop</code>
<hr/>		

Absorb the *⟨balanced text⟩* from the input stream delimited by the marker given in the function name, leaving nothing in the input stream.

<hr/>		
<code>\use_i_delimit_by_q_nil:nw</code>	*	<code>\use_i_delimit_by_q_nil:nw {⟨inserted tokens⟩} {⟨balanced text⟩}</code>
<code>\use_i_delimit_by_q_stop:nw</code>	*	<code>\q_nil</code>
<code>\use_i_delimit_by_q_recursion_stop:nw</code>	*	<code>\use_i_delimit_by_q_stop:nw {⟨inserted tokens⟩} {⟨balanced text⟩} \q_stop</code>
<hr/>		
		<code>\use_i_delimit_by_q_recursion_stop:nw {⟨inserted tokens⟩} {⟨balanced text⟩} \q_recursion_stop</code>

Absorb the *⟨balanced text⟩* from the input stream delimited by the marker given in the function name, leaving *⟨inserted tokens⟩* in the input stream for further processing.

4.6 Predicates and conditionals

L^AT_EX3 has three concepts for conditional flow processing:

Branching conditionals Functions that carry out a test and then execute, depending on its result, either the code supplied as the *⟨true code⟩* or the *⟨false code⟩*. These arguments are denoted with T and F, respectively. An example would be

```
\cs_if_free:cTF {abc} {⟨true code⟩} {⟨false code⟩}
```

a function that turns the first argument into a control sequence (since it’s marked as c) then checks whether this control sequence is still free and then depending on the result carries out the code in the second argument (true case) or in the third argument (false case).

These type of functions are known as “conditionals”; whenever a TF function is defined it is usually accompanied by T and F functions as well. These are provided for convenience when the branch only needs to go a single way. Package writers are free to choose which types to define but the kernel definitions always provide all three versions.

Important to note is that these branching conditionals with *⟨true code⟩* and/or *⟨false code⟩* are always defined in a way that the code of the chosen alternative can operate on following tokens in the input stream.

These conditional functions may or may not be fully expandable, but if they are expandable they are accompanied by a “predicate” for the same test as described below.

Predicates “Predicates” are functions that return a special type of boolean value which can be tested by the boolean expression parser. All functions of this type are expandable and have names that end with `_p` in the description part. For example,

```
\cs_if_free_p:N
```

would be a predicate function for the same type of test as the conditional described above. It would return “true” if its argument (a single token denoted by N) is still free for definition. It would be used in constructions like

```
\bool_if:nTF {
  \cs_if_free_p:N \l_tmpz_tl || \cs_if_free_p:N \g_tmpz_tl
} {⟨true code⟩} {⟨false code⟩}
```

For each predicate defined, a “branching conditional” also exists that behaves like a conditional described above.

Primitive conditionals There is a third variety of conditional, which is the original concept used in plain T_EX and L^AT_EX 2_ε. Their use is discouraged in `expl3` (although still used in low-level definitions) because they are more fragile and in many cases require more expansion control (hence more code) than the two types of conditionals described above.

```
\c_true_bool
\c_false_bool
```

Constants that represent `true` and `false`, respectively. Used to implement predicates.

4.6.1 Tests on control sequences

<code>\cs_if_eq_p:NN</code>	★	<code>\cs_if_eq_p:NN <cs₁> <cs₂></code>
<code>\cs_if_eq:NNTF</code>	★	<code>\cs_if_eq:NNTF <cs₁> <cs₂> {\true code} {\false code}</code>

Compares the definition of two *<control sequences>* and is logically **true** if they are the same, *i.e.* if they have exactly the same definition when examined with `\cs_show:N`.

<code>\cs_if_exist_p:N</code>	★	<code>\cs_if_exist_p:N <control sequence></code>
<code>\cs_if_exist_p:c</code>	★	<code>\cs_if_exist:NNTF <control sequence> {\true code} {\false code}</code>
<code>\cs_if_exist:NTF</code>	★	Tests whether the <i><control sequence></i> is currently defined (whether as a function or another control sequence type). Any definition of <i><control sequence></i> other than <code>\relax</code> evaluates as true .
<code>\cs_if_exist:cTF</code>	★	

<code>\cs_if_free_p:N</code>	★	<code>\cs_if_free_p:N <control sequence></code>
<code>\cs_if_free_p:c</code>	★	<code>\cs_if_free:NNTF <control sequence> {\true code} {\false code}</code>
<code>\cs_if_free:NTF</code>	★	Tests whether the <i><control sequence></i> is currently free to be defined. This test is false if the <i><control sequence></i> currently exists (as defined by <code>\cs_if_exist:NNTF</code>).
<code>\cs_if_free:cTF</code>	★	

4.6.2 Primitive conditionals

The ε -TeX engine itself provides many different conditionals. Some expand whatever comes after them and others don't. Hence the names for these underlying functions often contains a `:w` part but higher level functions are often available. See for instance `\int_compare_p:nNn` which is a wrapper for `\if_int_compare:w`.

Certain conditionals deal with specific data types like boxes and fonts and are described there. The ones described below are either the universal conditionals or deal with control sequences. We prefix primitive conditionals with `\if_`.

<code>\if_true:</code>	★	<code>\if_true: <true code> \else: <false code> \fi:</code>
<code>\if_false:</code>	★	<code>\if_false: <true code> \else: <false code> \fi:</code>
<code>\else:</code>	★	<code>\reverse_if:N <primitive conditional></code>
<code>\fi:</code>	★	<code>\if_true:</code> always executes <i><true code></i> , while <code>\if_false:</code> always executes <i><false code></i> . <code>\reverse_if:N</code> reverses any two-way primitive conditional. <code>\else:</code> and <code>\fi:</code> delimit the branches of the conditional. The function <code>\or:</code> is documented in <code>l3int</code> and used in case switches.
<code>\reverse_if:N</code>	★	

TeXhackers note: These are equivalent to their corresponding TeX primitive conditionals; `\reverse_if:N` is ε -TeX's `\unless`.

<code>\if_meaning:w</code>	★	<code>\if_meaning:w <arg₁> <arg₂> <true code> \else: <false code> \fi:</code>
----------------------------	---	---

`\if_meaning:w` executes *<true code>* when *<arg₁>* and *<arg₂>* are the same, otherwise it executes *<false code>*. *<arg₁>* and *<arg₂>* could be functions, variables, tokens; in all cases the *unexpanded* definitions are compared.

TeXhackers note: This is TeX's `\ifx`.

<code>\if:w</code>	★	<code>\if:w <token₁> <token₂> <true code> \else: <false code> \fi:</code>
<code>\if_charcode:w</code>	★	<code>\if_catcode:w <token₁> <token₂> <true code> \else: <false code> \fi:</code>
<code>\if_catcode:w</code>	★	These conditionals expand any following tokens until two unexpandable tokens are left. If you wish to prevent this expansion, prefix the token in question with <code>\exp_not:N</code> . <code>\if_catcode:w</code> tests if the category codes of the two tokens are the same whereas <code>\if:w</code> tests if the character codes are identical. <code>\if_charcode:w</code> is an alternative name for <code>\if:w</code> .

<code>\if_cs_exist:N</code>	★	<code>\if_cs_exist:N <cs> <true code> \else: <false code> \fi:</code>
<code>\if_cs_exist:w</code>	★	<code>\if_cs_exist:w <tokens> \cs_end: <true code> \else: <false code> \fi:</code>

Check if `<cs>` appears in the hash table or if the control sequence that can be formed from `<tokens>` appears in the hash table. The latter function does not turn the control sequence in question into `\scan_stop:!` This can be useful when dealing with control sequences which cannot be entered as a single token.

<code>\if_mode_horizontal:</code>	★	<code>\if_mode_horizontal: <true code> \else: <false code> \fi:</code>
<code>\if_mode_vertical:</code>	★	Execute <code><true code></code> if currently in horizontal mode, otherwise execute <code><false code></code> . Similar for the other functions.
<code>\if_mode_math:</code>	★	
<code>\if_mode_inner:</code>	★	

4.7 Starting a paragraph

<code>\mode_leave_vertical:</code>	<code>\mode_leave_vertical:</code>
------------------------------------	------------------------------------

New: 2017-07-04

Ensures that `TEX` is not in vertical (inter-paragraph) mode. In horizontal or math mode this command has no effect, in vertical mode it switches to horizontal mode, and inserts a box of width `\parindent`, followed by the `\everypar` token list.

T_EXhackers note: This results in the contents of the `\everypar` token register being inserted, after `\mode_leave_vertical:` is complete. Notice that in contrast to the L^AT_EX 2_ε `\leavevmode` approach, no box is used by the method implemented here.

4.8 Debugging support

`\debug_on:n`
`\debug_off:n`

New: 2017-07-16
Updated: 2017-08-02

`\debug_on:n { <comma-separated list> }`
`\debug_off:n { <comma-separated list> }`

Turn on and off within a group various debugging code, some of which is also available as `expl3` load-time options. The items that can be used in the *<list>* are

- **check-declarations** that checks all `expl3` variables used were previously declared and that local/global variables (based on their name or on their first assignment) are only locally/globally assigned;
- **check-expressions** that checks integer, dimension, skip, and muskip expressions are not terminated prematurely;
- **deprecation** that makes soon-to-be-deprecated commands produce errors;
- **log-functions** that logs function definitions;
- **all** that does all of the above.

Providing these as switches rather than options allows testing code even if it relies on other packages: load all other packages, call `\debug_on:n`, and load the code that one is interested in testing. These functions can only be used in L^AT_EX 2_ε package mode loaded with `enable-debug` or another option implying it.

`\debug_suspend:`
`\debug_resume:`

New: 2017-11-28

`\debug_suspend: ... \debug_resume:`

Suppress (locally) errors and logging from `debug` commands, except for the **deprecation** errors or warnings. These pairs of commands can be nested. This can be used around pieces of code that are known to fail checks, if such failures should be ignored. See for instance `l3coffins`.

Chapter 5

The l3expan package

Argument expansion

This module provides generic methods for expanding \TeX arguments in a systematic manner. The functions in this module all have prefix `exp`.

Not all possible variations are implemented for every base function. Instead only those that are used within the \LaTeX 3 kernel or otherwise seem to be of general interest are implemented. Consult the module description to find out which functions are actually defined. The next section explains how to define missing variants.

5.1 Defining new variants

The definition of variant forms for base functions may be necessary when writing new functions or when applying a kernel function in a situation that we haven't thought of before.

Internally preprocessing of arguments is done with functions of the form `\exp_....`. They all look alike, an example would be `\exp_args:NNo`. This function has three arguments, the first and the second are a single tokens, while the third argument should be given in braces. Applying `\exp_args:NNo` expands the content of third argument once before any expansion of the first and second arguments. If `\seq_gpush:No` was not defined it could be coded in the following way:

```
\exp_args:NNo \seq_gpush:Nn
  \g_file_name_stack
  { \l_tmpa_tl }
```

In other words, the first argument to `\exp_args:NNo` is the base function and the other arguments are preprocessed and then passed to this base function. In the example the first argument to the base function should be a single token which is left unchanged while the second argument is expanded once. From this example we can also see how the variants are defined. They just expand into the appropriate `\exp_` function followed by the desired base function, *e.g.*

```
\cs_generate_variant:Nn \seq_gpush:Nn { No }
```

results in the definition of `\seq_gpush:No`

```
\cs_new:Npn \seq_gpush:No { \exp_args:NNo \seq_gpush:Nn }
```

Providing variants in this way in style files is safe as the `\cs_generate_variant:Nn` function will only create new definitions if there is not already one available. Therefore adding such definition to later releases of the kernel will not make such style files obsolete.

The steps above may be automated by using the function `\cs_generate_variant:Nn`, described next.

5.2 Methods for defining variants

We recall the set of available argument specifiers.

- `N` is used for single-token arguments while `c` constructs a control sequence from its name and passes it to a parent function as an `N`-type argument.
- Many argument types extract or expand some tokens and provide it as an `n`-type argument, namely a braced multiple-token argument: `V` extracts the value of a variable, `v` extracts the value from the name of a variable, `n` uses the argument as it is, `o` expands once, `f` expands fully the front of the token list, `e` and `x` expand fully all tokens (differences are explained later).
- A few odd argument types remain: `T` and `F` for conditional processing, otherwise identical to `n`-type arguments, `p` for the parameter text in definitions, `w` for arguments with a specific syntax, and `D` to denote primitives that should not be used directly.

`\cs_generate_variant:Nn`
`\cs_generate_variant:cn`

Updated: 2017-11-28

`\cs_generate_variant:Nn` \langle parent control sequence \rangle $\{$ \langle variant argument specifiers \rangle $\}$

This function is used to define argument-specifier variants of the \langle parent control sequence \rangle for L^AT_EX3 code-level macros. The \langle parent control sequence \rangle is first separated into the \langle base name \rangle and \langle original argument specifier \rangle . The comma-separated list of \langle variant argument specifiers \rangle is then used to define variants of the \langle original argument specifier \rangle if these are not already defined. For each \langle variant \rangle given, a function is created that expands its arguments as detailed and passes them to the \langle parent control sequence \rangle . So for example

```
\cs_set:Npn \foo:Nn #1#2 { code here }
\cs_generate_variant:Nn \foo:Nn { c }
```

creates a new function `\foo:cn` which expands its first argument into a control sequence name and passes the result to `\foo:Nn`. Similarly

```
\cs_generate_variant:Nn \foo:Nn { NV , cV }
```

generates the functions `\foo:NV` and `\foo:cV` in the same way. The `\cs_generate_variant:Nn` function can only be applied if the \langle parent control sequence \rangle is already defined. If the \langle parent control sequence \rangle is protected or if the \langle variant \rangle involves any **x** argument, then the \langle variant control sequence \rangle is also protected. The \langle variant \rangle is created globally, as is any `\exp_args:N` \langle variant \rangle function needed to carry out the expansion.

Only **n** and **N** arguments can be changed to other types. The only allowed changes are

- **c** variant of an **N** parent;
- **o**, **V**, **v**, **f**, **e**, or **x** variant of an **n** parent;
- **N**, **n**, **T**, **F**, or **p** argument unchanged.

This means the \langle parent \rangle of a \langle variant \rangle form is always unambiguous, even in cases where both an **n**-type parent and an **N**-type parent exist, such as for `\tl_count:n` and `\tl_count:N`.

For backward compatibility it is currently possible to make **n**, **o**, **V**, **v**, **f**, **e**, or **x**-type variants of an **N**-type argument or **N** or **c**-type variants of an **n**-type argument. Both are deprecated. The first because passing more than one token to an **N**-type argument will typically break the parent function's code. The second because programmers who use that most often want to access the value of a variable given its name, hence should use a **V**-type or **v**-type variant instead of **c**-type. In those cases, using the lower-level `\exp_args:No` or `\exp_args:Nc` functions explicitly is preferred to defining confusing variants.

5.3 Introducing the variants

The **V** type returns the value of a register, which can be one of **tl**, **clist**, **int**, **skip**, **dim**, **muskip**, or built-in T_EX registers. The **v** type is the same except it first creates a control sequence out of its argument before returning the value.

In general, the programmer should not need to be concerned with expansion control. When simply using the content of a variable, functions with a **V** specifier should be used. For those referred to by (cs)name, the **v** specifier is available for the same purpose. Only

when specific expansion steps are needed, such as when using delimited arguments, should the lower-level functions with `o` specifiers be employed.

The `e` type expands all tokens fully, starting from the first. More precisely the expansion is identical to that of T_EX’s `\message` (in particular `#` needs not be doubled). It was added in May 2018. In recent enough engines (starting around 2019) it relies on the primitive `\expanded` hence is fast. In older engines it is very much slower. As a result it should only be used in performance critical code if typical users will have a recent installation of the T_EX ecosystem.

The `x` type expands all tokens fully, starting from the first. In contrast to `e`, all macro parameter characters `#` must be doubled, and omitting this leads to low-level errors. In addition this type of expansion is not expandable, namely functions that have `x` in their signature do not themselves expand when appearing inside `x` or `e` expansion.

The `f` type is so special that it deserves an example. It is typically used in contexts where only expandable commands are allowed. Then `x`-expansion cannot be used, and `f`-expansion provides an alternative that expands the front of the token list as much as can be done in such contexts. For instance, say that we want to evaluate the integer expression `3 + 4` and pass the result 7 as an argument to an expandable function `\example:n`. For this, one should define a variant using `\cs_generate_variant:Nn \example:n { f }`, then do

```
\example:f { \int_eval:n { 3 + 4 } }
```

Note that `x`-expansion would also expand `\int_eval:n` fully to its result 7, but the variant `\example:x` cannot be expandable. Note also that `o`-expansion would not expand `\int_eval:n` fully to its result since that function requires several expansions. Besides the fact that `x`-expansion is protected rather than expandable, another difference between `f`-expansion and `x`-expansion is that `f`-expansion expands tokens from the beginning and stops as soon as a non-expandable token is encountered, while `x`-expansion continues expanding further tokens. Thus, for instance

```
\example:f { \int_eval:n { 1 + 2 } , \int_eval:n { 3 + 4 } }
```

results in the call

```
\example:n { 3 , \int_eval:n { 3 + 4 } }
```

while using `\example:x` or `\example:e` instead results in

```
\example:n { 3 , 7 }
```

at the cost of being protected (for `x` type) or very much slower in old engines (for `e` type). If you use `f` type expansion in conditional processing then you should stick to using TF type functions only as the expansion does not finish any `\if... \fi`: itself!

It is important to note that both `f`- and `o`-type expansion are concerned with the expansion of tokens from left to right in their arguments. In particular, `o`-type expansion applies to the first *token* in the argument it receives: it is conceptually similar to

```
\exp_after:wN <base function> \exp_after:wN { <argument> }
```

At the same time, `f`-type expansion stops at the *first* non-expandable token. This means for example that both

```
\tl_set:No \l_tmpa_tl { { \g_tmpb_tl } }
```

and

```
\tl_set:Nf \l_tmpa_tl { { \g_tmpb_tl } }
```

leave `\g_tmpb_tl` unchanged: `{` is the first token in the argument and is non-expandable. It is usually best to keep the following in mind when using variant forms.

- Variants with `x`-type arguments (that are fully expanded before being passed to the `n`-type base function) are never expandable even when the base function is. Such variants cannot work correctly in arguments that are themselves subject to expansion. Consider using `f` or `e` expansion.
- In contrast, `e` expansion (full expansion, almost like `x` except for the treatment of `#`) does not prevent variants from being expandable (if the base function is). The drawback is that `e` expansion is very much slower in old engines (before 2019). Consider using `f` expansion if that type of expansion is sufficient to perform the required expansion, or `x` expansion if the variant will not itself need to be expandable.
- Finally `f` expansion only expands the front of the token list, stopping at the first non-expandable token. This may fail to fully expand the argument.

When speed is essential (for functions that do very little work and whose variants are used numerous times in a document) the following considerations apply because internal functions for argument expansion come in two flavours, some faster than others.

- Arguments that might need expansion should come first in the list of arguments.
- Arguments that should consist of single tokens `N`, `c`, `V`, or `v` should come first among these.
- Arguments that appear after the first multi-token argument `n`, `f`, `e`, or `o` require slightly slower special processing to be expanded. Therefore it is best to use the optimized functions, namely those that contain only `N`, `c`, `V`, and `v`, and, in the last position, `o`, `f`, `e`, with possible trailing `N` or `n` or `T` or `F`, which are not expanded. Any `x`-type argument causes slightly slower processing.

5.4 Manipulating the first argument

These functions are described in detail: expansion of multiple tokens follows the same rules but is described in a shorter fashion.

<code>\exp_args:Nc</code>	★	<code>\exp_args:Nc</code>	<code><function></code>	<code>{<tokens>}</code>
<code>\exp_args:cc</code>	★			

This function absorbs two arguments (the `<function>` name and the `<tokens>`). The `<tokens>` are expanded until only characters remain, and are then turned into a control sequence. The result is inserted into the input stream *after* reinsertion of the `<function>`. Thus the `<function>` may take more than one argument: all others are left unchanged.

The `:cc` variant constructs the `<function>` name in the same manner as described for the `<tokens>`.

T_EXhackers note: Protected macros that appear in a `c`-type argument are expanded despite being protected; `\exp_not:n` also has no effect. An internal error occurs if non-characters or active characters remain after full expansion, as the conversion to a control sequence is not possible.

`\exp_args:No` ★ `\exp_args:No` $\langle function \rangle$ $\{\langle tokens \rangle\}$...

This function absorbs two arguments (the $\langle function \rangle$ name and the $\langle tokens \rangle$). The $\langle tokens \rangle$ are expanded once, and the result is inserted in braces into the input stream *after* reinsertion of the $\langle function \rangle$. Thus the $\langle function \rangle$ may take more than one argument: all others are left unchanged.

`\exp_args:Nv` ★ `\exp_args:Nv` $\langle function \rangle$ $\langle variable \rangle$

This function absorbs two arguments (the names of the $\langle function \rangle$ and the $\langle variable \rangle$). The content of the $\langle variable \rangle$ are recovered and placed inside braces into the input stream *after* reinsertion of the $\langle function \rangle$. Thus the $\langle function \rangle$ may take more than one argument: all others are left unchanged.

`\exp_args:Nv` ★ `\exp_args:Nv` $\langle function \rangle$ $\{\langle tokens \rangle\}$

This function absorbs two arguments (the $\langle function \rangle$ name and the $\langle tokens \rangle$). The $\langle tokens \rangle$ are expanded until only characters remain, and are then turned into a control sequence. This control sequence should be the name of a $\langle variable \rangle$. The content of the $\langle variable \rangle$ are recovered and placed inside braces into the input stream *after* reinsertion of the $\langle function \rangle$. Thus the $\langle function \rangle$ may take more than one argument: all others are left unchanged.

TeXhackers note: Protected macros that appear in a v-type argument are expanded despite being protected; `\exp_not:n` also has no effect. An internal error occurs if non-characters or active characters remain after full expansion, as the conversion to a control sequence is not possible.

`\exp_args:Ne` ★ `\exp_args:Ne` $\langle function \rangle$ $\{\langle tokens \rangle\}$

New: 2018-05-15

This function absorbs two arguments (the $\langle function \rangle$ name and the $\langle tokens \rangle$) and exhaustively expands the $\langle tokens \rangle$. The result is inserted in braces into the input stream *after* reinsertion of the $\langle function \rangle$. Thus the $\langle function \rangle$ may take more than one argument: all others are left unchanged.

TeXhackers note: This relies on the `\expanded` primitive when available (in LuaTeX and starting around 2019 in other engines). Otherwise it uses some fall-back code that is very much slower. As a result it should only be used in performance-critical code if typical users have a recent installation of the TeX ecosystem.

`\exp_args:Nf` ★ `\exp_args:Nf` $\langle function \rangle$ $\{\langle tokens \rangle\}$

This function absorbs two arguments (the $\langle function \rangle$ name and the $\langle tokens \rangle$). The $\langle tokens \rangle$ are fully expanded until the first non-expandable token is found (if that is a space it is removed), and the result is inserted in braces into the input stream *after* reinsertion of the $\langle function \rangle$. Thus the $\langle function \rangle$ may take more than one argument: all others are left unchanged.

<code>\exp_args:Nx</code>	<code>\exp_args:Nx</code>	$\langle function \rangle$	$\{\langle tokens \rangle\}$
---------------------------	---------------------------	----------------------------	------------------------------

This function absorbs two arguments (the $\langle function \rangle$ name and the $\langle tokens \rangle$) and exhaustively expands the $\langle tokens \rangle$. The result is inserted in braces into the input stream *after* reinsertion of the $\langle function \rangle$. Thus the $\langle function \rangle$ may take more than one argument: all others are left unchanged.

5.5 Manipulating two arguments

<code>\exp_args:NNc</code>	<code>\exp_args:NNc</code>	$\langle token_1 \rangle$	$\langle token_2 \rangle$	$\{\langle tokens \rangle\}$
----------------------------	----------------------------	---------------------------	---------------------------	------------------------------

These optimized functions absorb three arguments and expand the second and third as detailed by their argument specifier. The first argument of the function is then the next item on the input stream, followed by the expansion of the second and third arguments.

`\exp_args:NNv` *
`\exp_args:NNe` *
`\exp_args:NNf` *
`\exp_args:Ncc` *
`\exp_args:Nco` *
`\exp_args:NcV` *
`\exp_args:Ncv` *
`\exp_args:Ncf` *
`\exp_args:NVV` *

Updated: 2018-05-15

<code>\exp_args:Nnc</code>	<code>\exp_args:Noo</code>	$\langle token \rangle$	$\{\langle tokens_1 \rangle\}$	$\{\langle tokens_2 \rangle\}$
----------------------------	----------------------------	-------------------------	--------------------------------	--------------------------------

These functions absorb three arguments and expand the second and third as detailed by their argument specifier. The first argument of the function is then the next item on the input stream, followed by the expansion of the second and third arguments. These functions need slower processing.

`\exp_args:Noc` *
`\exp_args:Noo` *
`\exp_args:Nof` *
`\exp_args:NVo` *
`\exp_args:Nfo` *
`\exp_args:Nff` *
`\exp_args:Nee` *

Updated: 2018-05-15

<code>\exp_args:NNx</code>	<code>\exp_args:NNx</code>	$\langle token_1 \rangle$	$\langle token_2 \rangle$	$\{\langle tokens \rangle\}$
----------------------------	----------------------------	---------------------------	---------------------------	------------------------------

These functions absorb three arguments and expand the second and third as detailed by their argument specifier. The first argument of the function is then the next item on the input stream, followed by the expansion of the second and third arguments. These functions are not expandable due to their x-type argument.

`\exp_args:Ncx`
`\exp_args:Nnx`
`\exp_args:Nox`
`\exp_args:Nxo`
`\exp_args:Nxx`

5.6 Manipulating three arguments

<hr/>	
<code>\exp_args:NNNo *</code>	<code>\exp_args:NNNo <token₁> <token₂> <token₃> {\tokens}</code>
<code>\exp_args:NNNV *</code>	
<code>\exp_args:NNNv *</code>	These optimized functions absorb four arguments and expand the second, third and fourth as detailed by their argument specifier. The first argument of the function is then
<code>\exp_args:Nccc *</code>	the next item on the input stream, followed by the expansion of the second argument,
<code>\exp_args:NcNc *</code>	<i>etc.</i>
<code>\exp_args:NcNo *</code>	
<code>\exp_args:Ncco *</code>	
<hr/>	
<code>\exp_args:NNcf *</code>	<code>\exp_args:NNoo <token₁> <token₂> {\tokens₃} {\tokens}</code>
<code>\exp_args:NNno *</code>	
<code>\exp_args:NNnV *</code>	These functions absorb four arguments and expand the second, third and fourth as de-
<code>\exp_args:NNoo *</code>	tailed by their argument specifier. The first argument of the function is then the next
<code>\exp_args:NNVV *</code>	item on the input stream, followed by the expansion of the second argument, <i>etc.</i> These
<code>\exp_args:Ncno *</code>	functions need slower processing.
<code>\exp_args:NcnV *</code>	
<code>\exp_args:Ncoo *</code>	
<code>\exp_args:NcVV *</code>	
<code>\exp_args:Nnnc *</code>	
<code>\exp_args:Nnno *</code>	
<code>\exp_args:Nnnf *</code>	
<code>\exp_args:Nnff *</code>	
<code>\exp_args:Nooo *</code>	
<code>\exp_args:Noof *</code>	
<code>\exp_args:Nffo *</code>	
<code>\exp_args:Neee *</code>	
<hr/>	
<code>\exp_args:NNNx *</code>	<code>\exp_args:NNnx <token₁> <token₂> {\tokens₁} {\tokens₂}</code>
<code>\exp_args:NNnx *</code>	
<code>\exp_args:NNox *</code>	These functions absorb four arguments and expand the second, third and fourth as de-
<code>\exp_args:Nccx *</code>	tailed by their argument specifier. The first argument of the function is then the next
<code>\exp_args:Ncnx *</code>	item on the input stream, followed by the expansion of the second argument, <i>etc.</i>
<code>\exp_args:Nnnx *</code>	
<code>\exp_args:Nnox *</code>	
<code>\exp_args:Noox *</code>	

New: 2015-08-12

5.7 Unbraced expansion

```

\exp_last_unbraced:No  *
\exp_last_unbraced:NV  *
\exp_last_unbraced:Nv  *
\exp_last_unbraced:Ne  *
\exp_last_unbraced:Nf  *
\exp_last_unbraced:NNo *
\exp_last_unbraced:NNV *
\exp_last_unbraced:NNf *
\exp_last_unbraced:Nco *
\exp_last_unbraced:NcV *
\exp_last_unbraced:Nno *
\exp_last_unbraced:Noo *
\exp_last_unbraced:Nfo *
\exp_last_unbraced:NNNo *
\exp_last_unbraced:NNNV *
\exp_last_unbraced:NNNf *
\exp_last_unbraced:NnNo *
\exp_last_unbraced:NNNNo *
\exp_last_unbraced:NNNNf *

```

Updated: 2018-05-15

```
\exp_last_unbraced:Nno <token> {\tokens1} {\tokens2}
```

These functions absorb the number of arguments given by their specification, carry out the expansion indicated and leave the results in the input stream, with the last argument not surrounded by the usual braces. Of these, the :Nno, :Noo, :Nfo and :NnNo variants need slower processing.

T_EXhackers note: As an optimization, the last argument is unbraced by some of those functions before expansion. This can cause problems if the argument is empty: for instance, `\exp_last_unbraced:Nf \foo_bar:w { } \q_stop` leads to an infinite loop, as the quark is f-expanded.

```
\exp_last_unbraced:Nx \exp_last_unbraced:Nx <function> {\tokens}
```

This function fully expands the $\langle tokens \rangle$ and leaves the result in the input stream after reinsertion of the $\langle function \rangle$. This function is not expandable.

```
\exp_last_two_unbraced:Noo * \exp_last_two_unbraced:Noo <token> {\tokens1} {\tokens2}
```

This function absorbs three arguments and expands the second and third once. The first argument of the function is then the next item on the input stream, followed by the expansion of the second and third arguments, which are not wrapped in braces. This function needs special (slower) processing.

```
\exp_after:wN * \exp_after:wN <token12


---



```

Carries out a single expansion of $\langle token_2 \rangle$ (which may consume arguments) prior to the expansion of $\langle token_1 \rangle$. If $\langle token_2 \rangle$ has no expansion (for example, if it is a character) then it is left unchanged. It is important to notice that $\langle token_1 \rangle$ may be *any* single token, including group-opening and -closing tokens (`{` or `}` assuming normal T_EX category codes). Unless specifically required this should be avoided: expansion should be carried out using an appropriate argument specifier variant or the appropriate `\exp_arg:N` function.

T_EXhackers note: This is the T_EX primitive `\expandafter` renamed.

5.8 Preventing expansion

Despite the fact that the following functions are all about preventing expansion, they're designed to be used in an expandable context and hence are all marked as being 'expand-

able' since they themselves disappear after the expansion has completed.

`\exp_not:N` ★ `\exp_not:N` $\langle token \rangle$

Prevents expansion of the $\langle token \rangle$ in a context where it would otherwise be expanded, for example an **x**-type argument or the first token in an **o** or **e** or **f** argument.

T_EXhackers note: This is the T_EX `\noexpand` primitive. It only prevents expansion. At the beginning of an **f**-type argument, a space $\langle token \rangle$ is removed even if it appears as `\exp_not:N \c_space_token`. In an **x**-expanding definition (`\cs_new:Npx`), a macro parameter introduces an argument even if it appears as `\exp_not:N # 1`. This differs from `\exp_not:n`.

`\exp_not:c` ★ `\exp_not:c` $\{\langle tokens \rangle\}$

Expands the $\langle tokens \rangle$ until only characters remain, and then converts this into a control sequence. Further expansion of this control sequence is then inhibited using `\exp_not:N`.

T_EXhackers note: Protected macros that appear in a **c**-type argument are expanded despite being protected; `\exp_not:n` also has no effect. An internal error occurs if non-characters or active characters remain after full expansion, as the conversion to a control sequence is not possible.

`\exp_not:n` ★ `\exp_not:n` $\{\langle tokens \rangle\}$

Prevents expansion of the $\langle tokens \rangle$ in an **e** or **x**-type argument. In all other cases the $\langle tokens \rangle$ continue to be expanded, for example in the input stream or in other types of arguments such as **c**, **f**, **v**. The argument of `\exp_not:n` *must* be surrounded by braces.

T_EXhackers note: This is the ε -T_EX `\unexpanded` primitive. In an **x**-expanding definition (`\cs_new:Npx`), `\exp_not:n {\#1}` is equivalent to `##1` rather than to `#1`, namely it inserts the two characters `#` and `1`. In an **e**-type argument `\exp_not:n {\#}` is equivalent to `#`, namely it inserts the character `#`.

`\exp_not:o` ★ `\exp_not:o` $\{\langle tokens \rangle\}$

Expands the $\langle tokens \rangle$ once, then prevents any further expansion in **x**-type or **e**-type arguments using `\exp_not:n`.

`\exp_not:V` ★ `\exp_not:V` $\langle variable \rangle$

Recovers the content of the $\langle variable \rangle$, then prevents expansion of this material in **x**-type or **e**-type arguments using `\exp_not:n`.

<hr/> <hr/>	<code>\exp_not:v *</code>	<code>\exp_not:v {<tokens>}</code>	Expands the <i><tokens></i> until only characters remains, and then converts this into a control sequence which should be a <i><variable></i> name. The content of the <i><variable></i> is recovered, and further expansion in x -type or e -type arguments is prevented using <code>\exp_not:n</code> .
<hr/> <hr/>	<code>\exp_not:e *</code>	<code>\exp_not:e {<tokens>}</code>	Expands <i><tokens></i> exhaustively, then protects the result of the expansion (including any tokens which were not expanded) from further expansion in e or x -type arguments using <code>\exp_not:n</code> . This is very rarely useful but is provided for consistency.
<hr/> <hr/>	<code>\exp_not:f *</code>	<code>\exp_not:f {<tokens>}</code>	Expands <i><tokens></i> fully until the first unexpandable token is found (if it is a space it is removed). Expansion then stops, and the result of the expansion (including any tokens which were not expanded) is protected from further expansion in x -type or e -type arguments using <code>\exp_not:n</code> .
<hr/> <hr/>	<code>\exp_stop_f: *</code>	<code>\foo_bar:f { <tokens> \exp_stop_f: <more tokens> }</code>	This function terminates an f -type expansion. Thus if a function <code>\foo_bar:f</code> starts an f -type expansion and all of <i><tokens></i> are expandable <code>\exp_stop_f:</code> terminates the expansion of tokens even if <i><more tokens></i> are also expandable. The function itself is an implicit space token. Inside an x -type or e -type expansion, it retains its form, but when typeset it produces the underlying space (\sqcup).

Updated: 2011-06-03

5.9 Controlled expansion

The `expl3` language makes all efforts to hide the complexity of `TeX` expansion from the programmer by providing concepts that evaluate/expand arguments of functions prior to calling the “base” functions. Thus, instead of using many `\expandafter` calls and other trickery it is usually a matter of choosing the right variant of a function to achieve a desired result.

Of course, deep down `TeX` is using expansion as always and there are cases where a programmer needs to control that expansion directly; typical situations are basic data manipulation tools. This section documents the functions for that level. These commands are used throughout the kernel code, but we hope that outside the kernel there will be little need to resort to them. Instead the argument manipulation methods document above should usually be sufficient.

While `\exp_after:wN` expands one token (out of order) it is sometimes necessary to expand several tokens in one go. The next set of commands provide this functionality. Be aware that it is absolutely required that the programmer has full control over the tokens to be expanded, i.e., it is not possible to use these functions to expand unknown input as part of *<expandable-tokens>* as that will break badly if unexpandable tokens are encountered in that place!

<code>\exp:w</code>	★	<code>\exp:w <expandable tokens> \exp_end:</code>
<code>\exp_end:</code>	★	Expands <code><expandable-tokens></code> until reaching <code>\exp_end:</code> at which point expansion stops. The full expansion of <code><expandable tokens></code> has to be empty. If any token in <code><expandable tokens></code> or any token generated by expanding the tokens therein is not expandable the expansion will end prematurely and as a result <code>\exp_end:</code> will be misinterpreted later on. ⁴
New: 2015-08-23		

In typical use cases the `\exp_end:` is hidden somewhere in the replacement text of `<expandable-tokens>` rather than being on the same expansion level than `\exp:w`, e.g., you may see code such as

```
\exp:w \@@_case:NnTF #1 {#2} { } { }
```

where somewhere during the expansion of `\@@_case:NnTF` the `\exp_end:` gets generated.

T_EXhackers note: The current implementation uses `\romannumeral` hence ignores space tokens and explicit signs + and - in the expansion of the `<expandable tokens>`, but this should not be relied upon.

<code>\exp:w</code>	★	<code>\exp:w <expandable-tokens> \exp_end_continue_f:w <further-tokens></code>
<code>\exp_end_continue_f:w</code>	★	Expands <code><expandable-tokens></code> until reaching <code>\exp_end_continue_f:w</code> at which point expansion continues as an f-type expansion expanding <code><further-tokens></code> until an unexpandable token is encountered (or the f-type expansion is explicitly terminated by <code>\exp_stop_f:</code>). As with all f-type expansions a space ending the expansion gets removed. The full expansion of <code><expandable-tokens></code> has to be empty. If any token in <code><expandable-tokens></code> or any token generated by expanding the tokens therein is not expandable the expansion will end prematurely and as a result <code>\exp_end_continue_f:w</code> will be misinterpreted later on. ⁵
New: 2015-08-23		

The full expansion of `<expandable-tokens>` has to be empty. If any token in `<expandable-tokens>` or any token generated by expanding the tokens therein is not expandable the expansion will end prematurely and as a result `\exp_end_continue_f:w` will be misinterpreted later on.⁵

In typical use cases `<expandable-tokens>` contains no tokens at all, e.g., you will see code such as

```
\exp_after:wN { \exp:w \exp_end_continue_f:w #2 }
```

where the `\exp_after:wN` triggers an f-expansion of the tokens in #2. For technical reasons this has to happen using two tokens (if they would be hidden inside another command `\exp_after:wN` would only expand the command but not trigger any additional f-expansion).

You might wonder why there are two different approaches available, after all the effect of

```
\exp:w <expandable-tokens> \exp_end:
```

can be alternatively achieved through an f-type expansion by using `\exp_stop_f:`, i.e.

```
\exp:w \exp_end_continue_f:w <expandable-tokens> \exp_stop_f:
```

The reason is simply that the first approach is slightly faster (one less token to parse and less expansion internally) so in places where such performance really matters and where we want to explicitly stop the expansion at a defined point the first form is preferable.

⁴Due to the implementation you might get the character in position 0 in the current font (typically “”) in the output without any error message!

⁵In this particular case you may get a character into the output as well as an error message.

<code>\exp:w</code>	★
<code>\exp_end_continue_f:nw</code>	★

New: 2015-08-23

`\exp:w` *<expandable-tokens>* `\exp_end_continue_f:nw` *<further-tokens>*

The difference to `\exp_end_continue_f:w` is that we first we pick up an argument which is then returned to the input stream. If *<further-tokens>* starts with space tokens then these space tokens are removed while searching for the argument. If it starts with a brace group then the braces are removed. Thus such spaces or braces will not terminate the f-type expansion.

5.10 Internal functions

`\::n` `\cs_new:Npn \exp_args:Ncof { \::c \::o \::f \::: }`

`\::N`

`\::p`

`\::c`

`\::o`

`\::e`

`\::f`

`\::x`

`\::v`

`\::V`

`\:::`

Internal forms for the base expansion types. These names do *not* conform to the general L^AT_EX3 approach as this makes them more readily visible in the log and so forth. They should not be used outside this module.

`\::o_unbraced` `\cs_new:Npn \exp_last_unbraced:Nno { \::n \::o_unbraced \::: }`

`\::e_unbraced`

`\::f_unbraced`

`\::x_unbraced`

`\::v_unbraced`

`\::V_unbraced`

Internal forms for the expansion types which leave the terminal argument unbraced. These names do *not* conform to the general L^AT_EX3 approach as this makes them more readily visible in the log and so forth. They should not be used outside this module.

Chapter 6

The l3sort package

Sorting functions

6.1 Controlling sorting

L^AT_EX3 comes with a facility to sort list variables (sequences, token lists, or comma-lists) according to some user-defined comparison. For instance,

```
\clist_set:Nn \l_foo_clist { 3 , 01 , -2 , 5 , +1 }
\clist_sort:Nn \l_foo_clist
{
  \int_compare:nNnTF { #1 } > { #2 }
  { \sort_return_swapped: }
  { \sort_return_same: }
}
```

results in `\l_foo_clist` holding the values `{ -2 , 01 , +1 , 3 , 5 }` sorted in non-decreasing order.

The code defining the comparison should call `\sort_return_swapped:` if the two items given as `#1` and `#2` are not in the correct order, and otherwise it should call `\sort_return_same:` to indicate that the order of this pair of items should not be changed.

For instance, a *comparison code* consisting only of `\sort_return_same:` with no test yields a trivial sort: the final order is identical to the original order. Conversely, using a *comparison code* consisting only of `\sort_return_swapped:` reverses the list (in a fairly inefficient way).

T_EXhackers note: The current implementation is limited to sorting approximately 20000 items (40000 in LuaT_EX), depending on what other packages are loaded.

Internally, the code from l3sort stores items in `\toks` registers allocated locally. Thus, the *comparison code* should not call `\newtoks` or other commands that allocate new `\toks` registers. On the other hand, altering the value of a previously allocated `\toks` register is not a problem.

`\sort_return_same:`
`\sort_return_swapped:`

New: 2017-02-06

`\seq_sort:Nn <seq var>`
`{ ... \sort_return_same: or \sort_return_swapped: ... }`

Indicates whether to keep the order or swap the order of two items that are compared in the sorting code. Only one of the `\sort_return_...` functions should be used by the code, according to the results of some tests on the items #1 and #2 to be compared.

Chapter 7

The l3tl-analysis package: Analysing token lists

This module provides functions that are particularly useful in the `l3regex` module for mapping through a token list one $\langle token \rangle$ at a time (including begin-group/end-group tokens). For `\tl_analysis_map_inline:Nn` or `\tl_analysis_map_inline:nn`, the token list is given as an argument; the analogous function `\peek_analysis_map_inline:n` documented in `l3token` finds tokens in the input stream instead. In both cases the user provides $\langle inline code \rangle$ that receives three arguments for each $\langle token \rangle$:

- $\langle tokens \rangle$, which both o-expand and x-expand to the $\langle token \rangle$. The detailed form of $\langle tokens \rangle$ may change in later releases.
- $\langle char code \rangle$, a decimal representation of the character code of the $\langle token \rangle$, -1 if it is a control sequence.
- $\langle catcode \rangle$, a capital hexadecimal digit which denotes the category code of the $\langle token \rangle$ (0: control sequence, 1: begin-group, 2: end-group, 3: math shift, 4: alignment tab, 6: parameter, 7: superscript, 8: subscript, A: space, B: letter, C: other, D: active). This can be converted to an integer by writing " $\langle catcode \rangle$ ".

In addition, there is a debugging function `\tl_analysis_show:n`, very similar to the `\ShowTokens` macro from the `ted` package.

```
\tl_analysis_show:N
\tl_analysis_show:n
\tl_analysis_log:N
\tl_analysis_log:n
```

New: 2021-05-11

```
\tl_analysis_show:n {\langle token list \rangle}
\tl_analysis_log:n {\langle token list \rangle}
```

Displays to the terminal (or log) the detailed decomposition of the $\langle token list \rangle$ into tokens, showing the category code of each character token, the meaning of control sequences and active characters, and the value of registers.

```
\tl_analysis_map_inline:nn
\tl_analysis_map_inline:Nn
```

New: 2018-04-09

```
\tl_analysis_map_inline:nn {\langle token list \rangle} {\langle inline function \rangle}
```

Applies the $\langle inline function \rangle$ to each individual $\langle token \rangle$ in the $\langle token list \rangle$. The $\langle inline function \rangle$ receives three arguments as explained above. As all other mappings the mapping is done at the current group level, *i.e.* any local assignments made by the $\langle inline function \rangle$ remain in effect after the loop.

Chapter 8

The `l3regex` package: Regular expressions in `TEX`

The `l3regex` package provides regular expression testing, extraction of submatches, splitting, and replacement, all acting on token lists. The syntax of regular expressions is mostly a subset of the PCRE syntax (and very close to POSIX), with some additions due to the fact that `TEX` manipulates tokens rather than characters. For performance reasons, only a limited set of features are implemented. Notably, back-references are not supported.

Let us give a few examples. After

```
\tl_set:Nn \l_my_tl { That~cat. }
\regex_replace_once:nnN { at } { is } \l_my_tl
```

the token list variable `\l_my_tl` holds the text “`This cat.`”, where the first occurrence of “`at`” was replaced by “`is`”. A more complicated example is a pattern to emphasize each word and add a comma after it:

```
\regex_replace_all:nnN { \w+ } { \c{emph}\cB\{ \0 \cE\} , } \l_my_tl
```

The `\w` sequence represents any “word” character, and `+` indicates that the `\w` sequence should be repeated as many times as possible (at least once), hence matching a word in the input token list. In the replacement text, `\0` denotes the full match (here, a word). The command `\emph` is inserted using `\c{emph}`, and its argument `\0` is put between braces `\cB\{` and `\cE\}`.

If a regular expression is to be used several times, it can be compiled once, and stored in a regex variable using `\regex_set:Nn`. For example,

```
\regex_new:N \l_foo_regex
\regex_set:Nn \l_foo_regex { \c{begin} \cB. (\c[^BE].*) \cE. }
```

stores in `\l_foo_regex` a regular expression which matches the starting marker for an environment: `\begin`, followed by a begin-group token (`\cB.`), then any number of tokens which are neither begin-group nor end-group character tokens (`\c[^BE].*`), ending with an end-group token (`\cE.`). As explained in the next section, the parentheses “capture” the result of `\c[^BE].*`, giving us access to the name of the environment when doing replacements.

8.1 Syntax of regular expressions

8.1.1 Regular expression examples

We start with a few examples, and encourage the reader to apply `\regex_show:n` to these regular expressions.

- `Cat` matches the word “Cat” capitalized in this way, but also matches the beginning of the word “Cattle”: use `\bCat\b` to match a complete word only.
- `[abc]` matches one letter among “a”, “b”, “c”; the pattern `(a|b|c)` matches the same three possible letters (but see the discussion of submatches below).
- `[A-Za-z]*` matches any number (due to the quantifier `*`) of Latin letters (not accented).
- `\c{[A-Za-z]*}` matches a control sequence made of Latin letters.
- `_[^_]*_` matches an underscore, any number of characters other than underscore, and another underscore; it is equivalent to `_.*?_` where `.` matches arbitrary characters and the lazy quantifier `*?` means to match as few characters as possible, thus avoiding matching underscores.
- `[\+|-]?d+` matches an explicit integer with at most one sign.
- `[\+|-_]*d+_*` matches an explicit integer with any number of `+` and `-` signs, with spaces allowed except within the mantissa, and surrounded by spaces.
- `[\+|-_]*(d+|\d*\.\d+)_*` matches an explicit integer or decimal number; using `[.,]` instead of `\.` would allow the comma as a decimal marker.
- `[\+|-_]*(d+|\d*\.\d+)_*((?i)pt|in|[cem]m|ex|[bs]p|[dn]d|[pcn]c)_*` matches an explicit dimension with any unit that T_EX knows, where `(?i)` means to treat lowercase and uppercase letters identically.
- `[\+|-_]*((?i)nan|inf|(d+|\d*\.\d+)_*(e[\+|-_]d+)?)_*` matches an explicit floating point number or the special values `nan` and `inf` (with signs and spaces allowed).
- `[\+|-_]*(d+|\cC.)_*` matches an explicit integer or control sequence (without checking whether it is an integer variable).
- `\G.*?K` at the beginning of a regular expression matches and discards (due to `\K`) everything between the end of the previous match (`\G`) and what is matched by the rest of the regular expression; this is useful in `\regex_replace_all:nnN` when the goal is to extract matches or submatches in a finer way than with `\regex_extract_all:nnN`.

While it is impossible for a regular expression to match only integer expressions, `[\+|-\(\)*d+\.]*([\+|-*/][\+|-\(\)*d+\.]*)*` matches among other things all valid integer expressions (made only with explicit integers). One should follow it with further testing.

8.1.2 Characters in regular expressions

Most characters match exactly themselves, with an arbitrary category code. Some characters are special and must be escaped with a backslash (*e.g.*, `*` matches a star character). Some escape sequences of the form backslash-letter also have a special meaning (for instance `\d` matches any digit). As a rule,

- every alphanumeric character (`A–Z`, `a–z`, `0–9`) matches exactly itself, and should not be escaped, because `\A`, `\B`, ... have special meanings;
- non-alphanumeric printable ascii characters can (and should) always be escaped: many of them have special meanings (*e.g.*, use `\(`, `\)`, `\?`, `\.`, `\^`);
- spaces should always be escaped (even in character classes);
- any other character may be escaped or not, without any effect: both versions match exactly that character.

Note that these rules play nicely with the fact that many non-alphanumeric characters are difficult to input into \TeX under normal category codes. For instance, `\abc%` matches the characters `\abc%` (with arbitrary category codes), but does not match the control sequence `\abc` followed by a percent character. Matching control sequences can be done using the `\c{<regex>}` syntax (see below).

Any special character which appears at a place where its special behaviour cannot apply matches itself instead (for instance, a quantifier appearing at the beginning of a string), after raising a warning.

Characters.

`\x{hh...}` Character with hex code `hh...`

`\xhh` Character with hex code `hh`.

`\a` Alarm (hex 07).

`\e` Escape (hex 1B).

`\f` Form-feed (hex 0C).

`\n` New line (hex 0A).

`\r` Carriage return (hex 0D).

`\t` Horizontal tab (hex 09).

8.1.3 Characters classes

Character types.

`.` A single period matches any token.

`\d` Any decimal digit.

`\h` Any horizontal space character, equivalent to `[\ \^~I]`: space and tab.

`\s` Any space character, equivalent to `[\ \^~I\^~J\^~L\^~M]`.

`\v` Any vertical space character, equivalent to `[\^J\^K\^L\^M]`. Note that `\^K` is a vertical space, but not a space, for compatibility with Perl.

`\w` Any word character, *i.e.*, alphanumerics and underscore, equivalent to the explicit class `[A-Za-z0-9_]`.

`\D` Any token not matched by `\d`.

`\H` Any token not matched by `\h`.

`\N` Any token other than the `\n` character (hex 0A).

`\S` Any token not matched by `\s`.

`\V` Any token not matched by `\v`.

`\W` Any token not matched by `\w`.

Of those, `.`, `\D`, `\H`, `\N`, `\S`, `\V`, and `\W` match arbitrary control sequences. Character classes match exactly one token in the subject.

`[...]` Positive character class. Matches any of the specified tokens.

`[^...]` Negative character class. Matches any token other than the specified characters.

`x-y` Within a character class, this denotes a range (can be used with escaped characters).

`[:<name>:]` Within a character class (one more set of brackets), this denotes the POSIX character class `<name>`, which can be `alnum`, `alpha`, `ascii`, `blank`, `cntrl`, `digit`, `graph`, `lower`, `print`, `punct`, `space`, `upper`, `word`, or `xdigit`.

`[:^<name>:]` Negative POSIX character class.

For instance, `[a-oq-z\cC.]` matches any lowercase latin letter except `p`, as well as control sequences (see below for a description of `\c`).

In character classes, only `[`, `^`, `-`, `]`, `\` and spaces are special, and should be escaped. Other non-alphanumeric characters can still be escaped without harm. Any escape sequence which matches a single character (`\d`, `\D`, *etc.*) is supported in character classes. If the first character is `^`, then the meaning of the character class is inverted; `^` appearing anywhere else in the range is not special. If the first character (possibly following a leading `^`) is `]` then it does not need to be escaped since ending the range there would make it empty. Ranges of characters can be expressed using `-`, for instance, `[\D 0-5]` and `[\^6-9]` are equivalent.

8.1.4 Structure: alternatives, groups, repetitions

Quantifiers (repetition).

`?` 0 or 1, greedy.

`??` 0 or 1, lazy.

`*` 0 or more, greedy.

`*?` 0 or more, lazy.

`+` 1 or more, greedy.

`+?` 1 or more, lazy.

`{n}` Exactly n .

`{n,}` n or more, greedy.

`{n,}?` n or more, lazy.

`{n,m}` At least n , no more than m , greedy.

`{n,m}?` At least n , no more than m , lazy.

For greedy quantifiers the regex code will first investigate matches that involve as many repetitions as possible, while for lazy quantifiers it investigates matches with as few repetitions as possible first.

Alternation and capturing groups.

`A|B|C` Either one of A, B, or C, investigating A first.

`(...)` Capturing group.

`(?:...)` Non-capturing group.

`(?|...)` Non-capturing group which resets the group number for capturing groups in each alternative. The following group is numbered with the first unused group number.

Capturing groups are a means of extracting information about the match. Parenthesized groups are labelled in the order of their opening parenthesis, starting at 1. The contents of those groups corresponding to the “best” match (leftmost longest) can be extracted and stored in a sequence of token lists using for instance `\regex_extract_once:nnTF`.

The `\K` escape sequence resets the beginning of the match to the current position in the token list. This only affects what is reported as the full match. For instance,

```
\regex_extract_all:nnN { a \K . } { a123aaxyz } \l_foo_seq
```

results in `\l_foo_seq` containing the items `{1}` and `{a}`: the true matches are `{a1}` and `{aa}`, but they are trimmed by the use of `\K`. The `\K` command does not affect capturing groups: for instance,

```
\regex_extract_once:nnN { (. \K c)+ \d } { acbc3 } \l_foo_seq
```

results in `\l_foo_seq` containing the items `{c3}` and `{bc}`: the true match is `{acbc3}`, with first submatch `{bc}`, but `\K` resets the beginning of the match to the last position where it appears.

8.1.5 Matching exact tokens

The `\c` escape sequence allows to test the category code of tokens, and match control sequences. Each character category is represented by a single uppercase letter:

- C for control sequences;
- B for begin-group tokens;
- E for end-group tokens;

- M for math shift;
- T for alignment tab tokens;
- P for macro parameter tokens;
- U for superscript tokens (up);
- D for subscript tokens (down);
- S for spaces;
- L for letters;
- O for others; and
- A for active characters.

The `\c` escape sequence is used as follows.

`\c{<regex>}` A control sequence whose `cname` matches the `<regex>`, anchored at the beginning and end, so that `\c{begin}` matches exactly `\begin`, and nothing else.

`\cX` Applies to the next object, which can be a character, escape character sequence such as `\x{0A}`, character class, or group, and forces this object to only match tokens with category X (any of CBEMTPUDSLOA). For instance, `\cL[A-Z\d]` matches uppercase letters and digits of category code letter, `\cC.` matches any control sequence, and `\cO(abc)` matches `abc` where each character has category other.⁶

`\c[XYZ]` Applies to the next object, and forces it to only match tokens with category X, Y, or Z (each being any of CBEMTPUDSLOA). For instance, `\c[LSO](..)` matches two tokens of category letter, space, or other.

`\c[^XYZ]` Applies to the next object and prevents it from matching any token with category X, Y, or Z (each being any of CBEMTPUDSLOA). For instance, `\c[^O]\d` matches digits which have any category different from other.

The category code tests can be used inside classes; for instance, `[\cO\d \c[L0][A-F]]` matches what T_EX considers as hexadecimal digits, namely digits with category other, or uppercase letters from A to F with category either letter or other. Within a group affected by a category code test, the outer test can be overridden by a nested test: for instance, `\cL(ab\cO*cd)` matches `ab*cd` where all characters are of category letter, except `*` which has category other.

The `\u` escape sequence allows to insert the contents of a token list directly into a regular expression or a replacement, avoiding the need to escape special characters. Namely, `\u{<var name>}` matches the exact contents (both character codes and category codes) of the variable `\<var name>`, which are obtained by applying `\exp_not:v{<var name>}` at the time the regular expression is compiled. Within a `\c{...}` control sequence matching, the `\u` escape sequence only expands its argument once, in effect performing `\tl_to_str:v`. Quantifiers are supported.

The `\ur` escape sequence allows to insert the contents of a `regex` variable into a larger regular expression. For instance, `A\ur{1_tmpa_regex}D` matches the tokens A and

⁶This last example also captures “`abc`” as a regex group; to avoid this use a non-capturing group `\cO(?:abc)`.

D separated by something that matches the regular expression `\l_tmpa_regex`. This behaves as if a non-capturing group were surrounding `\l_tmpa_regex`, and any group contained in `\l_tmpa_regex` is converted to a non-capturing group. Quantifiers are supported.

For instance, if `\l_tmpa_regex` has value `B|C`, then `A\ur{\l_tmpa_regex}D` is equivalent to `A(?:B|C)D` (matching `ABD` or `ACD`) and not to `AB|CD` (matching `AB` or `CD`). To get the latter effect, it is simplest to use TeX's expansion machinery directly: if `\l_mymodule_BC_tl` contains `B|C` then the following two lines show the same result:

```
\regex_show:n { A \u{\l_mymodule_BC_tl} D }
\regex_show:n { A B | C D }
```

8.1.6 Miscellaneous

Anchors and simple assertions.

`\b` Word boundary: either the previous token is matched by `\w` and the next by `\W`, or the opposite. For this purpose, the ends of the token list are considered as `\W`.

`\B` Not a word boundary: between two `\w` tokens or two `\W` tokens (including the boundary).

`^` or `\A` Start of the subject token list.

`$`, `\Z` or `\z` End of the subject token list.

`\G` Start of the current match. This is only different from `^` in the case of multiple matches: for instance `\regex_count:nnN { \G a } { aaba } \l_tmpa_int` yields 2, but replacing `\G` by `^` would result in `\l_tmpa_int` holding the value 1.

The option `(?i)` makes the match case insensitive (identifying `A-Z` with `a-z`; no Unicode support yet). This applies until the end of the group in which it appears, and can be reverted using `(?-i)`. For instance, in `(?i)(a(?-i)b|c)d`, the letters `a` and `d` are affected by the `i` option. Characters within ranges and classes are affected individually: `(?i)[Y-\]` is equivalent to `[YZ\[\]yz]`, and `(?i)[^aeiou]` matches any character which is not a vowel. Neither character properties, nor `\c{...}` nor `\u{...}` are affected by the `i` option.

8.2 Syntax of the replacement text

Most of the features described in regular expressions do not make sense within the replacement text. Backslash introduces various special constructions, described further below:

- `\0` is the whole match;
- `\1` is the submatch that was matched by the first (capturing) group (...); similarly for `\2`, ..., `\9` and `\g{<number>}`;
- `_` inserts a space (spaces are ignored when not escaped);
- `\a`, `\e`, `\f`, `\n`, `\r`, `\t`, `\xhh`, `\x{hhh}` correspond to single characters as in regular expressions;

- `\c{<cs name>}` inserts a control sequence;
- `\c{<category>}<character>` (see below);
- `\u{<tl var name>}` inserts the contents of the `<tl var>` (see below).

Characters other than backslash and space are simply inserted in the result (but since the replacement text is first converted to a string, one should also escape characters that are special for `TEX`, for instance use `\#`). Non-alphanumeric characters can always be safely escaped with a backslash.

For instance,

```
\tl_set:Nn \l_my_tl { Hello,~world! }
\regex_replace_all:nnN { ([er]?1|o) . } { (\0--\1) } \l_my_tl
```

results in `\l_my_tl` holding `H(ell--el)(o,--o) w(or--o)(ld--l)!`

The submatches are numbered according to the order in which the opening parenthesis of capturing groups appear in the regular expression to match. The n -th submatch is empty if there are fewer than n capturing groups or for capturing groups that appear in alternatives that were not used for the match. In case a capturing group matches several times during a match (due to quantifiers) only the last match is used in the replacement text. Submatches always keep the same category codes as in the original token list.

By default, the category code of characters inserted by the replacement are determined by the prevailing category code regime at the time where the replacement is made, with two exceptions:

- space characters (with character code 32) inserted with `_` or `\x20` or `\x{20}` have category code 10 regardless of the prevailing category code regime;
- if the category code would be 0 (escape), 5 (newline), 9 (ignore), 14 (comment) or 15 (invalid), it is replaced by 12 (other) instead.

The escape sequence `\c` allows to insert characters with arbitrary category codes, as well as control sequences.

`\cX(...)` Produces the characters “...” with category `X`, which must be one of `CBEMTPUDSLOA` as in regular expressions. Parentheses are optional for a single character (which can be an escape sequence). When nested, the innermost category code applies, for instance `\cL(Hello\cS\ world)!` gives this text with standard category codes.

`\c{<text>}` Produces the control sequence with csname `<text>`. The `<text>` may contain references to the submatches `\0`, `\1`, and so on, as in the example for `\u` below.

The escape sequence `\u{<var name>}` allows to insert the contents of the variable with name `<var name>` directly into the replacement, giving an easier control of category codes. When nested in `\c{...}` and `\u{...}` constructions, the `\u` and `\c` escape sequences perform `\tl_to_str:v`, namely extract the value of the control sequence and turn it into a string. Matches can also be used within the arguments of `\c` and `\u`. For instance,

```
\tl_set:Nn \l_my_one_tl { first }
\tl_set:Nn \l_my_two_tl { \emph{second} }
\tl_set:Nn \l_my_tl { one , two , one , one }
\regex_replace_all:nnN { [^,]+ } { \u{ \l_my_\0_tl } } \l_my_tl
```

results in `\l_my_tl` holding `first,\emph{second},first,first`.

Regex replacement is also a convenient way to produce token lists with arbitrary category codes. For instance

```
\tl_clear:N \l_tmpa_tl
\regex_replace_all:nnN { } { \cU\% \cA\~ } \l_tmpa_tl
```

results in `\l_tmpa_tl` containing the percent character with category code 7 (superscript) and an active tilde character.

8.3 Pre-compiling regular expressions

If a regular expression is to be used several times, it is better to compile it once rather than doing it each time the regular expression is used. The compiled regular expression is stored in a variable. All of the `l3regex` module's functions can be given their regular expression argument either as an explicit string or as a compiled regular expression.

<hr/> <code>\regex_new:N</code> <hr/>	<code>\regex_new:N <regex var></code>
<hr/> New: 2017-05-26 <hr/>	Creates a new <i><regex var></i> or raises an error if the name is already taken. The declaration is global. The <i><regex var></i> is initially such that it never matches.
<hr/> <code>\regex_set:Nn</code> <code>\regex_gset:Nn</code> <hr/>	<code>\regex_set:Nn <regex var> {<regex>}</code>
<hr/> New: 2017-05-26 <hr/>	Stores a compiled version of the <i><regular expression></i> in the <i><regex var></i> . The assignment is local for <code>\regex_set:Nn</code> and global for <code>\regex_gset:Nn</code> . For instance, this function can be used as
	<pre>\regex_new:N \l_my_regex \regex_set:Nn \l_my_regex { my\ (simple\)? reg(ex ular\ expression) }</pre>
<hr/> <code>\regex_const:Nn</code> <hr/>	<code>\regex_const:Nn <regex var> {<regex>}</code>
<hr/> New: 2017-05-26 <hr/>	Creates a new constant <i><regex var></i> or raises an error if the name is already taken. The value of the <i><regex var></i> is set globally to the compiled version of the <i><regular expression></i> .
<hr/> <code>\regex_show:N</code> <code>\regex_show:n</code> <code>\regex_log:N</code> <code>\regex_log:n</code> <hr/>	<code>\regex_show:n {<regex>}</code> <code>\regex_log:n {<regex>}</code>
<hr/> New: 2021-04-26 Updated: 2021-04-29 <hr/>	Displays in the terminal or writes in the log file (respectively) how <code>l3regex</code> interprets the <i><regex></i> . For instance, <code>\regex_show:n {\A X Y}</code> shows
	<pre>+--branch anchor at start (\A) char code 88 (X) +--branch char code 89 (Y)</pre>

indicating that the anchor `\A` only applies to the first branch: the second branch is not anchored to the beginning of the match.

8.4 Matching

All regular expression functions are available in both `:n` and `:N` variants. The former require a “standard” regular expression, while the later require a compiled expression as generated by `\regex_set:Nn`.

```
\regex_match:nnTF
\regex_match:NnTF
```

New: 2017-05-26

```
\regex_match:nnTF {<regex>} {<token list>} {<true code>} {<false code>}
```

Tests whether the *<regular expression>* matches any part of the *<token list>*. For instance,

```
\regex_match:nnTF { b [cde]* } { abecdcx } { TRUE } { FALSE }
\regex_match:nnTF { [b-dq-w] } { example } { TRUE } { FALSE }
```

leaves TRUE then FALSE in the input stream.

```
\regex_count:nnN
\regex_count:NnN
```

New: 2017-05-26

```
\regex_count:nnN {<regex>} {<token list>} <int var>
```

Sets *<int var>* within the current T_EX group level equal to the number of times *<regular expression>* appears in *<token list>*. The search starts by finding the left-most longest match, respecting greedy and lazy (non-greedy) operators. Then the search starts again from the character following the last character of the previous match, until reaching the end of the token list. Infinite loops are prevented in the case where the regular expression can match an empty token list: then we count one match between each pair of characters. For instance,

```
\int_new:N \l_foo_int
\regex_count:nnN { (b+|c) } { abbababcb } \l_foo_int
```

results in `\l_foo_int` taking the value 5.

```
\regex_match_case:nn
\regex_match_case:nnTF
```

New: 2022-01-10

```
\regex_match_case:nnTF
{
  {<regex1>} {<code case1>}
  {<regex2>} {<code case2>}
  ...
  {<regexn>} {<code casen>}
} {<token list>}
{<true code>} {<false code>}
```

Determines which of the *<regular expressions>* matches at the earliest point in the *<token list>*, and leaves the corresponding *<code_i>* followed by the *<true code>* in the input stream. If several *<regex>* match starting at the same point, then the first one in the list is selected and the others are discarded. If none of the *<regex>* match, the *<false code>* is left in the input stream. Each *<regex>* can either be given as a regex variable or as an explicit regular expression.

In detail, for each starting position in the *<token list>*, each of the *<regex>* is searched in turn. If one of them matches then the corresponding *<code>* is used and everything else is discarded, while if none of the *<regex>* match at a given position then the next starting position is attempted. If none of the *<regex>* match anywhere in the *<token list>* then nothing is left in the input stream. Note that this differs from nested `\regex_match:nnTF` statements since all *<regex>* are attempted at each position rather than attempting to match *<regex₁>* at every position before moving on to *<regex₂>*.

8.5 Submatch extraction

```
\regex_extract_once:nnN
\regex_extract_once:nnNTF
\regex_extract_once:NnN
\regex_extract_once:NnNTF
```

New: 2017-05-26

```
\regex_extract_once:nnN {<regex>} {<token list>} <seq var>
\regex_extract_once:nnNTF {<regex>} {<token list>} <seq var> {<true code>} {<false
code>}}
```

Finds the first match of the *<regular expression>* in the *<token list>*. If it exists, the match is stored as the first item of the *<seq var>*, and further items are the contents of capturing groups, in the order of their opening parenthesis. The *<seq var>* is assigned locally. If there is no match, the *<seq var>* is cleared. The testing versions insert the *<true code>* into the input stream if a match was found, and the *<false code>* otherwise.

For instance, assume that you type

```
\regex_extract_once:nnNTF { \A(La)?TeX(!*)\Z } { LaTeX!!! } \l_foo_seq
{ true } { false }
```

Then the regular expression (anchored at the start with `\A` and at the end with `\Z`) must match the whole token list. The first capturing group, `(La)?`, matches `La`, and the second capturing group, `(!*)`, matches `!!!`. Thus, `\l_foo_seq` contains as a result the items `{LaTeX!!!}`, `{La}`, and `{!!!}`, and the `true` branch is left in the input stream. Note that the n -th item of `\l_foo_seq`, as obtained using `\seq_item:Nn`, correspond to the submatch numbered $(n - 1)$ in functions such as `\regex_replace_once:nnN`.

```
\regex_extract_all:nnN
\regex_extract_all:nnNTF
\regex_extract_all:NnN
\regex_extract_all:NnNTF
```

New: 2017-05-26

```
\regex_extract_all:nnN {<regex>} {<token list>} <seq var>
\regex_extract_all:nnNTF {<regex>} {<token list>} <seq var> {<true code>} {<false
code>}}
```

Finds all matches of the *<regular expression>* in the *<token list>*, and stores all the submatch information in a single sequence (concatenating the results of multiple `\regex_extract_once:nnN` calls). The *<seq var>* is assigned locally. If there is no match, the *<seq var>* is cleared. The testing versions insert the *<true code>* into the input stream if a match was found, and the *<false code>* otherwise. For instance, assume that you type

```
\regex_extract_all:nnNTF { \w+ } { Hello,~world! } \l_foo_seq
{ true } { false }
```

Then the regular expression matches twice, the resulting sequence contains the two items `{Hello}` and `{world}`, and the `true` branch is left in the input stream.

```
\regex_split:nnN
\regex_split:nnNTF
\regex_split:NnN
\regex_split:NnNTF
```

New: 2017-05-26

```
\regex_split:nnN {<regular expression>} {<token list>} <seq var>
\regex_split:nnNTF {<regular expression>} {<token list>} <seq var> {<true code>}
{<false code>}
```

Splits the *<token list>* into a sequence of parts, delimited by matches of the *<regular expression>*. If the *<regular expression>* has capturing groups, then the token lists that they match are stored as items of the sequence as well. The assignment to *<seq var>* is local. If no match is found the resulting *<seq var>* has the *<token list>* as its sole item. If the *<regular expression>* matches the empty token list, then the *<token list>* is split into single tokens. The testing versions insert the *<true code>* into the input stream if a match was found, and the *<false code>* otherwise. For example, after

```
\seq_new:N \l_path_seq
\regex_split:nnNTF { / } { the/path/for/this/file.tex } \l_path_seq
{ true } { false }
```

the sequence `\l_path_seq` contains the items `{the}`, `{path}`, `{for}`, `{this}`, and `{file.tex}`, and the `true` branch is left in the input stream.

8.6 Replacement

```
\regex_replace_once:nnN
\regex_replace_once:nnNTF
\regex_replace_once:NnN
\regex_replace_once:NnNTF
```

New: 2017-05-26

```
\regex_replace_once:nnN {<regular expression>} {<replacement>} <tl var>
\regex_replace_once:nnNTF {<regular expression>} {<replacement>} <tl var> {<true
code>} {<false code>}
```

Searches for the *<regular expression>* in the contents of the *<tl var>* and replaces the first match with the *<replacement>*. In the *<replacement>*, `\0` represents the full match, `\1` represent the contents of the first capturing group, `\2` of the second, *etc.* The result is assigned locally to *<tl var>*.

```
\regex_replace_all:nnN
\regex_replace_all:nnNTF
\regex_replace_all:NnN
\regex_replace_all:NnNTF
```

New: 2017-05-26

```
\regex_replace_all:nnN {<regular expression>} {<replacement>} <tl var>
\regex_replace_all:nnNTF {<regular expression>} {<replacement>} <tl var> {<true
code>} {<false code>}
```

Replaces all occurrences of the *<regular expression>* in the contents of the *<tl var>* by the *<replacement>*, where `\0` represents the full match, `\1` represent the contents of the first capturing group, `\2` of the second, *etc.* Every match is treated independently, and matches cannot overlap. The result is assigned locally to *<tl var>*.

<u>\regex_replace_case_once:nN</u>	<u>\regex_replace_case_once:nNTF</u>
<u>\regex_replace_case_once:nNTF</u>	{
New: 2022-01-10	{\langle regex ₁ \rangle} {\langle replacement ₁ \rangle}
	{\langle regex ₂ \rangle} {\langle replacement ₂ \rangle}
	...
	{\langle regex _n \rangle} {\langle replacement _n \rangle}
	} \langle tl var \rangle
	{\langle true code \rangle} {\langle false code \rangle}

Replaces the earliest match of the regular expression $(\langle regex_1 \rangle | \dots | \langle regex_n \rangle)$ in the $\langle token list variable \rangle$ by the $\langle replacement \rangle$ corresponding to which $\langle regex_i \rangle$ matched, then leaves the $\langle true code \rangle$ in the input stream. If none of the $\langle regex \rangle$ match, then the $\langle tl var \rangle$ is not modified, and the $\langle false code \rangle$ is left in the input stream. Each $\langle regex \rangle$ can either be given as a regex variable or as an explicit regular expression.

In detail, for each starting position in the $\langle token list \rangle$, each of the $\langle regex \rangle$ is searched in turn. If one of them matches then it is replaced by the corresponding $\langle replacement \rangle$ as described for `\regex_replace_once:nnN`. This is equivalent to checking with `\regex_match_case:nn` which $\langle regex \rangle$ matches, then performing the replacement with `\regex_replace_once:nnN`.

<u>\regex_replace_case_all:nN</u>	<u>\regex_replace_case_all:nNTF</u>
<u>\regex_replace_case_all:nNTF</u>	{
New: 2022-01-10	{\langle regex ₁ \rangle} {\langle replacement ₁ \rangle}
	{\langle regex ₂ \rangle} {\langle replacement ₂ \rangle}
	...
	{\langle regex _n \rangle} {\langle replacement _n \rangle}
	} \langle tl var \rangle
	{\langle true code \rangle} {\langle false code \rangle}

Replaces all occurrences of all $\langle regex \rangle$ in the $\langle token list \rangle$ by the corresponding $\langle replacement \rangle$. Every match is treated independently, and matches cannot overlap. The result is assigned locally to $\langle tl var \rangle$, and the $\langle true code \rangle$ or $\langle false code \rangle$ is left in the input stream depending on whether any replacement was made or not.

In detail, for each starting position in the $\langle token list \rangle$, each of the $\langle regex \rangle$ is searched in turn. If one of them matches then it is replaced by the corresponding $\langle replacement \rangle$, and the search resumes at the position that follows this match (and replacement). For instance

```
\tl_set:Nn \l_tmpa_tl { Hello,~world! }
\regex_replace_case_all:nN
{
  { [A-Za-z]+ } { ‘\0’ }
  { \b } { --- }
  { . } { [\0] }
} \l_tmpa_tl
```

results in `\l_tmpa_tl` having the contents `‘Hello’---[,] [] ‘world’---[!]`. Note in particular that the word-boundary assertion `\b` did not match at the start of words because the case `[A-Za-z]+` matched at these positions. To change this, one could simply swap the order of the two cases in the argument of `\regex_replace_case_all:nN`.

8.7 Scratch regular expressions

<code>\l_tmpa_regex</code>
<code>\l_tmpb_regex</code>
New: 2017-12-11

Scratch regex for local assignment. These are never used by the kernel code, and so are safe for use with any L^AT_EX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

<code>\g_tmpa_regex</code>
<code>\g_tmpb_regex</code>
New: 2017-12-11

Scratch regex for global assignment. These are never used by the kernel code, and so are safe for use with any L^AT_EX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

8.8 Bugs, misfeatures, future work, and other possibilities

The following need to be done now.

- Rewrite the documentation in a more ordered way, perhaps add a BNF?
Additional error-checking to come.
- Clean up the use of messages.
- Cleaner error reporting in the replacement phase.
- Add tracing information.
- Detect attempts to use back-references and other non-implemented syntax.
- Test for the maximum register `\c_max_register_int`.
- Find out whether the fact that `\W` and friends match the end-marker leads to bugs. Possibly update `_regex_item_reverse:n`.
- The empty cs should be matched by `\c{}`, not by `\c{csname.?endcsname\s?}`.

Code improvements to come.

- Shift arrays so that the useful information starts at position 1.
- Only build `\c{...}` once.
- Use arrays for the left and right state stacks when compiling a regex.
- Should `_regex_action_free_group:n` only be used for greedy `{n,}` quantifier? (I think not.)
- Quantifiers for `\u` and assertions.
- When matching, keep track of an explicit stack of `curr_state` and `curr_submatches`.
- If possible, when a state is reused by the same thread, kill other subthreads.

- Use an array rather than `\g__regex_balance_tl` to build the function `__regex_replacement_balance_one_match:n`.
- Reduce the number of epsilon-transitions in alternatives.
- Optimize simple strings: use less states (`abcade` should give two states, for `abc` and `ade`). [Does that really make sense?]
- Optimize groups with no alternative.
- Optimize states with a single `__regex_action_free:n`.
- Optimize the use of `__regex_action_success:` by inserting it in state 2 directly instead of having an extra transition.
- Optimize the use of `\int_step...` functions.
- Groups don't capture within regexes for csnames; optimize and document.
- Better “show” for anchors, properties, and catcode tests.
- Does `\K` really need a new state for itself?
- When compiling, use a boolean `in_cs` and less magic numbers.
- Instead of checking whether the character is special or alphanumeric using its character code, check if it is special in regexes with `\cs_if_exist` tests.

The following features are likely to be implemented at some point in the future.

- General look-ahead/behind assertions.
- Regex matching on external files.
- Conditional subpatterns with look ahead/behind: “if what follows is [...], then [...]”.
- `(*..)` and `(?..)` sequences to set some options.
- UTF-8 mode for pdfTeX.
- Newline conventions are not done. In particular, we should have an option for `.` not to match newlines. Also, `\A` should differ from `^`, and `\Z`, `\z` and `$` should differ.
- Unicode properties: `\p{..}` and `\P{..}`; `\X` which should match any “extended” Unicode sequence. This requires to manipulate a lot of data, probably using tree-boxes.

The following features of PCRE or Perl may or may not be implemented.

- Callout with `(?C...)` or other syntax: some internal code changes make that possible, and it can be useful for instance in the replacement code to stop a regex replacement when some marker has been found; this raises the question of a potential `\regex_break:` and then of playing well with `\tl_map_break:` called from within the code in a regex. It also raises the question of nested calls to the regex machinery, which is a problem since `\fontdimen` are global.

- Conditional subpatterns (other than with a look-ahead or look-behind condition): this is non-regular, isn't it?
- Named subpatterns: \TeX programmers have lived so far without any need for named macro parameters.

The following features of PCRE or Perl will definitely not be implemented.

- Back-references: non-regular feature, this requires backtracking, which is prohibitively slow.
- Recursion: this is a non-regular feature.
- Atomic grouping, possessive quantifiers: those tools, mostly meant to fix catastrophic backtracking, are unnecessary in a non-backtracking algorithm, and difficult to implement.
- Subroutine calls: this syntactic sugar is difficult to include in a non-backtracking algorithm, in particular because the corresponding group should be treated as atomic.
- Backtracking control verbs: intrinsically tied to backtracking.
- $\backslash\text{ddd}$, matching the character with octal code ddd : we already have $\backslash\text{x}\{\dots\}$ and the syntax is confusingly close to what we could have used for backreferences ($\backslash 1$, $\backslash 2$, \dots), making it harder to produce useful error message.
- $\backslash\text{cx}$, similar to \TeX 's own $\backslash\text{\textasciicircum x}$.
- Comments: \TeX already has its own system for comments.
- $\backslash\text{Q}\dots\backslash\text{E}$ escaping: this would require to read the argument verbatim, which is not in the scope of this module.
- $\backslash\text{C}$ single byte in UTF-8 mode: $\text{Xe}\text{\TeX}$ and $\text{Lua}\text{\TeX}$ serve us characters directly, and splitting those into bytes is tricky, encoding dependent, and most likely not useful anyways.

Chapter 9

The l3prg package

Control structures

Conditional processing in L^AT_EX3 is defined as something that performs a series of tests, possibly involving assignments and calling other functions that do not read further ahead in the input stream. After processing the input, a *state* is returned. The states returned are *⟨true⟩* and *⟨false⟩*.

L^AT_EX3 has two forms of conditional flow processing based on these states. The first form is predicate functions that turn the returned state into a boolean *⟨true⟩* or *⟨false⟩*. For example, the function `\cs_if_free_p:N` checks whether the control sequence given as its argument is free and then returns the boolean *⟨true⟩* or *⟨false⟩* values to be used in testing with `\if_predicate:w` or in functions to be described below. The second form is the kind of functions choosing a particular argument from the input stream based on the result of the testing as in `\cs_if_free:NTF` which also takes one argument (the *N*) and then executes either `true` or `false` depending on the result.

T_EXhackers note: The arguments are executed after exiting the underlying `\if... \fi:` structure.

9.1 Defining a set of conditional functions

<code>\prg_new_conditional:Npnn</code>	<code>\prg_new_conditional:Npnn \⟨name⟩:⟨arg spec⟩ ⟨parameters⟩ {⟨conditions⟩} {⟨code⟩}</code>
<code>\prg_set_conditional:Npnn</code>	<code>\prg_new_conditional:Nnn \⟨name⟩:⟨arg spec⟩ {⟨conditions⟩} {⟨code⟩}</code>
<code>\prg_new_conditional:Nnn</code>	
<code>\prg_set_conditional:Nnn</code>	

Updated: 2012-02-06

These functions create a family of conditionals using the same *{⟨code⟩}* to perform the test created. Those conditionals are expandable if *⟨code⟩* is. The **new** versions check for existing definitions and perform assignments globally (cf. `\cs_new:Npn`) whereas the **set** versions do no check and perform assignments locally (cf. `\cs_set:Npn`). The conditionals created are dependent on the comma-separated list of *⟨conditions⟩*, which should be one or more of `p`, `T`, `F` and `TF`.

<code>\prg_new_protected_conditional:Npnn</code>	<code>\prg_new_protected_conditional:Npnn \<name>:\<arg spec> \<parameters></code>
<code>\prg_set_protected_conditional:Npnn</code>	<code>{\<conditions>} {\<code>}</code>
<code>\prg_new_protected_conditional:Nnn</code>	<code>\prg_new_protected_conditional:Nnn \<name>:\<arg spec></code>
<code>\prg_set_protected_conditional:Nnn</code>	<code>{\<conditions>} {\<code>}</code>

Updated: 2012-02-06

These functions create a family of protected conditionals using the same `{\<code>}` to perform the test created. The `\<code>` does not need to be expandable. The `new` version check for existing definitions and perform assignments globally (cf. `\cs_new:Npn`) whereas the `set` version do not (cf. `\cs_set:Npn`). The conditionals created are depended on the comma-separated list of `\<conditions>`, which should be one or more of T, F and TF (not p).

The conditionals are defined by `\prg_new_conditional:Npnn` and friends as:

- `\<name>_p:\<arg spec>` — a predicate function which will supply either a logical `true` or logical `false`. This function is intended for use in cases where one or more logical tests are combined to lead to a final outcome. This function cannot be defined for `protected` conditionals.
- `\<name>:\<arg spec>T` — a function with one more argument than the original `\<arg spec>` demands. The `\<true branch>` code in this additional argument will be left on the input stream only if the test is `true`.
- `\<name>:\<arg spec>F` — a function with one more argument than the original `\<arg spec>` demands. The `\<false branch>` code in this additional argument will be left on the input stream only if the test is `false`.
- `\<name>:\<arg spec>TF` — a function with two more argument than the original `\<arg spec>` demands. The `\<true branch>` code in the first additional argument will be left on the input stream if the test is `true`, while the `\<false branch>` code in the second argument will be left on the input stream if the test is `false`.

The `\<code>` of the test may use `\<parameters>` as specified by the second argument to `\prg_set_conditional:Npnn`: this should match the `\<argument specification>` but this is not enforced. The `Nnn` versions infer the number of arguments from the argument specification given (cf. `\cs_new:Nn`, etc.). Within the `\<code>`, the functions `\prg_return_true:` and `\prg_return_false:` are used to indicate the logical outcomes of the test.

An example can easily clarify matters here:

```
\prg_set_conditional:Npnn \foo_if_bar:NN #1#2 { p , T , TF }
{
  \if_meaning:w \l_tmpa_tl #1
  \prg_return_true:
\else:
  \if_meaning:w \l_tmpa_tl #2
  \prg_return_true:
\else:
  \prg_return_false:
\fi:
\fi:
}
```

This defines the function `\foo_if_bar_p:NN`, `\foo_if_bar:NNTF` and `\foo_if_bar:NNT` but not `\foo_if_bar:NNF` (because `F` is missing from the `\conditions` list). The return statements take care of resolving the remaining `\else:` and `\fi:` before returning the state. There must be a return statement for each branch; failing to do so will result in erroneous output if that branch is executed.

<code>\prg_new_eq_conditional:NNn</code>	<code>\prg_new_eq_conditional:NNn \langle name_1 \rangle \langle arg spec_1 \rangle \langle name_2 \rangle \langle arg spec_2 \rangle</code>
<code>\prg_set_eq_conditional:NNn</code>	<code>\{ \langle conditions \rangle \}</code>

These functions copy a family of conditionals. The **new** version checks for existing definitions (*cf.* `\cs_new_eq:NN`) whereas the **set** version does not (*cf.* `\cs_set_eq:NN`). The conditionals copied are depended on the comma-separated list of `\conditions`, which should be one or more of `p`, `T`, `F` and `TF`.

<code>\prg_return_true: *</code>	<code>\prg_return_true:</code>
<code>\prg_return_false: *</code>	<code>\prg_return_false:</code>

These “return” functions define the logical state of a conditional statement. They appear within the code for a conditional function generated by `\prg_set_conditional:Npnn`, *etc.*, to indicate when a true or false branch should be taken. While they may appear multiple times each within the code of such conditionals, the execution of the conditional must result in the expansion of one of these two functions *exactly once*.

The return functions trigger what is internally an `f`-expansion process to complete the evaluation of the conditional. Therefore, after `\prg_return_true:` or `\prg_return_false:` there must be no non-expandable material in the input stream for the remainder of the expansion of the conditional code. This includes other instances of either of these functions.

<code>\prg_generate_conditional_variant:Nnn</code>	<code>\prg_generate_conditional_variant:Nnn \langle name \rangle \langle arg spec \rangle</code>
	<code>\{ \langle variant argument specifiers \rangle \} \{ \langle condition specifiers \rangle \}</code>

New: 2017-12-12

Defines argument-specifier variants of conditionals. This is equivalent to running `\cs_generate_variant:Nn \langle conditional \rangle \{ \langle variant argument specifiers \rangle \}` on each `\conditional` described by the `\condition specifiers`. These base-form `\conditionals` are obtained from the `\name` and `\arg spec` as described for `\prg_new_conditional:Npnn`, and they should be defined.

9.2 The boolean data type

This section describes a boolean data type which is closely connected to conditional processing as sometimes you want to execute some code depending on the value of a switch (*e.g.*, draft/final) and other times you perhaps want to use it as a predicate function in an `\if_predicate:w` test. The problem of the primitive `\if_false:` and `\if_true:` tokens is that it is not always safe to pass them around as they may interfere with scanning for termination of primitive conditional processing. Therefore, we employ two canonical booleans: `\c_true_bool` or `\c_false_bool`. Besides preventing problems as described above, it also allows us to implement a simple boolean parser supporting the logical operations `And`, `Or`, `Not`, *etc.* which can then be used on both the boolean type and predicate functions.

All conditional `\bool_` functions except assignments are expandable and expect the input to also be fully expandable (which generally means being constructed from predicate functions and booleans, possibly nested).

T_EXhackers note: The `bool` data type is not implemented using the `\iffalse`/`\iftrue` primitives, in contrast to `\newif`, *etc.*, in plain T_EX, L^AT_EX 2_ε and so on. Programmers should not base use of `bool` switches on any particular expectation of the implementation.

<hr/> <code>\bool_new:N</code> <hr/> <code>\bool_new:c</code> <hr/>	<code>\bool_new:N</code> $\langle\textit{boolean}\rangle$ Creates a new $\langle\textit{boolean}\rangle$ or raises an error if the name is already taken. The declaration is global. The $\langle\textit{boolean}\rangle$ is initially <code>false</code> .
<hr/> <code>\bool_const:Nn</code> <hr/> <code>\bool_const:cn</code> <hr/> <div>New: 2017-11-28</div> <hr/>	<code>\bool_const:Nn</code> $\langle\textit{boolean}\rangle$ $\{\langle\textit{boolexpr}\rangle\}$ Creates a new constant $\langle\textit{boolean}\rangle$ or raises an error if the name is already taken. The value of the $\langle\textit{boolean}\rangle$ is set globally to the result of evaluating the $\langle\textit{boolexpr}\rangle$.
<hr/> <code>\bool_set_false:N</code> <hr/> <code>\bool_set_false:c</code> <hr/> <code>\bool_gset_false:N</code> <hr/> <code>\bool_gset_false:c</code> <hr/>	<code>\bool_set_false:N</code> $\langle\textit{boolean}\rangle$ Sets $\langle\textit{boolean}\rangle$ logically <code>false</code> .
<hr/> <code>\bool_set_true:N</code> <hr/> <code>\bool_set_true:c</code> <hr/> <code>\bool_gset_true:N</code> <hr/> <code>\bool_gset_true:c</code> <hr/>	<code>\bool_set_true:N</code> $\langle\textit{boolean}\rangle$ Sets $\langle\textit{boolean}\rangle$ logically <code>true</code> .
<hr/> <code>\bool_set_eq:NN</code> <hr/> <code>\bool_set_eq:(cN Nc cc)</code> <hr/> <code>\bool_gset_eq:NN</code> <hr/> <code>\bool_gset_eq:(cN Nc cc)</code> <hr/>	<code>\bool_set_eq:NN</code> $\langle\textit{boolean}_1\rangle$ $\langle\textit{boolean}_2\rangle$ Sets $\langle\textit{boolean}_1\rangle$ to the current value of $\langle\textit{boolean}_2\rangle$.
<hr/> <code>\bool_set:Nn</code> <hr/> <code>\bool_set:cn</code> <hr/> <code>\bool_gset:Nn</code> <hr/> <code>\bool_gset:cn</code> <hr/> <div>Updated: 2017-07-15</div> <hr/>	<code>\bool_set:Nn</code> $\langle\textit{boolean}\rangle$ $\{\langle\textit{boolexpr}\rangle\}$ Evaluates the $\langle\textit{boolean expression}\rangle$ as described for <code>\bool_if:nTF</code> , and sets the $\langle\textit{boolean}\rangle$ variable to the logical truth of this evaluation.
<hr/> <code>\bool_if_p:N</code> ★ <hr/> <code>\bool_if_p:c</code> ★ <hr/> <code>\bool_if:nTF</code> ★ <hr/> <code>\bool_if:cTF</code> ★ <hr/> <div>Updated: 2017-07-15</div> <hr/>	<code>\bool_if_p:N</code> $\langle\textit{boolean}\rangle$ <code>\bool_if:NTF</code> $\langle\textit{boolean}\rangle$ $\{\langle\textit{true code}\rangle\}$ $\{\langle\textit{false code}\rangle\}$ Tests the current truth of $\langle\textit{boolean}\rangle$, and continues expansion based on this result.
<hr/> <code>\bool_to_str:N</code> ★ <hr/> <code>\bool_to_str:c</code> ★ <hr/> <code>\bool_to_str:n</code> ★ <hr/> <div>New: 2021-11-01</div> <hr/>	<code>\bool_to_str:N</code> $\langle\textit{boolean}\rangle$ <code>\bool_to_str:n</code> $\langle\textit{boolean expression}\rangle$ Expands to the letters <code>true</code> or <code>false</code> depending on the logical truth of the $\langle\textit{boolean}\rangle$ or $\langle\textit{boolean expression}\rangle$.

<hr/> <code>\bool_show:N</code> <hr/> <code>\bool_show:c</code> <hr/> New: 2012-02-09 Updated: 2021-04-29 <hr/>	<code>\bool_show:N <boolean></code> Displays the logical truth of the <i><boolean></i> on the terminal.
<hr/> <code>\bool_show:n</code> <hr/> New: 2012-02-09 Updated: 2017-07-15 <hr/>	<code>\bool_show:n {(boolean expression)}</code> Displays the logical truth of the <i><boolean expression></i> on the terminal.
<hr/> <code>\bool_log:N</code> <hr/> <code>\bool_log:c</code> <hr/> New: 2014-08-22 Updated: 2021-04-29 <hr/>	<code>\bool_log:N <boolean></code> Writes the logical truth of the <i><boolean></i> in the log file.
<hr/> <code>\bool_log:n</code> <hr/> New: 2014-08-22 Updated: 2017-07-15 <hr/>	<code>\bool_log:n {(boolean expression)}</code> Writes the logical truth of the <i><boolean expression></i> in the log file.
<hr/> <code>\bool_if_exist_p:N *</code> <code>\bool_if_exist_p:c *</code> <code>\bool_if_exist:NTF *</code> <code>\bool_if_exist:cTF *</code> <hr/> New: 2012-03-03 <hr/>	<code>\bool_if_exist_p:N <boolean></code> <code>\bool_if_exist:NTF <boolean> {(true code)} {(false code)}</code> Tests whether the <i><boolean></i> is currently defined. This does not check that the <i><boolean></i> really is a boolean variable.

9.2.1 Scratch booleans

<hr/> <code>\l_tmpa_bool</code> <hr/> <code>\l_tmpb_bool</code> <hr/>	A scratch boolean for local assignment. It is never used by the kernel code, and so is safe for use with any L ^A T _E X3-defined function. However, it may be overwritten by other non-kernel code and so should only be used for short-term storage.
<hr/> <code>\g_tmpa_bool</code> <hr/> <code>\g_tmpb_bool</code> <hr/>	A scratch boolean for global assignment. It is never used by the kernel code, and so is safe for use with any L ^A T _E X3-defined function. However, it may be overwritten by other non-kernel code and so should only be used for short-term storage.

9.3 Boolean expressions

As we have a boolean datatype and predicate functions returning boolean *<true>* or *<false>* values, it seems only fitting that we also provide a parser for *<boolean expressions>*.

A boolean expression is an expression which given input in the form of predicate functions and boolean variables, return boolean *<true>* or *<false>*. It supports the logical operations And, Or and Not as the well-known infix operators `&&` and `||` and prefix `!` with their usual precedences (namely, `&&` binds more tightly than `||`). In addition to this, parentheses can be used to isolate sub-expressions. For example,


```

\int_compare_p:n { 1 = 1 } &&
(
  \int_compare_p:n { 2 = 3 } ||
  \int_compare_p:n { 4 <= 4 } ||
  \str_if_eq_p:nn { abc } { def }
) &&
! \int_compare_p:n { 2 = 4 }

```

is a valid boolean expression.

Contrarily to some other programming languages, the operators `&&` and `||` evaluate both operands in all cases, even when the first operand is enough to determine the result. This “eager” evaluation should be contrasted with the “lazy” evaluation of `\bool_lazy_...` functions.

T_EXhackers note: The eager evaluation of boolean expressions is unfortunately necessary in T_EX. Indeed, a lazy parser can get confused if `&&` or `||` or parentheses appear as (unbraced) arguments of some predicates. For instance, the innocuous-looking expression below would break (in a lazy parser) if `#1` were a closing parenthesis and `\l_tmpa_bool` were `true`.

```
( \l_tmpa_bool || \token_if_eq_meaning_p:NN X #1 )
```

Minimal (lazy) evaluation can be obtained using the conditionals `\bool_lazy_all:nTF`, `\bool_lazy_and:nnTF`, `\bool_lazy_any:nTF`, or `\bool_lazy_or:nnTF`, which only evaluate their boolean expression arguments when they are needed to determine the resulting truth value. For example, when evaluating the boolean expression

```

\bool_lazy_and_p:nn
{
  \bool_lazy_any_p:n
  {
    { \int_compare_p:n { 2 = 3 } }
    { \int_compare_p:n { 4 <= 4 } }
    { \int_compare_p:n { 1 = \error } } % skipped
  }
}
{ ! \int_compare_p:n { 2 = 4 } }

```

the line marked with `skipped` is not expanded because the result of `\bool_lazy_any_p:n` is known once the second boolean expression is found to be logically `true`. On the other hand, the last line is expanded because its logical value is needed to determine the result of `\bool_lazy_and_p:nn`.

<code>\bool_if_p:n</code> ★ <code>\bool_if:nTF</code> ★	<code>\bool_if_p:n {<boolean expression>}</code> <code>\bool_if:nTF {<boolean expression>} {<true code>} {<false code>}</code>
--	---

Updated: 2017-07-15

Tests the current truth of `<boolean expression>`, and continues expansion based on this result. The `<boolean expression>` should consist of a series of predicates or boolean variables with the logical relationship between these defined using `&&` (“And”), `||` (“Or”), `!` (“Not”) and parentheses. The logical Not applies to the next predicate or group.

<code>\bool_lazy_all_p:n *</code> <code>\bool_lazy_all:nTF *</code>	<code>\bool_lazy_all_p:n { {<boolean expression>} {<boolean expression>} ... {<boolean expression>} }</code> <code>\bool_lazy_all:nTF { {<boolean expression>} {<boolean expression>} ... {<boolean expression>} } {<true code>} {<false code>}</code>
--	---

New: 2015-11-15
Updated: 2017-07-15

Implements the “And” operation on the *<boolean expressions>*, hence is **true** if all of them are **true** and **false** if any of them is **false**. Contrarily to the infix operator `&&`, only the *<boolean expressions>* which are needed to determine the result of `\bool_lazy_all:nTF` are evaluated. See also `\bool_lazy_and:nnTF` when there are only two *<boolean expressions>*.

<code>\bool_lazy_and_p:nn *</code> <code>\bool_lazy_and:nnTF *</code>	<code>\bool_lazy_and_p:nn {<boolean expression>} {<boolean expression>}</code> <code>\bool_lazy_and:nnTF {<boolean expression>} {<boolean expression>} {<true code>} {<false code>}</code>
--	---

New: 2015-11-15
Updated: 2017-07-15

Implements the “And” operation between two boolean expressions, hence is **true** if both are **true**. Contrarily to the infix operator `&&`, the *<boolean expression>* is only evaluated if it is needed to determine the result of `\bool_lazy_and:nnTF`. See also `\bool_lazy_all:nTF` when there are more than two *<boolean expressions>*.

<code>\bool_lazy_any_p:n *</code> <code>\bool_lazy_any:nTF *</code>	<code>\bool_lazy_any_p:n { {<boolean expression>} {<boolean expression>} ... {<boolean expression>} }</code> <code>\bool_lazy_any:nTF { {<boolean expression>} {<boolean expression>} ... {<boolean expression>} } {<true code>} {<false code>}</code>
--	---

New: 2015-11-15
Updated: 2017-07-15

Implements the “Or” operation on the *<boolean expressions>*, hence is **true** if any of them is **true** and **false** if all of them are **false**. Contrarily to the infix operator `||`, only the *<boolean expressions>* which are needed to determine the result of `\bool_lazy_any:nTF` are evaluated. See also `\bool_lazy_or:nnTF` when there are only two *<boolean expressions>*.

<code>\bool_lazy_or_p:nn *</code> <code>\bool_lazy_or:nnTF *</code>	<code>\bool_lazy_or_p:nn {<boolean expression>} {<boolean expression>}</code> <code>\bool_lazy_or:nnTF {<boolean expression>} {<boolean expression>} {<true code>} {<false code>}</code>
--	---

New: 2015-11-15
Updated: 2017-07-15

Implements the “Or” operation between two boolean expressions, hence is **true** if either one is **true**. Contrarily to the infix operator `||`, the *<boolean expression>* is only evaluated if it is needed to determine the result of `\bool_lazy_or:nnTF`. See also `\bool_lazy_any:nTF` when there are more than two *<boolean expressions>*.

<code>\bool_not_p:n *</code>	<code>\bool_not_p:n {<boolean expression>}</code>
------------------------------	---

Updated: 2017-07-15

Function version of `!(<boolean expression>)` within a boolean expression.

<code>\bool_xor_p:nn *</code> <code>\bool_xor:nnTF *</code>	<code>\bool_xor_p:nn {<boolean expression>} {<boolean expression>}</code> <code>\bool_xor:nnTF {<boolean expression>} {<boolean expression>} {<true code>} {<false code>}</code>
--	---

New: 2018-05-09

Implements an “exclusive or” operation between two boolean expressions. There is no infix operation for this logical operation.

9.4 Logical loops

Loops using either boolean expressions or stored boolean values.

<code>\bool_do_until:Nn ☆</code> <code>\bool_do_until:cn ☆</code>	<code>\bool_do_until:Nn <boolean> {<code>}</code>
--	---

Updated: 2017-07-15

Places the *<code>* in the input stream for \TeX to process, and then checks the logical value of the *<boolean>*. If it is **false** then the *<code>* is inserted into the input stream again and the process loops until the *<boolean>* is **true**.

<code>\bool_do_while:Nn</code> ☆	<code>\bool_do_while:Nn <boolean> {<code>}</code>
<code>\bool_do_while:cn</code> ☆	
Updated: 2017-07-15	Places the <code><code></code> in the input stream for \TeX to process, and then checks the logical value of the <code><boolean></code> . If it is <code>true</code> then the <code><code></code> is inserted into the input stream again and the process loops until the <code><boolean></code> is <code>false</code> .
<code>\bool_until_do:Nn</code> ☆	<code>\bool_until_do:Nn <boolean> {<code>}</code>
<code>\bool_until_do:cn</code> ☆	
Updated: 2017-07-15	This function firsts checks the logical value of the <code><boolean></code> . If it is <code>false</code> the <code><code></code> is placed in the input stream and expanded. After the completion of the <code><code></code> the truth of the <code><boolean></code> is re-evaluated. The process then loops until the <code><boolean></code> is <code>true</code> .
<code>\bool_while_do:Nn</code> ☆	<code>\bool_while_do:Nn <boolean> {<code>}</code>
<code>\bool_while_do:cn</code> ☆	
Updated: 2017-07-15	This function firsts checks the logical value of the <code><boolean></code> . If it is <code>true</code> the <code><code></code> is placed in the input stream and expanded. After the completion of the <code><code></code> the truth of the <code><boolean></code> is re-evaluated. The process then loops until the <code><boolean></code> is <code>false</code> .
<code>\bool_do_until:nn</code> ☆	<code>\bool_do_until:nn {<boolean expression>} {<code>}</code>
Updated: 2017-07-15	Places the <code><code></code> in the input stream for \TeX to process, and then checks the logical value of the <code><boolean expression></code> as described for <code>\bool_if:nTF</code> . If it is <code>false</code> then the <code><code></code> is inserted into the input stream again and the process loops until the <code><boolean expression></code> evaluates to <code>true</code> .
<code>\bool_do_while:nn</code> ☆	<code>\bool_do_while:nn {<boolean expression>} {<code>}</code>
Updated: 2017-07-15	Places the <code><code></code> in the input stream for \TeX to process, and then checks the logical value of the <code><boolean expression></code> as described for <code>\bool_if:nTF</code> . If it is <code>true</code> then the <code><code></code> is inserted into the input stream again and the process loops until the <code><boolean expression></code> evaluates to <code>false</code> .
<code>\bool_until_do:nn</code> ☆	<code>\bool_until_do:nn {<boolean expression>} {<code>}</code>
Updated: 2017-07-15	This function firsts checks the logical value of the <code><boolean expression></code> (as described for <code>\bool_if:nTF</code>). If it is <code>false</code> the <code><code></code> is placed in the input stream and expanded. After the completion of the <code><code></code> the truth of the <code><boolean expression></code> is re-evaluated. The process then loops until the <code><boolean expression></code> is <code>true</code> .
<code>\bool_while_do:nn</code> ☆	<code>\bool_while_do:nn {<boolean expression>} {<code>}</code>
Updated: 2017-07-15	This function firsts checks the logical value of the <code><boolean expression></code> (as described for <code>\bool_if:nTF</code>). If it is <code>true</code> the <code><code></code> is placed in the input stream and expanded. After the completion of the <code><code></code> the truth of the <code><boolean expression></code> is re-evaluated. The process then loops until the <code><boolean expression></code> is <code>false</code> .

9.5 Producing multiple copies

<code>\prg_replicate:nn</code> ☆	<code>\prg_replicate:nn {<integer expression>} {<tokens>}</code>
Updated: 2011-07-04	Evaluates the <code><integer expression></code> (which should be zero or positive) and creates the resulting number of copies of the <code><tokens></code> . The function is both expandable and safe for nesting. It yields its result after two expansion steps.

9.6 Detecting T_EX’s mode

<code>\mode_if_horizontal_p: *</code>	<code>\mode_if_horizontal_p:</code>
<code>\mode_if_horizontal:TF *</code>	<code>\mode_if_horizontal:TF {\langle true code \rangle} {\langle false code \rangle}</code>

Detects if T_EX is currently in horizontal mode.

<code>\mode_if_inner_p: *</code>	<code>\mode_if_inner_p:</code>
<code>\mode_if_inner:TF *</code>	<code>\mode_if_inner:TF {\langle true code \rangle} {\langle false code \rangle}</code>

Detects if T_EX is currently in inner mode.

<code>\mode_if_math_p: *</code>	<code>\mode_if_math:TF {\langle true code \rangle} {\langle false code \rangle}</code>
<code>\mode_if_math:TF *</code>	

Detects if T_EX is currently in maths mode.

Updated: 2011-09-05

<code>\mode_if_vertical_p: *</code>	<code>\mode_if_vertical_p:</code>
<code>\mode_if_vertical:TF *</code>	<code>\mode_if_vertical:TF {\langle true code \rangle} {\langle false code \rangle}</code>

Detects if T_EX is currently in vertical mode.

9.7 Primitive conditionals

<code>\if_predicate:w *</code>	<code>\if_predicate:w \langle predicate \rangle \langle true code \rangle \else: \langle false code \rangle \fi:</code>
--------------------------------	---

This function takes a predicate function and branches according to the result. (In practice this function would also accept a single boolean variable in place of the *⟨predicate⟩* but to make the coding clearer this should be done through `\if_bool:N`.)

<code>\if_bool:N *</code>	<code>\if_bool:N \langle boolean \rangle \langle true code \rangle \else: \langle false code \rangle \fi:</code>
---------------------------	--

This function takes a boolean variable and branches according to the result.

9.8 Nestable recursions and mappings

There are a number of places where recursion or mapping constructs are used in `expl3`. At a low-level, these typically require insertion of tokens at the end of the content to allow “clean up”. To support such mappings in a nestable form, the following functions are provided.

<code>\prg_break_point:Nn *</code>	<code>\prg_break_point:Nn \langle type \rangle_map_break: {\langle code \rangle}</code>
------------------------------------	---

New: 2018-03-26

Used to mark the end of a recursion or mapping: the functions `\⟨type⟩_map_break:` and `\⟨type⟩_map_break:n` use this to break out of the loop (see `\prg_map_break:Nn` for how to set these up). After the loop ends, the *⟨code⟩* is inserted into the input stream. This occurs even if the break functions are *not* applied: `\prg_break_point:Nn` is functionally-equivalent in these cases to `\use_ii:nn`.

`\prg_map_break:Nn` ★
 New: 2018-03-26

`\prg_map_break:Nn \<type>_map_break: {\<user code>}`
`...`
`\prg_break_point:Nn \<type>_map_break: {\<ending code>}`

Breaks a recursion in mapping contexts, inserting in the input stream the $\langle user\ code \rangle$ after the $\langle ending\ code \rangle$ for the loop. The function breaks loops, inserting their $\langle ending\ code \rangle$, until reaching a loop with the same $\langle type \rangle$ as its first argument. This $\backslash\langle type \rangle_map_break:$ argument must be defined; it is simply used as a recognizable marker for the $\langle type \rangle$.

For types with mappings defined in the kernel, $\backslash\langle type \rangle_map_break:$ and $\backslash\langle type \rangle_map_break:n$ are defined as $\backslashprg_map_break:Nn \backslash\langle type \rangle_map_break: \{ \}$ and the same with $\{ \}$ omitted.

9.8.1 Simple mappings

In addition to the more complex mappings above, non-nestable mappings are used in a number of locations and support is provided for these.

`\prg_break_point: *`
 New: 2018-03-27

This copy of $\backslashprg_do_nothing:$ is used to mark the end of a fast short-term recursion: the function $\backslashprg_break:n$ uses this to break out of the loop.

`\prg_break: *`
`\prg_break:n *`
 New: 2018-03-27

`\prg_break:n {\<code>}` ... $\backslashprg_break_point:$
 Breaks a recursion which has no $\langle ending\ code \rangle$ and which is not a user-breakable mapping (see for instance $\backslashprop_get:Nn$), and inserts the $\langle code \rangle$ in the input stream.

9.9 Internal programming functions

`\group_align_safe_begin: *`
`\group_align_safe_end: *`
 Updated: 2011-08-11

`\group_align_safe_begin:`
`...`
`\group_align_safe_end:`

These functions are used to enclose material in a T_EX alignment environment within a specially-constructed group. This group is designed in such a way that it does not add brace groups to the output but does act as a group for the $\&$ token inside \backslashhalign . This is necessary to allow grabbing of tokens for testing purposes, as T_EX uses group level to determine the effect of alignment tokens. Without the special grouping, the use of a function such as $\backslashpeek_after:Nw$ would result in a forbidden comparison of the internal \backslashendtemplate token, yielding a fatal error. Each $\backslashgroup_align_safe_begin:$ must be matched by a $\backslashgroup_align_safe_end:$, although this does not have to occur within the same function.

Chapter 10

The l3sys package: System/runtime functions

10.1 The name of the job

`\c_sys_jobname_str`

New: 2015-09-19
Updated: 2019-10-27

Constant that gets the “job name” assigned when T_EX starts.

T_EXhackers note: This copies the contents of the primitive `\jobname`. For technical reasons, the string here is not of the same internal form as other, but may be manipulated using normal string functions.

10.2 Date and time

`\c_sys_minute_int`
`\c_sys_hour_int`
`\c_sys_day_int`
`\c_sys_month_int`
`\c_sys_year_int`

New: 2015-09-22

The date and time at which the current job was started: these are all reported as integers.

T_EXhackers note: Whilst the underlying primitives can be altered by the user, this interface to the time and date is intended to be the “real” values.

10.3 Engine

```

\sys_if_engine luatex_p: *
\sys_if_engine luatex:TF *
\sys_if_engine pdftex_p: *
\sys_if_engine pdftex:TF *
\sys_if_engine ptex_p: *
\sys_if_engine ptex:TF *
\sys_if_engine uptex_p: *
\sys_if_engine uptex:TF *
\sys_if_engine xetex_p: *
\sys_if_engine xetex:TF *

```

New: 2015-09-07

```

\c_sys_engine_str

```

New: 2015-09-19

```

\c_sys_engine_exec_str

```

New: 2020-08-20

```

\c_sys_engine_format_str

```

New: 2020-08-20

```

\sys_timer: *

```

New: 2020-09-24

```
\sys_if_engine pdftex:TF {\true code} {\false code}
```

Conditionals which allow engine-specific code to be used. The names follow naturally from those of the engine binaries: note that the (u)ptex tests are for ε -pTeX and ε -upTeX as expl3 requires the ε -TeX extensions. Each conditional is true for *exactly one* supported engine. In particular, `\sys_if_engine_ptex_p:` is true for ε -pTeX but false for ε -upTeX.

The current engine given as a lower case string: one of `luatex`, `pdftex`, `ptex`, `uptex` or `xetex`.

The name of the standard executable for the current TeX engine given as a lower case string: one of `luatex`, `luahtex`, `pdftex`, `eptex`, `euptex` or `xetex`.

The name of the preloaded format for the current TeX run given as a lower case string: one of `lualatex` (or `dvilualatex`), `pdflatex` (or `latex`), `platex`, `uplatex` or `xelatex` for L^ATeX, similar names for plain TeX (except pdfTeX in DVI mode yields `etex`), and `cont-en` for ConTeXt (i.e. the `\fmtname`).

`\sys_timer:`

Expands to the current value of the engine's timer clock, a non-negative integer. This function is only defined for engines with timer support. This command measures not just CPU time but real time (including time waiting for user input). The unit are scaled seconds (2^{-16} seconds).

10.4 Output format

```

\sys_if_output_dvi_p: *
\sys_if_output_dvi:TF *
\sys_if_output_pdf_p: *
\sys_if_output_pdf:TF *

```

New: 2015-09-19

```
\sys_if_output_dvi:TF {\true code} {\false code}
```

Conditionals which give the current output mode the TeX run is operating in. This is always one of two outcomes, DVI mode or PDF mode. The two sets of conditionals are thus complementary and are both provided to allow the programmer to emphasise the most appropriate case.

```

\c_sys_output_str

```

New: 2015-09-19

The current output mode given as a lower case string: one of `dvi` or `pdf`.

10.5 Platform

<code>\sys_if_platform_unix_p:</code>	★	<code>\sys_if_platform_unix:TF</code>	$\{\langle true\ code\rangle\}$	$\{\langle false\ code\rangle\}$
<code>\sys_if_platform_unix:TF</code>	★			
<code>\sys_if_platform_windows_p:</code>	★			
<code>\sys_if_platform_windows:TF</code>	★			

New: 2018-07-27

Conditionals which allow platform-specific code to be used. The names follow the Lua `os.type()` function, *i.e.* all Unix-like systems are `unix` (including Linux and MacOS).

<code>\c_sys_platform_str</code>	
----------------------------------	--

New: 2018-07-27

The current platform given as a lower case string: one of `unix`, `windows` or `unknown`.

10.6 Random numbers

<code>\sys_rand_seed:</code>	★	<code>\sys_rand_seed:</code>
------------------------------	---	------------------------------

New: 2017-05-27

Expands to the current value of the engine's random seed, a non-negative integer. In engines without random number support this expands to 0.

<code>\sys_gset_rand_seed:n</code>	
------------------------------------	--

New: 2017-05-27

`\sys_gset_rand_seed:n` $\{\langle intexpr\rangle\}$
 Globally sets the seed for the engine's pseudo-random number generator to the $\langle integer\ expression\rangle$. This random seed affects all `\..._rand` functions (such as `\int_rand:nn` or `\clist_rand_item:n`) as well as other packages relying on the engine's random number generator. In engines without random number support this produces an error.

TeXhackers note: While a 32-bit (signed) integer can be given as a seed, only the absolute value is used and any number beyond 2^{28} is divided by an appropriate power of 2. We recommend using an integer in $[0, 2^{28} - 1]$.

10.7 Access to the shell

<code>\sys_get_shell:nnN</code>	<code>\sys_get_shell:nnN</code>	$\{\langle shell\ command\rangle\}$	$\{\langle setup\rangle\}$	$\langle tl\ var\rangle$
<code>\sys_get_shell:nnNTF</code>	<code>\sys_get_shell:nnNTF</code>	$\{\langle shell\ command\rangle\}$	$\{\langle setup\rangle\}$	$\langle tl\ var\rangle$
				$\{\langle true\ code\rangle\}$
				$\{\langle false\ code\rangle\}$

New: 2019-09-20

Defines $\langle tl\ var\rangle$ to the text returned by the $\langle shell\ command\rangle$. The $\langle shell\ command\rangle$ is converted to a string using `\tl_to_str:n`. Category codes may need to be set appropriately via the $\langle setup\rangle$ argument, which is run just before running the $\langle shell\ command\rangle$ (in a group). If shell escape is disabled, the $\langle tl\ var\rangle$ will be set to `\q_no_value` in the non-branching version. Note that quote characters (") *cannot* be used inside the $\langle shell\ command\rangle$. The `\sys_get_shell:nnNTF` conditional inserts the `true code` if the shell is available and no quote is detected, and the `false code` otherwise.

<code>\c_sys_shell_escape_int</code>	This variable exposes the internal triple of the shell escape status. The possible values are
New: 2017-05-27	

- 0 Shell escape is disabled
- 1 Unrestricted shell escape is enabled
- 2 Restricted shell escape is enabled

<code>\sys_if_shell_p: *</code>	<code>\sys_if_shell_p:</code>
<code>\sys_if_shell:TF *</code>	<code>\sys_if_shell:TF {<true code>} {<false code>}</code>
New: 2017-05-27	Performs a check for whether shell escape is enabled. This returns true if either of restricted or unrestricted shell escape is enabled.

<code>\sys_if_shell_unrestricted_p: *</code>	<code>\sys_if_shell_unrestricted_p:</code>
<code>\sys_if_shell_unrestricted:TF *</code>	<code>\sys_if_shell_unrestricted:TF {<true code>} {<false code>}</code>
New: 2017-05-27	

Performs a check for whether *unrestricted* shell escape is enabled.

<code>\sys_if_shell_restricted_p: *</code>	<code>\sys_if_shell_restricted_p:</code>
<code>\sys_if_shell_restricted:TF *</code>	<code>\sys_if_shell_restricted:TF {<true code>} {<false code>}</code>
New: 2017-05-27	

Performs a check for whether *restricted* shell escape is enabled. This returns false if unrestricted shell escape is enabled. Unrestricted shell escape is not considered a superset of restricted shell escape in this case. To find whether any shell escape is enabled use `\sys_if_shell:`.

<code>\sys_shell_now:n</code>	<code>\sys_shell_now:n {<tokens>}</code>
<code>\sys_shell_now:x</code>	Execute <i><tokens></i> through shell escape immediately.
New: 2017-05-27	

<code>\sys_shell_shipout:n</code>	<code>\sys_shell_shipout:n {<tokens>}</code>
<code>\sys_shell_shipout:x</code>	Execute <i><tokens></i> through shell escape at shipout.
New: 2017-05-27	

10.8 Loading configuration data

<code>\sys_load_backend:n</code>	<code>\sys_load_backend:n {<backend>}</code>
New: 2019-09-12	Loads the additional configuration file needed for backend support. If the <i><backend></i> is empty, the standard backend for the engine in use will be loaded. This command may only be used once.

<code>\c_sys_backend_str</code>	Set to the name of the backend in use by <code>\sys_load_backend:n</code> when issued.
---------------------------------	--

<hr/> <code>\sys_load_debug:</code> <hr/>	<code>\sys_load_debug:</code>
<code>New: 2019-09-12</code>	Load the additional configuration file for debugging support.

10.8.1 Final settings

<hr/> <code>\sys_finalise:</code> <hr/>	<code>\sys_finalise:</code>
<code>New: 2019-10-06</code>	Finalises all system-dependent functionality: required before loading a backend.

Chapter 11

The l3msg package

Messages

Messages need to be passed to the user by modules, either when errors occur or to indicate how the code is proceeding. The `l3msg` module provides a consistent method for doing this (as opposed to writing directly to the terminal or log).

The system used by `l3msg` to create messages divides the process into two distinct parts. Named messages are created in the first part of the process; at this stage, no decision is made about the type of output that the message will produce. The second part of the process is actually producing a message. At this stage a choice of message *class* has to be made, for example `error`, `warning` or `info`.

By separating out the creation and use of messages, several benefits are available. First, the messages can be altered later without needing details of where they are used in the code. This makes it possible to alter the language used, the detail level and so on. Secondly, the output which results from a given message can be altered. This can be done on a message class, module or message name basis. In this way, message behaviour can be altered and messages can be entirely suppressed.

11.1 Creating new messages

All messages have to be created before they can be used. The text of messages is automatically wrapped to the length available in the console. As a result, formatting is only needed where it helps to show meaning. In particular, `\` may be used to force a new line and `_` forces an explicit space. Additionally, `\{`, `\#`, `\}`, `\%` and `\~` can be used to produce the corresponding character.

Messages may be subdivided *by one level* using the `/` character. This is used within the message filtering system to allow for example the L^AT_EX kernel messages to belong to the module `LaTeX` while still being filterable at a more granular level. Thus for example

```
\msg_new:nnnn { mymodule } { submodule / message } ...
```

will allow to filter out specifically messages from the `submodule`.

`\msg_new:nnnn`
`\msg_new:nnn`

Updated: 2011-08-16

`\msg_new:nnnn {<module>} {<message>} {<text>} {<more text>}`

Creates a *<message>* for a given *<module>*. The message is defined to first give *<text>* and then *<more text>* if the user requests it. If no *<more text>* is available then a standard text is given instead. Within *<text>* and *<more text>* four parameters (**#1** to **#4**) can be used: these will be supplied at the time the message is used. An error is raised if the *<message>* already exists.

`\msg_set:nnnn`
`\msg_set:nnn`
`\msg_gset:nnnn`
`\msg_gset:nnn`

`\msg_set:nnnn {<module>} {<message>} {<text>} {<more text>}`

Sets up the text for a *<message>* for a given *<module>*. The message is defined to first give *<text>* and then *<more text>* if the user requests it. If no *<more text>* is available then a standard text is given instead. Within *<text>* and *<more text>* four parameters (**#1** to **#4**) can be used: these will be supplied at the time the message is used.

`\msg_if_exist_p:nn *`
`\msg_if_exist:nnTF *`

New: 2012-03-03

`\msg_if_exist_p:nn {<module>} {<message>}`

`\msg_if_exist:nnTF {<module>} {<message>} {<true code>} {<false code>}`

Tests whether the *<message>* for the *<module>* is currently defined.

11.2 Customizable information for message modules

`\msg_module_name:n *`

New: 2018-10-10

`\msg_module_name:n {<module>}`

Expands to the public name of the *<module>* as defined by `\g_msg_module_name_prop` (or otherwise leaves the *<module>* unchanged).

`\msg_module_type:n *`

New: 2018-10-10

`\msg_module_type:n {<module>}`

Expands to the description which applies to the *<module>*, for example a **Package** or **Class**. The information here is defined in `\g_msg_module_type_prop`, and will default to **Package** if an entry is not present.

`\g_msg_module_name_prop`

New: 2018-10-10

Provides a mapping between the module name used for messages, and that for documentation. For example, L^AT_EX3 core messages are stored in the reserved L^AT_EX tree, but are printed as L^AT_EX3.

`\g_msg_module_type_prop`

New: 2018-10-10

Provides a mapping between the module name used for messages, and that type of module. For example, for L^AT_EX3 core messages, an empty entry is set here meaning that they are not described using the standard **Package** text.

11.3 Contextual information for messages

`\msg_line_context: ☆`

`\msg_line_context:`

Prints the current line number when a message is given, and thus suitable for giving context to messages. The number itself is preceded by the text **on line**.

<hr/> <hr/>	<code>\msg_line_number: *</code>	<code>\msg_line_number:</code>
		Prints the current line number when a message is given.
<hr/> <hr/>	<code>\msg_fatal_text:n *</code>	<code>\msg_fatal_text:n {<module>}</code>
		Produces the standard text
		Fatal Package <module> Error
		This function can be redefined to alter the language in which the message is given, using #1 as the name of the <module> to be included.
<hr/> <hr/>	<code>\msg_critical_text:n *</code>	<code>\msg_critical_text:n {<module>}</code>
		Produces the standard text
		Critical Package <module> Error
		This function can be redefined to alter the language in which the message is given, using #1 as the name of the <module> to be included.
<hr/> <hr/>	<code>\msg_error_text:n *</code>	<code>\msg_error_text:n {<module>}</code>
		Produces the standard text
		Package <module> Error
		This function can be redefined to alter the language in which the message is given, using #1 as the name of the <module> to be included.
<hr/> <hr/>	<code>\msg_warning_text:n *</code>	<code>\msg_warning_text:n {<module>}</code>
		Produces the standard text
		Package <module> Warning
		This function can be redefined to alter the language in which the message is given, using #1 as the name of the <module> to be included. The <type> of <module> may be adjusted: Package is the standard outcome: see <code>\msg_module_type:n</code> .
<hr/> <hr/>	<code>\msg_info_text:n *</code>	<code>\msg_info_text:n {<module>}</code>
		Produces the standard text:
		Package <module> Info
		This function can be redefined to alter the language in which the message is given, using #1 as the name of the <module> to be included. The <type> of <module> may be adjusted: Package is the standard outcome: see <code>\msg_module_type:n</code> .

<code>\msg_see_documentation_text:n</code>	★	<code>\msg_see_documentation_text:n {<module>}</code>
--	---	---

Updated: 2018-09-30

Produces the standard text

See the `<module>` documentation for further information.

This function can be redefined to alter the language in which the message is given, using #1 as the name of the `<module>` to be included. The name of the `<module>` is produced using `\msg_module_name:n`.

11.4 Issuing messages

Messages behave differently depending on the message class. In all cases, the message may be issued supplying 0 to 4 arguments. If the number of arguments supplied here does not match the number in the definition of the message, extra arguments are ignored, or empty arguments added (of course the sense of the message may be impaired). The four arguments are converted to strings before being added to the message text: the `x`-type variants should be used to expand material. Note that this expansion takes place with the standard definitions in effect, which means that shorthands such as `\~` or `\\` are *not* available; instead one should use `\iow_char:N \~` and `\iow_newline:`, respectively. The following message classes exist:

- **fatal**, ending the T_EX run;
- **critical**, ending the file being input;
- **error**, interrupting the T_EX run without ending it;
- **warning**, written to terminal and log file, for important messages that may require corrections by the user;
- **note** (less common than **info**) for important information messages written to the terminal and log file;
- **info** for normal information messages written to the log file only;
- **term** and **log** for un-decorated messages written to the terminal and log file, or to the log file only;
- **none** for suppressed messages.

<code>\msg_fatal:nnnnnn</code>	<code>\msg_fatal:nnnnnn {<module>} {<message>} {<arg one>} {<arg two>} {<arg three>} {<arg four>}</code>
<code>\msg_fatal:nnxxxx</code>	
<code>\msg_fatal:nnnnnn</code>	Issues <code><module></code> error <code><message></code> , passing <code><arg one></code> to <code><arg four></code> to the text-creating functions. After issuing a fatal error the T _E X run halts. No PDF file will be produced in this case (DVI mode runs may produce a truncated DVI file).
<code>\msg_fatal:nnxxx</code>	
<code>\msg_fatal:nnnn</code>	
<code>\msg_fatal:nnxx</code>	
<code>\msg_fatal:nnn</code>	
<code>\msg_fatal:nnx</code>	
<code>\msg_fatal:nn</code>	

Updated: 2012-08-11

```

\msg_critical:nnnnnn
\msg_critical:nnxxxx
\msg_critical:nnnnn
\msg_critical:nnxxx
\msg_critical:nnnn
\msg_critical:nnxx
\msg_critical:nnn
\msg_critical:nnx
\msg_critical:nn

```

Updated: 2012-08-11

```

\msg_critical:nnnnnn {\module} {\message} {\arg one} {\arg two} {\arg three}
{\arg four}

```

Issues $\langle module \rangle$ error $\langle message \rangle$, passing $\langle arg one \rangle$ to $\langle arg four \rangle$ to the text-creating functions. After issuing a critical error, T_EX stops reading the current input file. This may halt the T_EX run (if the current file is the main file) or may abort reading a sub-file.

T_EXhackers note: The T_EX `\endinput` primitive is used to exit the file. In particular, the rest of the current line remains in the input stream.

```

\msg_error:nnnnnn
\msg_error:nnxxxx
\msg_error:nnnnn
\msg_error:nnxxx
\msg_error:nnnn
\msg_error:nnxx
\msg_error:nnn
\msg_error:nnx
\msg_error:nn

```

Updated: 2012-08-11

```

\msg_error:nnnnnn {\module} {\message} {\arg one} {\arg two} {\arg three}
{\arg four}

```

Issues $\langle module \rangle$ error $\langle message \rangle$, passing $\langle arg one \rangle$ to $\langle arg four \rangle$ to the text-creating functions. The error interrupts processing and issues the text at the terminal. After user input, the run continues.

```

\msg_warning:nnnnnn
\msg_warning:nnxxxx
\msg_warning:nnnnn
\msg_warning:nnxxx
\msg_warning:nnnn
\msg_warning:nnxx
\msg_warning:nnn
\msg_warning:nnx
\msg_warning:nn

```

Updated: 2012-08-11

```

\msg_warning:nnxxxx {\module} {\message} {\arg one} {\arg two} {\arg three}
{\arg four}

```

Issues $\langle module \rangle$ warning $\langle message \rangle$, passing $\langle arg one \rangle$ to $\langle arg four \rangle$ to the text-creating functions. The warning text is added to the log file and the terminal, but the T_EX run is not interrupted.

<hr/>	
<code>\msg_note:nnnnnn</code>	<code>\msg_note:nnnnnn {<module>} {<message>} {<arg one>} {<arg two>} {<arg three>} {<arg four>}</code>
<code>\msg_note:nnxxxx</code>	
<code>\msg_note:nnnnn</code>	<code>\msg_info:nnnnnn {<module>} {<message>} {<arg one>} {<arg two>} {<arg three>} {<arg four>}</code>
<code>\msg_note:nnxxx</code>	
<code>\msg_note:nnnn</code>	Issues <i><module></i> information <i><message></i> , passing <i><arg one></i> to <i><arg four></i> to the text-creating functions. For the more common <code>\msg_info:nnnnnn</code> , the information text is added to the log file only, while <code>\msg_note:nnnnnn</code> adds the info text to both the log file and the terminal. The \TeX run is not interrupted.
<code>\msg_note:nnxx</code>	
<code>\msg_note:nnn</code>	
<code>\msg_note:nnx</code>	
<code>\msg_note:nn</code>	
<code>\msg_info:nnnnnn</code>	
<code>\msg_info:nnxxxx</code>	
<code>\msg_info:nnnnn</code>	
<code>\msg_info:nnxxx</code>	
<code>\msg_info:nnnn</code>	
<code>\msg_info:nnxx</code>	
<code>\msg_info:nnn</code>	
<code>\msg_info:nnx</code>	
<code>\msg_info:nn</code>	
<hr/>	
New: 2021-05-18	
<hr/>	
<code>\msg_term:nnnnnn</code>	<code>\msg_term:nnnnnn {<module>} {<message>} {<arg one>} {<arg two>} {<arg three>} {<arg four>}</code>
<code>\msg_term:nnxxxx</code>	
<code>\msg_term:nnnnn</code>	<code>\msg_log:nnnnnn {<module>} {<message>} {<arg one>} {<arg two>} {<arg three>} {<arg four>}</code>
<code>\msg_term:nnxxx</code>	
<code>\msg_term:nnnn</code>	Issues <i><module></i> information <i><message></i> , passing <i><arg one></i> to <i><arg four></i> to the text-creating functions. The output is briefer than <code>\msg_info:nnnnnn</code> , omitting for instance the module name. It is added to the log file by <code>\msg_log:nnnnnn</code> while <code>\msg_term:nnnnnn</code> also prints it on the terminal.
<code>\msg_term:nnxx</code>	
<code>\msg_term:nnn</code>	
<code>\msg_term:nnx</code>	
<code>\msg_term:nn</code>	
<code>\msg_log:nnnnnn</code>	
<code>\msg_log:nnxxxx</code>	
<code>\msg_log:nnnnn</code>	
<code>\msg_log:nnxxx</code>	
<code>\msg_log:nnnn</code>	
<code>\msg_log:nnxx</code>	
<code>\msg_log:nnn</code>	
<code>\msg_log:nnx</code>	
<code>\msg_log:nn</code>	
<hr/>	
Updated: 2012-08-11	
<hr/>	
<code>\msg_none:nnnnnn</code>	<code>\msg_none:nnnnnn {<module>} {<message>} {<arg one>} {<arg two>} {<arg three>} {<arg four>}</code>
<code>\msg_none:nnxxxx</code>	
<code>\msg_none:nnnnn</code>	Does nothing: used as a message class to prevent any output at all (see the discussion of message redirection).
<code>\msg_none:nnxxx</code>	
<code>\msg_none:nnnn</code>	
<code>\msg_none:nnxx</code>	
<code>\msg_none:nnn</code>	
<code>\msg_none:nnx</code>	
<code>\msg_none:nn</code>	
<hr/>	
Updated: 2012-08-11	

11.4.1 Messages for showing material

<code>\msg_show:nnnnnn</code> <code>\msg_show:nnxxxx</code> <code>\msg_show:nnnnn</code> <code>\msg_show:nnxxx</code> <code>\msg_show:nnnn</code> <code>\msg_show:nnxx</code> <code>\msg_show:nnn</code> <code>\msg_show:nnx</code> <code>\msg_show:nn</code>	<code>\msg_show:nnnnnn {<module>} {<message>} {<arg one>} {<arg two>} {<arg three>} {<arg four>}</code> <p>Issues <i><module></i> information <i><message></i>, passing <i><arg one></i> to <i><arg four></i> to the text-creating functions. The information text is shown on the terminal and the T_EX run is interrupted in a manner similar to <code>\tl_show:n</code>. This is used in conjunction with <code>\msg_show_item:n</code> and similar functions to print complex variable contents completely. If the formatted text does not contain <code>>~</code> at the start of a line, an additional line <code>>~.</code> will be put at the end. In addition, a final period is added if not present.</p>
---	--

New: 2017-12-04

11.4.2 Expandable error messages

In very rare cases it may be necessary to produce errors in an expansion-only context. The functions in this section should only be used if there is no alternative approach using `\msg_error:nnnnnn` or other non-expandable commands from the previous section. Despite having a similar interface as non-expandable messages, expandable errors must be handled internally very differently from normal error messages, as none of the tools to print to the terminal or the log file are expandable. As a result, short-hands such as `\{` or `\` do not work, and messages must be very short (with default settings, they are truncated after approximately 50 characters). It is advisable to ensure that the message is understandable even when truncated, by putting the most important information up front. Another particularity of expandable messages is that they cannot be redirected or turned off by the user.

<code>\msg_expandable_error:nnnnnn</code> * <code>\msg_expandable_error:nnffff</code> * <code>\msg_expandable_error:nnnnn</code> * <code>\msg_expandable_error:nnffff</code> * <code>\msg_expandable_error:nnnn</code> * <code>\msg_expandable_error:nnff</code> * <code>\msg_expandable_error:nnn</code> * <code>\msg_expandable_error:nnf</code> * <code>\msg_expandable_error:nn</code> *	<code>\msg_expandable_error:nnnnnn {<module>} {<message>} {<arg one>} {<arg two>} {<arg three>} {<arg four>}</code>
--	---

New: 2015-08-06

Updated: 2019-02-28

Issues an “Undefined error” message from T_EX itself using the undefined control sequence `\::error` then prints “! *<module>*: ”*<error message>*”, which should be short. With default settings, anything beyond approximately 60 characters long (or bytes in some engines) is cropped. A leading space might be removed as well.

11.5 Redirecting messages

Each message has a “name”, which can be used to alter the behaviour of the message when it is given. Thus we might have

```
\msg_new:nnnn { module } { my-message } { Some-text } { Some-more-text }
```

to define a message, with

```
\msg_error:nn { module } { my-message }
```

when it is used. With no filtering, this raises an error. However, we could alter the behaviour with

```
\msg_redirect_class:nn { error } { warning }
```

to turn all errors into warnings, or with

```
\msg_redirect_module:nnn { module } { error } { warning }
```

to alter only messages from that module, or even

```
\msg_redirect_name:nnn { module } { my-message } { warning }
```

to target just one message. Redirection applies first to individual messages, then to messages from one module and finally to messages of one class. Thus it is possible to select out an individual message for special treatment even if the entire class is already redirected.

Multiple redirections are possible. Redirections can be cancelled by providing an empty argument for the target class. Redirection to a missing class raises an error immediately. Infinite loops are prevented by eliminating the redirection starting from the target of the redirection that caused the loop to appear. Namely, if redirections are requested as $A \rightarrow B$, $B \rightarrow C$ and $C \rightarrow A$ in this order, then the $A \rightarrow B$ redirection is cancelled.

```
\msg_redirect_class:nn
```

Updated: 2012-04-27

```
\msg_redirect_class:nn {<class one>} {<class two>}
```

Changes the behaviour of messages of *<class one>* so that they are processed using the code for those of *<class two>*. Each *<class>* can be one of **fatal**, **critical**, **error**, **warning**, **note**, **info**, **term**, **log**, **none**.

```
\msg_redirect_module:nnn
```

Updated: 2012-04-27

```
\msg_redirect_module:nnn {<module>} {<class one>} {<class two>}
```

Redirects message of *<class one>* for *<module>* to act as though they were from *<class two>*. Messages of *<class one>* from sources other than *<module>* are not affected by this redirection. This function can be used to make some messages “silent” by default. For example, all of the **warning** messages of *<module>* could be turned off with:

```
\msg_redirect_module:nnn { module } { warning } { none }
```

```
\msg_redirect_name:nnn
```

Updated: 2012-04-27

```
\msg_redirect_name:nnn {<module>} {<message>} {<class>}
```

Redirects a specific *<message>* from a specific *<module>* to act as a member of *<class>* of messages. No further redirection is performed. This function can be used to make a selected message “silent” without changing global parameters:

```
\msg_redirect_name:nnn { module } { annoying-message } { none }
```

Chapter 12

The **l3file** package

File and I/O operations

This module provides functions for working with external files. Some of these functions apply to an entire file, and have prefix `\file_...`, while others are used to work with files on a line by line basis and have prefix `\ior_...` (reading) or `\iow_...` (writing).

It is important to remember that when reading external files \TeX attempts to locate them using both the operating system path and entries in the \TeX file database (most \TeX systems use such a database). Thus the “current path” for \TeX is somewhat broader than that for other programs.

For functions which expect a *file name* argument, this argument may contain both literal items and expandable content, which should on full expansion be the desired file name. Active characters (as declared in `\l_char_active_seq`) are *not* expanded, allowing the direct use of these in file names. Quote tokens (") are not permitted in file names as they are reserved for internal use by some \TeX primitives.

Spaces are trimmed at the beginning and end of the file name: this reflects the fact that some file systems do not allow or interact unpredictably with spaces in these positions. When no extension is given, this will trim spaces from the start of the name only.

12.1 Input–output stream management

As \TeX engines have a limited number of input and output streams, direct use of the streams by the programmer is not supported in \LaTeX 3. Instead, an internal pool of streams is maintained, and these are allocated and deallocated as needed by other modules. As a result, the programmer should close streams when they are no longer needed, to release them for other processes.

Note that I/O operations are global: streams should all be declared with global names and treated accordingly.

<hr/> <code>\ior_new:N</code> <code>\ior_new:c</code> <code>\iow_new:N</code> <code>\iow_new:c</code> <hr/> New: 2011-09-26 Updated: 2011-12-27	<code>\ior_new:N <stream></code> <code>\iow_new:N <stream></code> <p>Globally reserves the name of the $\langle stream \rangle$, either for reading or for writing as appropriate. The $\langle stream \rangle$ is not opened until the appropriate <code>\..._open:Nn</code> function is used. Attempting to use a $\langle stream \rangle$ which has not been opened is an error, and the $\langle stream \rangle$ will behave as the corresponding <code>\c_term_....</code></p>
<hr/> <code>\ior_open:Nn</code> <code>\ior_open:cn</code> <hr/> Updated: 2012-02-10	<code>\ior_open:Nn <stream> {<file name>}</code> <p>Opens $\langle file name \rangle$ for reading using $\langle stream \rangle$ as the control sequence for file access. If the $\langle stream \rangle$ was already open it is closed before the new operation begins. The $\langle stream \rangle$ is available for access immediately and will remain allocated to $\langle file name \rangle$ until a <code>\ior_close:N</code> instruction is given or the T_EX run ends. If the file is not found, an error is raised.</p>
<hr/> <code>\ior_open:NnTF</code> <code>\ior_open:cnTF</code> <hr/> New: 2013-01-12	<code>\ior_open:NnTF <stream> {<file name>} {<true code>} {<false code>}</code> <p>Opens $\langle file name \rangle$ for reading using $\langle stream \rangle$ as the control sequence for file access. If the $\langle stream \rangle$ was already open it is closed before the new operation begins. The $\langle stream \rangle$ is available for access immediately and will remain allocated to $\langle file name \rangle$ until a <code>\ior_close:N</code> instruction is given or the T_EX run ends. The $\langle true code \rangle$ is then inserted into the input stream. If the file is not found, no error is raised and the $\langle false code \rangle$ is inserted into the input stream.</p>
<hr/> <code>\iow_open:Nn</code> <code>\iow_open:cn</code> <hr/> Updated: 2012-02-09	<code>\iow_open:Nn <stream> {<file name>}</code> <p>Opens $\langle file name \rangle$ for writing using $\langle stream \rangle$ as the control sequence for file access. If the $\langle stream \rangle$ was already open it is closed before the new operation begins. The $\langle stream \rangle$ is available for access immediately and will remain allocated to $\langle file name \rangle$ until a <code>\iow_close:N</code> instruction is given or the T_EX run ends. Opening a file for writing clears any existing content in the file (<i>i.e.</i> writing is <i>not</i> additive).</p>
<hr/> <code>\ior_close:N</code> <code>\ior_close:c</code> <code>\iow_close:N</code> <code>\iow_close:c</code> <hr/> Updated: 2012-07-31	<code>\ior_close:N <stream></code> <code>\iow_close:N <stream></code> <p>Closes the $\langle stream \rangle$. Streams should always be closed when they are finished with as this ensures that they remain available to other programmers.</p>
<hr/> <code>\ior_show:N</code> <code>\ior_show:c</code> <code>\ior_log:N</code> <code>\ior_log:c</code> <code>\iow_show:N</code> <code>\iow_show:c</code> <code>\iow_log:N</code> <code>\iow_log:c</code> <hr/> New: 2021-05-11	<code>\ior_show:N <stream></code> <code>\ior_log:N <stream></code> <code>\iow_show:N <stream></code> <code>\iow_log:N <stream></code> <p>Display (to the terminal or log file) the file name associated to the (read or write) $\langle stream \rangle$.</p>

<code>\ior_show_list:</code>	<code>\ior_show_list:</code>
<code>\ior_log_list:</code>	<code>\ior_log_list:</code>
<code>\iow_show_list:</code>	<code>\iow_show_list:</code>
<code>\iow_log_list:</code>	<code>\iow_log_list:</code>

New: 2017-06-27

Display (to the terminal or log file) a list of the file names associated with each open (read or write) stream. This is intended for tracking down problems.

12.1.1 Reading from files

Reading from files and reading from the terminal are separate processes in `expl3`. The functions `\ior_get:NN` and `\ior_str_get:NN`, and their branching equivalents, are designed to work with *files*.

<code>\ior_get:NN</code>	<code>\ior_get:NN <stream> <token list variable></code>
<code>\ior_get:NNTF</code>	<code>\ior_get:NNTF <stream> <token list variable> <true code> <false code></code>

New: 2012-06-24
Updated: 2019-03-23

Function that reads one or more lines (until an equal number of left and right braces are found) from the file input `<stream>` and stores the result locally in the `<token list>` variable. The material read from the `<stream>` is tokenized by `TEX` according to the category codes and `\endlinechar` in force when the function is used. Assuming normal settings, any lines which do not end in a comment character `%` have the line ending converted to a space, so for example input

```
a b c
```

results in a token list `a_b_c_`. Any blank line is converted to the token `\par`. Therefore, blank lines can be skipped by using a test such as

```
\ior_get:NN \l_my_stream \l_tmpa_tl
\tl_set:Nn \l_tmpb_tl { \par }
\tl_if_eq:NNF \l_tmpa_tl \l_tmpb_tl
...
```

Also notice that if multiple lines are read to match braces then the resulting token list can contain `\par` tokens. In the non-branching version, where the `<stream>` is not open the `<tl var>` is set to `\q_no_value`.

T_EXhackers note: This protected macro is a wrapper around the `TEX` primitive `\read`. Regardless of settings, `TEX` replaces trailing space and tab characters (character codes 32 and 9) in each line by an end-of-line character (character code `\endlinechar`, omitted if `\endlinechar` is negative or too large) before turning characters into tokens according to current category codes. With default settings, spaces appearing at the beginning of lines are also ignored.

<code>\ior_str_get:NN</code>	<code>\ior_str_get:NN <stream> <token list variable></code>
<code>\ior_str_get:NNTF</code>	<code>\ior_str_get:NNTF <stream> <token list variable> <true code> <false code></code>

New: 2016-12-04
Updated: 2019-03-23

Function that reads one line from the file input $\langle stream \rangle$ and stores the result locally in the $\langle token list \rangle$ variable. The material is read from the $\langle stream \rangle$ as a series of tokens with category code 12 (other), with the exception of space characters which are given category code 10 (space). Multiple whitespace characters are retained by this process. It always only reads one line and any blank lines in the input result in the $\langle token list variable \rangle$ being empty. Unlike `\ior_get:NN`, line ends do not receive any special treatment. Thus input

a b c

results in a token list a b c with the letters a, b, and c having category code 12. In the non-branching version, where the $\langle stream \rangle$ is not open the $\langle tl var \rangle$ is set to `\q_no_value`.

TeXhackers note: This protected macro is a wrapper around the ε -TeX primitive `\readline`. Regardless of settings, TeX removes trailing space and tab characters (character codes 32 and 9). However, the end-line character normally added by this primitive is not included in the result of `\ior_str_get:NN`.

All mappings are done at the current group level, *i.e.* any local assignments made by the $\langle function \rangle$ or $\langle code \rangle$ discussed below remain in effect after the loop.

<code>\ior_map_inline:Nn</code>	<code>\ior_map_inline:Nn <stream> {\inline function}</code>
---------------------------------	---

New: 2012-02-11

Applies the $\langle inline function \rangle$ to each set of $\langle lines \rangle$ obtained by calling `\ior_get:NN` until reaching the end of the file. TeX ignores any trailing new-line marker from the file it reads. The $\langle inline function \rangle$ should consist of code which receives the $\langle line \rangle$ as #1.

<code>\ior_str_map_inline:Nn</code>	<code>\ior_str_map_inline:Nn <stream> {\inline function}</code>
-------------------------------------	---

New: 2012-02-11

Applies the $\langle inline function \rangle$ to every $\langle line \rangle$ in the $\langle stream \rangle$. The material is read from the $\langle stream \rangle$ as a series of tokens with category code 12 (other), with the exception of space characters which are given category code 10 (space). The $\langle inline function \rangle$ should consist of code which receives the $\langle line \rangle$ as #1. Note that TeX removes trailing space and tab characters (character codes 32 and 9) from every line upon input. TeX also ignores any trailing new-line marker from the file it reads.

<code>\ior_map_variable:NNn</code>	<code>\ior_map_variable:NNn <stream> <tl var> {\code}</code>
------------------------------------	--

New: 2019-01-13

For each set of $\langle lines \rangle$ obtained by calling `\ior_get:NN` until reaching the end of the file, stores the $\langle lines \rangle$ in the $\langle tl var \rangle$ then applies the $\langle code \rangle$. The $\langle code \rangle$ will usually make use of the $\langle variable \rangle$, but this is not enforced. The assignments to the $\langle variable \rangle$ are local. Its value after the loop is the last set of $\langle lines \rangle$, or its original value if the $\langle stream \rangle$ is empty. TeX ignores any trailing new-line marker from the file it reads. This function is typically faster than `\ior_map_inline:Nn`.

\ior_str_map_variable:Nn

New: 2019-01-13

\ior_str_map_variable:Nn $\langle stream \rangle$ $\langle variable \rangle$ $\{ \langle code \rangle \}$

For each $\langle line \rangle$ in the $\langle stream \rangle$, stores the $\langle line \rangle$ in the $\langle variable \rangle$ then applies the $\langle code \rangle$. The material is read from the $\langle stream \rangle$ as a series of tokens with category code 12 (other), with the exception of space characters which are given category code 10 (space). The $\langle code \rangle$ will usually make use of the $\langle variable \rangle$, but this is not enforced. The assignments to the $\langle variable \rangle$ are local. Its value after the loop is the last $\langle line \rangle$, or its original value if the $\langle stream \rangle$ is empty. Note that T_EX removes trailing space and tab characters (character codes 32 and 9) from every line upon input. T_EX also ignores any trailing new-line marker from the file it reads. This function is typically faster than **\ior_str_map_inline:Nn**.

\ior_map_break:

New: 2012-06-29

\ior_map_break:

Used to terminate a **\ior_map...** function before all lines from the $\langle stream \rangle$ have been processed. This normally takes place within a conditional statement, for example

```
\ior_map_inline:Nn \l_my_ior
{
  \str_if_eq:nnTF { #1 } { bingo }
  { \ior_map_break: }
  {
    % Do something useful
  }
}
```

Use outside of a **\ior_map...** scenario leads to low level T_EX errors.

T_EXhackers note: When the mapping is broken, additional tokens may be inserted before further items are taken from the input stream. This depends on the design of the mapping function.

\ior_map_break:n

New: 2012-06-29

\ior_map_break:n $\{ \langle code \rangle \}$

Used to terminate a **\ior_map...** function before all lines in the $\langle stream \rangle$ have been processed, inserting the $\langle code \rangle$ after the mapping has ended. This normally takes place within a conditional statement, for example

```
\ior_map_inline:Nn \l_my_ior
{
  \str_if_eq:nnTF { #1 } { bingo }
  { \ior_map_break:n { <code> } }
  {
    % Do something useful
  }
}
```

Use outside of a **\ior_map...** scenario leads to low level T_EX errors.

T_EXhackers note: When the mapping is broken, additional tokens may be inserted before the $\langle code \rangle$ is inserted into the input stream. This depends on the design of the mapping function.

<code>\ior_if_eof_p:N</code> *	<code>\ior_if_eof_p:N</code> $\langle stream \rangle$
<code>\ior_if_eof:NTF</code> *	<code>\ior_if_eof:NTF</code> $\langle stream \rangle$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$

Updated: 2012-02-10

Tests if the end of a file $\langle stream \rangle$ has been reached during a reading operation. The test also returns a `true` value if the $\langle stream \rangle$ is not open.

12.1.2 Writing to files

<code>\iow_now:Nn</code>	<code>\iow_now:Nn</code> $\langle stream \rangle$ $\{\langle tokens \rangle\}$
<code>\iow_now:(Nx cn cx)</code>	

Updated: 2012-06-05

This functions writes $\langle tokens \rangle$ to the specified $\langle stream \rangle$ immediately (*i.e.* the write operation is called on expansion of `\iow_now:Nn`).

<code>\iow_log:n</code>	<code>\iow_log:n</code> $\{\langle tokens \rangle\}$
<code>\iow_log:x</code>	

This function writes the given $\langle tokens \rangle$ to the log (transcript) file immediately: it is a dedicated version of `\iow_now:Nn`.

<code>\iow_term:n</code>	<code>\iow_term:n</code> $\{\langle tokens \rangle\}$
<code>\iow_term:x</code>	

This function writes the given $\langle tokens \rangle$ to the terminal file immediately: it is a dedicated version of `\iow_now:Nn`.

<code>\iow_shipout:Nn</code>	<code>\iow_shipout:Nn</code> $\langle stream \rangle$ $\{\langle tokens \rangle\}$
<code>\iow_shipout:(Nx cn cx)</code>	

This functions writes $\langle tokens \rangle$ to the specified $\langle stream \rangle$ when the current page is finalised (*i.e.* at shipout). The x-type variants expand the $\langle tokens \rangle$ at the point where the function is used but *not* when the resulting tokens are written to the $\langle stream \rangle$ (*cf.* `\iow_shipout_x:Nn`).

T_EXhackers note: When using `expl3` with a format other than L^AT_EX, new line characters inserted using `\iow_newline:` or using the line-wrapping code `\iow_wrap:nnnN` are not recognized in the argument of `\iow_shipout:Nn`. This may lead to the insertion of additional unwanted line-breaks.

<code>\iow_shipout_x:Nn</code>	<code>\iow_shipout_x:Nn</code> $\langle stream \rangle$ $\{\langle tokens \rangle\}$
<code>\iow_shipout_x:(Nx cn cx)</code>	

Updated: 2012-09-08

This functions writes $\langle tokens \rangle$ to the specified $\langle stream \rangle$ when the current page is finalised (*i.e.* at shipout). The $\langle tokens \rangle$ are expanded at the time of writing in addition to any expansion when the function is used. This makes these functions suitable for including material finalised during the page building process (such as the page number integer).

T_EXhackers note: This is a wrapper around the T_EX primitive `\write`. When using `expl3` with a format other than L^AT_EX, new line characters inserted using `\iow_newline:` or using the line-wrapping code `\iow_wrap:nnnN` are not recognized in the argument of `\iow_shipout:Nn`. This may lead to the insertion of additional unwanted line-breaks.

`\iow_char:N` ★ `\iow_char:N \langle char \rangle`

Inserts $\langle char \rangle$ into the output stream. Useful when trying to write difficult characters such as %, {, }, *etc.* in messages, for example:

`\iow_now:Nx \g_my_iow { \iow_char:N \{ text \iow_char:N \} }`

The function has no effect if writing is taking place without expansion (*e.g.* in the second argument of `\iow_now:Nn`).

`\iow_newline:` ★ `\iow_newline:`

Function to add a new line within the $\langle tokens \rangle$ written to a file. The function has no effect if writing is taking place without expansion (*e.g.* in the second argument of `\iow_now:Nn`).

TeXhackers note: When using `expl3` with a format other than \LaTeX , the character inserted by `\iow_newline:` is not recognized by \TeX , which may lead to the insertion of additional unwanted line-breaks. This issue only affects `\iow_shipout:Nn`, `\iow_shipout_x:Nn` and direct uses of primitive operations.

12.1.3 Wrapping lines in output

`\iow_wrap:nnnN`
`\iow_wrap:nxnN`

New: 2012-06-28
Updated: 2017-12-04

`\iow_wrap:nnnN` $\langle text \rangle$ $\langle run-on text \rangle$ $\langle set up \rangle$ $\langle function \rangle$

This function wraps the $\langle text \rangle$ to a fixed number of characters per line. At the start of each line which is wrapped, the $\langle run-on text \rangle$ is inserted. The line character count targeted is the value of `\l_iow_line_count_int` minus the number of characters in the $\langle run-on text \rangle$ for all lines except the first, for which the target number of characters is simply `\l_iow_line_count_int` since there is no run-on text. The $\langle text \rangle$ and $\langle run-on text \rangle$ are exhaustively expanded by the function, with the following substitutions:

- `\` or `\iow_newline`: may be used to force a new line,
- `_` may be used to represent a forced space (for example after a control sequence),
- `\#`, `\%`, `\{`, `\}`, `\~` may be used to represent the corresponding character,
- `\iow_allow_break`: may be used to allow a line-break without inserting a space (this is experimental),
- `\iow_indent:n` may be used to indent a part of the $\langle text \rangle$ (not the $\langle run-on text \rangle$).

Additional functions may be added to the wrapping by using the $\langle set up \rangle$, which is executed before the wrapping takes place: this may include overriding the substitutions listed.

Any expandable material in the $\langle text \rangle$ which is not to be expanded on wrapping should be converted to a string using `\token_to_str:N`, `\tl_to_str:n`, `\tl_to_str:N`, *etc.*

The result of the wrapping operation is passed as a braced argument to the $\langle function \rangle$, which is typically a wrapper around a write operation. The output of `\iow_wrap:nnnN` (*i.e.* the argument passed to the $\langle function \rangle$) consists of characters of category “other” (category code 12), with the exception of spaces which have category “space” (category code 10). This means that the output does *not* expand further when written to a file.

T_EXhackers note: Internally, `\iow_wrap:nnnN` carries out an x-type expansion on the $\langle text \rangle$ to expand it. This is done in such a way that `\exp_not:N` or `\exp_not:n` *could* be used to prevent expansion of material. However, this is less conceptually clear than conversion to a string, which is therefore the supported method for handling expandable material in the $\langle text \rangle$.

`\iow_indent:n`

New: 2011-09-21

`\iow_indent:n` $\langle text \rangle$

In the first argument of `\iow_wrap:nnnN` (for instance in messages), indents $\langle text \rangle$ by four spaces. This function does not cause a line break, and only affects lines which start within the scope of the $\langle text \rangle$. In case the indented $\langle text \rangle$ should appear on separate lines from the surrounding text, use `\` to force line breaks.

`\l_iow_line_count_int`

New: 2012-06-24

The maximum number of characters in a line to be written by the `\iow_wrap:nnnN` function. This value depends on the T_EX system in use: the standard value is 78, which is typically correct for unmodified T_EX Live and MiK_T_EX systems.

12.1.4 Constant input–output streams, and variables

<code>\g_tmpa_iow</code> <code>\g_tmpb_iow</code>	Scratch input stream for global use. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
<hr/> <small>New: 2017-12-11</small>	

<code>\c_log_iow</code> <code>\c_term_iow</code>	Constant output streams for writing to the log and to the terminal (plus the log), respectively.
---	--

<code>\g_tmpa_iow</code> <code>\g_tmpb_iow</code>	Scratch output stream for global use. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
<hr/> <small>New: 2017-12-11</small>	

12.1.5 Primitive conditionals

<code>\if_eof:w</code> ★	<code>\if_eof:w <stream></code> <code><true code></code> <code>\else:</code> <code><false code></code> <code>\fi:</code>
	Tests if the <code><stream></code> returns “end of file”, which is true for non-existent files. The <code>\else:</code> branch is optional.

T_EXhackers note: This is the T_EX primitive `\ifeof`.

12.2 File operation functions

<code>\g_file_curr_dir_str</code> <code>\g_file_curr_name_str</code> <code>\g_file_curr_ext_str</code>	Contain the directory, name and extension of the current file. The directory is empty if the file was loaded without an explicit path (<i>i.e.</i> if it is in the T _E X search path), and does <i>not</i> end in / other than the case that it is exactly equal to the root directory. The <code><name></code> and <code><ext></code> parts together make up the file name, thus the <code><name></code> part may be thought of as the “job name” for the current file. Note that T _E X does not provide information on the <code><ext></code> part for the main (top level) file and that this file always has an empty <code><dir></code> component. Also, the <code><name></code> here will be equal to <code>\c_sys_jobname_str</code> , which may be different from the real file name (if set using <code>--jobname</code> , for example).
<hr/> <small>New: 2017-06-21</small>	

<hr/> <code>\l_file_search_path_seq</code> <hr/> New: 2017-06-18	<p>Each entry is the path to a directory which should be searched when seeking a file. Each path can be relative or absolute, and should not include the trailing slash. The entries are not expanded when used so may contain active characters but should not feature any variable content. Spaces need not be quoted.</p> <p>TeXhackers note: When working as a package in L^AT_EX 2_ε, <code>expl3</code> will automatically append the current <code>\input@path</code> to the set of values from <code>\l_file_search_path_seq</code>.</p>
<hr/> <code>\file_if_exist:nTF</code> <hr/> Updated: 2012-02-10	<code>\file_if_exist:nTF {<file name>} {<true code>} {<false code>}</code> <p>Searches for <code><file name></code> using the current T_EX search path and the additional paths controlled by <code>\l_file_search_path_seq</code>.</p>
<hr/> <code>\file_get:nnN</code> <code>\file_get:nnNTF</code> <hr/> New: 2019-01-16 Updated: 2019-02-16	<code>\file_get:nnN {<filename>} {<setup>} <tl></code> <code>\file_get:nnNTF {<filename>} {<setup>} <tl> {<true code>} {<false code>}</code> <p>Defines <code><tl></code> to the contents of <code><filename></code>. Category codes may need to be set appropriately via the <code><setup></code> argument. The non-branching version sets the <code><tl></code> to <code>\q_no_value</code> if the file is not found. The branching version runs the <code><true code></code> after the assignment to <code><tl></code> if the file is found, and <code><false code></code> otherwise.</p>
<hr/> <code>\file_get_full_name:nN</code> <code>\file_get_full_name:VN</code> <code>\file_get_full_name:nNTF</code> <code>\file_get_full_name:VNTF</code> <hr/> Updated: 2019-02-16	<code>\file_get_full_name:nN {<file name>} <tl></code> <code>\file_get_full_name:nNTF {<file name>} <tl> {<true code>} {<false code>}</code> <p>Searches for <code><file name></code> in the path as detailed for <code>\file_if_exist:nTF</code>, and if found sets the <code><tl var></code> the fully-qualified name of the file, <i>i.e.</i> the path and file name. This includes an extension <code>.tex</code> when the given <code><file name></code> has no extension but the file found has that extension. In the non-branching version, the <code><tl var></code> will be set to <code>\q_no_value</code> in the case that the file does not exist.</p>
<hr/> <code>\file_full_name:n</code> ☆ <code>\file_full_name:V</code> ☆ <hr/> New: 2019-09-03	<code>\file_full_name:n {<file name>}</code> <p>Searches for <code><file name></code> in the path as detailed for <code>\file_if_exist:nTF</code>, and if found leaves the fully-qualified name of the file, <i>i.e.</i> the path and file name, in the input stream. This includes an extension <code>.tex</code> when the given <code><file name></code> has no extension but the file found has that extension. If the file is not found on the path, the expansion is empty.</p>

<code>\file_parse_full_name:nNNN</code>
<code>\file_parse_full_name:VNNN</code>

New: 2017-06-23
Updated: 2020-06-24

`\file_parse_full_name:nNNN {<full name>} <dir> <name> <ext>`

Parses the `<full name>` and splits it into three parts, each of which is returned by setting the appropriate local string variable:

- The `<dir>`: everything up to the last / (path separator) in the `<file path>`. As with system PATH variables and related functions, the `<dir>` does *not* include the trailing / unless it points to the root directory. If there is no path (only a file name), `<dir>` is empty.
- The `<name>`: everything after the last / up to the last ., where both of those characters are optional. The `<name>` may contain multiple . characters. It is empty if `<full name>` consists only of a directory name.
- The `<ext>`: everything after the last . (including the dot). The `<ext>` is empty if there is no . after the last /.

Before parsing, the `<full name>` is expanded until only non-expandable tokens remain, except that active characters are also not expanded. Quotes (") are invalid in file names and are discarded from the input.

<code>\file_parse_full_name:n *</code>
New: 2020-06-24

`\file_parse_full_name:n {<full name>}`

Parses the `<full name>` as described for `\file_parse_full_name:nNNN`, and leaves `<dir>`, `<name>`, and `<ext>` in the input stream, each inside a pair of braces.

<code>\file_parse_full_name_apply:nN *</code>	<code>\file_parse_full_name_apply:nN {<full name>} <function></code>
New: 2020-06-24	

Parses the `<full name>` as described for `\file_parse_full_name:nNNN`, and passes `<dir>`, `<name>`, and `<ext>` as arguments to `<function>`, as an n-type argument each, in this order.

<code>\file_hex_dump:n</code>	☆
<code>\file_hex_dump:nnn</code>	☆
New: 2019-11-19	

`\file_hex_dump:n {<file name>}`
`\file_hex_dump:nnn {<file name>} {<start index>} {<end index>}`

Searches for `<file name>` using the current T_EX search path and the additional paths controlled by `\l_file_search_path_seq`. It then expands to leave the hexadecimal dump of the file content in the input stream. The file is read as bytes, which means that in contrast to most T_EX behaviour there will be a difference in result depending on the line endings used in text files. The same file will produce the same result between different engines: the algorithm used is the same in all cases. When the file is not found, the result of expansion is empty. The `{<start index>}` and `{<end index>}` values work as described for `\str_range:nnn`.

<code>\file_get_hex_dump:nN</code>
<code>\file_get_hex_dump:nNTF</code>
<code>\file_get_hex_dump:nnnN</code>
<code>\file_get_hex_dump:nnnNTF</code>
New: 2019-11-19

`\file_get_hex_dump:nN {<file name>} <tl var>`
`\file_get_hex_dump:nnnN {<file name>} {<start index>} {<end index>} <tl var>`

Sets the `<tl var>` to the result of applying `\file_hex_dump:n/\file_hex_dump:nnn` to the `<file>`. If the file is not found, the `<tl var>` will be set to `\q_no_value`.

<hr/> <code>\file_md5ive_hash:n</code> ☆	<code>\file_md5ive_hash:n {⟨file name⟩}</code>
<hr/> New: 2019-09-03	<p>Searches for $\langle file\ name \rangle$ using the current \TeX search path and the additional paths controlled by <code>\l_file_search_path_seq</code>. It then expands to leave the MD5 sum generated from the contents of the file in the input stream. The file is read as bytes, which means that in contrast to most \TeX behaviour there will be a difference in result depending on the line endings used in text files. The same file will produce the same result between different engines: the algorithm used is the same in all cases. When the file is not found, the result of expansion is empty.</p>
<hr/> <code>\file_get_md5ive_hash:nN</code> <code>\file_get_md5ive_hash:nNTF</code>	<code>\file_get_md5ive_hash:nN {⟨file name⟩} ⟨tl var⟩</code>
<hr/> New: 2017-07-11 Updated: 2019-02-16	<p>Sets the $\langle tl\ var \rangle$ to the result of applying <code>\file_md5ive_hash:n</code> to the $\langle file \rangle$. If the file is not found, the $\langle tl\ var \rangle$ will be set to <code>\q_no_value</code>.</p>
<hr/> <code>\file_size:n</code> ☆	<code>\file_size:n {⟨file name⟩}</code>
<hr/> New: 2019-09-03	<p>Searches for $\langle file\ name \rangle$ using the current \TeX search path and the additional paths controlled by <code>\l_file_search_path_seq</code>. It then expands to leave the size of the file in bytes in the input stream. When the file is not found, the result of expansion is empty.</p>
<hr/> <code>\file_get_size:nN</code> <code>\file_get_size:nNTF</code>	<code>\file_get_size:nN {⟨file name⟩} ⟨tl var⟩</code>
<hr/> New: 2017-07-09 Updated: 2019-02-16	<p>Sets the $\langle tl\ var \rangle$ to the result of applying <code>\file_size:n</code> to the $\langle file \rangle$. If the file is not found, the $\langle tl\ var \rangle$ will be set to <code>\q_no_value</code>. This is not available in older versions of \LaTeX.</p>
<hr/> <code>\file_timestamp:n</code> ☆	<code>\file_timestamp:n {⟨file name⟩}</code>
<hr/> New: 2019-09-03	<p>Searches for $\langle file\ name \rangle$ using the current \TeX search path and the additional paths controlled by <code>\l_file_search_path_seq</code>. It then expands to leave the modification timestamp of the file in the input stream. The timestamp is of the form $D:\langle year \rangle \langle month \rangle \langle day \rangle \langle hour \rangle \langle minute \rangle \langle second \rangle \langle offset \rangle$, where the latter may be Z (UTC) or $\langle plus-minus \rangle \langle hours \rangle' \langle minutes \rangle'$. When the file is not found, the result of expansion is empty. This is not available in older versions of \LaTeX.</p>
<hr/> <code>\file_get_timestamp:nN</code> <code>\file_get_timestamp:nNTF</code>	<code>\file_get_timestamp:nN {⟨file name⟩} ⟨tl var⟩</code>
<hr/> New: 2017-07-09 Updated: 2019-02-16	<p>Sets the $\langle tl\ var \rangle$ to the result of applying <code>\file_timestamp:n</code> to the $\langle file \rangle$. If the file is not found, the $\langle tl\ var \rangle$ will be set to <code>\q_no_value</code>. This is not available in older versions of \LaTeX.</p>

<code>\file_compare_timestamp_p:nNn</code>	<code>★</code>	<code>\file_compare_timestamp:nNn {<file-1>} <comparator> {<file-2>} {<true</code>
<code>\file_compare_timestamp:nNnTF</code>	<code>★</code>	<code>code)} {<false code>}</code>

New: 2019-05-13

Updated: 2019-09-20

Compares the file stamps on the two *<files>* as indicated by the *<comparator>*, and inserts either the *<true code>* or *<false case>* as required. A file which is not found is treated as older than any file which is found. This allows for example the construct

```
\file_compare_timestamp:nNnT { source-file } > { derived-file }
{
  % Code to regenerate derived file
}
```

to work when the derived file is entirely absent. The timestamp of two absent files is regarded as different. This is not available in older versions of X_YTeX.

<code>\file_input:n</code>	<code>\file_input:n {<file name>}</code>
----------------------------	--

Updated: 2017-06-26

Searches for *<file name>* in the path as detailed for `\file_if_exist:nTF`, and if found reads in the file as additional L^AT_EX source. All files read are recorded for information and the file name stack is updated by this function. An error is raised if the file is not found.

<code>\file_if_exist_input:n</code>	<code>\file_if_exist_input:n {<file name>}</code>
<code>\file_if_exist_input:nF</code>	<code>\file_if_exist_input:nF {<file name>} {<false code>}</code>

New: 2014-07-02

Searches for *<file name>* using the current T_EX search path and the additional paths included in `\l_file_search_path_seq`. If found then reads in the file as additional L^AT_EX source as described for `\file_input:n`, otherwise inserts the *<false code>*. Note that these functions do not raise an error if the file is not found, in contrast to `\file_input:n`.

<code>\file_input_stop:</code>	<code>\file_input_stop:</code>
--------------------------------	--------------------------------

New: 2017-07-07

Ends the reading of a file started by `\file_input:n` or similar before the end of the file is reached. Where the file reading is being terminated due to an error, `\msg_critical:nn(nn)` should be preferred.

T_EXhackers note: This function must be used on a line on its own: T_EX reads files line-by-line and so any additional tokens in the “current” line will still be read.

This is also true if the function is hidden inside another function (which will be the normal case), i.e., all tokens on the same line in the source file are still processed. Putting it on a line by itself in the definition doesn’t help as it is the line where it is used that counts!

<code>\file_show_list:</code>	<code>\file_show_list:</code>
<code>\file_log_list:</code>	<code>\file_log_list:</code>

These functions list all files loaded by L^AT_EX 2_ε commands that populate `\@filelist` or by `\file_input:n`. While `\file_show_list:` displays the list in the terminal, `\file_log_list:` outputs it to the log file only.

Chapter 13

The `l3luatex` package: Lua_{TeX}-specific functions

The Lua_{TeX} engine provides access to the Lua programming language, and with it access to the “internals” of _{TeX}. In order to use this within the framework provided here, a family of functions is available. When used with pdf_{TeX}, p_{TeX}, up_{TeX} or X_{TeX} these raise an error: use `\sys_if_engine luatex:T` to avoid this. Details on using Lua with the Lua_{TeX} engine are given in the Lua_{TeX} manual.

13.1 Breaking out to Lua

`\lua_now:n ★`
`\lua_now:e ★`

New: 2018-06-18

`\lua_now:n` $\{ \langle token\ list \rangle \}$

The $\langle token\ list \rangle$ is first tokenized by _{TeX}, which includes converting line ends to spaces in the usual _{TeX} manner and which respects currently-applicable _{TeX} category codes. The resulting $\langle Lua\ input \rangle$ is passed to the Lua interpreter for processing. Each `\lua_now:n` block is treated by Lua as a separate chunk. The Lua interpreter executes the $\langle Lua\ input \rangle$ immediately, and in an expandable manner.

_{TeX}hackers note: `\lua_now:e` is a macro wrapper around `\directlua`: when Lua_{TeX} is in use two expansions are required to yield the result of the Lua code.

`\lua_shipout_e:n`
`\lua_shipout:n`

New: 2018-06-18

`\lua_shipout:n` $\{ \langle token\ list \rangle \}$

The $\langle token\ list \rangle$ is first tokenized by _{TeX}, which includes converting line ends to spaces in the usual _{TeX} manner and which respects currently-applicable _{TeX} category codes. The resulting $\langle Lua\ input \rangle$ is passed to the Lua interpreter when the current page is finalised (*i.e.* at shipout). Each `\lua_shipout:n` block is treated by Lua as a separate chunk. The Lua interpreter will execute the $\langle Lua\ input \rangle$ during the page-building routine: no _{TeX} expansion of the $\langle Lua\ input \rangle$ will occur at this stage.

In the case of the `\lua_shipout_e:n` version the input is fully expanded by _{TeX} in an e-type manner during the shipout operation.

_{TeX}hackers note: At a _{TeX} level, the $\langle Lua\ input \rangle$ is stored as a “whatsit”.

<code>\lua_escape:n</code>	★	<code>\lua_escape:n {⟨token list⟩}</code>
----------------------------	---	---

<code>\lua_escape:e</code>	★
----------------------------	---

New: 2015-06-29

Converts the $\langle token list \rangle$ such that it can safely be passed to Lua: embedded backslashes, double and single quotes, and newlines and carriage returns are escaped. This is done by prepending an extra token consisting of a backslash with category code 12, and for the line endings, converting them to $\backslash n$ and $\backslash r$, respectively.

TeXhackers note: `\lua_escape:e` is a macro wrapper around `\luaescapestring`: when LuaTeX is in use two expansions are required to yield the result of the Lua code.

13.2 Lua interfaces

As well as interfaces for TeX, there are a small number of Lua functions provided here.

<code>ltx.utils</code>

Most public interfaces provided by the module are stored within the `ltx.utils` table.

<code>ltx.utils.filedump</code>

$\langle dump \rangle = \text{ltx.utils.filedump}(\langle file \rangle, \langle offset \rangle, \langle length \rangle)$

Returns the uppercase hexadecimal representation of the content of the $\langle file \rangle$ read as bytes. If the $\langle length \rangle$ is given, only this part of the file is returned; similarly, one may specify the $\langle offset \rangle$ from the start of the file. If the $\langle length \rangle$ is not given, the entire file is read starting at the $\langle offset \rangle$.

<code>ltx.utils.filemd5sum</code>

$\langle hash \rangle = \text{ltx.utils.filemd5sum}(\langle file \rangle)$

Returns the MD5 sum of the file contents read as bytes; note that the result will depend on the nature of the line endings used in the file, in contrast to normal TeX behaviour. If the $\langle file \rangle$ is not found, nothing is returned with *no error raised*.

<code>ltx.utils.filemoddate</code>

$\langle date \rangle = \text{ltx.utils.filemoddate}(\langle file \rangle)$

Returns the date/time of last modification of the $\langle file \rangle$ in the format

$D:\langle year \rangle \langle month \rangle \langle day \rangle \langle hour \rangle \langle minute \rangle \langle second \rangle \langle offset \rangle$

where the latter may be Z (UTC) or $\langle plus-minus \rangle \langle hours \rangle' \langle minutes \rangle'$. If the $\langle file \rangle$ is not found, nothing is returned with *no error raised*.

<code>ltx.utils.filesize</code>

$size = \text{ltx.utils.filesize}(\langle file \rangle)$

Returns the size of the $\langle file \rangle$ in bytes. If the $\langle file \rangle$ is not found, nothing is returned with *no error raised*.

Chapter 14

The l3legacy package

Interfaces to legacy concepts

There are a small number of T_EX or L^AT_EX 2_ε concepts which are not used in expl3 code but which need to be manipulated when working as a L^AT_EX 2_ε package. To allow these to be integrated cleanly into expl3 code, a set of legacy interfaces are provided here.

<code>\legacy_if_p:n</code> *	<code>\legacy_if:nTF</code> { $\langle name \rangle$ } { $\langle true\ code \rangle$ } { $\langle false\ code \rangle$ }
-------------------------------	---

<code>\legacy_if:nTF</code> *	Tests if the L ^A T _E X 2 _ε /plain T _E X conditional (generated by <code>\newif</code>) if <code>true</code> or <code>false</code> and branches accordingly. The $\langle name \rangle$ of the conditional should <i>omit</i> the leading <code>if</code> .
-------------------------------	---

<code>\legacy_if_set_true:n</code>	<code>\legacy_if_set_true:n</code> { $\langle name \rangle$ }
<code>\legacy_if_set_false:n</code>	<code>\legacy_if_set_false:n</code> { $\langle name \rangle$ }

<code>\legacy_if_gset_true:n</code>	Sets the L ^A T _E X 2 _ε /plain T _E X conditional <code>\if</code> $\langle name \rangle$ (generated by <code>\newif</code>) to be <code>true</code> or <code>false</code> .
<code>\legacy_if_gset_false:n</code>	

New: 2021-05-10

<code>\legacy_if_set:nn</code>	<code>\legacy_if_set:nn</code> { $\langle name \rangle$ } { $\langle boolexpr \rangle$ }
--------------------------------	--

<code>\legacy_if_gset:nn</code>	Sets the L ^A T _E X 2 _ε /plain T _E X conditional <code>\if</code> $\langle name \rangle$ (generated by <code>\newif</code>) to the result of evaluating the $\langle boolean\ expression \rangle$.
---------------------------------	---

New: 2021-05-10

Part IV

Data types

Chapter 15

The `l3tl` package

Token lists

\TeX works with tokens, and \LaTeX3 therefore provides a number of functions to deal with lists of tokens. Token lists may be present directly in the argument to a function:

```
\foo:n { a collection of \tokens }
```

or may be stored in a so-called “token list variable”, which have the suffix `tl`: a token list variable can also be used as the argument to a function, for example

```
\foo:N \l_some_tl
```

In both cases, functions are available to test and manipulate the lists of tokens, and these have the module prefix `tl`. In many cases, functions which can be applied to token list variables are paired with similar functions for application to explicit lists of tokens: the two “views” of a token list are therefore collected together here.

A token list (explicit, or stored in a variable) can be seen either as a list of “items”, or a list of “tokens”. An item is whatever `\use:n` would grab as its argument: a single non-space token or a brace group, with optional leading explicit space characters (each item is thus itself a token list). A token is either a normal `N` argument, or `\`, `{`, or `}` (assuming normal \TeX category codes). Thus for example

```
{ Hello } ~ world
```

contains six items (`Hello`, `w`, `o`, `r`, `l` and `d`), but thirteen tokens (`{`, `H`, `e`, `l`, `l`, `o`, `}`, `\`, `w`, `o`, `r`, `l` and `d`). Functions which act on items are often faster than their analogue acting directly on tokens.

15.1 Creating and initialising token list variables

```
\tl_new:N \tl_new:N <tl var>
```

```
\tl_new:c
```

Creates a new `<tl var>` or raises an error if the name is already taken. The declaration is global. The `<tl var>` is initially empty.

<hr/>	
<code>\tl_const:Nn</code>	<code>\tl_const:Nn <tl var> {<token list>}</code>
<code>\tl_const:(Nx cn cx)</code>	Creates a new constant <code><tl var></code> or raises an error if the name is already taken. The value of the <code><tl var></code> is set globally to the <code><token list></code> .
<hr/>	
<code>\tl_clear:N</code>	<code>\tl_clear:N <tl var></code>
<code>\tl_clear:c</code>	
<code>\tl_gclear:N</code>	Clears all entries from the <code><tl var></code> .
<code>\tl_gclear:c</code>	
<hr/>	
<code>\tl_clear_new:N</code>	<code>\tl_clear_new:N <tl var></code>
<code>\tl_clear_new:c</code>	
<code>\tl_gclear_new:N</code>	Ensures that the <code><tl var></code> exists globally by applying <code>\tl_new:N</code> if necessary, then applies <code>\tl_(g)clear:N</code> to leave the <code><tl var></code> empty.
<code>\tl_gclear_new:c</code>	
<hr/>	
<code>\tl_set_eq:NN</code>	<code>\tl_set_eq:NN <tl var₁₂</code>
<code>\tl_set_eq:(cN Nc cc)</code>	Sets the content of <code><tl var_{1 equal to that of <code><tl var_{2.}</code>}</code>
<code>\tl_gset_eq:NN</code>	
<code>\tl_gset_eq:(cN Nc cc)</code>	
<hr/>	
<code>\tl_concat:NNN</code>	<code>\tl_concat:NNN <tl var₁₂₃</code>
<code>\tl_concat:ccc</code>	Concatenates the content of <code><tl var_{2 and <code><tl var_{3 together and saves the result in <code><tl var_{1. The <code><tl var_{2 is placed at the left side of the new token list.}</code>}</code>}</code>}</code>
<code>\tl_gconcat:NNN</code>	
<code>\tl_gconcat:ccc</code>	
<hr/>	
	New: 2012-05-18
<hr/>	
<code>\tl_if_exist_p:N *</code>	<code>\tl_if_exist_p:N <tl var></code>
<code>\tl_if_exist_p:c *</code>	<code>\tl_if_exist:NTF <tl var> {<true code>} {<false code>}</code>
<code>\tl_if_exist:N\overline{TF} *</code>	
<code>\tl_if_exist:c\overline{TF} *</code>	Tests whether the <code><tl var></code> is currently defined. This does not check that the <code><tl var></code> really is a token list variable.
<hr/>	
	New: 2012-03-03

15.2 Adding data to token list variables

<hr/>	
<code>\tl_set:Nn</code>	<code>\tl_set:Nn <tl var> {<tokens>}</code>
<code>\tl_set:(NV Nv No Nf Nx cn cV cv co cf cx)</code>	
<code>\tl_gset:Nn</code>	
<code>\tl_gset:(NV Nv No Nf Nx cn cV cv co cf cx)</code>	
<hr/>	
	Sets <code><tl var></code> to contain <code><tokens></code> , removing any previous content from the variable.
<hr/>	
<code>\tl_put_left:Nn</code>	<code>\tl_put_left:Nn <tl var> {<tokens>}</code>
<code>\tl_put_left:(NV No Nx cn cV co cx)</code>	
<code>\tl_gput_left:Nn</code>	
<code>\tl_gput_left:(NV No Nx cn cV co cx)</code>	
<hr/>	
	Appends <code><tokens></code> to the left side of the current content of <code><tl var></code> .

<code>\tl_put_right:Nn</code>	<code>\tl_put_right:Nn <tl var> {<tokens>}</code>
<code>\tl_put_right:(NV No Nx cn cV co cx)</code>	
<code>\tl_gput_right:Nn</code>	
<code>\tl_gput_right:(NV No Nx cn cV co cx)</code>	

Appends *<tokens>* to the right side of the current content of *<tl var>*.

15.3 Token list conditionals

<code>\tl_if_blank_p:n</code>	<code>\tl_if_blank_p:n {<token list>}</code>
<code>\tl_if_blank_p:(e V o)</code>	<code>\tl_if_blank:nTF {<token list>} {<true code>} {<false code>}</code>
<code>\tl_if_blank:nTF</code>	
<code>\tl_if_blank:(e V o)TF</code>	

Tests if the *<token list>* consists only of blank spaces (*i.e.* contains no item). The test is **true** if *<token list>* is zero or more explicit space characters (explicit tokens with character code 32 and category code 10), and is **false** otherwise.

Updated: 2019-09-04

<code>\tl_if_empty_p:N</code>	<code>\tl_if_empty_p:N <tl var></code>
<code>\tl_if_empty_p:c</code>	<code>\tl_if_empty:nTF <tl var> {<true code>} {<false code>}</code>
<code>\tl_if_empty:nTF</code>	
<code>\tl_if_empty:cTF</code>	

Tests if the *<token list variable>* is entirely empty (*i.e.* contains no tokens at all).

<code>\tl_if_empty_p:n</code>	<code>\tl_if_empty_p:n {<token list>}</code>
<code>\tl_if_empty_p:(V o)</code>	<code>\tl_if_empty:nTF {<token list>} {<true code>} {<false code>}</code>
<code>\tl_if_empty:nTF</code>	
<code>\tl_if_empty:(V o)TF</code>	

Tests if the *<token list>* is entirely empty (*i.e.* contains no tokens at all).

New: 2012-05-24
Updated: 2012-06-05

<code>\tl_if_eq_p:NN</code>	<code>\tl_if_eq_p:NN <tl var₁₂</code>
<code>\tl_if_eq_p:(Nc cN cc)</code>	<code>\tl_if_eq:NNTF <tl var₁₂</code>
<code>\tl_if_eq:NNTF</code>	
<code>\tl_if_eq:(Nc cN cc)TF</code>	

Compares the content of two *<token list variables>* and is logically **true** if the two contain the same list of tokens (*i.e.* identical in both the list of characters they contain and the category codes of those characters). Thus for example

```
\tl_set:Nn \l_tmpa_tl { abc }
\tl_set:Nx \l_tmpb_tl { \tl_to_str:n { abc } }
\tl_if_eq:NNTF \l_tmpa_tl \l_tmpb_tl { true } { false }
```

yields **false**. See also `\str_if_eq:nnTF` for a comparison that ignores category codes.

<code>\tl_if_eq:NnTF</code>	<code>\tl_if_eq:NnTF <tl var₁₂</code>
<code>\tl_if_eq:cnTF</code>	

Tests if the *<token list variable_{1 and the *<token list_{2 contain the same list of tokens, both in respect of character codes and category codes. This conditional is not expandable: see `\tl_if_eq:NNTF` for an expandable version when both token lists are stored in variables, or `\str_if_eq:nnTF` if category codes are not important.}*}*

New: 2020-07-14

<code>\tl_if_eq:nnTF</code>	<code>\tl_if_eq:nnTF {<token list₁>} {<token list₂>} {<true code>} {<false code>}</code>
-----------------------------	--

Tests if $\langle token list_1 \rangle$ and $\langle token list_2 \rangle$ contain the same list of tokens, both in respect of character codes and category codes. This conditional is not expandable: see `\tl_if_eq:NNTF` for an expandable version when token lists are stored in variables, or `\str_if_eq:nnTF` if category codes are not important.

<code>\tl_if_in:NnTF</code>	<code>\tl_if_in:NnTF <tl var> {<token list>} {<true code>} {<false code>}</code>
<code>\tl_if_in:cnTF</code>	

Tests if the $\langle token list \rangle$ is found in the content of the $\langle tl var \rangle$. The $\langle token list \rangle$ cannot contain the tokens `{`, `}` or `#` (more precisely, explicit character tokens with category code 1 (begin-group) or 2 (end-group), and tokens with category code 6).

<code>\tl_if_in:nnTF</code>	<code>\tl_if_in:nnTF {<token list₁>} {<token list₂>} {<true code>} {<false code>}</code>
<code>\tl_if_in:(Vn on no)TF</code>	

Tests if $\langle token list_2 \rangle$ is found inside $\langle token list_1 \rangle$. The $\langle token list_2 \rangle$ cannot contain the tokens `{`, `}` or `#` (more precisely, explicit character tokens with category code 1 (begin-group) or 2 (end-group), and tokens with category code 6). The search does *not* enter brace (category code 1/2) groups.

<code>\tl_if_novalue_p:n *</code>	<code>\tl_if_novalue_p:n {<token list>}</code>
<code>\tl_if_novalue:nTF *</code>	<code>\tl_if_novalue:nTF {<token list>} {<true code>} {<false code>}</code>

New: 2017-11-14

Tests if the $\langle token list \rangle$ is exactly equal to the special `\c_novalue_tl` marker. This function is intended to allow construction of flexible document interface structures in which missing optional arguments are detected.

<code>\tl_if_single_p:N *</code>	<code>\tl_if_single_p:N <tl var></code>
<code>\tl_if_single_p:c *</code>	<code>\tl_if_single:NNTF <tl var> {<true code>} {<false code>}</code>
<code>\tl_if_single:NNTF *</code>	
<code>\tl_if_single:cNTF *</code>	

Updated: 2011-08-13

Tests if the content of the $\langle tl var \rangle$ consists of a single $\langle item \rangle$, *i.e.* is a single normal token (neither an explicit space character nor a begin-group character) or a single brace group, surrounded by optional spaces on both sides. In other words, such a token list has token count 1 according to `\tl_count:N`.

<code>\tl_if_single_p:n *</code>	<code>\tl_if_single_p:n {<token list>}</code>
<code>\tl_if_single:nNTF *</code>	<code>\tl_if_single:nNTF {<token list>} {<true code>} {<false code>}</code>

Updated: 2011-08-13

Tests if the $\langle token list \rangle$ has exactly one $\langle item \rangle$, *i.e.* is a single normal token (neither an explicit space character nor a begin-group character) or a single brace group, surrounded by optional spaces on both sides. In other words, such a token list has token count 1 according to `\tl_count:n`.

<code>\tl_if_single_token_p:n *</code>	<code>\tl_if_single_token_p:n {<token list>}</code>
<code>\tl_if_single_token:nTF *</code>	<code>\tl_if_single_token:nTF {<token list>} {<true code>} {<false code>}</code>

Tests if the token list consists of exactly one token, *i.e.* is either a single space character or a single normal token. Token groups (`{...}`) are not single tokens.

<code>\tl_case:Nn</code>	<code>*</code>	<code>\tl_case:NnTF</code>	<code><test token list variable></code>
<code>\tl_case:cn</code>	<code>*</code>	<code>{</code>	
<code>\tl_case:NnTF</code>	<code>*</code>		<code><token list variable case₁> {<code case₁>}</code>
<code>\tl_case:cnTF</code>	<code>*</code>		<code><token list variable case₂> {<code case₂>}</code>
			<code>...</code>
			<code><token list variable case_n> {<code case_n>}</code>
		<code>}</code>	
			<code>{<true code>}</code>
			<code>{<false code>}</code>

New: 2013-07-24

This function compares the `<test token list variable>` in turn with each of the `<token list variable cases>`. If the two are equal (as described for `\tl_if_eq:NNTF`) then the associated `<code>` is left in the input stream and other cases are discarded. If any of the cases are matched, the `<true code>` is also inserted into the input stream (after the code for the appropriate case), while if none match then the `<false code>` is inserted. The function `\tl_case:Nn`, which does nothing if there is no match, is also available.

15.3.1 Testing the first token

<code>\tl_if_head_eq_catcode_p:nN</code>	<code>*</code>	<code>\tl_if_head_eq_catcode_p:nN</code>	<code>{<token list>}</code>	<code><test token></code>
<code>\tl_if_head_eq_catcode_p:oN</code>	<code>*</code>	<code>\tl_if_head_eq_catcode:nNTF</code>	<code>{<token list>}</code>	<code><test token></code>
<code>\tl_if_head_eq_catcode:nNTF</code>	<code>*</code>		<code>{<true code>}</code>	<code>{<false code>}</code>
<code>\tl_if_head_eq_catcode:oNTF</code>	<code>*</code>			

Updated: 2012-07-09

Tests if the first `<token>` in the `<token list>` has the same category code as the `<test token>`. In the case where the `<token list>` is empty, the test is always **false**.

<code>\tl_if_head_eq_charcode_p:nN</code>	<code>*</code>	<code>\tl_if_head_eq_charcode_p:nN</code>	<code>{<token list>}</code>	<code><test token></code>
<code>\tl_if_head_eq_charcode_p:fN</code>	<code>*</code>	<code>\tl_if_head_eq_charcode:nNTF</code>	<code>{<token list>}</code>	<code><test token></code>
<code>\tl_if_head_eq_charcode:nNTF</code>	<code>*</code>		<code>{<true code>}</code>	<code>{<false code>}</code>
<code>\tl_if_head_eq_charcode:fNTF</code>	<code>*</code>			

Updated: 2012-07-09

Tests if the first `<token>` in the `<token list>` has the same character code as the `<test token>`. In the case where the `<token list>` is empty, the test is always **false**.

<code>\tl_if_head_eq_meaning_p:nN</code>	<code>*</code>	<code>\tl_if_head_eq_meaning_p:nN</code>	<code>{<token list>}</code>	<code><test token></code>
<code>\tl_if_head_eq_meaning:nNTF</code>	<code>*</code>	<code>\tl_if_head_eq_meaning:nNTF</code>	<code>{<token list>}</code>	<code><test token></code>
			<code>{<true code>}</code>	<code>{<false code>}</code>

Updated: 2012-07-09

Tests if the first `<token>` in the `<token list>` has the same meaning as the `<test token>`. In the case where `<token list>` is empty, the test is always **false**.

<code>\tl_if_head_is_group_p:n</code>	<code>*</code>	<code>\tl_if_head_is_group_p:n</code>	<code>{<token list>}</code>
<code>\tl_if_head_is_group:nTF</code>	<code>*</code>	<code>\tl_if_head_is_group:nTF</code>	<code>{<token list>}</code>
			<code>{<true code>}</code>
			<code>{<false code>}</code>

New: 2012-07-08

Tests if the first `<token>` in the `<token list>` is an explicit begin-group character (with category code 1 and any character code), in other words, if the `<token list>` starts with a brace group. In particular, the test is **false** if the `<token list>` starts with an implicit token such as `\c_group_begin_token`, or if it is empty. This function is useful to implement actions on token lists on a token by token basis.

<code>\tl_if_head_is_N_type_p:n</code>	★	<code>\tl_if_head_is_N_type_p:n {⟨token list⟩}</code>
<code>\tl_if_head_is_N_type:nTF</code>	★	<code>\tl_if_head_is_N_type:nTF {⟨token list⟩} {⟨true code⟩} {⟨false code⟩}</code>

New: 2012-07-08

Tests if the first *⟨token⟩* in the *⟨token list⟩* is a normal N-type argument. In other words, it is neither an explicit space character (explicit token with character code 32 and category code 10) nor an explicit begin-group character (with category code 1 and any character code). An empty argument yields **false**, as it does not have a normal first token. This function is useful to implement actions on token lists on a token by token basis.

<code>\tl_if_head_is_space_p:n</code>	★	<code>\tl_if_head_is_space_p:n {⟨token list⟩}</code>
<code>\tl_if_head_is_space:nTF</code>	★	<code>\tl_if_head_is_space:nTF {⟨token list⟩} {⟨true code⟩} {⟨false code⟩}</code>

Updated: 2012-07-08

Tests if the first *⟨token⟩* in the *⟨token list⟩* is an explicit space character (explicit token with character code 12 and category code 10). In particular, the test is **false** if the *⟨token list⟩* starts with an implicit token such as `\c_space_token`, or if it is empty. This function is useful to implement actions on token lists on a token by token basis.

15.4 Working with token lists as a whole

15.4.1 Using token lists

<code>\tl_to_str:n</code>	★	<code>\tl_to_str:n {⟨token list⟩}</code>
<code>\tl_to_str:V</code>	★	

Converts the *⟨token list⟩* to a *⟨string⟩*, leaving the resulting character tokens in the input stream. A *⟨string⟩* is a series of tokens with category code 12 (other) with the exception of spaces, which retain category code 10 (space). This function requires only a single expansion. Its argument *must* be braced.

TeXhackers note: This is the ε -TeX primitive `\detokenize`. Converting a *⟨token list⟩* to a *⟨string⟩* yields a concatenation of the string representations of every token in the *⟨token list⟩*. The string representation of a control sequence is

- an escape character, whose character code is given by the internal parameter `\escapechar`, absent if the `\escapechar` is negative or greater than the largest character code;
- the control sequence name, as defined by `\cs_to_str:N`;
- a space, unless the control sequence name is a single character whose category at the time of expansion of `\tl_to_str:n` is not “letter”.

The string representation of an explicit character token is that character, doubled in the case of (explicit) macro parameter characters (normally `#`). In particular, the string representation of a token list may depend on the category codes in effect when it is evaluated, and the value of the `\escapechar`: for instance `\tl_to_str:n {\a}` normally produces the three character “backslash”, “lower-case a”, “space”, but it may also produce a single “lower-case a” if the escape character is negative and `a` is currently not a letter.

<code>\tl_to_str:N</code>	★	<code>\tl_to_str:N ⟨tl var⟩</code>
<code>\tl_to_str:c</code>	★	

Converts the content of the *⟨tl var⟩* into a series of characters with category code 12 (other) with the exception of spaces, which retain category code 10 (space). This *⟨string⟩* is then left in the input stream. For low-level details, see the notes given for `\tl_to_str:n`.

<code>\tl_use:N</code>	★	<code>\tl_use:N <tl var></code>
------------------------	---	---------------------------------------

<code>\tl_use:c</code>	★
------------------------	---

Recovers the content of a *<tl var>* and places it directly in the input stream. An error is raised if the variable does not exist or if it is invalid. Note that it is possible to use a *<tl var>* directly without an accessor function.

15.4.2 Counting and reversing token lists

<code>\tl_count:n</code>	★	<code>\tl_count:n {(tokens)}</code>
--------------------------	---	-------------------------------------

<code>\tl_count:(V o)</code>	★
------------------------------	---

New: 2012-05-13

Counts the number of *<items>* in *<tokens>* and leaves this information in the input stream. Unbraced tokens count as one element as do each token group *{...}*. This process ignores any unprotected spaces within *<tokens>*. See also `\tl_count:N`. This function requires three expansions, giving an *<integer denotation>*.

<code>\tl_count:N</code>	★	<code>\tl_count:N <tl var></code>
--------------------------	---	---

<code>\tl_count:c</code>	★
--------------------------	---

New: 2012-05-13

Counts the number of *<items>* in the *<tl var>* and leaves this information in the input stream. Unbraced tokens count as one element as do each token group *{...}*. This process ignores any unprotected spaces within the *<tl var>*. See also `\tl_count:n`. This function requires three expansions, giving an *<integer denotation>*.

<code>\tl_count_tokens:n</code>	★	<code>\tl_count_tokens:n {(tokens)}</code>
---------------------------------	---	--

New: 2019-02-25

Counts the number of \TeX tokens in the *<tokens>* and leaves this information in the input stream. Every token, including spaces and braces, contributes one to the total; thus for instance, the token count of `a~{bc}` is 6.

<code>\tl_reverse:n</code>	★	<code>\tl_reverse:n {(token list)}</code>
----------------------------	---	---

<code>\tl_reverse:(V o)</code>	★
--------------------------------	---

Updated: 2012-01-08

Reverses the order of the *<items>* in the *<token list>*, so that *<item₁><item₂><item₃>...<item_n>* becomes *<item_n>...<item₃><item₂><item₁>*. This process preserves unprotected space within the *<token list>*. Tokens are not reversed within braced token groups, which keep their outer set of braces. In situations where performance is important, consider `\tl_reverse_items:n`. See also `\tl_reverse:N`.

\TeX hackers note: The result is returned within `\unexpanded`, which means that the token list does not expand further when appearing in an *x*-type or *e*-type argument expansion.

<code>\tl_reverse:N</code>		<code>\tl_reverse:N <tl var></code>
----------------------------	--	---

<code>\tl_reverse:c</code>	
----------------------------	--

<code>\tl_greverse:N</code>	
-----------------------------	--

<code>\tl_greverse:c</code>	
-----------------------------	--

Updated: 2012-01-08

Sets the *<tl var>* to contain the result of reversing the order of its *<items>*, so that *<item₁><item₂><item₃>...<item_n>* becomes *<item_n>...<item₃><item₂><item₁>*. This process preserves unprotected spaces within the *<token list variable>*. Braced token groups are copied without reversing the order of tokens, but keep the outer set of braces. This is equivalent to a combination of an assignment and `\tl_reverse:V`. See also `\tl_reverse_items:n` for improved performance.

<hr/> <code>\tl_reverse_items:n</code> ★ <hr/>	<code>\tl_reverse_items:n</code> $\{\langle token\ list\rangle\}$
New: 2012-01-08	Reverses the order of the $\langle items\rangle$ stored in $\langle tl\ var\rangle$, so that $\{\langle item_1\rangle\}\{\langle item_2\rangle\}\{\langle item_3\rangle\}\dots\{\langle item_n\rangle\}$ becomes $\{\langle item_n\rangle\}\dots\{\langle item_3\rangle\}\{\langle item_2\rangle\}\{\langle item_1\rangle\}$. This process removes any unprotected space within the $\langle token\ list\rangle$. Braced token groups are copied without reversing the order of tokens, and keep the outer set of braces. Items which are initially not braced are copied with braces in the result. In cases where preserving spaces is important, consider the slower function <code>\tl_reverse:n</code> .

TeXhackers note: The result is returned within `\unexpanded`, which means that the token list does not expand further when appearing in an **x**-type or **e**-type argument expansion.

<hr/> <code>\tl_trim_spaces:n</code> ★ <code>\tl_trim_spaces:o</code> ★ <hr/>	<code>\tl_trim_spaces:n</code> $\{\langle token\ list\rangle\}$
New: 2011-07-09 Updated: 2012-06-25	Removes any leading and trailing explicit space characters (explicit tokens with character code 32 and category code 10) from the $\langle token\ list\rangle$ and leaves the result in the input stream.

TeXhackers note: The result is returned within `\unexpanded`, which means that the token list does not expand further when appearing in an **x**-type or **e**-type argument expansion.

<hr/> <code>\tl_trim_spaces_apply:nN</code> ★ <code>\tl_trim_spaces_apply:oN</code> ★ <hr/>	<code>\tl_trim_spaces_apply:nN</code> $\{\langle token\ list\rangle\}$ $\langle function\rangle$
New: 2018-04-12	Removes any leading and trailing explicit space characters (explicit tokens with character code 32 and category code 10) from the $\langle token\ list\rangle$ and passes the result to the $\langle function\rangle$ as an n -type argument.

<hr/> <code>\tl_trim_spaces:N</code> <code>\tl_trim_spaces:c</code> <code>\tl_gtrim_spaces:N</code> <code>\tl_gtrim_spaces:c</code> <hr/>	<code>\tl_trim_spaces:N</code> $\langle tl\ var\rangle$
New: 2011-07-09	Sets the $\langle tl\ var\rangle$ to contain the result of removing any leading and trailing explicit space characters (explicit tokens with character code 32 and category code 10) from its contents.

15.4.3 Viewing token lists

<hr/> <code>\tl_show:N</code> <code>\tl_show:c</code> <hr/>	<code>\tl_show:N</code> $\langle tl\ var\rangle$
Updated: 2021-04-29	Displays the content of the $\langle tl\ var\rangle$ on the terminal.

TeXhackers note: This is similar to the TeX primitive `\show`, wrapped to a fixed number of characters per line.

<hr/> <code>\tl_show:n</code> <hr/>	<code>\tl_show:n</code> $\{\langle token\ list\rangle\}$
Updated: 2015-08-07	Displays the $\langle token\ list\rangle$ on the terminal.

TeXhackers note: This is similar to the ϵ -TeX primitive `\showtokens`, wrapped to a fixed number of characters per line.

<code>\tl_log:N</code>	<code>\tl_log:N <tl var></code>
<code>\tl_log:c</code>	Writes the content of the <code><tl var></code> in the log file. See also <code>\tl_show:N</code> which displays the result in the terminal.
New: 2014-08-22	
Updated: 2021-04-29	

<code>\tl_log:n</code>	<code>\tl_log:n {<token list>}</code>
New: 2014-08-22	Writes the <code><token list></code> in the log file. See also <code>\tl_show:n</code> which displays the result in the terminal.
Updated: 2015-08-07	

15.5 Manipulating items in token lists

15.5.1 Mapping over token lists

All mappings are done at the current group level, *i.e.* any local assignments made by the `<function>` or `<code>` discussed below remain in effect after the loop.

<code>\tl_map_function:NN</code> ☆	<code>\tl_map_function:NN <tl var> <function></code>
<code>\tl_map_function:cN</code> ☆	Applies <code><function></code> to every <code><item></code> in the <code><tl var></code> . The <code><function></code> receives one argument for each iteration. This may be a number of tokens if the <code><item></code> was stored within braces. Hence the <code><function></code> should anticipate receiving <code>n</code> -type arguments. See also <code>\tl_map_function:nN</code> .
Updated: 2012-06-29	

<code>\tl_map_function:nN</code> ☆	<code>\tl_map_function:nN {<token list>} <function></code>
Updated: 2012-06-29	Applies <code><function></code> to every <code><item></code> in the <code><token list></code> . The <code><function></code> receives one argument for each iteration. This may be a number of tokens if the <code><item></code> was stored within braces. Hence the <code><function></code> should anticipate receiving <code>n</code> -type arguments. See also <code>\tl_map_function:NN</code> .

<code>\tl_map_inline:Nn</code>	<code>\tl_map_inline:Nn <tl var> {<inline function>}</code>
<code>\tl_map_inline:cn</code>	Applies the <code><inline function></code> to every <code><item></code> stored within the <code><tl var></code> . The <code><inline function></code> should consist of code which receives the <code><item></code> as <code>#1</code> . See also <code>\tl_map_function:NN</code> .
Updated: 2012-06-29	

<code>\tl_map_inline:nn</code>	<code>\tl_map_inline:nn {<token list>} {<inline function>}</code>
Updated: 2012-06-29	Applies the <code><inline function></code> to every <code><item></code> stored within the <code><token list></code> . The <code><inline function></code> should consist of code which receives the <code><item></code> as <code>#1</code> . See also <code>\tl_map_function:nN</code> .

<code>\tl_map_tokens:Nn</code> ☆	<code>\tl_map_tokens:Nn <tl var> {<code>}</code>
<code>\tl_map_tokens:cn</code> ☆	<code>\tl_map_tokens:nn {<tokens>} {<code>}</code>
<code>\tl_map_tokens:nn</code> ☆	Analogue of <code>\tl_map_function:NN</code> which maps several tokens instead of a single function. The <code><code></code> receives each <code><item></code> in the <code><tl var></code> or in <code><tokens></code> as a trailing brace group. For instance,
New: 2019-09-02	

```
\tl_map_tokens:Nn \l_my_tl { \prg_replicate:nn { 2 } }
```

expands to twice each `<item>` in the `<tl var>`: for each `<item>` in `\l_my_tl` the function `\prg_replicate:nn` receives 2 and `<item>` as its two arguments. The function `\tl_map_inline:Nn` is typically faster but is not expandable.

<code>\tl_map_variable:NNn</code>	<code>\tl_map_variable:NNn <tl var> <variable> {<code>}</code>
<code>\tl_map_variable:cNn</code>	Stores each <code><item></code> of the <code><tl var></code> in turn in the (token list) <code><variable></code> and applies the <code><code></code> . The <code><code></code> will usually make use of the <code><variable></code> , but this is not enforced. The assignments to the <code><variable></code> are local. Its value after the loop is the last <code><item></code> in the <code><tl var></code> , or its original value if the <code><tl var></code> is blank. See also <code>\tl_map_inline:Nn</code> .
Updated: 2012-06-29	

<code>\tl_map_variable:nNn</code>	<code>\tl_map_variable:nNn {<token list>} <variable> {<code>}</code>
Updated: 2012-06-29	Stores each <code><item></code> of the <code><token list></code> in turn in the (token list) <code><variable></code> and applies the <code><code></code> . The <code><code></code> will usually make use of the <code><variable></code> , but this is not enforced. The assignments to the <code><variable></code> are local. Its value after the loop is the last <code><item></code> in the <code><tl var></code> , or its original value if the <code><tl var></code> is blank. See also <code>\tl_map_inline:nn</code> .

<code>\tl_map_break:</code> ☆	<code>\tl_map_break:</code>
Updated: 2012-06-29	Used to terminate a <code>\tl_map...</code> function before all entries in the <code><token list variable></code> have been processed. This normally takes place within a conditional statement, for example

```
\tl_map_inline:Nn \l_my_tl
{
  \str_if_eq:nnT { #1 } { bingo } { \tl_map_break: }
  % Do something useful
}
```

See also `\tl_map_break:n`. Use outside of a `\tl_map...` scenario leads to low level TeX errors.

TeXhackers note: When the mapping is broken, additional tokens may be inserted before the `<tokens>` are inserted into the input stream. This depends on the design of the mapping function.

`\tl_map_break:n` ☆

Updated: 2012-06-29

`\tl_map_break:n {<code>}`

Used to terminate a `\tl_map...` function before all entries in the *<token list variable>* have been processed, inserting the *<code>* after the mapping has ended. This normally takes place within a conditional statement, for example

```
\tl_map_inline:Nn \l_my_tl
{
  \str_if_eq:nnT { #1 } { bingo }
  { \tl_map_break:n { <code> } }
  % Do something useful
}
```

Use outside of a `\tl_map...` scenario leads to low level TeX errors.

TeXhackers note: When the mapping is broken, additional tokens may be inserted before the *<code>* is inserted into the input stream. This depends on the design of the mapping function.

15.5.2 Head and tail of token lists

Functions which deal with either only the very first item (balanced text or single normal token) in a token list, or the remaining tokens.

`\tl_head:N` ★

`\tl_head:n` ★

`\tl_head:(V|v|f)` ★

Updated: 2012-09-09

`\tl_head:n {<token list>}`

Leaves in the input stream the first *<item>* in the *<token list>*, discarding the rest of the *<token list>*. All leading explicit space characters (explicit tokens with character code 32 and category code 10) are discarded; for example

```
\tl_head:n { abc }
```

and

```
\tl_head:n { ~ abc }
```

both leave `a` in the input stream. If the “head” is a brace group, rather than a single token, the braces are removed, and so

```
\tl_head:n { ~ { ~ ab } c }
```

yields `␣ab`. A blank *<token list>* (see `\tl_if_blank:nTF`) results in `\tl_head:n` leaving nothing in the input stream.

TeXhackers note: The result is returned within `\exp_not:n`, which means that the token list does not expand further when appearing in an *x*-type or *e*-type argument expansion.

<code>\tl_head:w</code>	★	<code>\tl_head:w <token list> { } \q_stop</code>
-------------------------	---	--

Leaves in the input stream the first *<item>* in the *<token list>*, discarding the rest of the *<token list>*. All leading explicit space characters (explicit tokens with character code 32 and category code 10) are discarded. A blank *<token list>* (which consists only of space characters) results in a low-level T_EX error, which may be avoided by the inclusion of an empty group in the input (as shown), without the need for an explicit test. Alternatively, `\tl_if_blank:nF` may be used to avoid using the function with a “blank” argument. This function requires only a single expansion, and thus is suitable for use within an o-type expansion. In general, `\tl_head:n` should be preferred if the number of expansions is not critical.

<code>\tl_tail:N</code>	★	<code>\tl_tail:n {<token list>}</code>
<code>\tl_tail:n</code>	★	
<code>\tl_tail:(V v f)</code>	★	

Updated: 2012-09-01

Discards all leading explicit space characters (explicit tokens with character code 32 and category code 10) and the first *<item>* in the *<token list>*, and leaves the remaining tokens in the input stream. Thus for example

`\tl_tail:n { a ~ {bc} d }`

and

`\tl_tail:n { ~ a ~ {bc} d }`

both leave `\tl_tail:n` leaving nothing in the input stream. A blank *<token list>* (see `\tl_if_blank:nTF`) results in `\tl_tail:n` leaving nothing in the input stream.

T_EXhackers note: The result is returned within `\exp_not:n`, which means that the token list does not expand further when appearing in an x-type or e-type argument expansion.

15.5.3 Items and ranges in token lists

<code>\tl_item:nn</code>	★	<code>\tl_item:nn {<token list>} {<integer expression>}</code>
<code>\tl_item:Nn</code>	★	
<code>\tl_item:cn</code>	★	

New: 2014-07-17

Indexing items in the *<token list>* from 1 on the left, this function evaluates the *<integer expression>* and leaves the appropriate item from the *<token list>* in the input stream. If the *<integer expression>* is negative, indexing occurs from the right of the token list, starting at -1 for the right-most item. If the index is out of bounds, then the function expands to nothing.

T_EXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the *<item>* does not expand further when appearing in an x-type or e-type argument expansion.

<code>\tl_rand_item:N *</code>	<code>\tl_rand_item:N <tl var></code>
<code>\tl_rand_item:c *</code>	<code>\tl_rand_item:n <token list></code>
<code>\tl_rand_item:n *</code>	Selects a pseudo-random item of the <i><token list></i> . If the <i><token list></i> is blank, the result is empty. This is not available in older versions of X _Y TeX.

New: 2016-12-06

TeXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the *<item>* does not expand further when appearing in an **x**-type or **e**-type argument expansion.

<code>\tl_range:Nnn</code> \star <code>\tl_range:nnn</code> \star	<code>\tl_range:Nnn</code> $\langle \text{tl var} \rangle$ $\{\langle \text{start index} \rangle\}$ $\{\langle \text{end index} \rangle\}$ <code>\tl_range:nnn</code> $\{\langle \text{token list} \rangle\}$ $\{\langle \text{start index} \rangle\}$ $\{\langle \text{end index} \rangle\}$
--	--

New: 2017-02-17
Updated: 2017-07-15

Leaves in the input stream the items from the $\langle \text{start index} \rangle$ to the $\langle \text{end index} \rangle$ inclusive. Spaces and braces are preserved between the items returned (but never at either end of the list). Here $\langle \text{start index} \rangle$ and $\langle \text{end index} \rangle$ should be $\langle \text{integer expressions} \rangle$. For describing in detail the functions' behavior, let m and n be the start and end index respectively. If either is 0, the result is empty. A positive index means 'start counting from the left end', and a negative index means 'from the right end'. Let l be the count of the token list.

The *actual start point* is determined as $M = m$ if $m > 0$ and as $M = l + m + 1$ if $m < 0$. Similarly the *actual end point* is $N = n$ if $n > 0$ and $N = l + n + 1$ if $n < 0$. If $M > N$, the result is empty. Otherwise it consists of all items from position M to position N inclusive; for the purpose of this rule, we can imagine that the token list extends at infinity on either side, with void items at positions s for $s \leq 0$ or $s > l$.

Spaces in between items in the actual range are preserved. Spaces at either end of the token list will be removed anyway (think to the token list being passed to `\tl_trim_spaces:n` to begin with).

Thus, with $l = 7$ as in the examples below, all of the following are equivalent and result in the whole token list

```
\tl_range:nnn { abcd~{e{}}fg } { 1 } { 7 }
\tl_range:nnn { abcd~{e{}}fg } { 1 } { 12 }
\tl_range:nnn { abcd~{e{}}fg } { -7 } { 7 }
\tl_range:nnn { abcd~{e{}}fg } { -12 } { 7 }
```

Here are some more interesting examples. The calls

```
\iow_term:x { \tl_range:nnn { abcd~{e{}}fg } { 2 } { 5 } }
\iow_term:x { \tl_range:nnn { abcd~{e{}}fg } { 2 } { -3 } }
\iow_term:x { \tl_range:nnn { abcd~{e{}}fg } { -6 } { 5 } }
\iow_term:x { \tl_range:nnn { abcd~{e{}}fg } { -6 } { -3 } }
```

are all equivalent and will print `bcd{e{}}` on the terminal; similarly

```
\iow_term:x { \tl_range:nnn { abcd~{e{}}fg } { 2 } { 5 } }
\iow_term:x { \tl_range:nnn { abcd~{e{}}fg } { 2 } { -3 } }
\iow_term:x { \tl_range:nnn { abcd~{e{}}fg } { -6 } { 5 } }
\iow_term:x { \tl_range:nnn { abcd~{e{}}fg } { -6 } { -3 } }
```

are all equivalent and will print `bcd {e{}}` on the terminal (note the space in the middle). To the contrary,

```
\tl_range:nnn { abcd~{e{}}f } { 2 } { 4 }
```

will discard the space after 'd'.

If we want to get the items from, say, the third to the last in a token list $\langle \text{tl} \rangle$, the call is `\tl_range:nnn { <tl> } { 3 } { -1 }`. Similarly, for discarding the last item, we can do `\tl_range:nnn { <tl> } { 1 } { -2 }`.

For better performance, see `\tl_range_braced:nnn` and `\tl_range_unbraced:nnn`.

T_EXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the $\langle \text{item} \rangle$ does not expand further when appearing in an `x`-type or `e`-type argument expansion.

15.5.4 Sorting token lists

<hr/>	
<code>\tl_sort:Nn</code>	<code>\tl_sort:Nn <tl var> {<comparison code>}</code>
<code>\tl_sort:cn</code>	Sorts the items in the <code><tl var></code> according to the <code><comparison code></code> , and assigns the result to <code><tl var></code> . The details of sorting comparison are described in Section 6.1.
<code>\tl_gsort:Nn</code>	
<code>\tl_gsort:cn</code>	
<hr/>	
New: 2017-02-06	
<hr/>	
<code>\tl_sort:nN</code> ★	<code>\tl_sort:nN {<token list>} {<conditional>}</code>
<hr/>	
New: 2017-02-06	Sorts the items in the <code><token list></code> , using the <code><conditional></code> to compare items, and leaves the result in the input stream. The <code><conditional></code> should have signature <code>:nnTF</code> , and return <code>true</code> if the two items being compared should be left in the same order, and <code>false</code> if the items should be swapped. The details of sorting comparison are described in Section 6.1.

TeXhackers note: The result is returned within `\exp_not:n`, which means that the token list does not expand further when appearing in an `x`-type or `e`-type argument expansion.

15.6 Manipulating tokens in token lists

15.6.1 Replacing tokens

Within token lists, replacement takes place at the top level: there is no recursion into brace groups (more precisely, within a group defined by a category code 1/2 pair).

<hr/>	
<code>\tl_replace_once:Nnn</code>	<code>\tl_replace_once:Nnn <tl var> {<old tokens>} {<new tokens>}</code>
<code>\tl_replace_once:cnn</code>	Replaces the first (leftmost) occurrence of <code><old tokens></code> in the <code><tl var></code> with <code><new tokens></code> . <code><Old tokens></code> cannot contain <code>{</code> , <code>}</code> or <code>#</code> (more precisely, explicit character tokens with category code 1 (begin-group) or 2 (end-group), and tokens with category code 6).
<code>\tl_greplace_once:Nnn</code>	
<code>\tl_greplace_once:cnn</code>	
<hr/>	
Updated: 2011-08-11	
<hr/>	
<code>\tl_replace_all:Nnn</code>	<code>\tl_replace_all:Nnn <tl var> {<old tokens>} {<new tokens>}</code>
<code>\tl_replace_all:cnn</code>	Replaces all occurrences of <code><old tokens></code> in the <code><tl var></code> with <code><new tokens></code> . <code><Old tokens></code> cannot contain <code>{</code> , <code>}</code> or <code>#</code> (more precisely, explicit character tokens with category code 1 (begin-group) or 2 (end-group), and tokens with category code 6). As this function operates from left to right, the pattern <code><old tokens></code> may remain after the replacement (see <code>\tl_remove_all:Nn</code> for an example).
<code>\tl_greplace_all:Nnn</code>	
<code>\tl_greplace_all:cnn</code>	
<hr/>	
Updated: 2011-08-11	
<hr/>	
<code>\tl_remove_once:Nn</code>	<code>\tl_remove_once:Nn <tl var> {<tokens>}</code>
<code>\tl_remove_once:cn</code>	Removes the first (leftmost) occurrence of <code><tokens></code> from the <code><tl var></code> . <code><Tokens></code> cannot contain <code>{</code> , <code>}</code> or <code>#</code> (more precisely, explicit character tokens with category code 1 (begin-group) or 2 (end-group), and tokens with category code 6).
<code>\tl_gremove_once:Nn</code>	
<code>\tl_gremove_once:cn</code>	
<hr/>	
Updated: 2011-08-11	

```

\tl_remove_all:Nn
\tl_remove_all:cn
\tl_gremove_all:Nn
\tl_gremove_all:cn

```

Updated: 2011-08-11

```
\tl_remove_all:Nn <tl var> {<tokens>}
```

Removes all occurrences of *<tokens>* from the *<tl var>*. *<Tokens>* cannot contain `{`, `}` or `#` (more precisely, explicit character tokens with category code 1 (begin-group) or 2 (end-group), and tokens with category code 6). As this function operates from left to right, the pattern *<tokens>* may remain after the removal, for instance,

```
\tl_set:Nn \l_tmpa_tl {abbccd} \tl_remove_all:Nn \l_tmpa_tl {bc}
```

results in `\l_tmpa_tl` containing `abcd`.

15.6.2 Reassigning category codes

These functions allow the rescanning of tokens: re-apply T_EX's tokenization process to apply category codes different from those in force when the tokens were absorbed. Whilst this functionality is supported, it is often preferable to find alternative approaches to achieving outcomes rather than rescanning tokens (for example construction of token lists token-by-token with intervening category code changes or using `\char_generate:nn`).

```

\tl_set_rescan:Nnn
\tl_set_rescan:(Nno|Nnx|cnn|cno|cnx)
\tl_gset_rescan:Nnn
\tl_gset_rescan:(Nno|Nnx|cnn|cno|cnx)

```

Updated: 2015-08-11

```
\tl_set_rescan:Nnn <tl var> {<setup>} {<tokens>}
```

Sets *<tl var>* to contain *<tokens>*, applying the category code régime specified in the *<setup>* before carrying out the assignment. (Category codes applied to tokens not explicitly covered by the *<setup>* are those in force at the point of use of `\tl_set_rescan:Nnn`.) This allows the *<tl var>* to contain material with category codes other than those that apply when *<tokens>* are absorbed. The *<setup>* is run within a group and may contain any valid input, although only changes in category codes, such as uses of `\cctab_select:N`, are relevant. See also `\tl_rescan:nn`.

T_EXhackers note: The *<tokens>* are first turned into a string (using `\tl_to_str:n`). If the string contains one or more characters with character code `\newlinechar` (set equal to `\endlinechar` unless that is equal to 32, before the user *<setup>*), then it is split into lines at these characters, then read as if reading multiple lines from a file, ignoring spaces (catcode 10) at the beginning and spaces and tabs (character code 32 or 9) at the end of every line. Otherwise, spaces (and tabs) are retained at both ends of the single-line string, as if it appeared in the middle of a line read from a file.

<hr/> <code>\tl_rescan:nn</code> <hr/>	<code>\tl_rescan:nn {<setup>} {<tokens>}</code>
Updated: 2015-08-11	<p>Rescans <i><tokens></i> applying the category code régime specified in the <i><setup></i>, and leaves the resulting tokens in the input stream. (Category codes applied to tokens not explicitly covered by the <i><setup></i> are those in force at the point of use of <code>\tl_rescan:nn</code>.) The <i><setup></i> is run within a group and may contain any valid input, although only changes in category codes, such as uses of <code>\cctab_select:N</code>, are relevant. See also <code>\tl_set_rescan:Nnn</code>, which is more robust than using <code>\tl_set:Nn</code> in the <i><tokens></i> argument of <code>\tl_rescan:nn</code>.</p>

T_EXhackers note: The *<tokens>* are first turned into a string (using `\tl_to_str:n`). If the string contains one or more characters with character code `\newlinechar` (set equal to `\endlinechar` unless that is equal to 32, before the user *<setup>*), then it is split into lines at these characters, then read as if reading multiple lines from a file, ignoring spaces (catcode 10) at the beginning and spaces and tabs (character code 32 or 9) at the end of every line. Otherwise, spaces (and tabs) are retained at both ends of the single-line string, as if it appeared in the middle of a line read from a file.

Contrarily to the `\scantokens` primitive, `\tl_rescan:nn` tokenizes the whole string in the same category code regime rather than one token at a time, so that directives such as `\verb` that rely on changing category codes will not function properly.

15.7 Constant token lists

<hr/> <code>\c_empty_tl</code> <hr/>	Constant that is always empty.
<hr/> <code>\c_novalue_tl</code> <hr/>	A marker for the absence of an argument. This constant <code>tl</code> can safely be typeset (<i>cf.</i> <code>\q_nil</code>), with the result being <code>-NoValue-</code> . It is important to note that <code>\c_novalue_tl</code> is constructed such that it will <i>not</i> match the simple text input <code>-NoValue-</code> , <i>i.e.</i> that
New: 2017-11-14	

`\tl_if_eq:NnTF \c_novalue_tl { -NoValue- }`

is logically **false**. The `\c_novalue_tl` marker is intended for use in creating document-level interfaces, where it serves as an indicator that an (optional) argument was omitted. In particular, it is distinct from a simple empty `tl`.

<hr/> <code>\c_space_tl</code> <hr/>	An explicit space character contained in a token list (compare this with <code>\c_space_token</code>). For use where an explicit space is required.
--------------------------------------	--

15.8 Scratch token lists

<hr/> <code>\l_tmpa_tl</code> <hr/>	Scratch token lists for local assignment. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
<hr/> <code>\l_tmppb_tl</code> <hr/>	

<u>\g_tmpa_tl</u>	Scratch token lists for global assignment. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
<u>\g_tmpb_tl</u>	

Chapter 16

The l3str package: Strings

TeX associates each character with a category code: as such, there is no concept of a “string” as commonly understood in many other programming languages. However, there are places where we wish to manipulate token lists while in some sense “ignoring” category codes: this is done by treating token lists as strings in a TeX sense.

A TeX string (and thus an expl3 string) is a series of characters which have category code 12 (“other”) with the exception of space characters which have category code 10 (“space”). Thus at a technical level, a TeX string is a token list with the appropriate category codes. In this documentation, these are simply referred to as strings.

String variables are simply specialised token lists, but by convention should be named with the suffix `...str`. Such variables should contain characters with category code 12 (other), except spaces, which have category code 10 (blank space). All the functions in this module which accept a token list argument first convert it to a string using `\tl_to_str:n` for internal processing, and do not treat a token list or the corresponding string representation differently.

As a string is a subset of the more general token list, it is sometimes unclear when one should be used over the other. Use a string variable for data that isn’t primarily intended for typesetting and for which a level of protection from unwanted expansion is suitable. This data type simplifies comparison of variables since there are no concerns about expansion of their contents.

The functions `\cs_to_str:N`, `\tl_to_str:n`, `\tl_to_str:N` and `\token_to_str:N` (and variants) generate strings from the appropriate input: these are documented in `l3basics`, `l3tl` and `l3token`, respectively.

Most expandable functions in this module come in three flavours:

- `\str_...:N`, which expect a token list or string variable as their argument;
- `\str_...:n`, taking any token list (or string) as an argument;
- `\str_..._ignore_spaces:n`, which ignores any space encountered during the operation: these functions are typically faster than those which take care of escaping spaces appropriately.

16.1 Creating and initialising string variables

<hr/> $\backslash\text{str_new:N}$ $\backslash\text{str_new:c}$ <hr/> New: 2015-09-18	$\backslash\text{str_new:N}$ $\langle\text{str var}\rangle$ Creates a new $\langle\text{str var}\rangle$ or raises an error if the name is already taken. The declaration is global. The $\langle\text{str var}\rangle$ is initially empty.
<hr/> $\backslash\text{str_const:Nn}$ $\backslash\text{str_const:(NV Nx cn cV cx)}$ <hr/> New: 2015-09-18 Updated: 2018-07-28	$\backslash\text{str_const:Nn}$ $\langle\text{str var}\rangle$ $\{(\text{token list})\}$ Creates a new constant $\langle\text{str var}\rangle$ or raises an error if the name is already taken. The value of the $\langle\text{str var}\rangle$ is set globally to the $\langle\text{token list}\rangle$, converted to a string.
<hr/> $\backslash\text{str_clear:N}$ $\backslash\text{str_clear:c}$ $\backslash\text{str_gclear:N}$ $\backslash\text{str_gclear:c}$ <hr/> New: 2015-09-18	$\backslash\text{str_clear:N}$ $\langle\text{str var}\rangle$ Clears the content of the $\langle\text{str var}\rangle$.
<hr/> $\backslash\text{str_clear_new:N}$ $\backslash\text{str_clear_new:c}$ <hr/> New: 2015-09-18	$\backslash\text{str_clear_new:N}$ $\langle\text{str var}\rangle$ Ensures that the $\langle\text{str var}\rangle$ exists globally by applying $\backslash\text{str_new:N}$ if necessary, then applies $\backslash\text{str_clear:N}$ to leave the $\langle\text{str var}\rangle$ empty.
<hr/> $\backslash\text{str_set_eq:NN}$ $\backslash\text{str_set_eq:(cN Nc cc)}$ $\backslash\text{str_gset_eq:NN}$ $\backslash\text{str_gset_eq:(cN Nc cc)}$ <hr/> New: 2015-09-18	$\backslash\text{str_set_eq:NN}$ $\langle\text{str var}_1\rangle$ $\langle\text{str var}_2\rangle$ Sets the content of $\langle\text{str var}_1\rangle$ equal to that of $\langle\text{str var}_2\rangle$.
<hr/> $\backslash\text{str_concat:NNN}$ $\backslash\text{str_concat:ccc}$ $\backslash\text{str_gconcat:NNN}$ $\backslash\text{str_gconcat:ccc}$ <hr/> New: 2017-10-08	$\backslash\text{str_concat:NNN}$ $\langle\text{str var}_1\rangle$ $\langle\text{str var}_2\rangle$ $\langle\text{str var}_3\rangle$ Concatenates the content of $\langle\text{str var}_2\rangle$ and $\langle\text{str var}_3\rangle$ together and saves the result in $\langle\text{str var}_1\rangle$. The $\langle\text{str var}_2\rangle$ is placed at the left side of the new string variable. The $\langle\text{str var}_2\rangle$ and $\langle\text{str var}_3\rangle$ must indeed be strings, as this function does not convert their contents to a string.
<hr/> $\backslash\text{str_if_exist_p:N}$ \star $\backslash\text{str_if_exist_p:c}$ \star $\backslash\text{str_if_exist:N}\underline{TF}$ \star $\backslash\text{str_if_exist:c}\underline{TF}$ \star <hr/> New: 2015-09-18	$\backslash\text{str_if_exist_p:N}$ $\langle\text{str var}\rangle$ $\backslash\text{str_if_exist:N}\underline{TF}$ $\langle\text{str var}\rangle$ $\{(\text{true code})\}$ $\{(\text{false code})\}$ Tests whether the $\langle\text{str var}\rangle$ is currently defined. This does not check that the $\langle\text{str var}\rangle$ really is a string.

16.2 Adding data to string variables

```
\str_set:Nn
\str_set:(NV|Nx|cn|cV|cx)
\str_gset:Nn
\str_gset:(NV|Nx|cn|cV|cx)
```

New: 2015-09-18
Updated: 2018-07-28

`\str_set:Nn <str var> {<token list>}`
Converts the *<token list>* to a *<string>*, and stores the result in *<str var>*.

```
\str_put_left:Nn
\str_put_left:(NV|Nx|cn|cV|cx)
\str_gput_left:Nn
\str_gput_left:(NV|Nx|cn|cV|cx)
```

New: 2015-09-18
Updated: 2018-07-28

`\str_put_left:Nn <str var> {<token list>}`

Converts the *<token list>* to a *<string>*, and prepends the result to *<str var>*. The current contents of the *<str var>* are not automatically converted to a string.

```
\str_put_right:Nn
\str_put_right:(NV|Nx|cn|cV|cx)
\str_gput_right:Nn
\str_gput_right:(NV|Nx|cn|cV|cx)
```

New: 2015-09-18
Updated: 2018-07-28

`\str_put_right:Nn <str var> {<token list>}`

Converts the *<token list>* to a *<string>*, and appends the result to *<str var>*. The current contents of the *<str var>* are not automatically converted to a string.

16.3 String conditionals

```
\str_if_empty_p:N *
\str_if_empty_p:c *
\str_if_empty:NTF *
\str_if_empty:cTF *
```

New: 2015-09-18

`\str_if_empty_p:N <str var>`
`\str_if_empty:NTF <str var> {<true code>} {<false code>}`
Tests if the *<string variable>* is entirely empty (*i.e.* contains no characters at all).

```
\str_if_eq_p:NN *
\str_if_eq_p:(Nc|cN|cc) *
\str_if_eq:NNTF *
\str_if_eq:(Nc|cN|cc)TF *
```

New: 2015-09-18

`\str_if_eq_p:NN <str var12
\str_if_eq:NNTF <str var12
Compares the content of two <str variables> and is logically true if the two contain the same characters in the same order. See \tl_if_eq:NNTF to compare tokens (including their category codes) rather than characters.`

<code>\str_if_eq_p:nn</code>	★	<code>\str_if_eq_p:nn {<tl₁>} {<tl₂>}</code>
<code>\str_if_eq_p:(Vn on no nV VV vn nv ee)</code>	★	<code>\str_if_eq:nnTF {<tl₁>} {<tl₂>} {<true code>} {<false code>}</code>
<code>\str_if_eq:nnTF</code>	★	
<code>\str_if_eq:(Vn on no nV VV vn nv ee)TF</code>	★	

Updated: 2018-06-18

Compares the two *<token lists>* on a character by character basis (namely after converting them to strings), and is **true** if the two *<strings>* contain the same characters in the same order. Thus for example

`\str_if_eq_p:no { abc } { \tl_to_str:n { abc } }`

is logically **true**. See `\tl_if_eq:nnTF` to compare tokens (including their category codes) rather than characters.

<code>\str_if_in:NnTF</code>	<code>\str_if_in:NnTF <str var> {<token list>} {<true code>} {<false code>}</code>
<code>\str_if_in:cnTF</code>	Converts the <i><token list></i> to a <i><string></i> and tests if that <i><string></i> is found in the content of the <i><str var></i> .

New: 2017-10-08

<code>\str_if_in:nnTF</code>	<code>\str_if_in:nnTF {<tl₁>} {<tl₂>} {<true code>} {<false code>}</code>
	Converts both <i><token lists></i> to <i><strings></i> and tests whether <i><string₂></i> is found inside <i><string₁></i> .

New: 2017-10-08

<code>\str_case:nn</code>	★	<code>\str_case:nnTF {<test string>}</code>
<code>\str_case:(Vn on nV nv)</code>	★	{
<code>\str_case:nnTF</code>	★	{<string case ₁ >} {<code case ₁ >}
<code>\str_case:(Vn on nV nv)TF</code>	★	{<string case ₂ >} {<code case ₂ >}
		...
		{<string case _n >} {<code case _n >}
		}
		{<true code>}
		{<false code>}

New: 2013-07-24
Updated: 2015-02-28

Compares the *<test string>* in turn with each of the *<string cases>* (all token lists are converted to strings). If the two are equal (as described for `\str_if_eq:nnTF`) then the associated *<code>* is left in the input stream and other cases are discarded. If any of the cases are matched, the *<true code>* is also inserted into the input stream (after the code for the appropriate case), while if none match then the *<false code>* is inserted. The function `\str_case:nn`, which does nothing if there is no match, is also available.

<hr/>	<hr/>
<code>\str_case_e:nn</code> ★	<code>\str_case_e:nnTF</code> { <i>test string</i> }
<code>\str_case_e:nnTF</code> ★	{
	{ <i>string case</i> ₁ } { <i>code case</i> ₁ }
	{ <i>string case</i> ₂ } { <i>code case</i> ₂ }
	...
	{ <i>string case</i> _n } { <i>code case</i> _n }
	}
	{ <i>true code</i> }
	{ <i>false code</i> }
<hr/>	<hr/>
New: 2018-06-19	

Compares the full expansion of the *test string* in turn with the full expansion of the *string cases* (all token lists are converted to strings). If the two full expansions are equal (as described for `\str_if_eq:nnTF`) then the associated *code* is left in the input stream and other cases are discarded. If any of the cases are matched, the *true code* is also inserted into the input stream (after the code for the appropriate case), while if none match then the *false code* is inserted. The function `\str_case_e:nn`, which does nothing if there is no match, is also available. The *test string* is expanded in each comparison, and must always yield the same result: for example, random numbers must not be used within this string.

<hr/>	<hr/>
<code>\str_compare_p:nNn</code> ★	<code>\str_compare_p:nNn</code> { <i>tl</i> ₁ } <i>relation</i> { <i>tl</i> ₂ }
<code>\str_compare_p:eNe</code> ★	<code>\str_compare:nNnTF</code> { <i>tl</i> ₁ } <i>relation</i> { <i>tl</i> ₂ } { <i>true code</i> } { <i>false code</i> }
<code>\str_compare:nNnTF</code> ★	
<code>\str_compare:eNeTF</code> ★	
<hr/>	<hr/>

New: 2021-05-17

Compares the two *token lists* on a character by character basis (namely after converting them to strings) in a lexicographic order according to the character codes of the characters. The *relation* can be *<*, *=*, or *>* and the test is **true** under the following conditions:

- for *<*, if the first string is earlier than the second in lexicographic order;
- for *=*, if the two strings have exactly the same characters;
- for *>*, if the first string is later than the second in lexicographic order.

Thus for example the following is logically **true**:

```
\str_compare_p:nNn { ab } < { abc }
```

T_EXhackers note: This is a wrapper around the T_EX primitive `\(pdf)strcmp`. It is meant for programming and not for sorting textual contents, as it simply considers character codes and not more elaborate considerations of grapheme clusters, locale, etc.

16.4 Mapping over strings

All mappings are done at the current group level, *i.e.* any local assignments made by the *function* or *code* discussed below remain in effect after the loop.

<hr/>	<hr/>
<code>\str_map_function:nN</code> ☆	<code>\str_map_function:nN</code> { <i>token list</i> } <i>function</i>
<code>\str_map_function:NN</code> ☆	<code>\str_map_function:NN</code> <i>str var</i> <i>function</i>
<code>\str_map_function:cN</code> ☆	Converts the <i>token list</i> to a <i>string</i> then applies <i>function</i> to every <i>character</i> in the <i>string</i> including spaces.
<hr/>	<hr/>
New: 2017-11-14	

<hr/> <code>\str_map_inline:nn</code> <code>\str_map_inline:Nn</code> <code>\str_map_inline:cn</code> <hr/> New: 2017-11-14	<code>\str_map_inline:nn {⟨token list⟩} {⟨inline function⟩}</code> <code>\str_map_inline:Nn ⟨str var⟩ {⟨inline function⟩}</code> Converts the <i>⟨token list⟩</i> to a <i>⟨string⟩</i> then applies the <i>⟨inline function⟩</i> to every <i>⟨character⟩</i> in the <i>⟨str var⟩</i> including spaces. The <i>⟨inline function⟩</i> should consist of code which receives the <i>⟨character⟩</i> as #1.
<hr/> <code>\str_map_tokens:nn ☆</code> <code>\str_map_tokens:Nn ☆</code> <code>\str_map_tokens:cn ☆</code> <hr/> New: 2021-05-05	<code>\str_map_tokens:nn {⟨token list⟩} {⟨code⟩}</code> <code>\str_map_tokens:Nn ⟨str var⟩ {⟨code⟩}</code> Converts the <i>⟨token list⟩</i> to a <i>⟨string⟩</i> then applies <i>⟨code⟩</i> to every <i>⟨character⟩</i> in the <i>⟨string⟩</i> including spaces. The <i>⟨code⟩</i> receives each character as a trailing brace group. This is equivalent to <code>\str_map_function:nN</code> if the <i>⟨code⟩</i> consists of a single function.
<hr/> <code>\str_map_variable:nNn</code> <code>\str_map_variable:NNn</code> <code>\str_map_variable:cNn</code> <hr/> New: 2017-11-14	<code>\str_map_variable:nNn {⟨token list⟩} ⟨variable⟩ {⟨code⟩}</code> <code>\str_map_variable:NNn ⟨str var⟩ ⟨variable⟩ {⟨code⟩}</code> Converts the <i>⟨token list⟩</i> to a <i>⟨string⟩</i> then stores each <i>⟨character⟩</i> in the <i>⟨string⟩</i> (including spaces) in turn in the (string or token list) <i>⟨variable⟩</i> and applies the <i>⟨code⟩</i> . The <i>⟨code⟩</i> will usually make use of the <i>⟨variable⟩</i> , but this is not enforced. The assignments to the <i>⟨variable⟩</i> are local. Its value after the loop is the last <i>⟨character⟩</i> in the <i>⟨string⟩</i> , or its original value if the <i>⟨string⟩</i> is empty. See also <code>\str_map_inline:Nn</code> .
<hr/> <code>\str_map_break: ☆</code> <hr/> New: 2017-10-08	<code>\str_map_break:</code> Used to terminate a <code>\str_map...</code> function before all characters in the <i>⟨string⟩</i> have been processed. This normally takes place within a conditional statement, for example <pre> \str_map_inline:Nn \l_my_str { \str_if_eq:nnT { #1 } { bingo } { \str_map_break: } % Do something useful } </pre> <p>See also <code>\str_map_break:n</code>. Use outside of a <code>\str_map...</code> scenario leads to low level \TeX errors.</p> <p>\TeXhackers note: When the mapping is broken, additional tokens may be inserted before continuing with the code that follows the loop. This depends on the design of the mapping function.</p>

`\str_map_break:n` ☆

New: 2017-10-08

`\str_map_break:n` {*<code>*}

Used to terminate a `\str_map...` function before all characters in the *<string>* have been processed, inserting the *<code>* after the mapping has ended. This normally takes place within a conditional statement, for example

```
\str_map_inline:Nn \l_my_str
{
  \str_if_eq:nnT { #1 } { bingo }
  { \str_map_break:n { <code> } }
  % Do something useful
}
```

Use outside of a `\str_map...` scenario leads to low level TeX errors.

TeXhackers note: When the mapping is broken, additional tokens may be inserted before the *<code>* is inserted into the input stream. This depends on the design of the mapping function.

16.5 Working with the content of strings

`\str_use:N` ★

`\str_use:c` ★

New: 2015-09-18

`\str_use:N` *<str var>*

Recovers the content of a *<str var>* and places it directly in the input stream. An error is raised if the variable does not exist or if it is invalid. Note that it is possible to use a *<str>* directly without an accessor function.

<code>\str_count:N</code>	★	<code>\str_count:n</code> { <i><token list></i> }
<code>\str_count:c</code>	★	
<code>\str_count:n</code>	★	
<code>\str_count_ignore_spaces:n</code>	★	

New: 2015-09-18

Leaves in the input stream the number of characters in the string representation of *<token list>*, as an integer denotation. The functions differ in their treatment of spaces. In the case of `\str_count:N` and `\str_count:n`, all characters including spaces are counted. The `\str_count_ignore_spaces:n` function leaves the number of non-space characters in the input stream.

`\str_count_spaces:N` ★

`\str_count_spaces:c` ★

`\str_count_spaces:n` ★

New: 2015-09-18

`\str_count_spaces:n` {*<token list>*}

Leaves in the input stream the number of space characters in the string representation of *<token list>*, as an integer denotation. Of course, this function has no `_ignore_spaces` variant.

<code>\str_head:N</code>	★	<code>\str_head:n {⟨token list⟩}</code>
<code>\str_head:c</code>	★	
<code>\str_head:n</code>	★	
<code>\str_head_ignore_spaces:n</code>	★	

New: 2015-09-18

Converts the $\langle token\ list \rangle$ into a $\langle string \rangle$. The first character in the $\langle string \rangle$ is then left in the input stream, with category code “other”. The functions differ if the first character is a space: `\str_head:N` and `\str_head:n` return a space token with category code 10 (blank space), while the `\str_head_ignore_spaces:n` function ignores this space character and leaves the first non-space character in the input stream. If the $\langle string \rangle$ is empty (or only contains spaces in the case of the `_ignore_spaces` function), then nothing is left on the input stream.

<code>\str_tail:N</code>	★	<code>\str_tail:n {⟨token list⟩}</code>
<code>\str_tail:c</code>	★	
<code>\str_tail:n</code>	★	
<code>\str_tail_ignore_spaces:n</code>	★	

New: 2015-09-18

Converts the $\langle token\ list \rangle$ to a $\langle string \rangle$, removes the first character, and leaves the remaining characters (if any) in the input stream, with category codes 12 and 10 (for spaces). The functions differ in the case where the first character is a space: `\str_tail:N` and `\str_tail:n` only trim that space, while `\str_tail_ignore_spaces:n` removes the first non-space character and any space before it. If the $\langle token\ list \rangle$ is empty (or blank in the case of the `_ignore_spaces` variant), then nothing is left on the input stream.

<code>\str_item:Nn</code>	★	<code>\str_item:nn {⟨token list⟩} {⟨integer expression⟩}</code>
<code>\str_item:nn</code>	★	
<code>\str_item_ignore_spaces:nn</code>	★	

New: 2015-09-18

Converts the $\langle token\ list \rangle$ to a $\langle string \rangle$, and leaves in the input stream the character in position $\langle integer\ expression \rangle$ of the $\langle string \rangle$, starting at 1 for the first (left-most) character. In the case of `\str_item:Nn` and `\str_item:nn`, all characters including spaces are taken into account. The `\str_item_ignore_spaces:nn` function skips spaces when counting characters. If the $\langle integer\ expression \rangle$ is negative, characters are counted from the end of the $\langle string \rangle$. Hence, -1 is the right-most character, *etc.*

```

\str_range:Nnn      * \str_range:nnn {⟨token list⟩} {⟨start index⟩} {⟨end index⟩}
\str_range:cnn      *
\str_range:nnn      *
\str_range_ignore_spaces:nnn *

```

New: 2015-09-18

Converts the $\langle token\ list \rangle$ to a $\langle string \rangle$, and leaves in the input stream the characters from the $\langle start\ index \rangle$ to the $\langle end\ index \rangle$ inclusive. Spaces are preserved and counted as items (contrast this with `\tl_range:nnn` where spaces are not counted as items and are possibly discarded from the output).

Here $\langle start\ index \rangle$ and $\langle end\ index \rangle$ should be integer denotations. For describing in detail the functions' behavior, let m and n be the start and end index respectively. If either is 0, the result is empty. A positive index means 'start counting from the left end', a negative index means 'start counting from the right end'. Let l be the count of the token list.

The *actual start point* is determined as $M = m$ if $m > 0$ and as $M = l + m + 1$ if $m < 0$. Similarly the *actual end point* is $N = n$ if $n > 0$ and $N = l + n + 1$ if $n < 0$. If $M > N$, the result is empty. Otherwise it consists of all items from position M to position N inclusive; for the purpose of this rule, we can imagine that the token list extends at infinity on either side, with void items at positions s for $s \leq 0$ or $s > l$. For instance,

```

\iow_term:x { \str_range:nnn { abcdef } { 2 } { 5 } }
\iow_term:x { \str_range:nnn { abcdef } { -4 } { -1 } }
\iow_term:x { \str_range:nnn { abcdef } { -2 } { -1 } }
\iow_term:x { \str_range:nnn { abcdef } { 0 } { -1 } }

```

prints `bcde`, `cdef`, `ef`, and an empty line to the terminal. The $\langle start\ index \rangle$ must always be smaller than or equal to the $\langle end\ index \rangle$: if this is not the case then no output is generated. Thus

```

\iow_term:x { \str_range:nnn { abcdef } { 5 } { 2 } }
\iow_term:x { \str_range:nnn { abcdef } { -1 } { -4 } }

```

both yield empty strings.

The behavior of `\str_range_ignore_spaces:nnn` is similar, but spaces are removed before starting the job. The input

```

\iow_term:x { \str_range:nnn { abcdefg } { 2 } { 5 } }
\iow_term:x { \str_range:nnn { abcdefg } { 2 } { -3 } }
\iow_term:x { \str_range:nnn { abcdefg } { -6 } { 5 } }
\iow_term:x { \str_range:nnn { abcdefg } { -6 } { -3 } }

\iow_term:x { \str_range:nnn { abc~efg } { 2 } { 5 } }
\iow_term:x { \str_range:nnn { abc~efg } { 2 } { -3 } }
\iow_term:x { \str_range:nnn { abc~efg } { -6 } { 5 } }
\iow_term:x { \str_range:nnn { abc~efg } { -6 } { -3 } }

\iow_term:x { \str_range_ignore_spaces:nnn { abcdefg } { 2 } { 5 } }
\iow_term:x { \str_range_ignore_spaces:nnn { abcdefg } { 2 } { -3 } }
\iow_term:x { \str_range_ignore_spaces:nnn { abcdefg } { -6 } { 5 } }
\iow_term:x { \str_range_ignore_spaces:nnn { abcdefg } { -6 } { -3 } }

```

```

\iow_term:x { \str_range_ignore_spaces:nnn { abcd~efg } { 2 } { 5 } }
\iow_term:x { \str_range_ignore_spaces:nnn { abcd~efg } { 2 } { -3 } }
\iow_term:x { \str_range_ignore_spaces:nnn { abcd~efg } { -6 } { 5 } }
\iow_term:x { \str_range_ignore_spaces:nnn { abcd~efg } { -6 } { -3 } }

```

will print four instances of bcde, four instances of bc e and eight instances of bcde.

16.6 Modifying string variables

```

\str_replace_once:Nnn
\str_replace_once:cnn
\str_greplace_once:Nnn
\str_greplace_once:cnn

```

New: 2017-10-08

`\str_replace_once:Nnn` $\langle str\ var \rangle$ $\{\langle old \rangle\}$ $\{\langle new \rangle\}$

Converts the $\langle old \rangle$ and $\langle new \rangle$ token lists to strings, then replaces the first (leftmost) occurrence of $\langle old\ string \rangle$ in the $\langle str\ var \rangle$ with $\langle new\ string \rangle$.

```

\str_replace_all:Nnn
\str_replace_all:cnn
\str_greplace_all:Nnn
\str_greplace_all:cnn

```

New: 2017-10-08

`\str_replace_all:Nnn` $\langle str\ var \rangle$ $\{\langle old \rangle\}$ $\{\langle new \rangle\}$

Converts the $\langle old \rangle$ and $\langle new \rangle$ token lists to strings, then replaces all occurrences of $\langle old\ string \rangle$ in the $\langle str\ var \rangle$ with $\langle new\ string \rangle$. As this function operates from left to right, the pattern $\langle old\ string \rangle$ may remain after the replacement (see `\str_remove_all:Nn` for an example).

```

\str_remove_once:Nn
\str_remove_once:cn
\str_gremove_once:Nn
\str_gremove_once:cn

```

New: 2017-10-08

`\str_remove_once:Nn` $\langle str\ var \rangle$ $\{\langle token\ list \rangle\}$

Converts the $\langle token\ list \rangle$ to a $\langle string \rangle$ then removes the first (leftmost) occurrence of $\langle string \rangle$ from the $\langle str\ var \rangle$.

```

\str_remove_all:Nn
\str_remove_all:cn
\str_gremove_all:Nn
\str_gremove_all:cn

```

New: 2017-10-08

`\str_remove_all:Nn` $\langle str\ var \rangle$ $\{\langle token\ list \rangle\}$

Converts the $\langle token\ list \rangle$ to a $\langle string \rangle$ then removes all occurrences of $\langle string \rangle$ from the $\langle str\ var \rangle$. As this function operates from left to right, the pattern $\langle string \rangle$ may remain after the removal, for instance,

```

\str_set:Nn \l_tmpa_str {abbccd} \str_remove_all:Nn \l_tmpa_str
{bc}

```

results in `\l_tmpa_str` containing `abcd`.

16.7 String manipulation

<code>\str_lowercase:n *</code>	<code>\str_lowercase:n {(tokens)}</code>
<code>\str_lowercase:f *</code>	<code>\str_uppercase:n {(tokens)}</code>
<code>\str_uppercase:n *</code>	
<code>\str_uppercase:f *</code>	

New: 2019-11-26

Converts the input $\langle tokens \rangle$ to their string representation, as described for `\tl_to_str:n`, and then to the lower or upper case representation using a one-to-one mapping as described by the Unicode Consortium file `UnicodeData.txt`.

These functions are intended for case changing programmatic data in places where upper/lower case distinctions are meaningful. One example would be automatically generating a function name from user input where some case changing is needed. In this situation the input is programmatic, not textual, case does have meaning and a language-independent one-to-one mapping is appropriate. For example

```
\cs_new_protected:Npn \myfunc:nn #1#2
{
  \cs_set_protected:cpn
  {
    user
    \str_uppercase:f { \tl_head:n {#1} }
    \str_lowercase:f { \tl_tail:n {#1} }
  }
  { #2 }
}
```

would be used to generate a function with an auto-generated name consisting of the upper case equivalent of the supplied name followed by the lower case equivalent of the rest of the input.

These functions should *not* be used for

- Caseless comparisons: use `\str_foldcase:n` for this situation (case folding is distinct from lower casing).
- Case changing text for typesetting: see the `\text_lowercase:n(n)`, `\text_uppercase:n(n)` and `\text_titlecase:n(n)` functions which correctly deal with context-dependence and other factors appropriate to text case changing.

TeXhackers note: As with all expl3 functions, the input supported by `\str_foldcase:n` is *engine-native* characters which are or interoperate with UTF-8. As such, when used with pdfTeX *only* the Latin alphabet characters A–Z are case-folded (*i.e.* the ASCII range which coincides with UTF-8). Full UTF-8 support is available with both XeTeX and LuaTeX.

```
\str_foldcase:n ★
\str_foldcase:V ★
```

New: 2019-11-26

```
\str_foldcase:n {⟨tokens⟩}
```

Converts the input $\langle tokens \rangle$ to their string representation, as described for `\tl_to_str:n`, and then folds the case of the resulting $\langle string \rangle$ to remove case information. The result of this process is left in the input stream.

String folding is a process used for material such as identifiers rather than for “text”. The folding provided by `\str_foldcase:n` follows the mappings provided by the [Unicode Consortium](#), who [state](#):

Case folding is primarily used for caseless comparison of text, such as identifiers in a computer program, rather than actual text transformation. Case folding in Unicode is based on the lowercase mapping, but includes additional changes to the source text to help make it language-insensitive and consistent. As a result, case-folded text should be used solely for internal processing and generally should not be stored or displayed to the end user.

The folding approach implemented by `\str_foldcase:n` follows the “full” scheme defined by the Unicode Consortium (*e.g.* SSfolds to SS). As case-folding is a language-insensitive process, there is no special treatment of Turkic input (*i.e.* I always folds to i and not to ı).

TeXhackers note: As with all `expl3` functions, the input supported by `\str_foldcase:n` is *engine-native* characters which are or interoperate with UTF-8. As such, when used with `pdfTeX` *only* the Latin alphabet characters A–Z are case-folded (*i.e.* the ASCII range which coincides with UTF-8). Full UTF-8 support is available with both `XYTeX` and `LuaTeX`, subject only to the fact that `XYTeX` in particular has issues with characters of code above hexadecimal 0xFFFF when interacting with `\tl_to_str:n`.

16.8 Viewing strings

```
\str_show:N
\str_show:c
\str_show:n
```

New: 2015-09-18
Updated: 2021-04-29

```
\str_show:N ⟨str var⟩
```

Displays the content of the $\langle str var \rangle$ on the terminal.

```
\str_log:N
\str_log:c
\str_log:n
```

New: 2019-02-15
Updated: 2021-04-29

```
\str_log:N ⟨str var⟩
```

Writes the content of the $\langle str var \rangle$ in the log file.

16.9 Constant strings

`\c_ampersand_str`
`\c_atsign_str`
`\c_backslash_str`
`\c_left_brace_str`
`\c_right_brace_str`
`\c_circumflex_str`
`\c_colon_str`
`\c_dollar_str`
`\c_hash_str`
`\c_percent_str`
`\c_tilde_str`
`\c_underscore_str`
`\c_zero_str`

Constant strings, containing a single character token, with category code 12.

New: 2015-09-19
Updated: 2020-12-22

16.10 Scratch strings

`\l_tmpa_str`
`\l_tmpb_str`

Scratch strings for local assignment. These are never used by the kernel code, and so are safe for use with any \LaTeX 3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

`\g_tmpa_str`
`\g_tmpb_str`

Scratch strings for global assignment. These are never used by the kernel code, and so are safe for use with any \LaTeX 3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

Chapter 17

The l3str-convert package: string encoding conversions

17.1 Encoding and escaping schemes

Traditionally, string encodings only specify how strings of characters should be stored as bytes. However, the resulting lists of bytes are often to be used in contexts where only a restricted subset of bytes are permitted (*e.g.*, PDF string objects, URLs). Hence, storing a string of characters is done in two steps.

- The code points (“character codes”) are expressed as bytes following a given “encoding”. This can be UTF-16, ISO 8859-1, *etc.* See Table 1 for a list of supported encodings.⁷
- Bytes are translated to T_EX tokens through a given “escaping”. Those are defined for the most part by the pdf file format. See Table 2 for a list of escaping methods supported.⁷

⁷Encodings and escapings will be added as they are requested.

Table 1: Supported encodings. Non-alphanumeric characters are ignored, and capital letters are lower-cased before searching for the encoding in this list.

<i>⟨Encoding⟩</i>	description
<code>utf8</code>	UTF-8
<code>utf16</code>	UTF-16, with byte-order mark
<code>utf16be</code>	UTF-16, big-endian
<code>utf16le</code>	UTF-16, little-endian
<code>utf32</code>	UTF-32, with byte-order mark
<code>utf32be</code>	UTF-32, big-endian
<code>utf32le</code>	UTF-32, little-endian
<code>iso88591, latin1</code>	ISO 8859-1
<code>iso88592, latin2</code>	ISO 8859-2
<code>iso88593, latin3</code>	ISO 8859-3
<code>iso88594, latin4</code>	ISO 8859-4
<code>iso88595</code>	ISO 8859-5
<code>iso88596</code>	ISO 8859-6
<code>iso88597</code>	ISO 8859-7
<code>iso88598</code>	ISO 8859-8
<code>iso88599, latin5</code>	ISO 8859-9
<code>iso885910, latin6</code>	ISO 8859-10
<code>iso885911</code>	ISO 8859-11
<code>iso885913, latin7</code>	ISO 8859-13
<code>iso885914, latin8</code>	ISO 8859-14
<code>iso885915, latin9</code>	ISO 8859-15
<code>iso885916, latin10</code>	ISO 8859-16
<code>clist</code>	comma-list of integers
<code>⟨empty⟩</code>	native (Unicode) string
<code>default</code>	like <code>utf8</code> with 8-bit engines, and like native with unicode-engines

Table 2: Supported escapings. Non-alphanumeric characters are ignored, and capital letters are lower-cased before searching for the escaping in this list.

<i>⟨Escaping⟩</i>	description
<code>bytes</code> , or <code>empty</code>	arbitrary bytes
<code>hex</code> , <code>hexadecimal</code>	byte = two hexadecimal digits
<code>name</code>	see <code>\pdfescapename</code>
<code>string</code>	see <code>\pdfescapestring</code>
<code>url</code>	encoding used in URLs

17.2 Conversion functions

```
\str_set_convert:Nnnn
\str_gset_convert:Nnnn
```

```
\str_set_convert:Nnnn <str var> {<string>} {<name 1>} {<name 2>}
```

This function converts the $\langle string \rangle$ from the encoding given by $\langle name 1 \rangle$ to the encoding given by $\langle name 2 \rangle$, and stores the result in the $\langle str var \rangle$. Each $\langle name \rangle$ can have the form $\langle encoding \rangle$ or $\langle encoding \rangle / \langle escaping \rangle$, where the possible values of $\langle encoding \rangle$ and $\langle escaping \rangle$ are given in Tables 1 and 2, respectively. The default escaping is to input and output bytes directly. The special case of an empty $\langle name \rangle$ indicates the use of “native” strings, 8-bit for pdfTeX, and Unicode strings for the other two engines.

For example,

```
\str_set_convert:Nnnn \l_foo_str { Hello! } { } { utf16/hex }
```

results in the variable `\l_foo_str` holding the string `FEFF00480065006C006C006F0021`. This is obtained by converting each character in the (native) string `Hello!` to the UTF-16 encoding, and expressing each byte as a pair of hexadecimal digits. Note the presence of a (big-endian) byte order mark “FEFF”, which can be avoided by specifying the encoding `utf16be/hex`.

An error is raised if the $\langle string \rangle$ is not valid according to the $\langle escaping 1 \rangle$ and $\langle encoding 1 \rangle$, or if it cannot be reencoded in the $\langle encoding 2 \rangle$ and $\langle escaping 2 \rangle$ (for instance, if a character does not exist in the $\langle encoding 2 \rangle$). Erroneous input is replaced by the Unicode replacement character “FFFD”, and characters which cannot be reencoded are replaced by either the replacement character “FFFD” if it exists in the $\langle encoding 2 \rangle$, or an encoding-specific replacement character, or the question mark character.

```
\str_set_convert:NnnnTF
\str_gset_convert:NnnnTF
```

```
\str_set_convert:NnnnTF <str var> {<string>} {<name 1>} {<name 2>} {<true code>}
{<false code>}
```

As `\str_set_convert:Nnnn`, converts the $\langle string \rangle$ from the encoding given by $\langle name 1 \rangle$ to the encoding given by $\langle name 2 \rangle$, and assigns the result to $\langle str var \rangle$. Contrarily to `\str_set_convert:Nnnn`, the conditional variant does not raise errors in case the $\langle string \rangle$ is not valid according to the $\langle name 1 \rangle$ encoding, or cannot be expressed in the $\langle name 2 \rangle$ encoding. Instead, the $\langle false code \rangle$ is performed.

17.3 Conversion by expansion (for PDF contexts)

A small number of expandable functions are provided for use in PDF string/name contexts. These *assume UTF-8* and *no escaping* in the input.

```
\str_convert_pdfname:n *
```

```
\str_convert_pdfname:n <string>
```

As `\str_set_convert:Nnnn`, converts the $\langle string \rangle$ on a byte-by-byte basis with non-ASCII codepoints escaped using hashes.

17.4 Possibilities, and things to do

Encoding/escaping-related tasks.

- In Xe_{La}TeX/Lua_{La}TeX, would it be better to use the `^^^~....` approach to build a string from a given list of character codes? Namely, within a group, assign 0-9a-f and all characters we want to category “other”, then assign `^` the category superscript, and use `\scantokens`.
- Change `\str_set_convert:Nnnn` to expand its last two arguments.
- Describe the internal format in the code comments. Refuse code points in [“D800,”DFFF] in the internal representation?
- Add documentation about each encoding and escaping method, and add examples.
- The `hex` unescaping should raise an error for odd-token count strings.
- Decide what bytes should be escaped in the `url` escaping. Perhaps the characters `! ' () * - . / 0 1 2 3 4 5 6 7 8 9 _` are safe, and all other characters should be escaped?
- Automate generation of 8-bit mapping files.
- Change the framework for 8-bit encodings: for decoding from 8-bit to Unicode, use 256 integer registers; for encoding, use a tree-box.
- More encodings (see Heiko’s `stringenc`). CESU?
- More escapings: ASCII85, shell escapes, lua escapes, *etc.*?

Chapter 18

The l3quark package

Quarks

Two special types of constants in L^AT_EX3 are “quarks” and “scan marks”. By convention all constants of type quark start out with `\q_`, and scan marks start with `\s_`.

18.1 Quarks

Quarks are control sequences (and in fact, token lists) that expand to themselves and should therefore *never* be executed directly in the code. This would result in an endless loop!

They are meant to be used as delimiter in weird functions, the most common use case being the ‘stop token’ (*i.e.* `\q_stop`). For example, when writing a macro to parse a user-defined date

```
\date_parse:n {19/June/1981}
```

one might write a command such as

```
\cs_new:Npn \date_parse:n #1 { \date_parse_aux:w #1 \q_stop }
\cs_new:Npn \date_parse_aux:w #1 / #2 / #3 \q_stop
{ <do something with the date> }
```

Quarks are sometimes also used as error return values for functions that receive erroneous input. For example, in the function `\prop_get:NnN` to retrieve a value stored in some key of a property list, if the key does not exist then the return value is the quark `\q_no_value`. As mentioned above, such quarks are extremely fragile and it is imperative when using such functions that code is carefully written to check for pathological cases to avoid leakage of a quark into an uncontrolled environment.

Quarks also permit the following ingenious trick when parsing tokens: when you pick up a token in a temporary variable and you want to know whether you have picked up a particular quark, all you have to do is compare the temporary variable to the quark using `\tl_if_eq:NNTF`. A set of special quark testing functions is set up below. All the quark testing functions are expandable although the ones testing only single tokens are much faster.

18.2 Defining quarks

<u><code>\quark_new:N</code></u>	<code>\quark_new:N <quark></code> Creates a new <code><quark></code> which expands only to <code><quark></code> . The <code><quark></code> is defined globally, and an error message is raised if the name was already taken.
<u><code>\q_stop</code></u>	Used as a marker for delimited arguments, such as <code>\cs_set:Npn \tmp:w #1#2 \q_stop {#1}</code>
<u><code>\q_mark</code></u>	Used as a marker for delimited arguments when <code>\q_stop</code> is already in use.
<u><code>\q_nil</code></u>	Quark to mark a null value in structured variables or functions. Used as an end delimiter when this may itself need to be tested (in contrast to <code>\q_stop</code> , which is only ever used as a delimiter).
<u><code>\q_no_value</code></u>	A canonical value for a missing value, when one is requested from a data structure. This is therefore used as a “return” value by functions such as <code>\prop_get:NnN</code> if there is no data to return.

18.3 Quark tests

The method used to define quarks means that the single token (N) tests are faster than the multi-token (n) tests. The latter should therefore only be used when the argument can definitely take more than a single token.

<u><code>\quark_if_nil_p:N *</code></u>	<code>\quark_if_nil_p:N <token></code>
<u><code>\quark_if_nil:NTF *</code></u>	<code>\quark_if_nil:NTF <token> {\true code} {\false code}</code>
	Tests if the <code><token></code> is equal to <code>\q_nil</code> .
<u><code>\quark_if_nil_p:n *</code></u>	<code>\quark_if_nil_p:n {\token list}</code>
<u><code>\quark_if_nil_p:(o V) *</code></u>	<code>\quark_if_nil:nTF {\token list} {\true code} {\false code}</code>
<u><code>\quark_if_nil:nTF *</code></u>	Tests if the <code><token list></code> contains only <code>\q_nil</code> (distinct from <code><token list></code> being empty or containing <code>\q_nil</code> plus one or more other tokens).
<u><code>\quark_if_nil:(o V)TF *</code></u>	
<u><code>\quark_if_no_value_p:N *</code></u>	<code>\quark_if_no_value_p:N <token></code>
<u><code>\quark_if_no_value_p:c *</code></u>	<code>\quark_if_no_value:NTF <token> {\true code} {\false code}</code>
<u><code>\quark_if_no_value:NTF *</code></u>	Tests if the <code><token></code> is equal to <code>\q_no_value</code> .
<u><code>\quark_if_no_value:cTF *</code></u>	
<u><code>\quark_if_no_value_p:n *</code></u>	<code>\quark_if_no_value_p:n {\token list}</code>
<u><code>\quark_if_no_value:nTF *</code></u>	<code>\quark_if_no_value:nTF {\token list} {\true code} {\false code}</code>
	Tests if the <code><token list></code> contains only <code>\q_no_value</code> (distinct from <code><token list></code> being empty or containing <code>\q_no_value</code> plus one or more other tokens).

18.4 Recursion

This module provides a uniform interface to intercepting and terminating loops as when one is doing tail recursion. The building blocks follow below and an example is shown in Section 18.4.1.

<code>\q_recursion_tail</code>	This quark is appended to the data structure in question and appears as a real element there. This means it gets any list separators around it.
--------------------------------	---

<code>\q_recursion_stop</code>	This quark is added <i>after</i> the data structure. Its purpose is to make it possible to terminate the recursion at any point easily.
--------------------------------	---

<code>\quark_if_recursion_tail_stop:N *</code>	<code>\quark_if_recursion_tail_stop:N <token></code>
--	--

Tests if $\langle token \rangle$ contains only the marker `\q_recursion_tail`, and if so uses `\use_none_delimit_by_q_recursion_stop:w` to terminate the recursion that this belongs to. The recursion input must include the marker tokens `\q_recursion_tail` and `\q_recursion_stop` as the last two items.

<code>\quark_if_recursion_tail_stop:n *</code>	<code>\quark_if_recursion_tail_stop:n {<token list>}</code>
<code>\quark_if_recursion_tail_stop:o *</code>	

Updated: 2011-09-06

Tests if the $\langle token list \rangle$ contains only `\q_recursion_tail`, and if so uses `\use_none_delimit_by_q_recursion_stop:w` to terminate the recursion that this belongs to. The recursion input must include the marker tokens `\q_recursion_tail` and `\q_recursion_stop` as the last two items.

<code>\quark_if_recursion_tail_stop_do:Nn *</code>	<code>\quark_if_recursion_tail_stop_do:Nn <token> {<insertion>}</code>
--	--

Tests if $\langle token \rangle$ contains only the marker `\q_recursion_tail`, and if so uses `\use_i_delimit_by_q_recursion_stop:w` to terminate the recursion that this belongs to. The recursion input must include the marker tokens `\q_recursion_tail` and `\q_recursion_stop` as the last two items. The $\langle insertion \rangle$ code is then added to the input stream after the recursion has ended.

<code>\quark_if_recursion_tail_stop_do:nn *</code>	<code>\quark_if_recursion_tail_stop_do:nn {<token list>} {<insertion>}</code>
<code>\quark_if_recursion_tail_stop_do:on *</code>	

Updated: 2011-09-06

Tests if the $\langle token list \rangle$ contains only `\q_recursion_tail`, and if so uses `\use_i_delimit_by_q_recursion_stop:w` to terminate the recursion that this belongs to. The recursion input must include the marker tokens `\q_recursion_tail` and `\q_recursion_stop` as the last two items. The $\langle insertion \rangle$ code is then added to the input stream after the recursion has ended.

```

\quark_if_recursion_tail_break:NN * \quark_if_recursion_tail_break:nN {\token list}
\quark_if_recursion_tail_break:nN * \langle type \rangle_map_break:

```

New: 2018-04-10

Tests if $\langle token list \rangle$ contains only `\q_recursion_tail`, and if so terminates the recursion using `\langle type \rangle_map_break:.` The recursion end should be marked by `\prg_break_point:Nn \langle type \rangle_map_break:.`

18.4.1 An example of recursion with quarks

Quarks are mainly used internally in the `expl3` code to define recursion functions such as `\tl_map_inline:nn` and so on. Here is a small example to demonstrate how to use quarks in this fashion. We shall define a command called `\my_map_dbl:nn` which takes a token list and applies an operation to every *pair* of tokens. For example, `\my_map_dbl:nn {abcd} {[--#1--#2--]~}` would produce “`[-a-b-] [-c-d-]`”. Using quarks to define such functions simplifies their logic and ensures robustness in many cases.

Here’s the definition of `\my_map_dbl:nn`. First of all, define the function that does the processing based on the inline function argument `#2`. Then initiate the recursion using an internal function. The token list `#1` is terminated using `\q_recursion_tail`, with delimiters according to the type of recursion (here a pair of `\q_recursion_tail`), concluding with `\q_recursion_stop`. These quarks are used to mark the end of the token list being operated upon.

```

\cs_new:Npn \my_map_dbl:nn #1#2
{
  \cs_set:Npn \__my_map_dbl_fn:nn ##1 ##2 {#2}
  \__my_map_dbl:nn #1 \q_recursion_tail \q_recursion_tail
  \q_recursion_stop
}

```

The definition of the internal recursion function follows. First check if either of the input tokens are the termination quarks. Then, if not, apply the inline function to the two arguments.

```

\cs_new:Nn \__my_map_dbl:nn
{
  \quark_if_recursion_tail_stop:n {#1}
  \quark_if_recursion_tail_stop:n {#2}
  \__my_map_dbl_fn:nn {#1} {#2}
}

```

Finally, recurse:

```

  \__my_map_dbl:nn
}

```

Note that contrarily to L^AT_EX3 built-in mapping functions, this mapping function cannot be nested, since the second map would overwrite the definition of `__my_map_dbl_fn:nn`.

18.5 Scan marks

Scan marks are control sequences set equal to `\scan_stop:`, hence never expand in an expansion context and are (largely) invisible if they are encountered in a typesetting context.

Like quarks, they can be used as delimiters in weird functions and are often safer to use for this purpose. Since they are harmless when executed by $\mathrm{T}_{\mathrm{E}}\mathrm{X}$ in non-expandable contexts, they can be used to mark the end of a set of instructions. This allows to skip to that point if the end of the instructions should not be performed (see `l3regex`).

<hr/> <code>\scan_new:N</code> <hr/>	<code>\scan_new:N</code> $\langle scan\ mark \rangle$
<hr/> <small>New: 2018-04-01</small> <hr/>	Creates a new $\langle scan\ mark \rangle$ which is set equal to <code>\scan_stop:</code> . The $\langle scan\ mark \rangle$ is defined globally, and an error message is raised if the name was already taken by another scan mark.

<hr/> <code>\s_stop</code> <hr/>	Used at the end of a set of instructions, as a marker that can be jumped to using <code>\use_none_delimit_by_s_stop:w</code> .
<hr/> <small>New: 2018-04-01</small> <hr/>	

<hr/> <code>\use_none_delimit_by_s_stop:w</code> ★ <hr/>	<code>\use_none_delimit_by_s_stop:w</code> $\langle tokens \rangle$ <code>\s_stop</code>
<hr/> <small>New: 2018-04-01</small> <hr/>	Removes the $\langle tokens \rangle$ and <code>\s_stop</code> from the input stream. This leads to a low-level $\mathrm{T}_{\mathrm{E}}\mathrm{X}$ error if <code>\s_stop</code> is absent.

Chapter 19

The l3seq package

Sequences and stacks

L^AT_EX3 implements a “sequence” data type, which contain an ordered list of entries which may contain any *balanced text*. It is possible to map functions to sequences such that the function is applied to every item in the sequence.

Sequences are also used to implement stack functions in L^AT_EX3. This is achieved using a number of dedicated stack functions.

19.1 Creating and initialising sequences

<code>\seq_new:N</code>
<code>\seq_new:c</code>

`\seq_new:N` $\langle sequence \rangle$

Creates a new $\langle sequence \rangle$ or raises an error if the name is already taken. The declaration is global. The $\langle sequence \rangle$ initially contains no items.

<code>\seq_clear:N</code>
<code>\seq_clear:c</code>
<code>\seq_gclear:N</code>
<code>\seq_gclear:c</code>

`\seq_clear:N` $\langle sequence \rangle$

Clears all items from the $\langle sequence \rangle$.

<code>\seq_clear_new:N</code>
<code>\seq_clear_new:c</code>
<code>\seq_gclear_new:N</code>
<code>\seq_gclear_new:c</code>

`\seq_clear_new:N` $\langle sequence \rangle$

Ensures that the $\langle sequence \rangle$ exists globally by applying `\seq_new:N` if necessary, then applies `\seq_(g)clear:N` to leave the $\langle sequence \rangle$ empty.

<code>\seq_set_eq:NN</code>
<code>\seq_set_eq:(cN Nc cc)</code>
<code>\seq_gset_eq:NN</code>
<code>\seq_gset_eq:(cN Nc cc)</code>

`\seq_set_eq:NN` $\langle sequence_1 \rangle$ $\langle sequence_2 \rangle$

Sets the content of $\langle sequence_1 \rangle$ equal to that of $\langle sequence_2 \rangle$.

<code>\seq_set_from_clist:NN</code>	<code>\seq_set_from_clist:NN <sequence> <comma-list></code>
<code>\seq_set_from_clist:(cN Nc cc)</code>	
<code>\seq_set_from_clist:Nn</code>	
<code>\seq_set_from_clist:cn</code>	
<code>\seq_gset_from_clist:NN</code>	
<code>\seq_gset_from_clist:(cN Nc cc)</code>	
<code>\seq_gset_from_clist:Nn</code>	
<code>\seq_gset_from_clist:cn</code>	

New: 2014-07-17

Converts the data in the *<comma list>* into a *<sequence>*: the original *<comma list>* is unchanged.

<code>\seq_const_from_clist:Nn</code>	<code>\seq_const_from_clist:Nn <seq var> {<comma-list>}</code>
<code>\seq_const_from_clist:cn</code>	

New: 2017-11-28

Creates a new constant *<seq var>* or raises an error if the name is already taken. The *<seq var>* is set globally to contain the items in the *<comma list>*.

<code>\seq_set_split:Nnn</code>	<code>\seq_set_split:Nnn <sequence> {<delimiter>} {<token list>}</code>
<code>\seq_set_split:NnV</code>	
<code>\seq_gset_split:Nnn</code>	
<code>\seq_gset_split:NnV</code>	

New: 2011-08-15

Updated: 2012-07-02

Splits the *<token list>* into *<items>* separated by *<delimiter>*, and assigns the result to the *<sequence>*. Spaces on both sides of each *<item>* are ignored, then one set of outer braces is removed (if any); this space trimming behaviour is identical to that of `l3clist` functions. Empty *<items>* are preserved by `\seq_set_split:Nnn`, and can be removed afterwards using `\seq_remove_all:Nn <sequence> {}`. The *<delimiter>* may not contain `{`, `}` or `#` (assuming \TeX 's normal category code régime). If the *<delimiter>* is empty, the *<token list>* is split into *<items>* as a *<token list>*. See also `\seq_set_split_keep_spaces:Nnn`, which omits space stripping.

<code>\seq_set_split_keep_spaces:Nnn</code>	<code>\seq_set_split_keep_spaces:Nnn <sequence> {<delimiter>} {<token list>}</code>
<code>\seq_set_split_keep_spaces:NnV</code>	
<code>\seq_gset_split_keep_spaces:Nnn</code>	
<code>\seq_gset_split_keep_spaces:NnV</code>	

New: 2021-03-24

Splits the *<token list>* into *<items>* separated by *<delimiter>*, and assigns the result to the *<sequence>*. One set of outer braces is removed (if any) but any surrounding spaces are retained: any braces *inside* one or more spaces are therefore kept. Empty *<items>* are preserved by `\seq_set_split_keep_spaces:Nnn`, and can be removed afterwards using `\seq_remove_all:Nn <sequence> {}`. The *<delimiter>* may not contain `{`, `}` or `#` (assuming \TeX 's normal category code régime). If the *<delimiter>* is empty, the *<token list>* is split into *<items>* as a *<token list>*. See also `\seq_set_split:Nnn`, which removes spaces around the delimiters.

<code>\seq_concat:NNN</code>	<code>\seq_concat:NNN <sequence₁₂₃</code>
<code>\seq_concat:ccc</code>	
<code>\seq_gconcat:NNN</code>	
<code>\seq_gconcat:ccc</code>	

Concatenates the content of *<sequence_{2 and *<sequence_{3 together and saves the result in *<sequence_{1. The items in *<sequence_{2 are placed at the left side of the new sequence.}*}*}*}*

<code>\seq_if_exist_p:N *</code>	<code>\seq_if_exist_p:N <sequence></code>
<code>\seq_if_exist_p:c *</code>	<code>\seq_if_exist:NTF <sequence> {\true code} {\false code}</code>
<code>\seq_if_exist:N\underline{TF} *</code>	Tests whether the $\langle sequence \rangle$ is currently defined. This does not check that the $\langle sequence \rangle$
<code>\seq_if_exist:c\underline{TF} *</code>	really is a sequence variable.

New: 2012-03-03

19.2 Appending data to sequences

<code>\seq_put_left:Nn</code>	<code>\seq_put_left:Nn <sequence> {\item}</code>
<code>\seq_put_left:(NV Nv No Nx cn cV cv co cx)</code>	
<code>\seq_gput_left:Nn</code>	
<code>\seq_gput_left:(NV Nv No Nx cn cV cv co cx)</code>	

Appends the $\langle item \rangle$ to the left of the $\langle sequence \rangle$.

<code>\seq_put_right:Nn</code>	<code>\seq_put_right:Nn <sequence> {\item}</code>
<code>\seq_put_right:(NV Nv No Nx cn cV cv co cx)</code>	
<code>\seq_gput_right:Nn</code>	
<code>\seq_gput_right:(NV Nv No Nx cn cV cv co cx)</code>	

Appends the $\langle item \rangle$ to the right of the $\langle sequence \rangle$.

19.3 Recovering items from sequences

Items can be recovered from either the left or the right of sequences. For implementation reasons, the actions at the left of the sequence are faster than those acting on the right. These functions all assign the recovered material locally, *i.e.* setting the $\langle token list variable \rangle$ used with `\tl_set:Nn` and *never* `\tl_gset:Nn`.

<code>\seq_get_left:NN</code>	<code>\seq_get_left:NN <sequence> <token list variable></code>
<code>\seq_get_left:cN</code>	Stores the left-most item from a $\langle sequence \rangle$ in the $\langle token list variable \rangle$ without removing it from the $\langle sequence \rangle$. The $\langle token list variable \rangle$ is assigned locally. If $\langle sequence \rangle$ is empty the $\langle token list variable \rangle$ is set to the special marker <code>\q_no_value</code> .

Updated: 2012-05-14

<code>\seq_get_right:NN</code>	<code>\seq_get_right:NN <sequence> <token list variable></code>
<code>\seq_get_right:cN</code>	Stores the right-most item from a $\langle sequence \rangle$ in the $\langle token list variable \rangle$ without removing it from the $\langle sequence \rangle$. The $\langle token list variable \rangle$ is assigned locally. If $\langle sequence \rangle$ is empty the $\langle token list variable \rangle$ is set to the special marker <code>\q_no_value</code> .

Updated: 2012-05-19

<code>\seq_pop_left:NN</code>	<code>\seq_pop_left:NN <sequence> <token list variable></code>
<code>\seq_pop_left:cN</code>	Pops the left-most item from a $\langle sequence \rangle$ into the $\langle token list variable \rangle$, <i>i.e.</i> removes the item from the sequence and stores it in the $\langle token list variable \rangle$. Both of the variables are assigned locally. If $\langle sequence \rangle$ is empty the $\langle token list variable \rangle$ is set to the special marker <code>\q_no_value</code> .

Updated: 2012-05-14

`\seq_gpop_left:NN`
`\seq_gpop_left:cN`
Updated: 2012-05-14

`\seq_gpop_left:NN` $\langle sequence \rangle$ $\langle token list variable \rangle$

Pops the left-most item from a $\langle sequence \rangle$ into the $\langle token list variable \rangle$, *i.e.* removes the item from the sequence and stores it in the $\langle token list variable \rangle$. The $\langle sequence \rangle$ is modified globally, while the assignment of the $\langle token list variable \rangle$ is local. If $\langle sequence \rangle$ is empty the $\langle token list variable \rangle$ is set to the special marker `\q_no_value`.

`\seq_pop_right:NN`
`\seq_pop_right:cN`
Updated: 2012-05-19

`\seq_pop_right:NN` $\langle sequence \rangle$ $\langle token list variable \rangle$

Pops the right-most item from a $\langle sequence \rangle$ into the $\langle token list variable \rangle$, *i.e.* removes the item from the sequence and stores it in the $\langle token list variable \rangle$. Both of the variables are assigned locally. If $\langle sequence \rangle$ is empty the $\langle token list variable \rangle$ is set to the special marker `\q_no_value`.

`\seq_gpop_right:NN`
`\seq_gpop_right:cN`
Updated: 2012-05-19

`\seq_gpop_right:NN` $\langle sequence \rangle$ $\langle token list variable \rangle$

Pops the right-most item from a $\langle sequence \rangle$ into the $\langle token list variable \rangle$, *i.e.* removes the item from the sequence and stores it in the $\langle token list variable \rangle$. The $\langle sequence \rangle$ is modified globally, while the assignment of the $\langle token list variable \rangle$ is local. If $\langle sequence \rangle$ is empty the $\langle token list variable \rangle$ is set to the special marker `\q_no_value`.

`\seq_item:Nn` ★
`\seq_item:cn` ★
New: 2014-07-17

`\seq_item:Nn` $\langle sequence \rangle$ $\{\langle integer expression \rangle\}$

Indexing items in the $\langle sequence \rangle$ from 1 at the top (left), this function evaluates the $\langle integer expression \rangle$ and leaves the appropriate item from the sequence in the input stream. If the $\langle integer expression \rangle$ is negative, indexing occurs from the bottom (right) of the sequence. If the $\langle integer expression \rangle$ is larger than the number of items in the $\langle sequence \rangle$ (as calculated by `\seq_count:N`) then the function expands to nothing.

TeXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the $\langle item \rangle$ does not expand further when appearing in an **x**-type or **e**-type argument expansion.

`\seq_rand_item:N` ★
`\seq_rand_item:c` ★
New: 2016-12-06

`\seq_rand_item:N` $\langle seq var \rangle$

Selects a pseudo-random item of the $\langle sequence \rangle$. If the $\langle sequence \rangle$ is empty the result is empty. This is not available in older versions of Xe_{La}TeX.

TeXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the $\langle item \rangle$ does not expand further when appearing in an **x**-type or **e**-type argument expansion.

19.4 Recovering values from sequences with branching

The functions in this section combine tests for non-empty sequences with recovery of an item from the sequence. They offer increased readability and performance over separate testing and recovery phases.

<hr/> <code>\seq_get_left:NNTF</code> <code>\seq_get_left:cNTF</code> <hr/> New: 2012-05-14 Updated: 2012-05-19 <hr/>	<code>\seq_get_left:NNTF <sequence> <token list variable> {<true code>} {<false code>}</code> <p>If the <i><sequence></i> is empty, leaves the <i><false code></i> in the input stream. The value of the <i><token list variable></i> is not defined in this case and should not be relied upon. If the <i><sequence></i> is non-empty, stores the left-most item from the <i><sequence></i> in the <i><token list variable></i> without removing it from the <i><sequence></i>, then leaves the <i><true code></i> in the input stream. The <i><token list variable></i> is assigned locally.</p>
<hr/> <code>\seq_get_right:NNTF</code> <code>\seq_get_right:cNTF</code> <hr/> New: 2012-05-19 <hr/>	<code>\seq_get_right:NNTF <sequence> <token list variable> {<true code>} {<false code>}</code> <p>If the <i><sequence></i> is empty, leaves the <i><false code></i> in the input stream. The value of the <i><token list variable></i> is not defined in this case and should not be relied upon. If the <i><sequence></i> is non-empty, stores the right-most item from the <i><sequence></i> in the <i><token list variable></i> without removing it from the <i><sequence></i>, then leaves the <i><true code></i> in the input stream. The <i><token list variable></i> is assigned locally.</p>
<hr/> <code>\seq_pop_left:NNTF</code> <code>\seq_pop_left:cNTF</code> <hr/> New: 2012-05-14 Updated: 2012-05-19 <hr/>	<code>\seq_pop_left:NNTF <sequence> <token list variable> {<true code>} {<false code>}</code> <p>If the <i><sequence></i> is empty, leaves the <i><false code></i> in the input stream. The value of the <i><token list variable></i> is not defined in this case and should not be relied upon. If the <i><sequence></i> is non-empty, pops the left-most item from the <i><sequence></i> in the <i><token list variable></i>, <i>i.e.</i> removes the item from the <i><sequence></i>, then leaves the <i><true code></i> in the input stream. Both the <i><sequence></i> and the <i><token list variable></i> are assigned locally.</p>
<hr/> <code>\seq_gpop_left:NNTF</code> <code>\seq_gpop_left:cNTF</code> <hr/> New: 2012-05-14 Updated: 2012-05-19 <hr/>	<code>\seq_gpop_left:NNTF <sequence> <token list variable> {<true code>} {<false code>}</code> <p>If the <i><sequence></i> is empty, leaves the <i><false code></i> in the input stream. The value of the <i><token list variable></i> is not defined in this case and should not be relied upon. If the <i><sequence></i> is non-empty, pops the left-most item from the <i><sequence></i> in the <i><token list variable></i>, <i>i.e.</i> removes the item from the <i><sequence></i>, then leaves the <i><true code></i> in the input stream. The <i><sequence></i> is modified globally, while the <i><token list variable></i> is assigned locally.</p>
<hr/> <code>\seq_pop_right:NNTF</code> <code>\seq_pop_right:cNTF</code> <hr/> New: 2012-05-19 <hr/>	<code>\seq_pop_right:NNTF <sequence> <token list variable> {<true code>} {<false code>}</code> <p>If the <i><sequence></i> is empty, leaves the <i><false code></i> in the input stream. The value of the <i><token list variable></i> is not defined in this case and should not be relied upon. If the <i><sequence></i> is non-empty, pops the right-most item from the <i><sequence></i> in the <i><token list variable></i>, <i>i.e.</i> removes the item from the <i><sequence></i>, then leaves the <i><true code></i> in the input stream. Both the <i><sequence></i> and the <i><token list variable></i> are assigned locally.</p>
<hr/> <code>\seq_gpop_right:NNTF</code> <code>\seq_gpop_right:cNTF</code> <hr/> New: 2012-05-19 <hr/>	<code>\seq_gpop_right:NNTF <sequence> <token list variable> {<true code>} {<false code>}</code> <p>If the <i><sequence></i> is empty, leaves the <i><false code></i> in the input stream. The value of the <i><token list variable></i> is not defined in this case and should not be relied upon. If the <i><sequence></i> is non-empty, pops the right-most item from the <i><sequence></i> in the <i><token list variable></i>, <i>i.e.</i> removes the item from the <i><sequence></i>, then leaves the <i><true code></i> in the input stream. The <i><sequence></i> is modified globally, while the <i><token list variable></i> is assigned locally.</p>

19.5 Modifying sequences

While sequences are normally used as ordered lists, it may be necessary to modify the content. The functions here may be used to update sequences, while retaining the order of the unaffected entries.

```
\seq_remove_duplicates:N
\seq_remove_duplicates:c
\seq_gremove_duplicates:N
\seq_gremove_duplicates:c
```

`\seq_remove_duplicates:N` $\langle sequence \rangle$

Removes duplicate items from the $\langle sequence \rangle$, leaving the left most copy of each item in the $\langle sequence \rangle$. The $\langle item \rangle$ comparison takes place on a token basis, as for `\tl_if_eq:nnTF`.

TeXhackers note: This function iterates through every item in the $\langle sequence \rangle$ and does a comparison with the $\langle items \rangle$ already checked. It is therefore relatively slow with large sequences.

```
\seq_remove_all:Nn
\seq_remove_all:cn
\seq_gremove_all:Nn
\seq_gremove_all:cn
```

`\seq_remove_all:Nn` $\langle sequence \rangle$ $\{\langle item \rangle\}$

Removes every occurrence of $\langle item \rangle$ from the $\langle sequence \rangle$. The $\langle item \rangle$ comparison takes place on a token basis, as for `\tl_if_eq:nnTF`.

```
\seq_reverse:N
\seq_reverse:c
\seq_greverse:N
\seq_greverse:c
```

`\seq_reverse:N` $\langle sequence \rangle$

Reverses the order of the items stored in the $\langle sequence \rangle$.

New: 2014-07-18

```
\seq_sort:Nn
\seq_sort:cn
\seq_gsort:Nn
\seq_gsort:cn
```

`\seq_sort:Nn` $\langle sequence \rangle$ $\{\langle comparison code \rangle\}$

Sorts the items in the $\langle sequence \rangle$ according to the $\langle comparison code \rangle$, and assigns the result to $\langle sequence \rangle$. The details of sorting comparison are described in Section 6.1.

New: 2017-02-06

```
\seq_shuffle:N
\seq_shuffle:c
\seq_gshuffle:N
\seq_gshuffle:c
```

`\seq_shuffle:N` $\langle seq var \rangle$

Sets the $\langle seq var \rangle$ to the result of placing the items of the $\langle seq var \rangle$ in a random order. Each item is (roughly) as likely to end up in any given position.

TeXhackers note: For sequences with more than 13 items or so, only a small proportion of all possible permutations can be reached, because the random seed `\sys_rand_seed` only has 28-bits. The use of `\toks` internally means that sequences with more than 32767 or 65535 items (depending on the engine) cannot be shuffled.

New: 2018-04-29

19.6 Sequence conditionals

```
\seq_if_empty_p:N *
\seq_if_empty_p:c *
\seq_if_empty:NTF *
\seq_if_empty:cTF *
```

`\seq_if_empty_p:N` $\langle sequence \rangle$

`\seq_if_empty:NNTF` $\langle sequence \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Tests if the $\langle sequence \rangle$ is empty (containing no items).

<code>\seq_if_in:NnTF</code>	<code>\seq_if_in:NnTF <sequence> {(item)} {(true code)} {(false code)}</code>
<code>\seq_if_in:(NV Nv No Nx cn cV cv co cx)TF</code>	

Tests if the $\langle item \rangle$ is present in the $\langle sequence \rangle$.

19.7 Mapping over sequences

All mappings are done at the current group level, *i.e.* any local assignments made by the $\langle function \rangle$ or $\langle code \rangle$ discussed below remain in effect after the loop.

<code>\seq_map_function:Nn ☆</code>
<code>\seq_map_function:cn ☆</code>
Updated: 2012-06-29

`\seq_map_function:Nn <sequence> <function>`

Applies $\langle function \rangle$ to every $\langle item \rangle$ stored in the $\langle sequence \rangle$. The $\langle function \rangle$ will receive one argument for each iteration. The $\langle items \rangle$ are returned from left to right. To pass further arguments to the $\langle function \rangle$, see `\seq_map_tokens:Nn`. The function `\seq_map_inline:Nn` is faster than `\seq_map_function:Nn` for sequences with more than about 10 items.

<code>\seq_map_inline:Nn</code>
<code>\seq_map_inline:cn</code>
Updated: 2012-06-29

`\seq_map_inline:Nn <sequence> {(inline function)}`

Applies $\langle inline function \rangle$ to every $\langle item \rangle$ stored within the $\langle sequence \rangle$. The $\langle inline function \rangle$ should consist of code which will receive the $\langle item \rangle$ as #1. The $\langle items \rangle$ are returned from left to right.

<code>\seq_map_tokens:Nn ☆</code>
<code>\seq_map_tokens:cn ☆</code>
New: 2019-08-30

`\seq_map_tokens:Nn <sequence> {(code)}`

Analogue of `\seq_map_function:Nn` which maps several tokens instead of a single function. The $\langle code \rangle$ receives each item in the $\langle sequence \rangle$ as a trailing brace group. For instance,

`\seq_map_tokens:Nn \l_my_seq { \prg_replicate:nn { 2 } }`

expands to twice each item in the $\langle sequence \rangle$: for each item in `\l_my_seq` the function `\prg_replicate:nn` receives 2 and $\langle item \rangle$ as its two arguments. The function `\seq_map_inline:Nn` is typically faster but it is not expandable.

<code>\seq_map_variable:NNn</code>
<code>\seq_map_variable:(Ncn cNn ccn)</code>
Updated: 2012-06-29

`\seq_map_variable:NNn <sequence> <variable> {(code)}`

Stores each $\langle item \rangle$ of the $\langle sequence \rangle$ in turn in the (token list) $\langle variable \rangle$ and applies the $\langle code \rangle$. The $\langle code \rangle$ will usually make use of the $\langle variable \rangle$, but this is not enforced. The assignments to the $\langle variable \rangle$ are local. Its value after the loop is the last $\langle item \rangle$ in the $\langle sequence \rangle$, or its original value if the $\langle sequence \rangle$ is empty. The $\langle items \rangle$ are returned from left to right.

<code>\seq_map_indexed_function:Nn ☆</code>
New: 2018-05-03

`\seq_map_indexed_function:Nn <seq var> <function>`

Applies $\langle function \rangle$ to every entry in the $\langle sequence variable \rangle$. The $\langle function \rangle$ should have signature `:nn`. It receives two arguments for each iteration: the $\langle index \rangle$ (namely 1 for the first entry, then 2 and so on) and the $\langle item \rangle$.

`\seq_map_indexed_inline:Nn`

New: 2018-05-03

`\seq_map_indexed_inline:Nn` $\langle seq\ var \rangle$ $\{\langle inline\ function \rangle\}$

Applies $\langle inline\ function \rangle$ to every entry in the $\langle sequence\ variable \rangle$. The $\langle inline\ function \rangle$ should consist of code which receives the $\langle index \rangle$ (namely 1 for the first entry, then 2 and so on) as $\#1$ and the $\langle item \rangle$ as $\#2$.

`\seq_map_break:` ☆

Updated: 2012-06-29

`\seq_map_break:`

Used to terminate a `\seq_map...` function before all entries in the $\langle sequence \rangle$ have been processed. This normally takes place within a conditional statement, for example

```
\seq_map_inline:Nn \l_my_seq
{
  \str_if_eq:nnTF { #1 } { bingo }
  { \seq_map_break: }
  {
    % Do something useful
  }
}
```

Use outside of a `\seq_map...` scenario leads to low level T_EX errors.

T_EXhackers note: When the mapping is broken, additional tokens may be inserted before further items are taken from the input stream. This depends on the design of the mapping function.

`\seq_map_break:n` ☆

Updated: 2012-06-29

`\seq_map_break:n` $\{\langle code \rangle\}$

Used to terminate a `\seq_map...` function before all entries in the $\langle sequence \rangle$ have been processed, inserting the $\langle code \rangle$ after the mapping has ended. This normally takes place within a conditional statement, for example

```
\seq_map_inline:Nn \l_my_seq
{
  \str_if_eq:nnTF { #1 } { bingo }
  { \seq_map_break:n { <code> } }
  {
    % Do something useful
  }
}
```

Use outside of a `\seq_map...` scenario leads to low level T_EX errors.

T_EXhackers note: When the mapping is broken, additional tokens may be inserted before the $\langle code \rangle$ is inserted into the input stream. This depends on the design of the mapping function.

`\seq_set_map:NNn`
`\seq_gset_map:NNn`

New: 2011-12-22
Updated: 2020-07-16

`\seq_set_map:NNn` $\langle sequence_1 \rangle$ $\langle sequence_2 \rangle$ $\{ \langle inline function \rangle \}$

Applies $\langle inline function \rangle$ to every $\langle item \rangle$ stored within the $\langle sequence_2 \rangle$. The $\langle inline function \rangle$ should consist of code which will receive the $\langle item \rangle$ as #1. The sequence resulting applying $\langle inline function \rangle$ to each $\langle item \rangle$ is assigned to $\langle sequence_1 \rangle$.

T_EXhackers note: Contrarily to other mapping functions, `\seq_map_break:` cannot be used in this function, and would lead to low-level T_EX errors.

`\seq_set_map_x:NNn`
`\seq_gset_map_x:NNn`

New: 2020-07-16

`\seq_set_map_x:NNn` $\langle sequence_1 \rangle$ $\langle sequence_2 \rangle$ $\{ \langle inline function \rangle \}$

Applies $\langle inline function \rangle$ to every $\langle item \rangle$ stored within the $\langle sequence_2 \rangle$. The $\langle inline function \rangle$ should consist of code which will receive the $\langle item \rangle$ as #1. The sequence resulting from x-expanding $\langle inline function \rangle$ applied to each $\langle item \rangle$ is assigned to $\langle sequence_1 \rangle$. As such, the code in $\langle inline function \rangle$ should be expandable.

T_EXhackers note: Contrarily to other mapping functions, `\seq_map_break:` cannot be used in this function, and would lead to low-level T_EX errors.

`\seq_count:N` ★
`\seq_count:c` ★

New: 2012-07-13

`\seq_count:N` $\langle sequence \rangle$

Leaves the number of items in the $\langle sequence \rangle$ in the input stream as an $\langle integer denotation \rangle$. The total number of items in a $\langle sequence \rangle$ includes those which are empty and duplicates, *i.e.* every item in a $\langle sequence \rangle$ is unique.

19.8 Using the content of sequences directly

`\seq_use:Nnnn` ★
`\seq_use:cnnn` ★

New: 2013-05-26

`\seq_use:Nnnn` $\langle seq var \rangle$ $\{ \langle separator between two \rangle \}$
 $\{ \langle separator between more than two \rangle \} \{ \langle separator between final two \rangle \}$

Places the contents of the $\langle seq var \rangle$ in the input stream, with the appropriate $\langle separator \rangle$ between the items. Namely, if the sequence has more than two items, the $\langle separator between more than two \rangle$ is placed between each pair of items except the last, for which the $\langle separator between final two \rangle$ is used. If the sequence has exactly two items, then they are placed in the input stream separated by the $\langle separator between two \rangle$. If the sequence has a single item, it is placed in the input stream, and an empty sequence produces no output. An error is raised if the variable does not exist or if it is invalid.

For example,

```
\seq_set_split:Nnn \l_tmpa_seq { | } { a | b | c | {de} | f }
\seq_use:Nnnn \l_tmpa_seq { ~and~ } { ,~ } { ,~and~ }
```

inserts “a, b, c, de, and f” in the input stream. The first separator argument is not used in this case because the sequence has more than 2 items.

T_EXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the $\langle items \rangle$ do not expand further when appearing in an x-type or e-type argument expansion.

`\seq_use:Nn` ★
`\seq_use:cn` ★

New: 2013-05-26

`\seq_use:Nn` $\langle seq\ var \rangle$ $\{\langle separator \rangle\}$

Places the contents of the $\langle seq\ var \rangle$ in the input stream, with the $\langle separator \rangle$ between the items. If the sequence has a single item, it is placed in the input stream with no $\langle separator \rangle$, and an empty sequence produces no output. An error is raised if the variable does not exist or if it is invalid.

For example,

```
\seq_set_split:Nnn \l_tmpa_seq { | } { a | b | c | {de} | f }
\seq_use:Nn \l_tmpa_seq { ~and~ }
```

inserts “a and b and c and de and f” in the input stream.

TpXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the $\langle items \rangle$ do not expand further when appearing in an `x`-type or `e`-type argument expansion.

19.9 Sequences as stacks

Sequences can be used as stacks, where data is pushed to and popped from the top of the sequence. (The left of a sequence is the top, for performance reasons.) The stack functions for sequences are not intended to be mixed with the general ordered data functions detailed in the previous section: a sequence should either be used as an ordered data type or as a stack, but not in both ways.

`\seq_get:NN`
`\seq_get:cn`

Updated: 2012-05-14

`\seq_get:NN` $\langle sequence \rangle$ $\langle token\ list\ variable \rangle$

Reads the top item from a $\langle sequence \rangle$ into the $\langle token\ list\ variable \rangle$ without removing it from the $\langle sequence \rangle$. The $\langle token\ list\ variable \rangle$ is assigned locally. If $\langle sequence \rangle$ is empty the $\langle token\ list\ variable \rangle$ is set to the special marker `\q_no_value`.

`\seq_pop:NN`
`\seq_pop:cn`

Updated: 2012-05-14

`\seq_pop:NN` $\langle sequence \rangle$ $\langle token\ list\ variable \rangle$

Pops the top item from a $\langle sequence \rangle$ into the $\langle token\ list\ variable \rangle$. Both of the variables are assigned locally. If $\langle sequence \rangle$ is empty the $\langle token\ list\ variable \rangle$ is set to the special marker `\q_no_value`.

`\seq_gpop:NN`
`\seq_gpop:cn`

Updated: 2012-05-14

`\seq_gpop:NN` $\langle sequence \rangle$ $\langle token\ list\ variable \rangle$

Pops the top item from a $\langle sequence \rangle$ into the $\langle token\ list\ variable \rangle$. The $\langle sequence \rangle$ is modified globally, while the $\langle token\ list\ variable \rangle$ is assigned locally. If $\langle sequence \rangle$ is empty the $\langle token\ list\ variable \rangle$ is set to the special marker `\q_no_value`.

`\seq_get:NNTF`
`\seq_get:cNTF`

New: 2012-05-14
Updated: 2012-05-19

`\seq_get:NNTF` $\langle sequence \rangle$ $\langle token\ list\ variable \rangle$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$

If the $\langle sequence \rangle$ is empty, leaves the $\langle false\ code \rangle$ in the input stream. The value of the $\langle token\ list\ variable \rangle$ is not defined in this case and should not be relied upon. If the $\langle sequence \rangle$ is non-empty, stores the top item from a $\langle sequence \rangle$ in the $\langle token\ list\ variable \rangle$ without removing it from the $\langle sequence \rangle$. The $\langle token\ list\ variable \rangle$ is assigned locally.

`\seq_pop:NNTF`
`\seq_pop:cNTF`

New: 2012-05-14
Updated: 2012-05-19

`\seq_pop:NNTF` $\langle sequence \rangle$ $\langle token list variable \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

If the $\langle sequence \rangle$ is empty, leaves the $\langle false code \rangle$ in the input stream. The value of the $\langle token list variable \rangle$ is not defined in this case and should not be relied upon. If the $\langle sequence \rangle$ is non-empty, pops the top item from the $\langle sequence \rangle$ in the $\langle token list variable \rangle$, *i.e.* removes the item from the $\langle sequence \rangle$. Both the $\langle sequence \rangle$ and the $\langle token list variable \rangle$ are assigned locally.

`\seq_gpop:NNTF`
`\seq_gpop:cNTF`

New: 2012-05-14
Updated: 2012-05-19

`\seq_gpop:NNTF` $\langle sequence \rangle$ $\langle token list variable \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

If the $\langle sequence \rangle$ is empty, leaves the $\langle false code \rangle$ in the input stream. The value of the $\langle token list variable \rangle$ is not defined in this case and should not be relied upon. If the $\langle sequence \rangle$ is non-empty, pops the top item from the $\langle sequence \rangle$ in the $\langle token list variable \rangle$, *i.e.* removes the item from the $\langle sequence \rangle$. The $\langle sequence \rangle$ is modified globally, while the $\langle token list variable \rangle$ is assigned locally.

`\seq_push:Nn`
`\seq_push:(NV|Nv|No|Nx|cn|cV|cv|co|cx)`
`\seq_gpush:Nn`
`\seq_gpush:(NV|Nv|No|Nx|cn|cV|cv|co|cx)`

`\seq_push:Nn` $\langle sequence \rangle$ $\{\langle item \rangle\}$

Adds the $\{\langle item \rangle\}$ to the top of the $\langle sequence \rangle$.

19.10 Sequences as sets

Sequences can also be used as sets, such that all of their items are distinct. Usage of sequences as sets is not currently widespread, hence no specific set function is provided. Instead, it is explained here how common set operations can be performed by combining several functions described in earlier sections. When using sequences to implement sets, one should be careful not to rely on the order of items in the sequence representing the set.

Sets should not contain several occurrences of a given item. To make sure that a $\langle sequence variable \rangle$ only has distinct items, use `\seq_remove_duplicates:N` $\langle sequence variable \rangle$. This function is relatively slow, and to avoid performance issues one should only use it when necessary.

Some operations on a set $\langle seq var \rangle$ are straightforward. For instance, `\seq_count:N` $\langle seq var \rangle$ expands to the number of items, while `\seq_if_in:NnTF` $\langle seq var \rangle$ $\{\langle item \rangle\}$ tests if the $\langle item \rangle$ is in the set.

Adding an $\langle item \rangle$ to a set $\langle seq var \rangle$ can be done by appending it to the $\langle seq var \rangle$ if it is not already in the $\langle seq var \rangle$:

`\seq_if_in:NnF` $\langle seq var \rangle$ $\{\langle item \rangle\}$
`{ \seq_put_right:Nn` $\langle seq var \rangle$ $\{\langle item \rangle\}$ `}`

Removing an $\langle item \rangle$ from a set $\langle seq var \rangle$ can be done using `\seq_remove_all:Nn`,

`\seq_remove_all:Nn` $\langle seq var \rangle$ $\{\langle item \rangle\}$

The intersection of two sets $\langle seq var_1 \rangle$ and $\langle seq var_2 \rangle$ can be stored into $\langle seq var_3 \rangle$ by collecting items of $\langle seq var_1 \rangle$ which are in $\langle seq var_2 \rangle$.

```

\seq_clear:N <seq var3>
\seq_map_inline:Nn <seq var1>
{
\seq_if_in:NnT <seq var2> {#1}
{ \seq_put_right:Nn <seq var3> {#1} }
}

```

The code as written here only works if $\langle seq\ var_3 \rangle$ is different from the other two sequence variables. To cover all cases, items should first be collected in a sequence $\backslash l_ \langle pkg \rangle_internal_seq$, then $\langle seq\ var_3 \rangle$ should be set equal to this internal sequence. The same remark applies to other set functions.

The union of two sets $\langle seq\ var_1 \rangle$ and $\langle seq\ var_2 \rangle$ can be stored into $\langle seq\ var_3 \rangle$ through

```

\seq_concat:NNN <seq var3> <seq var1> <seq var2>
\seq_remove_duplicates:N <seq var3>

```

or by adding items to (a copy of) $\langle seq\ var_1 \rangle$ one by one

```

\seq_set_eq:NN <seq var3> <seq var1>
\seq_map_inline:Nn <seq var2>
{
\seq_if_in:NnF <seq var3> {#1}
{ \seq_put_right:Nn <seq var3> {#1} }
}

```

The second approach is faster than the first when the $\langle seq\ var_2 \rangle$ is short compared to $\langle seq\ var_1 \rangle$.

The difference of two sets $\langle seq\ var_1 \rangle$ and $\langle seq\ var_2 \rangle$ can be stored into $\langle seq\ var_3 \rangle$ by removing items of the $\langle seq\ var_2 \rangle$ from (a copy of) the $\langle seq\ var_1 \rangle$ one by one.

```

\seq_set_eq:NN <seq var3> <seq var1>
\seq_map_inline:Nn <seq var2>
{ \seq_remove_all:Nn <seq var3> {#1} }

```

The symmetric difference of two sets $\langle seq\ var_1 \rangle$ and $\langle seq\ var_2 \rangle$ can be stored into $\langle seq\ var_3 \rangle$ by computing the difference between $\langle seq\ var_1 \rangle$ and $\langle seq\ var_2 \rangle$ and storing the result as $\backslash l_ \langle pkg \rangle_internal_seq$, then the difference between $\langle seq\ var_2 \rangle$ and $\langle seq\ var_1 \rangle$, and finally concatenating the two differences to get the symmetric differences.

```

\seq_set_eq:NN \l\_ \langle pkg \rangle\_internal\_seq <seq var1>
\seq_map_inline:Nn <seq var2>
{ \seq_remove_all:Nn \l\_ \langle pkg \rangle\_internal\_seq {#1} }
\seq_set_eq:NN <seq var3> <seq var2>
\seq_map_inline:Nn <seq var1>
{ \seq_remove_all:Nn <seq var3> {#1} }
\seq_concat:NNN <seq var3> <seq var3> \l\_ \langle pkg \rangle\_internal\_seq

```

19.11 Constant and scratch sequences

$\backslash c_empty_seq$ Constant that is always empty.

New: 2012-07-02

<code>\l_tmpa_seq</code>	Scratch sequences for local assignment. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
<code>\l_tmpb_seq</code>	
New: 2012-04-26	

<code>\g_tmpa_seq</code>	Scratch sequences for global assignment. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
<code>\g_tmpb_seq</code>	
New: 2012-04-26	

19.12 Viewing sequences

<code>\seq_show:N</code>	<code>\seq_show:N</code> <i><sequence></i>
<code>\seq_show:c</code>	
Updated: 2021-04-29	Displays the entries in the <i><sequence></i> in the terminal.

<code>\seq_log:N</code>	<code>\seq_log:N</code> <i><sequence></i>
<code>\seq_log:c</code>	
New: 2014-08-12	Writes the entries in the <i><sequence></i> in the log file.
Updated: 2021-04-29	

Chapter 20

The l3int package

Integers

Calculation and comparison of integer values can be carried out using literal numbers, `int` registers, constants and integers stored in token list variables. The standard operators `+`, `-`, `/` and `*` and parentheses can be used within such expressions to carry arithmetic operations. This module carries out these functions on *integer expressions* (“`intexpr`”).

20.1 Integer expressions

Throughout this module, (almost) all `n`-type argument allow for an *⟨intexpr⟩* argument with the following syntax. The *⟨integer expression⟩* should consist, after expansion, of `+`, `-`, `*`, `/`, `(`, `)` and of course integer operands. The result is calculated by applying standard mathematical rules with the following peculiarities:

- `/` denotes division rounded to the closest integer with ties rounded away from zero;
- there is an error and the overall expression evaluates to zero whenever the absolute value of any intermediate result exceeds $2^{31} - 1$, except in the case of scaling operations $a*b/c$, for which $a*b$ may be arbitrarily large (but the operands a , b , c are still constrained to an absolute value at most $2^{31} - 1$);
- parentheses may not appear after unary `+` or `-`, namely placing `+(` or `-(` at the start of an expression or after `+`, `-`, `*`, `/` or `(` leads to an error.

Each integer operand can be either an integer variable (with no need for `\int_use:N`) or an integer denotation. For example both

```
\int_show:n { 5 + 4 * 3 - ( 3 + 4 * 5 ) }
```

and

```
\tl_new:N \l_my_tl  
\tl_set:Nn \l_my_tl { 5 }  
\int_new:N \l_my_int  
\int_set:Nn \l_my_int { 4 }  
\int_show:n { \l_my_tl + \l_my_int * 3 - ( 3 + 4 * 5 ) }
```

show the same result -6 because `\l_my_tl` expands to the integer denotation 5 while the integer variable `\l_my_int` takes the value 4 . As the *integer expression* is fully expanded from left to right during evaluation, fully expandable and restricted-expandable functions can both be used, and `\exp_not:n` and its variants have no effect while `\exp_not:N` may incorrectly interrupt the expression.

T_EXhackers note: Exactly two expansions are needed to evaluate `\int_eval:n`. The result is *not* an *internal integer*, and therefore should be terminated by a space if used in `\int_value:w` or in a T_EX-style integer assignment.

As all T_EX integers, integer operands can also be: `\value{<LATEX 2ε counter>}`; dimension or skip variables, converted to integers in `sp`; the character code of some character given as `'<char>` or `\<char>`; octal numbers given as `'` followed by digits from 0 to 7 ; or hexadecimal numbers given as `"` followed by digits and upper case letters from A to F .

<code>\int_eval:n</code> ★	<code>\int_eval:n {<integer expression>}</code>
----------------------------	---

Evaluates the *integer expression* and leaves the result in the input stream as an integer denotation: for positive results an explicit sequence of decimal digits not starting with 0 , for negative results $-$ followed by such a sequence, and 0 for zero.

<code>\int_eval:w</code> ★	<code>\int_eval:w <integer expression></code>
----------------------------	---

New: 2018-03-30

Evaluates the *integer expression* as described for `\int_eval:n`. The end of the expression is the first token encountered that cannot form part of such an expression. If that token is `\scan_stop`: it is removed, otherwise not. Spaces do *not* terminate the expression. However, spaces terminate explicit integers, and this may terminate the expression: for instance, `\int_eval:w 1_+1_9` (with explicit space tokens inserted using `~` in a code setting) expands to 29 since the digit 9 is not part of the expression.

<code>\int_sign:n</code> ★	<code>\int_sign:n {<intexpr>}</code>
----------------------------	--

New: 2018-11-03

Evaluates the *integer expression* then leaves 1 or 0 or -1 in the input stream according to the sign of the result.

<code>\int_abs:n</code> ★	<code>\int_abs:n {<integer expression>}</code>
---------------------------	--

Updated: 2012-09-26

Evaluates the *integer expression* as described for `\int_eval:n` and leaves the absolute value of the result in the input stream as an *integer denotation* after two expansions.

<code>\int_div_round:nn</code> ★	<code>\int_div_round:nn {<intexpr₁>} {<intexpr₂>}</code>
----------------------------------	--

Updated: 2012-09-26

Evaluates the two *integer expressions* as described earlier, then divides the first value by the second, and rounds the result to the closest integer. Ties are rounded away from zero. Note that this is identical to using `/` directly in an *integer expression*. The result is left in the input stream as an *integer denotation* after two expansions.

<code>\int_div_truncate:nn</code> ★	<code>\int_div_truncate:nn {<intexpr₁>} {<intexpr₂>}</code>
-------------------------------------	---

Updated: 2012-02-09

Evaluates the two *integer expressions* as described earlier, then divides the first value by the second, and rounds the result towards zero. Note that division using `/` rounds to the closest integer instead. The result is left in the input stream as an *integer denotation* after two expansions.

<code>\int_max:nn</code>	★	<code>\int_max:nn {\langle integer \rangle_1} {\langle integer \rangle_2}</code>
<code>\int_min:nn</code>	★	<code>\int_min:nn {\langle integer \rangle_1} {\langle integer \rangle_2}</code>
Updated: 2012-09-26		Evaluates the $\langle integer expressions \rangle$ as described for <code>\int_eval:n</code> and leaves either the larger or smaller value in the input stream as an $\langle integer denotation \rangle$ after two expansions.

<code>\int_mod:nn</code>	★	<code>\int_mod:nn {\langle integer \rangle_1} {\langle integer \rangle_2}</code>
Updated: 2012-09-26		Evaluates the two $\langle integer expressions \rangle$ as described earlier, then calculates the integer remainder of dividing the first expression by the second. This is obtained by subtracting <code>\int_div_truncate:nn {\langle integer \rangle_1} {\langle integer \rangle_2}</code> times $\langle integer \rangle_2$ from $\langle integer \rangle_1$. Thus, the result has the same sign as $\langle integer \rangle_1$ and its absolute value is strictly less than that of $\langle integer \rangle_2$. The result is left in the input stream as an $\langle integer denotation \rangle$ after two expansions.

20.2 Creating and initialising integers

<code>\int_new:N</code>	<code>\int_new:N \langle integer \rangle</code>
<code>\int_new:c</code>	Creates a new $\langle integer \rangle$ or raises an error if the name is already taken. The declaration is global. The $\langle integer \rangle$ is initially equal to 0.

<code>\int_const:Nn</code>	<code>\int_const:Nn \langle integer \rangle {\langle integer expression \rangle}</code>
<code>\int_const:cn</code>	Creates a new constant $\langle integer \rangle$ or raises an error if the name is already taken. The value of the $\langle integer \rangle$ is set globally to the $\langle integer expression \rangle$.
Updated: 2011-10-22	

<code>\int_zero:N</code>	<code>\int_zero:N \langle integer \rangle</code>
<code>\int_zero:c</code>	Sets $\langle integer \rangle$ to 0.
<code>\int_gzero:N</code>	
<code>\int_gzero:c</code>	

<code>\int_zero_new:N</code>	<code>\int_zero_new:N \langle integer \rangle</code>
<code>\int_zero_new:c</code>	Ensures that the $\langle integer \rangle$ exists globally by applying <code>\int_new:N</code> if necessary, then applies <code>\int_(g)zero:N</code> to leave the $\langle integer \rangle$ set to zero.
<code>\int_gzero_new:N</code>	
<code>\int_gzero_new:c</code>	

New: 2011-12-13

<code>\int_set_eq:NN</code>	<code>\int_set_eq:NN \langle integer_1 \rangle \langle integer_2 \rangle</code>
<code>\int_set_eq:(cN Nc cc)</code>	Sets the content of $\langle integer_1 \rangle$ equal to that of $\langle integer_2 \rangle$.
<code>\int_gset_eq:NN</code>	
<code>\int_gset_eq:(cN Nc cc)</code>	

<code>\int_if_exist_p:N</code>	★	<code>\int_if_exist_p:N \langle int \rangle</code>
<code>\int_if_exist_p:c</code>	★	<code>\int_if_exist:NTF \langle int \rangle {\langle true code \rangle} {\langle false code \rangle}</code>
<code>\int_if_exist:N\overline{TF}</code>	★	Tests whether the $\langle int \rangle$ is currently defined. This does not check that the $\langle int \rangle$ really is an integer variable.
<code>\int_if_exist:c\overline{TF}</code>	★	

New: 2012-03-03

20.3 Setting and incrementing integers

<hr/>	
<code>\int_add:Nn</code>	<code>\int_add:Nn <integer> {<integer expression>}</code>
<code>\int_add:cn</code>	
<code>\int_gadd:Nn</code>	Adds the result of the <i><integer expression></i> to the current content of the <i><integer></i> .
<code>\int_gadd:cn</code>	
<hr/>	
Updated: 2011-10-22	
<hr/>	
<code>\int_decr:N</code>	<code>\int_decr:N <integer></code>
<code>\int_decr:c</code>	
<code>\int_gdecr:N</code>	Decreases the value stored in <i><integer></i> by 1.
<code>\int_gdecr:c</code>	
<hr/>	
<code>\int_incr:N</code>	<code>\int_incr:N <integer></code>
<code>\int_incr:c</code>	
<code>\int_gincr:N</code>	Increases the value stored in <i><integer></i> by 1.
<code>\int_gincr:c</code>	
<hr/>	
<code>\int_set:Nn</code>	<code>\int_set:Nn <integer> {<integer expression>}</code>
<code>\int_set:cn</code>	
<code>\int_gset:Nn</code>	Sets <i><integer></i> to the value of <i><integer expression></i> , which must evaluate to an integer (as described for <code>\int_eval:n</code>).
<code>\int_gset:cn</code>	
<hr/>	
Updated: 2011-10-22	
<hr/>	
<code>\int_sub:Nn</code>	<code>\int_sub:Nn <integer> {<integer expression>}</code>
<code>\int_sub:cn</code>	
<code>\int_gsub:Nn</code>	Subtracts the result of the <i><integer expression></i> from the current content of the <i><integer></i> .
<code>\int_gsub:cn</code>	
<hr/>	
Updated: 2011-10-22	

20.4 Using integers

<hr/>	
<code>\int_use:N</code> ★	<code>\int_use:N <integer></code>
<code>\int_use:c</code> ★	
<hr/>	
Updated: 2011-10-22	
<hr/>	

Recovers the content of an *<integer>* and places it directly in the input stream. An error is raised if the variable does not exist or if it is invalid. Can be omitted in places where an *<integer>* is required (such as in the first and third arguments of `\int_compare:nNnTF`).

TeXhackers note: `\int_use:N` is the TeX primitive `\the`: this is one of several L^AT_EX3 names for this primitive.

20.5 Integer expression conditionals

```
\int_compare_p:nNn * \int_compare_p:nNn {<intexpr1>} <relation> {<intexpr2>}
\int_compare:nNnTF * \int_compare:nNnTF
                        {<intexpr1>} <relation> {<intexpr2>}
                        {<true code>} {<false code>}
```

This function first evaluates each of the *<integer expressions>* as described for `\int_eval:n`. The two results are then compared using the *<relation>*:

Equal	=
Greater than	>
Less than	<

This function is less flexible than `\int_compare:nTF` but around 5 times faster.

```
\int_compare_p:n * \int_compare_p:n
\int_compare:nTF * {
                    <intexpr1> <relation1>
                    ...
                    <intexprN> <relationN>
                    <intexprN+1>
                }
\int_compare:nTF {
                <intexpr1> <relation1>
                ...
                <intexprN> <relationN>
                <intexprN+1>
            }
                {<true code>} {<false code>}
```

Updated: 2013-01-13

This function evaluates the *<integer expressions>* as described for `\int_eval:n` and compares consecutive result using the corresponding *<relation>*, namely it compares *<intexpr₁>* and *<intexpr₂>* using the *<relation₁>*, then *<intexpr₂>* and *<intexpr₃>* using the *<relation₂>*, until finally comparing *<intexpr_N>* and *<intexpr_{N+1}>* using the *<relation_N>*. The test yields **true** if all comparisons are **true**. Each *<integer expression>* is evaluated only once, and the evaluation is lazy, in the sense that if one comparison is **false**, then no other *<integer expression>* is evaluated and no other comparison is performed. The *<relations>* can be any of the following:

Equal	= or ==
Greater than or equal to	>=
Greater than	>
Less than or equal to	<=
Less than	<
Not equal	!=

This function is more flexible than `\int_compare:nNnTF` but around 5 times slower.

<code>\int_case:nn</code> *	<code>\int_case:nnTF {⟨test integer expression⟩}</code>
<code>\int_case:nnTF</code> *	{
	{⟨intexpr case ₁ ⟩} {⟨code case ₁ ⟩}
	{⟨intexpr case ₂ ⟩} {⟨code case ₂ ⟩}
	...
	{⟨intexpr case _n ⟩} {⟨code case _n ⟩}
	}
	{⟨true code⟩}
	{⟨false code⟩}

New: 2013-07-24

This function evaluates the *⟨test integer expression⟩* and compares this in turn to each of the *⟨integer expression cases⟩*. If the two are equal then the associated *⟨code⟩* is left in the input stream and other cases are discarded. If any of the cases are matched, the *⟨true code⟩* is also inserted into the input stream (after the code for the appropriate case), while if none match then the *⟨false code⟩* is inserted. The function `\int_case:nn`, which does nothing if there is no match, is also available. For example

```
\int_case:nnF
{ 2 * 5 }
{
  { 5 }      { Small }
  { 4 + 6 }   { Medium }
  { -2 * 10 } { Negative }
}
{ No idea! }
```

leaves “Medium” in the input stream.

<code>\int_if_even_p:n</code> *	<code>\int_if_odd_p:n {⟨integer expression⟩}</code>
<code>\int_if_even:nTF</code> *	<code>\int_if_odd:nTF {⟨integer expression⟩}</code>
<code>\int_if_odd_p:n</code> *	{⟨true code⟩} {⟨false code⟩}
<code>\int_if_odd:nTF</code> *	

This function first evaluates the *⟨integer expression⟩* as described for `\int_eval:n`. It then evaluates if this is odd or even, as appropriate.

20.6 Integer expression loops

<code>\int_do_until:nNnn</code> ☆	<code>\int_do_until:nNnn {⟨intexpr₁⟩} ⟨relation⟩ {⟨intexpr₂⟩} {⟨code⟩}</code>
-----------------------------------	---

Places the *⟨code⟩* in the input stream for T_EX to process, and then evaluates the relationship between the two *⟨integer expressions⟩* as described for `\int_compare:nNnTF`. If the test is **false** then the *⟨code⟩* is inserted into the input stream again and a loop occurs until the *⟨relation⟩* is **true**.

<code>\int_do_while:nNnn</code> ☆	<code>\int_do_while:nNnn {⟨intexpr₁⟩} ⟨relation⟩ {⟨intexpr₂⟩} {⟨code⟩}</code>
-----------------------------------	---

Places the *⟨code⟩* in the input stream for T_EX to process, and then evaluates the relationship between the two *⟨integer expressions⟩* as described for `\int_compare:nNnTF`. If the test is **true** then the *⟨code⟩* is inserted into the input stream again and a loop occurs until the *⟨relation⟩* is **false**.

<hr/> <code>\int_until_do:nNnn</code> ☆ <hr/>	<code>\int_until_do:nNnn {<intexpr1>} <relation> {<intexpr2>} {<code>}</code>
	Evaluates the relationship between the two <i><integer expressions></i> as described for <code>\int_compare:nNnTF</code> , and then places the <i><code></i> in the input stream if the <i><relation></i> is false . After the <i><code></i> has been processed by T _E X the test is repeated, and a loop occurs until the test is true .
<hr/> <code>\int_while_do:nNnn</code> ☆ <hr/>	<code>\int_while_do:nNnn {<intexpr1>} <relation> {<intexpr2>} {<code>}</code>
	Evaluates the relationship between the two <i><integer expressions></i> as described for <code>\int_compare:nNnTF</code> , and then places the <i><code></i> in the input stream if the <i><relation></i> is true . After the <i><code></i> has been processed by T _E X the test is repeated, and a loop occurs until the test is false .
<hr/> <code>\int_do_until:nn</code> ☆ <hr/>	<code>\int_do_until:nn {<integer relation>} {<code>}</code>
Updated: 2013-01-13	Places the <i><code></i> in the input stream for T _E X to process, and then evaluates the <i><integer relation></i> as described for <code>\int_compare:nTF</code> . If the test is false then the <i><code></i> is inserted into the input stream again and a loop occurs until the <i><relation></i> is true .
<hr/> <code>\int_do_while:nn</code> ☆ <hr/>	<code>\int_do_while:nn {<integer relation>} {<code>}</code>
Updated: 2013-01-13	Places the <i><code></i> in the input stream for T _E X to process, and then evaluates the <i><integer relation></i> as described for <code>\int_compare:nTF</code> . If the test is true then the <i><code></i> is inserted into the input stream again and a loop occurs until the <i><relation></i> is false .
<hr/> <code>\int_until_do:nn</code> ☆ <hr/>	<code>\int_until_do:nn {<integer relation>} {<code>}</code>
Updated: 2013-01-13	Evaluates the <i><integer relation></i> as described for <code>\int_compare:nTF</code> , and then places the <i><code></i> in the input stream if the <i><relation></i> is false . After the <i><code></i> has been processed by T _E X the test is repeated, and a loop occurs until the test is true .
<hr/> <code>\int_while_do:nn</code> ☆ <hr/>	<code>\int_while_do:nn {<integer relation>} {<code>}</code>
Updated: 2013-01-13	Evaluates the <i><integer relation></i> as described for <code>\int_compare:nTF</code> , and then places the <i><code></i> in the input stream if the <i><relation></i> is true . After the <i><code></i> has been processed by T _E X the test is repeated, and a loop occurs until the test is false .

20.7 Integer step functions

<code>\int_step_function:nN</code>	☆	<code>\int_step_function:nN {⟨final value⟩} ⟨function⟩</code>
<code>\int_step_function:nnN</code>	☆	<code>\int_step_function:nnN {⟨initial value⟩} {⟨final value⟩} ⟨function⟩</code>
<code>\int_step_function:nnnN</code>	☆	<code>\int_step_function:nnnN {⟨initial value⟩} {⟨step⟩} {⟨final value⟩} ⟨function⟩</code>

New: 2012-06-04
Updated: 2018-04-22

This function first evaluates the $\langle initial\ value \rangle$, $\langle step \rangle$ and $\langle final\ value \rangle$, all of which should be integer expressions. The $\langle function \rangle$ is then placed in front of each $\langle value \rangle$ from the $\langle initial\ value \rangle$ to the $\langle final\ value \rangle$ in turn (using $\langle step \rangle$ between each $\langle value \rangle$). The $\langle step \rangle$ must be non-zero. If the $\langle step \rangle$ is positive, the loop stops when the $\langle value \rangle$ becomes larger than the $\langle final\ value \rangle$. If the $\langle step \rangle$ is negative, the loop stops when the $\langle value \rangle$ becomes smaller than the $\langle final\ value \rangle$. The $\langle function \rangle$ should absorb one numerical argument. For example

```
\cs_set:Npn \my_func:n #1 { [I~saw~#1] \quad }
\int_step_function:nnnN { 1 } { 1 } { 5 } \my_func:n
```

would print

```
[I saw 1] [I saw 2] [I saw 3] [I saw 4] [I saw 5]
```

The functions `\int_step_function:nN` and `\int_step_function:nnN` both use a fixed $\langle step \rangle$ of 1, and in the case of `\int_step_function:nN` the $\langle initial\ value \rangle$ is also fixed as 1. These functions are provided as simple short-cuts for code clarity.

<code>\int_step_inline:nn</code>	<code>\int_step_inline:nn {⟨final value⟩} {⟨code⟩}</code>
<code>\int_step_inline:nnn</code>	<code>\int_step_inline:nnn {⟨initial value⟩} {⟨final value⟩} {⟨code⟩}</code>
<code>\int_step_inline:nnnn</code>	<code>\int_step_inline:nnnn {⟨initial value⟩} {⟨step⟩} {⟨final value⟩} {⟨code⟩}</code>

New: 2012-06-04
Updated: 2018-04-22

This function first evaluates the $\langle initial\ value \rangle$, $\langle step \rangle$ and $\langle final\ value \rangle$, all of which should be integer expressions. Then for each $\langle value \rangle$ from the $\langle initial\ value \rangle$ to the $\langle final\ value \rangle$ in turn (using $\langle step \rangle$ between each $\langle value \rangle$), the $\langle code \rangle$ is inserted into the input stream with `#1` replaced by the current $\langle value \rangle$. Thus the $\langle code \rangle$ should define a function of one argument (`#1`).

The functions `\int_step_inline:nn` and `\int_step_inline:nnn` both use a fixed $\langle step \rangle$ of 1, and in the case of `\int_step_inline:nn` the $\langle initial\ value \rangle$ is also fixed as 1. These functions are provided as simple short-cuts for code clarity.

<code>\int_step_variable:nNn</code>	<code>\int_step_variable:nNn {⟨final value⟩} ⟨tl var⟩ {⟨code⟩}</code>
<code>\int_step_variable:nnNn</code>	<code>\int_step_variable:nnNn {⟨initial value⟩} {⟨final value⟩} ⟨tl var⟩ {⟨code⟩}</code>
<code>\int_step_variable:nnnNn</code>	<code>\int_step_variable:nnnNn {⟨initial value⟩} {⟨step⟩} {⟨final value⟩} ⟨tl var⟩ {⟨code⟩}</code>

New: 2012-06-04
Updated: 2018-04-22

This function first evaluates the $\langle initial\ value \rangle$, $\langle step \rangle$ and $\langle final\ value \rangle$, all of which should be integer expressions. Then for each $\langle value \rangle$ from the $\langle initial\ value \rangle$ to the $\langle final\ value \rangle$ in turn (using $\langle step \rangle$ between each $\langle value \rangle$), the $\langle code \rangle$ is inserted into the input stream, with the $\langle tl\ var \rangle$ defined as the current $\langle value \rangle$. Thus the $\langle code \rangle$ should make use of the $\langle tl\ var \rangle$.

The functions `\int_step_variable:nNn` and `\int_step_variable:nnNn` both use a fixed $\langle step \rangle$ of 1, and in the case of `\int_step_variable:nNn` the $\langle initial\ value \rangle$ is also fixed as 1. These functions are provided as simple short-cuts for code clarity.

20.8 Formatting integers

Integers can be placed into the output stream with formatting. These conversions apply to any integer expressions.

<code>\int_to_arabic:n</code> *	<code>\int_to_arabic:n {⟨integer expression⟩}</code>
---------------------------------	--

Updated: 2011-10-22

Places the value of the $\langle integer\ expression \rangle$ in the input stream as digits, with category code 12 (other).

<code>\int_to_alph:n</code> *	<code>\int_to_alph:n {⟨integer expression⟩}</code>
<code>\int_to_Alph:n</code> *	

Updated: 2011-09-17

Evaluates the $\langle integer\ expression \rangle$ and converts the result into a series of letters, which are then left in the input stream. The conversion rule uses the 26 letters of the English alphabet, in order, adding letters when necessary to increase the total possible range of representable numbers. Thus

```
\int_to_alph:n { 1 }
```

places a in the input stream,

```
\int_to_alph:n { 26 }
```

is represented as z and

```
\int_to_alph:n { 27 }
```

is converted to aa. For conversions using other alphabets, use `\int_to_symbols:nnn` to define an alphabet-specific function. The basic `\int_to_alph:n` and `\int_to_Alph:n` functions should not be modified. The resulting tokens are digits with category code 12 (other) and letters with category code 11 (letter).

<code>\int_to_symbols:nnn</code> *	<code>\int_to_symbols:nnn</code> <code>{⟨integer expression⟩} {⟨total symbols⟩}</code> <code>{⟨value to symbol mapping⟩}</code>
------------------------------------	---

Updated: 2011-09-17

This is the low-level function for conversion of an $\langle integer\ expression \rangle$ into a symbolic form (often letters). The $\langle total\ symbols \rangle$ available should be given as an integer expression. Values are actually converted to symbols according to the $\langle value\ to\ symbol\ mapping \rangle$. This should be given as $\langle total\ symbols \rangle$ pairs of entries, a number and the appropriate symbol. Thus the `\int_to_alph:n` function is defined as

```
\cs_new:Npn \int_to_alph:n #1
{
  \int_to_symbols:nnn {#1} { 26 }
  {
    { 1 } { a }
    { 2 } { b }
    ...
    { 26 } { z }
  }
}
```

<hr/>	
<code>\int_to_bin:n *</code>	<code>\int_to_bin:n {⟨integer expression⟩}</code>
<hr/>	
<code>New: 2014-02-11</code>	Calculates the value of the <i>⟨integer expression⟩</i> and places the binary representation of the result in the input stream.
<hr/>	
<code>\int_to_hex:n *</code>	<code>\int_to_hex:n {⟨integer expression⟩}</code>
<code>\int_to_Hex:n *</code>	Calculates the value of the <i>⟨integer expression⟩</i> and places the hexadecimal (base 16) representation of the result in the input stream. Letters are used for digits beyond 9: lower case letters for <code>\int_to_hex:n</code> and upper case ones for <code>\int_to_Hex:n</code> . The resulting tokens are digits with category code 12 (other) and letters with category code 11 (letter).
<hr/>	
<code>New: 2014-02-11</code>	
<hr/>	
<code>\int_to_oct:n *</code>	<code>\int_to_oct:n {⟨integer expression⟩}</code>
<hr/>	
<code>New: 2014-02-11</code>	Calculates the value of the <i>⟨integer expression⟩</i> and places the octal (base 8) representation of the result in the input stream. The resulting tokens are digits with category code 12 (other) and letters with category code 11 (letter).
<hr/>	
<code>\int_to_base:nn *</code>	<code>\int_to_base:nn {⟨integer expression⟩} {⟨base⟩}</code>
<code>\int_to_Base:nn *</code>	Calculates the value of the <i>⟨integer expression⟩</i> and converts it into the appropriate representation in the <i>⟨base⟩</i> ; the later may be given as an integer expression. For bases greater than 10 the higher “digits” are represented by letters from the English alphabet: lower case letters for <code>\int_to_base:n</code> and upper case ones for <code>\int_to_Base:n</code> . The maximum <i>⟨base⟩</i> value is 36. The resulting tokens are digits with category code 12 (other) and letters with category code 11 (letter).
<hr/>	
<code>Updated: 2014-02-11</code>	
<hr/>	
TeXhackers note: This is a generic version of <code>\int_to_bin:n</code> , <i>etc.</i>	
<hr/>	
<code>\int_to_roman:n ☆</code>	<code>\int_to_roman:n {⟨integer expression⟩}</code>
<code>\int_to_Roman:n ☆</code>	Places the value of the <i>⟨integer expression⟩</i> in the input stream as Roman numerals, either lower case (<code>\int_to_roman:n</code>) or upper case (<code>\int_to_Roman:n</code>). If the value is negative or zero, the output is empty. The Roman numerals are letters with category code 11 (letter). The letters used are <code>mdclxvi</code> , repeated as needed: the notation with bars (such as <code>v̄</code> for 5000) is <i>not</i> used. For instance <code>\int_to_roman:n { 8249 }</code> expands to <code>mmmmmmmmccxlix</code> .
<hr/>	
<code>Updated: 2011-10-22</code>	
<hr/>	

20.9 Converting from other formats to integers

<hr/>	
<code>\int_from_alph:n *</code>	<code>\int_from_alph:n {⟨letters⟩}</code>
<hr/>	
<code>Updated: 2014-08-25</code>	Converts the <i>⟨letters⟩</i> into the integer (base 10) representation and leaves this in the input stream. The <i>⟨letters⟩</i> are first converted to a string, with no expansion. Lower and upper case letters from the English alphabet may be used, with “a” equal to 1 through to “z” equal to 26. The function also accepts a leading sign, made of + and -. This is the inverse function of <code>\int_to_alph:n</code> and <code>\int_to_Alph:n</code> .
<hr/>	

<hr/> <code>\int_from_bin:n</code> ★	<code>\int_from_bin:n {⟨binary number⟩}</code>
New: 2014-02-11 Updated: 2014-08-25	<p>Converts the <i>⟨binary number⟩</i> into the integer (base 10) representation and leaves this in the input stream. The <i>⟨binary number⟩</i> is first converted to a string, with no expansion. The function accepts a leading sign, made of + and -, followed by binary digits. This is the inverse function of <code>\int_to_bin:n</code>.</p>
<hr/> <code>\int_from_hex:n</code> ★	<code>\int_from_hex:n {⟨hexadecimal number⟩}</code>
New: 2014-02-11 Updated: 2014-08-25	<p>Converts the <i>⟨hexadecimal number⟩</i> into the integer (base 10) representation and leaves this in the input stream. Digits greater than 9 may be represented in the <i>⟨hexadecimal number⟩</i> by upper or lower case letters. The <i>⟨hexadecimal number⟩</i> is first converted to a string, with no expansion. The function also accepts a leading sign, made of + and -. This is the inverse function of <code>\int_to_hex:n</code> and <code>\int_to_Hex:n</code>.</p>
<hr/> <code>\int_from_oct:n</code> ★	<code>\int_from_oct:n {⟨octal number⟩}</code>
New: 2014-02-11 Updated: 2014-08-25	<p>Converts the <i>⟨octal number⟩</i> into the integer (base 10) representation and leaves this in the input stream. The <i>⟨octal number⟩</i> is first converted to a string, with no expansion. The function accepts a leading sign, made of + and -, followed by octal digits. This is the inverse function of <code>\int_to_oct:n</code>.</p>
<hr/> <code>\int_from_roman:n</code> ★	<code>\int_from_roman:n {⟨roman numeral⟩}</code>
Updated: 2014-08-25	<p>Converts the <i>⟨roman numeral⟩</i> into the integer (base 10) representation and leaves this in the input stream. The <i>⟨roman numeral⟩</i> is first converted to a string, with no expansion. The <i>⟨roman numeral⟩</i> may be in upper or lower case; if the numeral contains characters besides <code>mdclxvi</code> or <code>MDCLXVI</code> then the resulting value is -1. This is the inverse function of <code>\int_to_roman:n</code> and <code>\int_to_Roman:n</code>.</p>
<hr/> <code>\int_from_base:nn</code> ★	<code>\int_from_base:nn {⟨number⟩} {⟨base⟩}</code>
Updated: 2014-08-25	<p>Converts the <i>⟨number⟩</i> expressed in <i>⟨base⟩</i> into the appropriate value in base 10. The <i>⟨number⟩</i> is first converted to a string, with no expansion. The <i>⟨number⟩</i> should consist of digits and letters (either lower or upper case), plus optionally a leading sign. The maximum <i>⟨base⟩</i> value is 36. This is the inverse function of <code>\int_to_base:nn</code> and <code>\int_to_Base:nn</code>.</p>

20.10 Random integers

<hr/> <code>\int_rand:nn</code> ★	<code>\int_rand:nn {⟨intexpr₁⟩} {⟨intexpr₂⟩}</code>
New: 2016-12-06 Updated: 2018-04-27	<p>Evaluates the two <i>⟨integer expressions⟩</i> and produces a pseudo-random number between the two (with bounds included). This is not available in older versions of X_YTeX.</p>
<hr/> <code>\int_rand:n</code> ★	<code>\int_rand:n {⟨intexpr⟩}</code>
New: 2018-05-05	<p>Evaluates the <i>⟨integer expression⟩</i> then produces a pseudo-random number between 1 and the <i>⟨intexpr⟩</i> (included). This is not available in older versions of X_YTeX.</p>

20.11 Viewing integers

<hr/> <code>\int_show:N</code> <code>\int_show:c</code> <hr/>	<code>\int_show:N <integer></code> Displays the value of the <i><integer></i> on the terminal.
<hr/> <code>\int_show:n</code> <hr/> <div>New: 2011-11-22 Updated: 2015-08-07</div>	<code>\int_show:n {(integer expression)}</code> Displays the result of evaluating the <i><integer expression></i> on the terminal.
<hr/> <code>\int_log:N</code> <code>\int_log:c</code> <hr/> <div>New: 2014-08-22 Updated: 2015-08-03</div>	<code>\int_log:N <integer></code> Writes the value of the <i><integer></i> in the log file.
<hr/> <code>\int_log:n</code> <hr/> <div>New: 2014-08-22 Updated: 2015-08-07</div>	<code>\int_log:n {(integer expression)}</code> Writes the result of evaluating the <i><integer expression></i> in the log file.

20.12 Constant integers

<hr/> <code>\c_zero_int</code> <code>\c_one_int</code> <hr/> <div>New: 2018-05-07</div>	Integer values used with primitive tests and assignments: their self-terminating nature makes these more convenient and faster than literal numbers.
<hr/> <code>\c_max_int</code> <hr/>	The maximum value that can be stored as an integer.
<hr/> <code>\c_max_register_int</code> <hr/>	Maximum number of registers.
<hr/> <code>\c_max_char_int</code> <hr/>	Maximum character code completely supported by the engine.

20.13 Scratch integers

<hr/> <code>\l_tmpa_int</code> <code>\l_tmpb_int</code> <hr/>	Scratch integer for local assignment. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
<hr/> <code>\g_tmpa_int</code> <code>\g_tmpb_int</code> <hr/>	Scratch integer for global assignment. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

20.14 Direct number expansion

`\int_value:w` ★
 New: 2018-03-27

`\int_value:w` $\langle integer \rangle$
`\int_value:w` $\langle integer\ denotation \rangle$ $\langle optional\ space \rangle$

Expands the following tokens until an $\langle integer \rangle$ is formed, and leaves a normalized form (no leading sign except for negative numbers, no leading digit 0 except for zero) in the input stream as category code 12 (other) characters. The $\langle integer \rangle$ can consist of any number of signs (with intervening spaces) followed by

- an integer variable (in fact, any T_EX register except `\toks`) or
- explicit digits (or by ‘ $\langle octal\ digits \rangle$ ’ or “ $\langle hexadecimal\ digits \rangle$ ” or ‘ $\langle character \rangle$ ’).

In this last case expansion stops once a non-digit is found; if that is a space it is removed as in `f`-expansion, and so `\exp_stop_f:` may be employed as an end marker. Note that protected functions *are* expanded by this process.

This function requires exactly one expansion to produce a value, and so is suitable for use in cases where a number is required “directly”. In general, `\int_eval:n` is the preferred approach to generating numbers.

T_EXhackers note: This is the T_EX primitive `\number`.

20.15 Primitive conditionals

`\if_int_compare:w` ★

`\if_int_compare:w` $\langle integer_1 \rangle$ $\langle relation \rangle$ $\langle integer_2 \rangle$
 $\langle true\ code \rangle$
`\else:`
 $\langle false\ code \rangle$
`\fi:`

Compare two integers using $\langle relation \rangle$, which must be one of =, < or > with category code 12. The `\else:` branch is optional.

T_EXhackers note: These are both names for the T_EX primitive `\ifnum`.

`\if_case:w` ★
`\or:` ★

`\if_case:w` $\langle integer \rangle$ $\langle case_0 \rangle$
`\or:` $\langle case_1 \rangle$
`\or:` ...
`\else:` $\langle default \rangle$
`\fi:`

Selects a case to execute based on the value of the $\langle integer \rangle$. The first case ($\langle case_0 \rangle$) is executed if $\langle integer \rangle$ is 0, the second ($\langle case_1 \rangle$) if the $\langle integer \rangle$ is 1, *etc.* The $\langle integer \rangle$ may be a literal, a constant or an integer expression (*e.g.* using `\int_eval:n`).

T_EXhackers note: These are the T_EX primitives `\ifcase` and `\or`.

<code>\if_int_odd:w</code> ★	<code>\if_int_odd:w</code> $\langle tokens \rangle$ $\langle optional\ space \rangle$ $\langle true\ code \rangle$ <code>\else:</code> $\langle true\ code \rangle$ <code>\fi:</code>
------------------------------	---

Expands $\langle tokens \rangle$ until a non-numeric token or a space is found, and tests whether the resulting $\langle integer \rangle$ is odd. If so, $\langle true\ code \rangle$ is executed. The `\else:` branch is optional.

TeXhackers note: This is the TeX primitive `\ifodd`.

Chapter 21

The l3flag package: Expandable flags

Flags are the only data-type that can be modified in expansion-only contexts. This module is meant mostly for kernel use: in almost all cases, booleans or integers should be preferred to flags because they are very significantly faster.

A flag can hold any non-negative value, which we call its *height*. In expansion-only contexts, a flag can only be “raised”: this increases the *height* by 1. The *height* can also be queried expandably. However, decreasing it, or setting it to zero requires non-expandable assignments.

Flag variables are always local. They are referenced by a *flag name* such as `str_missing`. The *flag name* is used as part of `\use:c` constructions hence is expanded at point of use. It must expand to character tokens only, with no spaces.

A typical use case of flags would be to keep track of whether an exceptional condition has occurred during expandable processing, and produce a meaningful (non-expandable) message after the end of the expandable processing. This is exemplified by `l3str-convert`, which for performance reasons performs conversions of individual characters expandably and for readability reasons produces a single error message describing incorrect inputs that were encountered.

Flags should not be used without carefully considering the fact that raising a flag takes a time and memory proportional to its height. Flags should not be used unless unavoidable.

21.1 Setting up flags

<code>\flag_new:n</code>	<code>\flag_new:n {<flag name>}</code>
--------------------------	--

Creates a new flag with a name given by *flag name*, or raises an error if the name is already taken. The *flag name* may not contain spaces. The declaration is global, but flags are always local variables. The *flag* initially has zero height.

<code>\flag_clear:n</code>	<code>\flag_clear:n {<flag name>}</code>
----------------------------	--

The *flag*’s height is set to zero. The assignment is local.

<hr/> <hr/>	<code>\flag_clear_new:n</code>	<code>\flag_clear_new:n {⟨flag name⟩}</code>	Ensures that the $\langle flag \rangle$ exists globally by applying <code>\flag_new:n</code> if necessary, then applies <code>\flag_clear:n</code> , setting the height to zero locally.
<hr/> <hr/>	<code>\flag_show:n</code>	<code>\flag_show:n {⟨flag name⟩}</code>	Displays the $\langle flag \rangle$'s height in the terminal.
<hr/> <hr/>	<code>\flag_log:n</code>	<code>\flag_log:n {⟨flag name⟩}</code>	Writes the $\langle flag \rangle$'s height to the log file.

21.2 Expandable flag commands

<hr/> <hr/>	<code>\flag_if_exist_p:n *</code>	<code>\flag_if_exist:n {⟨flag name⟩}</code>	
<hr/> <hr/>	<code>\flag_if_exist:nTF *</code>		This function returns <code>true</code> if the $\langle flag name \rangle$ references a flag that has been defined previously, and <code>false</code> otherwise.
<hr/> <hr/>	<code>\flag_if_raised_p:n *</code>	<code>\flag_if_raised:n {⟨flag name⟩}</code>	
<hr/> <hr/>	<code>\flag_if_raised:nTF *</code>		This function returns <code>true</code> if the $\langle flag \rangle$ has non-zero height, and <code>false</code> if the $\langle flag \rangle$ has zero height.
<hr/> <hr/>	<code>\flag_height:n *</code>	<code>\flag_height:n {⟨flag name⟩}</code>	Expands to the height of the $\langle flag \rangle$ as an integer denotation.
<hr/> <hr/>	<code>\flag_raise:n *</code>	<code>\flag_raise:n {⟨flag name⟩}</code>	The $\langle flag \rangle$'s height is increased by 1 locally.

Chapter 22

The l3clist package

Comma separated lists

Comma lists (in short, `clist`) contain ordered data where items can be added to the left or right end of the list. This data type allows basic list manipulations such as adding/removing items, applying a function to every item, removing duplicate items, extracting a given item, using the comma list with specified separators, and so on. Sequences (defined in `l3seq`) are safer, faster, and provide more features, so they should often be preferred to comma lists. Comma lists are mostly useful when interfacing with $\text{\LaTeX}2_{\epsilon}$ or other code that expects or provides items separated by commas.

Several items can be added at once. To ease input of comma lists from data provided by a user outside an `\ExplSyntaxOn ... \ExplSyntaxOff` block, spaces are removed from both sides of each comma-delimited argument upon input. Blank arguments are ignored, to allow for trailing commas or repeated commas (which may otherwise arise when concatenating comma lists “by hand”). In addition, a set of braces is removed if the result of space-trimming is braced: this allows the storage of any item in a comma list. For instance,

```
\clist_new:N \l_my_clist
\clist_put_left:Nn \l_my_clist { ~a~ , ~{b}~ , c~\d }
\clist_put_right:Nn \l_my_clist { ~{e~} , , {{f}} , }
```

results in `\l_my_clist` containing `a,b,c~\d,{e~},{{f}}` namely the five items `a`, `b`, `c~\d`, `e~` and `{f}`. Comma lists normally do not contain empty or blank items so the following gives an empty comma list:

```
\clist_clear_new:N \l_my_clist
\clist_set:Nn \l_my_clist { , ~ , , }
\clist_if_empty:NTF \l_my_clist { true } { false }
```

and it leaves `true` in the input stream. To include an “unsafe” item (empty, or one that contains a comma, or starts or ends with a space, or is a single brace group), surround it with braces.

Any `n`-type token list is a valid comma list input for `l3clist` functions, which will split the token list at every comma and process the items as described above. On the other hand, `N`-type functions expect comma list variables, which are particular token list variables in which this processing of items (and removal of blank items) has already

occurred. Because comma list variables are token list variables, expanding them once yields their items separated by commas, and `\l3tl` functions such as `\tl_show:N` can be applied to them. (These functions often have `\clist` analogues, which should be preferred.)

Almost all operations on comma lists are noticeably slower than those on sequences so converting the data to sequences using `\seq_set_from_clist:Nn` (see `\l3seq`) may be advisable if speed is important. The exception is that `\clist_if_in:NnTF` and `\clist_remove_duplicates:N` may be faster than their sequence analogues for large lists. However, these functions work slowly for “unsafe” items that must be braced, and may produce errors when their argument contains `{`, `}` or `#` (assuming the usual `TeX` category codes apply). The sequence data type should thus certainly be preferred to comma lists to store such items.

22.1 Creating and initialising comma lists

<code>\clist_new:N</code>	<code>\clist_new:N</code> \langle <i>comma list</i> \rangle
---------------------------	---

<code>\clist_new:c</code>	
---------------------------	--

Creates a new \langle *comma list* \rangle or raises an error if the name is already taken. The declaration is global. The \langle *comma list* \rangle initially contains no items.

<code>\clist_const:Nn</code>	<code>\clist_const:Nn</code> \langle <i>clist var</i> \rangle $\{\langle$ <i>comma list</i> $\rangle\}$
------------------------------	---

<code>\clist_const:(Nx cn cx)</code>	
--------------------------------------	--

New: 2014-07-05

Creates a new constant \langle *clist var* \rangle or raises an error if the name is already taken. The value of the \langle *clist var* \rangle is set globally to the \langle *comma list* \rangle .

<code>\clist_clear:N</code>	<code>\clist_clear:N</code> \langle <i>comma list</i> \rangle
-----------------------------	---

<code>\clist_clear:c</code>	
-----------------------------	--

<code>\clist_gclear:N</code>	
------------------------------	--

<code>\clist_gclear:c</code>	
------------------------------	--

Clears all items from the \langle *comma list* \rangle .

<code>\clist_clear_new:N</code>	<code>\clist_clear_new:N</code> \langle <i>comma list</i> \rangle
---------------------------------	---

<code>\clist_clear_new:c</code>	
---------------------------------	--

<code>\clist_gclear_new:N</code>	
----------------------------------	--

<code>\clist_gclear_new:c</code>	
----------------------------------	--

Ensures that the \langle *comma list* \rangle exists globally by applying `\clist_new:N` if necessary, then applies `\clist_(g)clear:N` to leave the list empty.

<code>\clist_set_eq:NN</code>	<code>\clist_set_eq:NN</code> \langle <i>comma list</i> ₁ \rangle \langle <i>comma list</i> ₂ \rangle
-------------------------------	---

<code>\clist_set_eq:(cN Nc cc)</code>	
---------------------------------------	--

<code>\clist_gset_eq:NN</code>	
--------------------------------	--

<code>\clist_gset_eq:(cN Nc cc)</code>	
--	--

Sets the content of \langle *comma list*₁ \rangle equal to that of \langle *comma list*₂ \rangle . To set a token list variable equal to a comma list variable, use `\tl_set_eq:NN`. Conversely, setting a comma list variable to a token list is unadvisable unless one checks space-trimming and related issues.

<code>\clist_set_from_seq:NN</code>	<code>\clist_set_from_seq:NN</code> \langle <i>comma list</i> \rangle \langle <i>sequence</i> \rangle
-------------------------------------	---

<code>\clist_set_from_seq:(cN Nc cc)</code>	
---	--

<code>\clist_gset_from_seq:NN</code>	
--------------------------------------	--

<code>\clist_gset_from_seq:(cN Nc cc)</code>	
--	--

New: 2014-07-17

Converts the data in the \langle *sequence* \rangle into a \langle *comma list* \rangle : the original \langle *sequence* \rangle is unchanged. Items which contain either spaces or commas are surrounded by braces.

<hr/>	
<code>\clist_concat:NNN</code>	<code>\clist_concat:NNN <comma list₁> <comma list₂> <comma list₃></code>
<code>\clist_concat:ccc</code>	
<code>\clist_gconcat:NNN</code>	Concatenates the content of <code><comma list₂></code> and <code><comma list₃></code> together and saves the result in <code><comma list₁></code> . The items in <code><comma list₂></code> are placed at the left side of the new comma list.
<code>\clist_gconcat:ccc</code>	
<hr/>	
<code>\clist_if_exist_p:N *</code>	<code>\clist_if_exist_p:N <comma list></code>
<code>\clist_if_exist_p:c *</code>	<code>\clist_if_exist:NTF <comma list> {\<true code>} {\<false code>}</code>
<code>\clist_if_exist:N\overline{TF} *</code>	
<code>\clist_if_exist:c\overline{TF} *</code>	Tests whether the <code><comma list></code> is currently defined. This does not check that the <code><comma list></code> really is a comma list.
<hr/>	
New: 2012-03-03	
<hr/>	

22.2 Adding data to comma lists

<hr/>	
<code>\clist_set:Nn</code>	<code>\clist_set:Nn <comma list> {\<item₁>,...,<item_n>}</code>
<code>\clist_set:(NV No Nx cn cV co cx)</code>	
<code>\clist_gset:Nn</code>	
<code>\clist_gset:(NV No Nx cn cV co cx)</code>	
<hr/>	
New: 2011-09-06	
<hr/>	

Sets `<comma list>` to contain the `<items>`, removing any previous content from the variable. Blank items are omitted, spaces are removed from both sides of each item, then a set of braces is removed if the resulting space-trimmed item is braced. To store some `<tokens>` as a single `<item>` even if the `<tokens>` contain commas or spaces, add a set of braces: `\clist_set:Nn <comma list> { {\<tokens>} }`.

<hr/>	
<code>\clist_put_left:Nn</code>	<code>\clist_put_left:Nn <comma list> {\<item₁>,...,<item_n>}</code>
<code>\clist_put_left:(NV No Nx cn cV co cx)</code>	
<code>\clist_gput_left:Nn</code>	
<code>\clist_gput_left:(NV No Nx cn cV co cx)</code>	
<hr/>	
Updated: 2011-09-05	
<hr/>	

Appends the `<items>` to the left of the `<comma list>`. Blank items are omitted, spaces are removed from both sides of each item, then a set of braces is removed if the resulting space-trimmed item is braced. To append some `<tokens>` as a single `<item>` even if the `<tokens>` contain commas or spaces, add a set of braces: `\clist_put_left:Nn <comma list> { {\<tokens>} }`.

<hr/>	
<code>\clist_put_right:Nn</code>	<code>\clist_put_right:Nn <comma list> {\<item₁>,...,<item_n>}</code>
<code>\clist_put_right:(NV No Nx cn cV co cx)</code>	
<code>\clist_gput_right:Nn</code>	
<code>\clist_gput_right:(NV No Nx cn cV co cx)</code>	
<hr/>	
Updated: 2011-09-05	
<hr/>	

Appends the `<items>` to the right of the `<comma list>`. Blank items are omitted, spaces are removed from both sides of each item, then a set of braces is removed if the resulting space-trimmed item is braced. To append some `<tokens>` as a single `<item>` even if the `<tokens>` contain commas or spaces, add a set of braces: `\clist_put_right:Nn <comma list> { {\<tokens>} }`.

22.3 Modifying comma lists

While comma lists are normally used as ordered lists, it may be necessary to modify the content. The functions here may be used to update comma lists, while retaining the order of the unaffected entries.

```

\clist_remove_duplicates:N    \clist_remove_duplicates:N <comma list>
\clist_remove_duplicates:c
\clist_gremove_duplicates:N
\clist_gremove_duplicates:c

```

Removes duplicate items from the $\langle comma list \rangle$, leaving the left most copy of each item in the $\langle comma list \rangle$. The $\langle item \rangle$ comparison takes place on a token basis, as for $\texttt{\backslash tl_if_eq:nnTF}$.

T_EXhackers note: This function iterates through every item in the $\langle comma list \rangle$ and does a comparison with the $\langle items \rangle$ already checked. It is therefore relatively slow with large comma lists. Furthermore, it may fail if any of the items in the $\langle comma list \rangle$ contains $\{$, $\}$, or $\#$ (assuming the usual T_EX category codes apply).

```

\clist_remove_all:Nn    \clist_remove_all:Nn <comma list> {\item}
\clist_remove_all:(cn|NV|cV)
\clist_gremove_all:Nn
\clist_gremove_all:(cn|NV|cV)

```

Updated: 2011-09-06

Removes every occurrence of $\langle item \rangle$ from the $\langle comma list \rangle$. The $\langle item \rangle$ comparison takes place on a token basis, as for $\texttt{\backslash tl_if_eq:nnTF}$.

T_EXhackers note: The function may fail if the $\langle item \rangle$ contains $\{$, $\}$, or $\#$ (assuming the usual T_EX category codes apply).

```

\clist_reverse:N    \clist_reverse:N <comma list>
\clist_reverse:c
\clist_greverse:N
\clist_greverse:c

```

New: 2014-07-18

```

\clist_reverse:n    \clist_reverse:n {\comma list}

```

New: 2014-07-18

Leaves the items in the $\langle comma list \rangle$ in the input stream in reverse order. Contrarily to other what is done for other n-type $\langle comma list \rangle$ arguments, braces and spaces are preserved by this process.

T_EXhackers note: The result is returned within $\texttt{\backslash unexpanded}$, which means that the comma list does not expand further when appearing in an x-type or e-type argument expansion.

<hr/> <code>\clist_sort:Nn</code>	<code>\clist_sort:Nn <clist var> {<comparison code>}</code>
<code>\clist_sort:cn</code>	
<code>\clist_gsort:Nn</code>	Sorts the items in the <code><clist var></code> according to the <code><comparison code></code> , and assigns the
<code>\clist_gsort:cn</code>	result to <code><clist var></code> . The details of sorting comparison are described in Section 6.1.

New: 2017-02-06

22.4 Comma list conditionals

<hr/> <code>\clist_if_empty_p:N</code> *	<code>\clist_if_empty_p:N <comma list></code>
<code>\clist_if_empty_p:c</code> *	<code>\clist_if_empty:NTF <comma list> {<true code>} {<false code>}</code>
<code>\clist_if_empty:NTF</code> *	Tests if the <code><comma list></code> is empty (containing no items).
<code>\clist_if_empty:cTF</code> *	

<hr/> <code>\clist_if_empty_p:n</code> *	<code>\clist_if_empty_p:n {<comma list>}</code>
<code>\clist_if_empty:nTF</code> *	<code>\clist_if_empty:nTF {<comma list>} {<true code>} {<false code>}</code>

New: 2014-07-05

Tests if the `<comma list>` is empty (containing no items). The rules for space trimming are as for other n-type comma-list functions, hence the comma list `{~,~,~}` (without outer braces) is empty, while `{~,{}},}` (without outer braces) contains one element, which happens to be empty: the comma-list is not empty.

<hr/> <code>\clist_if_in:NnTF</code>	<code>\clist_if_in:NnTF <comma list> {<item>} {<true code>} {<false code>}</code>
<code>\clist_if_in:(NV No cn cV co)TF</code>	
<code>\clist_if_in:nnTF</code>	
<code>\clist_if_in:(nV no)TF</code>	

Updated: 2011-09-06

Tests if the `<item>` is present in the `<comma list>`. In the case of an n-type `<comma list>`, the usual rules of space trimming and brace stripping apply. Hence,

`\clist_if_in:nnTF { a , {b}~ , {b} , c } { b } {true} {false}`

yields true.

T_EXhackers note: The function may fail if the `<item>` contains `{`, `}`, or `#` (assuming the usual T_EX category codes apply).

22.5 Mapping over comma lists

The functions described in this section apply a specified function to each item of a comma list. All mappings are done at the current group level, *i.e.* any local assignments made by the `<function>` or `<code>` discussed below remain in effect after the loop.

When the comma list is given explicitly, as an n-type argument, spaces are trimmed around each item. If the result of trimming spaces is empty, the item is ignored. Otherwise, if the item is surrounded by braces, one set is removed, and the result is passed to the mapped function. Thus, if the comma list that is being mapped is `{a_,{b}_},_{},_{c},}` then the arguments passed to the mapped function are ‘a’, ‘b’_␣, an empty argument, and ‘c’.

When the comma list is given as an N-type argument, spaces have already been trimmed on input, and items are simply stripped of one set of braces if any. This case is more efficient than using n-type comma lists.

`\clist_map_function:NN` ☆
`\clist_map_function:cN` ☆
`\clist_map_function:nN` ☆

Updated: 2012-06-29

`\clist_map_function:NN` $\langle comma list \rangle$ $\langle function \rangle$

Applies $\langle function \rangle$ to every $\langle item \rangle$ stored in the $\langle comma list \rangle$. The $\langle function \rangle$ receives one argument for each iteration. The $\langle items \rangle$ are returned from left to right. The function `\clist_map_inline:Nn` is in general more efficient than `\clist_map_function:NN`.

`\clist_map_inline:Nn`
`\clist_map_inline:cn`
`\clist_map_inline:nn`

Updated: 2012-06-29

`\clist_map_inline:Nn` $\langle comma list \rangle$ $\{ \langle inline function \rangle \}$

Applies $\langle inline function \rangle$ to every $\langle item \rangle$ stored within the $\langle comma list \rangle$. The $\langle inline function \rangle$ should consist of code which receives the $\langle item \rangle$ as #1. The $\langle items \rangle$ are returned from left to right.

`\clist_map_variable:NNn`
`\clist_map_variable:cNn`
`\clist_map_variable:nNn`

Updated: 2012-06-29

`\clist_map_variable:NNn` $\langle comma list \rangle$ $\langle variable \rangle$ $\{ \langle code \rangle \}$

Stores each $\langle item \rangle$ of the $\langle comma list \rangle$ in turn in the (token list) $\langle variable \rangle$ and applies the $\langle code \rangle$. The $\langle code \rangle$ will usually make use of the $\langle variable \rangle$, but this is not enforced. The assignments to the $\langle variable \rangle$ are local. Its value after the loop is the last $\langle item \rangle$ in the $\langle comma list \rangle$, or its original value if there were no $\langle item \rangle$. The $\langle items \rangle$ are returned from left to right.

`\clist_map_tokens:Nn` ☆
`\clist_map_tokens:cn` ☆
`\clist_map_tokens:nn` ☆

New: 2021-05-05

`\clist_map_tokens:Nn` $\langle clist var \rangle$ $\{ \langle code \rangle \}$
`\clist_map_tokens:nn` $\{ \langle comma list \rangle \}$ $\{ \langle code \rangle \}$

Calls $\langle code \rangle$ $\{ \langle item \rangle \}$ for every $\langle item \rangle$ stored in the $\langle comma list \rangle$. The $\langle code \rangle$ receives each $\langle item \rangle$ as a trailing brace group. If the $\langle code \rangle$ consists of a single function this is equivalent to `\clist_map_function:nN`.

`\clist_map_break:` ☆

Updated: 2012-06-29

`\clist_map_break:`

Used to terminate a `\clist_map_...` function before all entries in the $\langle comma list \rangle$ have been processed. This normally takes place within a conditional statement, for example

```
\clist_map_inline:Nn \l_my_clist
{
  \str_if_eq:nnTF { #1 } { bingo }
  { \clist_map_break: }
  {
    % Do something useful
  }
}
```

Use outside of a `\clist_map_...` scenario leads to low level TeX errors.

TeXhackers note: When the mapping is broken, additional tokens may be inserted before further items are taken from the input stream. This depends on the design of the mapping function.

`\clist_map_break:n` ☆

Updated: 2012-06-29

`\clist_map_break:n {<code>}`

Used to terminate a `\clist_map_...` function before all entries in the *<comma list>* have been processed, inserting the *<code>* after the mapping has ended. This normally takes place within a conditional statement, for example

```
\clist_map_inline:Nn \l_my_clist
{
  \str_if_eq:nnTF { #1 } { bingo }
  { \clist_map_break:n { <code> } }
  {
    % Do something useful
  }
}
```

Use outside of a `\clist_map_...` scenario leads to low level T_EX errors.

T_EXhackers note: When the mapping is broken, additional tokens may be inserted before the *<code>* is inserted into the input stream. This depends on the design of the mapping function.

`\clist_count:N` ★

`\clist_count:c` ★

`\clist_count:n` ★

New: 2012-07-13

`\clist_count:N <comma list>`

Leaves the number of items in the *<comma list>* in the input stream as an *<integer denotation>*. The total number of items in a *<comma list>* includes those which are duplicates, *i.e.* every item in a *<comma list>* is counted.

22.6 Using the content of comma lists directly

`\clist_use:Nnnn` ★

`\clist_use:cnnn` ★

New: 2013-05-26

`\clist_use:Nnnn <clist var> {<separator between two>}`

`{<separator between more than two>} {<separator between final two>}`

Places the contents of the *<clist var>* in the input stream, with the appropriate *<separator>* between the items. Namely, if the comma list has more than two items, the *<separator between more than two>* is placed between each pair of items except the last, for which the *<separator between final two>* is used. If the comma list has exactly two items, then they are placed in the input stream separated by the *<separator between two>*. If the comma list has a single item, it is placed in the input stream, and a comma list with no items produces no output. An error is raised if the variable does not exist or if it is invalid.

For example,

```
\clist_set:Nn \l_tmpa_clist { a , b , , c , {de} , f }
\clist_use:Nnnn \l_tmpa_clist { ~and~ } { ,~ } { ,~and~ }
```

inserts “a, b, c, de, and f” in the input stream. The first separator argument is not used in this case because the comma list has more than 2 items.

T_EXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the *<items>* do not expand further when appearing in an x-type or e-type argument expansion.

```
\clist_use:Nn ★ \clist_use:Nn <clist var> {<separator>}
```

```
\clist_use:cn ★
```

New: 2013-05-26

Places the contents of the $\langle\textit{clist var}\rangle$ in the input stream, with the $\langle\textit{separator}\rangle$ between the items. If the comma list has a single item, it is placed in the input stream, and a comma list with no items produces no output. An error is raised if the variable does not exist or if it is invalid.

For example,

```
\clist_set:Nn \l_tmpa_clist { a , b , , c , {de} , f }
\clist_use:Nn \l_tmpa_clist { ~and~ }
```

inserts “a and b and c and de and f” in the input stream.

T_EXhackers note: The result is returned within the $\backslash\textit{unexpanded}$ primitive ($\backslash\textit{exp_not:n}$), which means that the $\langle\textit{items}\rangle$ do not expand further when appearing in an \textit{x} -type or \textit{e} -type argument expansion.

```
\clist_use:nnnn ★ \clist_use:nnnn <comma list> {<separator between two>}
```

```
\clist_use:nn ★ {<separator between more than two>} {<separator between final two>}
```

New: 2021-05-10

```
\clist_use:nn <comma list> {<separator>}
```

Places the contents of the $\langle\textit{comma list}\rangle$ in the input stream, with the appropriate $\langle\textit{separator}\rangle$ between the items. As for $\backslash\textit{clist_set:Nn}$, blank items are omitted, spaces are removed from both sides of each item, then a set of braces is removed if the resulting space-trimmed item is braced. The $\langle\textit{separators}\rangle$ are then inserted in the same way as for $\backslash\textit{clist_use:Nnnn}$ and $\backslash\textit{clist_use:Nn}$, respectively.

T_EXhackers note: The result is returned within the $\backslash\textit{unexpanded}$ primitive ($\backslash\textit{exp_not:n}$), which means that the $\langle\textit{items}\rangle$ do not expand further when appearing in an \textit{x} -type or \textit{e} -type argument expansion.

22.7 Comma lists as stacks

Comma lists can be used as stacks, where data is pushed to and popped from the top of the comma list. (The left of a comma list is the top, for performance reasons.) The stack functions for comma lists are not intended to be mixed with the general ordered data functions detailed in the previous section: a comma list should either be used as an ordered data type or as a stack, but not in both ways.

```
\clist_get:NN \clist_get:NN <comma list> <token list variable>
```

```
\clist_get:cN
```

```
\clist_get:NNTF
```

```
\clist_get:cNTF
```

New: 2012-05-14

Updated: 2019-02-16

Stores the left-most item from a $\langle\textit{comma list}\rangle$ in the $\langle\textit{token list variable}\rangle$ without removing it from the $\langle\textit{comma list}\rangle$. The $\langle\textit{token list variable}\rangle$ is assigned locally. In the non-branching version, if the $\langle\textit{comma list}\rangle$ is empty the $\langle\textit{token list variable}\rangle$ is set to the marker value $\backslash\textit{q_no_value}$.

```
\clist_pop:NN \clist_pop:NN <comma list> <token list variable>
```

```
\clist_pop:cN
```

Updated: 2011-09-06

Pops the left-most item from a $\langle\textit{comma list}\rangle$ into the $\langle\textit{token list variable}\rangle$, *i.e.* removes the item from the comma list and stores it in the $\langle\textit{token list variable}\rangle$. Both of the variables are assigned locally.

`\clist_gpop:Nn`
`\clist_gpop:cn`

`\clist_gpop:Nn` $\langle comma list \rangle$ $\langle token list variable \rangle$

Pops the left-most item from a $\langle comma list \rangle$ into the $\langle token list variable \rangle$, *i.e.* removes the item from the comma list and stores it in the $\langle token list variable \rangle$. The $\langle comma list \rangle$ is modified globally, while the assignment of the $\langle token list variable \rangle$ is local.

`\clist_pop:NNTF`
`\clist_pop:cNTF`

New: 2012-05-14

`\clist_pop:NNTF` $\langle comma list \rangle$ $\langle token list variable \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

If the $\langle comma list \rangle$ is empty, leaves the $\langle false code \rangle$ in the input stream. The value of the $\langle token list variable \rangle$ is not defined in this case and should not be relied upon. If the $\langle comma list \rangle$ is non-empty, pops the top item from the $\langle comma list \rangle$ in the $\langle token list variable \rangle$, *i.e.* removes the item from the $\langle comma list \rangle$. Both the $\langle comma list \rangle$ and the $\langle token list variable \rangle$ are assigned locally.

`\clist_gpop:NNTF`
`\clist_gpop:cNTF`

New: 2012-05-14

`\clist_gpop:NNTF` $\langle comma list \rangle$ $\langle token list variable \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

If the $\langle comma list \rangle$ is empty, leaves the $\langle false code \rangle$ in the input stream. The value of the $\langle token list variable \rangle$ is not defined in this case and should not be relied upon. If the $\langle comma list \rangle$ is non-empty, pops the top item from the $\langle comma list \rangle$ in the $\langle token list variable \rangle$, *i.e.* removes the item from the $\langle comma list \rangle$. The $\langle comma list \rangle$ is modified globally, while the $\langle token list variable \rangle$ is assigned locally.

`\clist_push:Nn`
`\clist_push:(NV|No|Nx|cn|cV|co|cx)`
`\clist_gpush:Nn`
`\clist_gpush:(NV|No|Nx|cn|cV|co|cx)`

`\clist_push:Nn` $\langle comma list \rangle$ $\{\langle items \rangle\}$

Adds the $\{\langle items \rangle\}$ to the top of the $\langle comma list \rangle$. Spaces are removed from both sides of each item as for any n-type comma list.

22.8 Using a single item

`\clist_item:Nn` ★
`\clist_item:cn` ★
`\clist_item:nn` ★

New: 2014-07-17

`\clist_item:Nn` $\langle comma list \rangle$ $\{\langle integer expression \rangle\}$

Indexing items in the $\langle comma list \rangle$ from 1 at the top (left), this function evaluates the $\langle integer expression \rangle$ and leaves the appropriate item from the comma list in the input stream. If the $\langle integer expression \rangle$ is negative, indexing occurs from the bottom (right) of the comma list. When the $\langle integer expression \rangle$ is larger than the number of items in the $\langle comma list \rangle$ (as calculated by `\clist_count:N`) then the function expands to nothing.

T_EXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the $\langle item \rangle$ does not expand further when appearing in an x-type or e-type argument expansion.

<hr/> <code>\clist_rand_item:N</code> *	<code>\clist_rand_item:N</code> \langle <i>clist var</i> \rangle
<code>\clist_rand_item:c</code> *	<code>\clist_rand_item:n</code> $\{\langle$ <i>comma list</i> $\rangle\}$
<code>\clist_rand_item:n</code> *	Selects a pseudo-random item of the \langle <i>comma list</i> \rangle . If the \langle <i>comma list</i> \rangle has no item, the result is empty.
<hr/> New: 2016-12-06 <hr/>	

T_EXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the \langle *item* \rangle does not expand further when appearing in an **x**-type or **e**-type argument expansion.

22.9 Viewing comma lists

<hr/> <code>\clist_show:N</code>	<code>\clist_show:N</code> \langle <i>comma list</i> \rangle
<code>\clist_show:c</code>	Displays the entries in the \langle <i>comma list</i> \rangle in the terminal.
<hr/> Updated: 2021-04-29 <hr/>	

<hr/> <code>\clist_show:n</code>	<code>\clist_show:n</code> $\{\langle$ <i>tokens</i> $\rangle\}$
<code>\clist_show:n</code>	Displays the entries in the comma list in the terminal.
<hr/> Updated: 2013-08-03 <hr/>	

<hr/> <code>\clist_log:N</code>	<code>\clist_log:N</code> \langle <i>comma list</i> \rangle
<code>\clist_log:c</code>	Writes the entries in the \langle <i>comma list</i> \rangle in the log file. See also <code>\clist_show:N</code> which displays the result in the terminal.
<hr/> New: 2014-08-22 <hr/> Updated: 2021-04-29 <hr/>	

<hr/> <code>\clist_log:n</code>	<code>\clist_log:n</code> $\{\langle$ <i>tokens</i> $\rangle\}$
<code>\clist_log:n</code>	Writes the entries in the comma list in the log file. See also <code>\clist_show:n</code> which displays the result in the terminal.
<hr/> New: 2014-08-22 <hr/>	

22.10 Constant and scratch comma lists

<hr/> <code>\c_empty_clist</code>	Constant that is always empty.
<hr/> New: 2012-07-02 <hr/>	

<hr/> <code>\l_tmpa_clist</code>	Scratch comma lists for local assignment. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
<code>\l_tmpb_clist</code>	
<hr/> New: 2011-09-06 <hr/>	

<hr/> <code>\g_tmpa_clist</code>	Scratch comma lists for global assignment. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
<code>\g_tmpb_clist</code>	
<hr/> New: 2011-09-06 <hr/>	

Chapter 23

The l3token package

Token manipulation

This module deals with tokens. Now this is perhaps not the most precise description so let's try with a better description: When programming in T_EX, it is often desirable to know just what a certain token is: is it a control sequence or something else. Similarly one often needs to know if a control sequence is expandable or not, a macro or a primitive, how many arguments it takes etc. Another thing of great importance (especially when it comes to document commands) is looking ahead in the token stream to see if a certain character is present and maybe even remove it or disregard other tokens while scanning. This module provides functions for both and as such has two primary function categories: `\token_` for anything that deals with tokens and `\peek_` for looking ahead in the token stream.

Most functions we describe here can be used on control sequences, as those are tokens as well.

It is important to distinguish two aspects of a token: its “shape” (for lack of a better word), which affects the matching of delimited arguments and the comparison of token lists containing this token, and its “meaning”, which affects whether the token expands or what operation it performs. One can have tokens of different shapes with the same meaning, but not the converse.

For instance, `\if:w`, `\if_charcode:w`, and `\tex_if:D` are three names for the same internal operation of T_EX, namely the primitive testing the next two characters for equality of their character code. They have the same meaning hence behave identically in many situations. However, T_EX distinguishes them when searching for a delimited argument. Namely, the example function `\show_until_if:w` defined below takes everything until `\if:w` as an argument, despite the presence of other copies of `\if:w` under different names.

```
\cs_new:Npn \show_until_if:w #1 \if:w { \tl_show:n {#1} }  
\show_until_if:w \tex_if:D \if_charcode:w \if:w
```

A list of all possible shapes and a list of all possible meanings are given in section [23.7](#).

23.1 Creating character tokens

```
\char_set_active_eq:NN
\char_set_active_eq:Nc
\char_gset_active_eq:NN
\char_gset_active_eq:Nc
```

Updated: 2015-11-12

```
\char_set_active_eq:nN {\integer expression} {\function}
\char_set_active_eq:nc
\char_gset_active_eq:nN
\char_gset_active_eq:nc
```

New: 2015-11-12

```
\char_generate:nn *
```

New: 2015-09-09

Updated: 2019-01-16

```
\char_set_active_eq:NN <char> <function>
```

Sets the behaviour of the $\langle char \rangle$ in situations where it is active (category code 13) to be equivalent to that of the $\langle function \rangle$. The category code of the $\langle char \rangle$ is *unchanged* by this process. The $\langle function \rangle$ may itself be an active character.

```
\char_set_active_eq:nN {\integer expression} {\function}
```

Sets the behaviour of the $\langle char \rangle$ which has character code as given by the $\langle integer expression \rangle$ in situations where it is active (category code 13) to be equivalent to that of the $\langle function \rangle$. The category code of the $\langle char \rangle$ is *unchanged* by this process. The $\langle function \rangle$ may itself be an active character.

```
\char_generate:nn {\charcode} {\catcode}
```

Generates a character token of the given $\langle charcode \rangle$ and $\langle catcode \rangle$ (both of which may be integer expressions). The $\langle catcode \rangle$ may be one of

- 1 (begin group)
- 2 (end group)
- 3 (math toggle)
- 4 (alignment)
- 6 (parameter)
- 7 (math superscript)
- 8 (math subscript)
- 11 (letter)
- 12 (other)
- 13 (active)

and other values raise an error. The $\langle charcode \rangle$ may be any one valid for the engine in use. Active characters cannot be generated in older versions of \TeX . Another way to build token lists with unusual category codes is `\regex_replace:nnN {.*} {\replacement} <tl var>`.

\TeX hackers note: Exactly two expansions are needed to produce the character.

<code>\char_lowercase:N</code>	*	<code>\char_lowercase:N</code> $\langle char \rangle$
<code>\char_uppercase:N</code>	*	
<code>\char_titlecase:N</code>	*	Converts the $\langle char \rangle$ to the equivalent case-changed character as detailed by the function name (see <code>\str_foldcase:n</code> and <code>\text_titlecase:n</code> for details of these terms).
<code>\char_foldcase:N</code>	*	
<code>\char_str_lowercase:N</code>	*	The case mapping is carried out with no context-dependence (<i>cf.</i> <code>\text_uppercase:n</code> , <i>etc.</i>) The <code>str</code> versions always generate “other” (category code 12) characters, whilst the standard versions generate characters with the category code of the $\langle char \rangle$ (i.e. only the character code changes).
<code>\char_str_uppercase:N</code>	*	
<code>\char_str_titlecase:N</code>	*	
<code>\char_str_foldcase:N</code>	*	

New: 2020-01-09

<code>\c_catcode_other_space_tl</code>	Token list containing one character with category code 12, (“other”), and character code 32 (space).
--	--

New: 2011-09-05

23.2 Manipulating and interrogating character tokens

<code>\char_set_catcode_escape:N</code>	<code>\char_set_catcode_letter:N</code> $\langle character \rangle$
<code>\char_set_catcode_group_begin:N</code>	
<code>\char_set_catcode_group_end:N</code>	
<code>\char_set_catcode_math_toggle:N</code>	
<code>\char_set_catcode_alignment:N</code>	
<code>\char_set_catcode_end_line:N</code>	
<code>\char_set_catcode_parameter:N</code>	
<code>\char_set_catcode_math_superscript:N</code>	
<code>\char_set_catcode_math_subscript:N</code>	
<code>\char_set_catcode_ignore:N</code>	
<code>\char_set_catcode_space:N</code>	
<code>\char_set_catcode_letter:N</code>	
<code>\char_set_catcode_other:N</code>	
<code>\char_set_catcode_active:N</code>	
<code>\char_set_catcode_comment:N</code>	
<code>\char_set_catcode_invalid:N</code>	

Updated: 2015-11-11

Sets the category code of the $\langle character \rangle$ to that indicated in the function name. Depending on the current category code of the $\langle token \rangle$ the escape token may also be needed:

`\char_set_catcode_other:N \%`

The assignment is local.

<code>\char_set_catcode_escape:n</code>	<code>\char_set_catcode_letter:n {⟨integer expression⟩}</code>
<code>\char_set_catcode_group_begin:n</code>	
<code>\char_set_catcode_group_end:n</code>	
<code>\char_set_catcode_math_toggle:n</code>	
<code>\char_set_catcode_alignment:n</code>	
<code>\char_set_catcode_end_line:n</code>	
<code>\char_set_catcode_parameter:n</code>	
<code>\char_set_catcode_math_superscript:n</code>	
<code>\char_set_catcode_math_subscript:n</code>	
<code>\char_set_catcode_ignore:n</code>	
<code>\char_set_catcode_space:n</code>	
<code>\char_set_catcode_letter:n</code>	
<code>\char_set_catcode_other:n</code>	
<code>\char_set_catcode_active:n</code>	
<code>\char_set_catcode_comment:n</code>	
<code>\char_set_catcode_invalid:n</code>	

Updated: 2015-11-11

Sets the category code of the $\langle character \rangle$ which has character code as given by the $\langle integer expression \rangle$. This version can be used to set up characters which cannot otherwise be given (*cf.* the N-type variants). The assignment is local.

<code>\char_set_catcode:nn</code>	<code>\char_set_catcode:nn {⟨intexpr₁⟩} {⟨intexpr₂⟩}</code>
-----------------------------------	---

Updated: 2015-11-11

These functions set the category code of the $\langle character \rangle$ which has character code as given by the $\langle integer expression \rangle$. The first $\langle integer expression \rangle$ is the character code and the second is the category code to apply. The setting applies within the current \TeX group. In general, the symbolic functions `\char_set_catcode_⟨type⟩` should be preferred, but there are cases where these lower-level functions may be useful.

<code>\char_value_catcode:n *</code>	<code>\char_value_catcode:n {⟨integer expression⟩}</code>
--------------------------------------	---

Expands to the current category code of the $\langle character \rangle$ with character code given by the $\langle integer expression \rangle$.

<code>\char_show_value_catcode:n</code>	<code>\char_show_value_catcode:n {⟨integer expression⟩}</code>
---	--

Displays the current category code of the $\langle character \rangle$ with character code given by the $\langle integer expression \rangle$ on the terminal.

<code>\char_set_lccode:nn</code>	<code>\char_set_lccode:nn {⟨intexpr₁⟩} {⟨intexpr₂⟩}</code>
----------------------------------	--

Updated: 2015-08-06

Sets up the behaviour of the $\langle character \rangle$ when found inside `\text_lowercase:n`, such that $\langle character_1 \rangle$ will be converted into $\langle character_2 \rangle$. The two $\langle characters \rangle$ may be specified using an $\langle integer expression \rangle$ for the character code concerned. This may include the \TeX ‘ $\langle character \rangle$ ’ method for converting a single character into its character code:

```
\char_set_lccode:nn { ‘\A } { ‘\a } % Standard behaviour
\char_set_lccode:nn { ‘\A } { ‘\A + 32 }
\char_set_lccode:nn { 50 } { 60 }
```

The setting applies within the current \TeX group.

<hr/> <hr/>	<hr/>
<code>\char_value_lccode:n *</code>	<code>\char_value_lccode:n {⟨integer expression⟩}</code>
	Expands to the current lower case code of the $\langle character \rangle$ with character code given by the $\langle integer expression \rangle$.
<hr/> <hr/>	<hr/>
<code>\char_show_value_lccode:n</code>	<code>\char_show_value_lccode:n {⟨integer expression⟩}</code>
	Displays the current lower case code of the $\langle character \rangle$ with character code given by the $\langle integer expression \rangle$ on the terminal.
<hr/> <hr/>	<hr/>
<code>\char_set_uccode:nn</code>	<code>\char_set_uccode:nn {⟨intexpr₁⟩} {⟨intexpr₂⟩}</code>
<hr/> <hr/>	<hr/>
Updated: 2015-08-06	Sets up the behaviour of the $\langle character \rangle$ when found inside <code>\text_uppercase:n</code> , such that $\langle character_1 \rangle$ will be converted into $\langle character_2 \rangle$. The two $\langle characters \rangle$ may be specified using an $\langle integer expression \rangle$ for the character code concerned. This may include the TeX ‘ $\langle character \rangle$ ’ method for converting a single character into its character code:
	<pre> \char_set_uccode:nn { ‘\a } { ‘\A } % Standard behaviour \char_set_uccode:nn { ‘\A } { ‘\A - 32 } \char_set_uccode:nn { 60 } { 50 } </pre>
	The setting applies within the current TeX group.
<hr/> <hr/>	<hr/>
<code>\char_value_uccode:n *</code>	<code>\char_value_uccode:n {⟨integer expression⟩}</code>
	Expands to the current upper case code of the $\langle character \rangle$ with character code given by the $\langle integer expression \rangle$.
<hr/> <hr/>	<hr/>
<code>\char_show_value_uccode:n</code>	<code>\char_show_value_uccode:n {⟨integer expression⟩}</code>
	Displays the current upper case code of the $\langle character \rangle$ with character code given by the $\langle integer expression \rangle$ on the terminal.
<hr/> <hr/>	<hr/>
<code>\char_set_mathcode:nn</code>	<code>\char_set_mathcode:nn {⟨intexpr₁⟩} {⟨intexpr₂⟩}</code>
<hr/> <hr/>	<hr/>
Updated: 2015-08-06	This function sets up the math code of $\langle character \rangle$. The $\langle character \rangle$ is specified as an $\langle integer expression \rangle$ which will be used as the character code of the relevant character. The setting applies within the current TeX group.
<hr/> <hr/>	<hr/>
<code>\char_value_mathcode:n *</code>	<code>\char_value_mathcode:n {⟨integer expression⟩}</code>
	Expands to the current math code of the $\langle character \rangle$ with character code given by the $\langle integer expression \rangle$.
<hr/> <hr/>	<hr/>
<code>\char_show_value_mathcode:n</code>	<code>\char_show_value_mathcode:n {⟨integer expression⟩}</code>
	Displays the current math code of the $\langle character \rangle$ with character code given by the $\langle integer expression \rangle$ on the terminal.
<hr/> <hr/>	<hr/>
<code>\char_set_sfcode:nn</code>	<code>\char_set_sfcode:nn {⟨intexpr₁⟩} {⟨intexpr₂⟩}</code>
<hr/> <hr/>	<hr/>
Updated: 2015-08-06	This function sets up the space factor for the $\langle character \rangle$. The $\langle character \rangle$ is specified as an $\langle integer expression \rangle$ which will be used as the character code of the relevant character. The setting applies within the current TeX group.

<hr/> <hr/>	
<code>\char_value_sfcode:n</code> ★	<code>\char_value_sfcode:n {⟨integer expression⟩}</code>
	Expands to the current space factor for the $\langle character \rangle$ with character code given by the $\langle integer expression \rangle$.
<hr/> <hr/>	
<code>\char_show_value_sfcode:n</code>	<code>\char_show_value_sfcode:n {⟨integer expression⟩}</code>
	Displays the current space factor for the $\langle character \rangle$ with character code given by the $\langle integer expression \rangle$ on the terminal.
<hr/> <hr/>	
<code>\l_char_active_seq</code>	Used to track which tokens may require special handling at the document level as they
New: 2012-01-23	are (or have been at some point) of category $\langle active \rangle$ (catcode 13). Each entry in the
Updated: 2015-11-11	sequence consists of a single escaped token, for example <code>\~</code> . Active tokens should be
	added to the sequence when they are defined for general document use.
<hr/> <hr/>	
<code>\l_char_special_seq</code>	Used to track which tokens will require special handling when working with verbatim-
New: 2012-01-23	like material at the document level as they are not of categories $\langle letter \rangle$ (catcode 11)
Updated: 2015-11-11	or $\langle other \rangle$ (catcode 12). Each entry in the sequence consists of a single escaped token,
	for example <code>\</code> for the backslash or <code>\{</code> for an opening brace. Escaped tokens should be
	added to the sequence when they are defined for general document use.

23.3 Generic tokens

<hr/> <hr/>	
<code>\c_group_begin_token</code>	These are implicit tokens which have the category code described by their name. They are used internally for test purposes but are also available to the programmer for other uses.
<code>\c_group_end_token</code>	
<code>\c_math_toggle_token</code>	
<code>\c_alignment_token</code>	
<code>\c_parameter_token</code>	
<code>\c_math_superscript_token</code>	
<code>\c_math_subscript_token</code>	
<code>\c_space_token</code>	
<hr/> <hr/>	
<code>\c_catcode_letter_token</code>	These are implicit tokens which have the category code described by their name. They are used internally for test purposes and should not be used other than for category code tests.
<code>\c_catcode_other_token</code>	
<hr/> <hr/>	
<code>\c_catcode_active_tl</code>	A token list containing an active token. This is used internally for test purposes and should not be used other than in appropriately-constructed category code tests.

23.4 Converting tokens

<code>\token_to_meaning:N</code>	★	<code>\token_to_meaning:N</code>	$\langle token \rangle$
<code>\token_to_meaning:c</code>	★		

Inserts the current meaning of the $\langle token \rangle$ into the input stream as a series of characters of category code 12 (other). This is the primitive \TeX description of the $\langle token \rangle$, thus for example both functions defined by `\cs_set_nopar:Npn` and token list variables defined using `\tl_new:N` are described as macros.

\TeX hackers note: This is the \TeX primitive `\meaning`. The $\langle token \rangle$ can thus be an explicit space tokens or an explicit begin-group or end-group character token (`{` or `}` when normal \TeX category codes apply) even though these are not valid N-type arguments.

<code>\token_to_str:N</code>	★	<code>\token_to_str:N</code>	$\langle token \rangle$
<code>\token_to_str:c</code>	★		

Converts the given $\langle token \rangle$ into a series of characters with category code 12 (other). If the $\langle token \rangle$ is a control sequence, this will start with the current escape character with category code 12 (the escape character is part of the $\langle token \rangle$). This function requires only a single expansion.

\TeX hackers note: `\token_to_str:N` is the \TeX primitive `\string` renamed. The $\langle token \rangle$ can thus be an explicit space tokens or an explicit begin-group or end-group character token (`{` or `}` when normal \TeX category codes apply) even though these are not valid N-type arguments.

23.5 Token conditionals

<code>\token_if_group_begin_p:N</code>	★	<code>\token_if_group_begin_p:N</code>	$\langle token \rangle$
<code>\token_if_group_begin:NTF</code>	★	<code>\token_if_group_begin:NTF</code>	$\langle token \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Tests if $\langle token \rangle$ has the category code of a begin group token (`{` when normal \TeX category codes are in force). Note that an explicit begin group token cannot be tested in this way, as it is not a valid N-type argument.

<code>\token_if_group_end_p:N</code>	★	<code>\token_if_group_end_p:N</code>	$\langle token \rangle$
<code>\token_if_group_end:NTF</code>	★	<code>\token_if_group_end:NTF</code>	$\langle token \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Tests if $\langle token \rangle$ has the category code of an end group token (`}` when normal \TeX category codes are in force). Note that an explicit end group token cannot be tested in this way, as it is not a valid N-type argument.

<code>\token_if_math_toggle_p:N</code>	★	<code>\token_if_math_toggle_p:N</code>	$\langle token \rangle$
<code>\token_if_math_toggle:NTF</code>	★	<code>\token_if_math_toggle:NTF</code>	$\langle token \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Tests if $\langle token \rangle$ has the category code of a math shift token (`$` when normal \TeX category codes are in force).

<code>\token_if_alignment_p:N *</code>	<code>\token_if_alignment_p:N <token></code>
<code>\token_if_alignment:NTF *</code>	<code>\token_if_alignment:NTF <token> {\true code} {\false code}</code>

Tests if *<token>* has the category code of an alignment token (& when normal T_EX category codes are in force).

<code>\token_if_parameter_p:N *</code>	<code>\token_if_parameter_p:N <token></code>
<code>\token_if_parameter:NTF *</code>	<code>\token_if_parameter:NTF <token> {\true code} {\false code}</code>

Tests if *<token>* has the category code of a macro parameter token (# when normal T_EX category codes are in force).

<code>\token_if_math_superscript_p:N *</code>	<code>\token_if_math_superscript_p:N <token></code>
<code>\token_if_math_superscript:NTF *</code>	<code>\token_if_math_superscript:NTF <token> {\true code} {\false code}</code>

Tests if *<token>* has the category code of a superscript token (^ when normal T_EX category codes are in force).

<code>\token_if_math_subscript_p:N *</code>	<code>\token_if_math_subscript_p:N <token></code>
<code>\token_if_math_subscript:NTF *</code>	<code>\token_if_math_subscript:NTF <token> {\true code} {\false code}</code>

Tests if *<token>* has the category code of a subscript token (_ when normal T_EX category codes are in force).

<code>\token_if_space_p:N *</code>	<code>\token_if_space_p:N <token></code>
<code>\token_if_space:NTF *</code>	<code>\token_if_space:NTF <token> {\true code} {\false code}</code>

Tests if *<token>* has the category code of a space token. Note that an explicit space token with character code 32 cannot be tested in this way, as it is not a valid N-type argument.

<code>\token_if_letter_p:N *</code>	<code>\token_if_letter_p:N <token></code>
<code>\token_if_letter:NTF *</code>	<code>\token_if_letter:NTF <token> {\true code} {\false code}</code>

Tests if *<token>* has the category code of a letter token.

<code>\token_if_other_p:N *</code>	<code>\token_if_other_p:N <token></code>
<code>\token_if_other:NTF *</code>	<code>\token_if_other:NTF <token> {\true code} {\false code}</code>

Tests if *<token>* has the category code of an “other” token.

<code>\token_if_active_p:N *</code>	<code>\token_if_active_p:N <token></code>
<code>\token_if_active:NTF *</code>	<code>\token_if_active:NTF <token> {\true code} {\false code}</code>

Tests if *<token>* has the category code of an active character.

<code>\token_if_eq_catcode_p:NN *</code>	<code>\token_if_eq_catcode_p:NN <token₁₂</code>
<code>\token_if_eq_catcode:NNTF *</code>	<code>\token_if_eq_catcode:NNTF <token₁₂</code>

Tests if the two *<tokens>* have the same category code.

<code>\token_if_eq_charcode_p:NN *</code>	<code>\token_if_eq_charcode_p:NN <token₁₂</code>
<code>\token_if_eq_charcode:NNTF *</code>	<code>\token_if_eq_charcode:NNTF <token₁₂</code>

Tests if the two *<tokens>* have the same character code.

<code>\token_if_eq_meaning_p:NN</code>	<code>*</code>	<code>\token_if_eq_meaning_p:NN</code>	<code><token₁></code>	<code><token₂></code>
<code>\token_if_eq_meaning:NNTF</code>	<code>*</code>	<code>\token_if_eq_meaning:NNTF</code>	<code><token₁></code>	<code><token₂></code>

Tests if the two `<tokens>` have the same meaning when expanded.

<code>\token_if_macro_p:N</code>	<code>*</code>	<code>\token_if_macro_p:N</code>	<code><token></code>
<code>\token_if_macro:NNTF</code>	<code>*</code>	<code>\token_if_macro:NNTF</code>	<code><token></code>

Updated: 2011-05-23

Tests if the `<token>` is a \TeX macro.

<code>\token_if_cs_p:N</code>	<code>*</code>	<code>\token_if_cs_p:N</code>	<code><token></code>
<code>\token_if_cs:NNTF</code>	<code>*</code>	<code>\token_if_cs:NNTF</code>	<code><token></code>

Tests if the `<token>` is a control sequence.

<code>\token_if_expandable_p:N</code>	<code>*</code>	<code>\token_if_expandable_p:N</code>	<code><token></code>
<code>\token_if_expandable:NNTF</code>	<code>*</code>	<code>\token_if_expandable:NNTF</code>	<code><token></code>

Tests if the `<token>` is expandable. This test returns `<false>` for an undefined token.

<code>\token_if_long_macro_p:N</code>	<code>*</code>	<code>\token_if_long_macro_p:N</code>	<code><token></code>
<code>\token_if_long_macro:NNTF</code>	<code>*</code>	<code>\token_if_long_macro:NNTF</code>	<code><token></code>

Updated: 2012-01-20

Tests if the `<token>` is a long macro.

<code>\token_if_protected_macro_p:N</code>	<code>*</code>	<code>\token_if_protected_macro_p:N</code>	<code><token></code>
<code>\token_if_protected_macro:NNTF</code>	<code>*</code>	<code>\token_if_protected_macro:NNTF</code>	<code><token></code>

Updated: 2012-01-20

Tests if the `<token>` is a protected macro: for a macro which is both protected and long this returns `false`.

<code>\token_if_protected_long_macro_p:N</code>	<code>*</code>	<code>\token_if_protected_long_macro_p:N</code>	<code><token></code>
<code>\token_if_protected_long_macro:NNTF</code>	<code>*</code>	<code>\token_if_protected_long_macro:NNTF</code>	<code><token></code>

Updated: 2012-01-20

Tests if the `<token>` is a protected long macro.

<code>\token_if_chardef_p:N</code>	<code>*</code>	<code>\token_if_chardef_p:N</code>	<code><token></code>
<code>\token_if_chardef:NNTF</code>	<code>*</code>	<code>\token_if_chardef:NNTF</code>	<code><token></code>

Updated: 2012-01-20

Tests if the `<token>` is defined to be a chardef.

\TeX hackers note: Booleans, boxes and small integer constants are implemented as `\chardefs`.

<code>\token_if_mathchardef_p:N</code>	<code>*</code>	<code>\token_if_mathchardef_p:N</code>	<code><token></code>
<code>\token_if_mathchardef:NNTF</code>	<code>*</code>	<code>\token_if_mathchardef:NNTF</code>	<code><token></code>

Updated: 2012-01-20

Tests if the `<token>` is defined to be a mathchardef.

<code>\token_if_font_selection_p:N</code>	<code>*</code>	<code>\token_if_font_selection_p:N</code>	<code><token></code>
<code>\token_if_font_selection:NTF</code>	<code>*</code>	<code>\token_if_font_selection:NTF</code>	<code><token> {\true code} {\false code}</code>

New: 2020-10-27

Tests if the `<token>` is defined to be a font selection command.

<code>\token_if_dim_register_p:N</code>	<code>*</code>	<code>\token_if_dim_register_p:N</code>	<code><token></code>
<code>\token_if_dim_register:NTF</code>	<code>*</code>	<code>\token_if_dim_register:NTF</code>	<code><token> {\true code} {\false code}</code>

Updated: 2012-01-20

Tests if the `<token>` is defined to be a dimension register.

<code>\token_if_int_register_p:N</code>	<code>*</code>	<code>\token_if_int_register_p:N</code>	<code><token></code>
<code>\token_if_int_register:NTF</code>	<code>*</code>	<code>\token_if_int_register:NTF</code>	<code><token> {\true code} {\false code}</code>

Updated: 2012-01-20

Tests if the `<token>` is defined to be a integer register.

TeXhackers note: Constant integers may be implemented as integer registers, `\chardefs`, or `\mathchardefs` depending on their value.

<code>\token_if_muskip_register_p:N</code>	<code>*</code>	<code>\token_if_muskip_register_p:N</code>	<code><token></code>
<code>\token_if_muskip_register:NTF</code>	<code>*</code>	<code>\token_if_muskip_register:NTF</code>	<code><token> {\true code} {\false code}</code>

New: 2012-02-15

Tests if the `<token>` is defined to be a muskip register.

<code>\token_if_skip_register_p:N</code>	<code>*</code>	<code>\token_if_skip_register_p:N</code>	<code><token></code>
<code>\token_if_skip_register:NTF</code>	<code>*</code>	<code>\token_if_skip_register:NTF</code>	<code><token> {\true code} {\false code}</code>

Updated: 2012-01-20

Tests if the `<token>` is defined to be a skip register.

<code>\token_if_toks_register_p:N</code>	<code>*</code>	<code>\token_if_toks_register_p:N</code>	<code><token></code>
<code>\token_if_toks_register:NTF</code>	<code>*</code>	<code>\token_if_toks_register:NTF</code>	<code><token> {\true code} {\false code}</code>

Updated: 2012-01-20

Tests if the `<token>` is defined to be a toks register (not used by L^AT_EX3).

<code>\token_if_primitive_p:N</code>	<code>*</code>	<code>\token_if_primitive_p:N</code>	<code><token></code>
<code>\token_if_primitive:NTF</code>	<code>*</code>	<code>\token_if_primitive:NTF</code>	<code><token> {\true code} {\false code}</code>

Updated: 2020-09-11

Tests if the `<token>` is an engine primitive. In LuaTeX this includes primitive-like commands defined using `{token.set_lua}`.

<hr/>	
<code>\token_case_catcode:Nn</code>	<code>*</code>
<code>\token_case_catcode:NnTF</code>	<code>*</code>
<code>\token_case_charcode:Nn</code>	<code>*</code>
<code>\token_case_charcode:NnTF</code>	<code>*</code>
<code>\token_case_meaning:Nn</code>	<code>*</code>
<code>\token_case_meaning:NnTF</code>	<code>*</code>
<hr/>	
New: 2020-12-03	
<hr/>	

```

\token_case_meaning:NnTF <test token>
{
  <token case1> {\code case1>}
  <token case2> {\code case2>}
  ...
  <token casen> {\code casen>}
}
{\true code}&
{\false code}&

```

This function compares the $\langle test\ token \rangle$ in turn with each of the $\langle token\ cases \rangle$. If the two are equal (as described for `\token_if_eq_catcode:NNTF`, `\token_if_eq_charcode:NNTF` and `\token_if_eq_meaning:NNTF`, respectively) then the associated $\langle code \rangle$ is left in the input stream and other cases are discarded. If any of the cases are matched, the $\langle true\ code \rangle$ is also inserted into the input stream (after the code for the appropriate case), while if none match then the $\langle false\ code \rangle$ is inserted. The functions `\token_case_catcode:Nn`, `\token_case_charcode:Nn`, and `\token_case_meaning:Nn`, which do nothing if there is no match, are also available.

23.6 Peeking ahead at the next token

There is often a need to look ahead at the next token in the input stream while leaving it in place. This is handled using the “peek” functions. The generic `\peek_after:Nw` is provided along with a family of predefined tests for common cases. As peeking ahead does *not* skip spaces the predefined tests include both a space-respecting and space-skipping version. In addition, using `\peek_analysis_map_inline:n`, one can map through the following tokens in the input stream and repeatedly perform some tests.

<hr/>	
<code>\peek_after:Nw</code>	<code>\peek_after:Nw <function> <token></code>
<hr/>	
Locally sets the test variable <code>\l_peek_token</code> equal to $\langle token \rangle$ (as an implicit token, <i>not</i> as a token list), and then expands the $\langle function \rangle$. The $\langle token \rangle$ remains in the input stream as the next item after the $\langle function \rangle$. The $\langle token \rangle$ here may be \sqcup , $\{$ or $\}$ (assuming normal T _E X category codes), <i>i.e.</i> it is not necessarily the next argument which would be grabbed by a normal function.	
<hr/>	
<code>\peek_gafter:Nw</code>	<code>\peek_gafter:Nw <function> <token></code>
<hr/>	
Globally sets the test variable <code>\g_peek_token</code> equal to $\langle token \rangle$ (as an implicit token, <i>not</i> as a token list), and then expands the $\langle function \rangle$. The $\langle token \rangle$ remains in the input stream as the next item after the $\langle function \rangle$. The $\langle token \rangle$ here may be \sqcup , $\{$ or $\}$ (assuming normal T _E X category codes), <i>i.e.</i> it is not necessarily the next argument which would be grabbed by a normal function.	
<hr/>	
<code>\l_peek_token</code>	Token set by <code>\peek_after:Nw</code> and available for testing as described above.
<hr/>	
<code>\g_peek_token</code>	Token set by <code>\peek_gafter:Nw</code> and available for testing as described above.
<hr/>	

<hr/> \peek_catcode:NTF <hr/>	\peek_catcode:NTF <i><test token></i> <i>{<true code>}</i> <i>{<false code>}</i>
Updated: 2012-12-20	Tests if the next <i><token></i> in the input stream has the same category code as the <i><test token></i> (as defined by the test \token_if_eq_catcode:NNTF). Spaces are respected by the test and the <i><token></i> is left in the input stream after the <i><true code></i> or <i><false code></i> (as appropriate to the result of the test).
<hr/> \peek_catcode_remove:NTF <hr/>	\peek_catcode_remove:NTF <i><test token></i> <i>{<true code>}</i> <i>{<false code>}</i>
Updated: 2012-12-20	Tests if the next <i><token></i> in the input stream has the same category code as the <i><test token></i> (as defined by the test \token_if_eq_catcode:NNTF). Spaces are respected by the test and the <i><token></i> is removed from the input stream if the test is true. The function then places either the <i><true code></i> or <i><false code></i> in the input stream (as appropriate to the result of the test).
<hr/> \peek_charcode:NTF <hr/>	\peek_charcode:NTF <i><test token></i> <i>{<true code>}</i> <i>{<false code>}</i>
Updated: 2012-12-20	Tests if the next <i><token></i> in the input stream has the same character code as the <i><test token></i> (as defined by the test \token_if_eq_charcode:NNTF). Spaces are respected by the test and the <i><token></i> is left in the input stream after the <i><true code></i> or <i><false code></i> (as appropriate to the result of the test).
<hr/> \peek_charcode_remove:NTF <hr/>	\peek_charcode_remove:NTF <i><test token></i> <i>{<true code>}</i> <i>{<false code>}</i>
Updated: 2012-12-20	Tests if the next <i><token></i> in the input stream has the same character code as the <i><test token></i> (as defined by the test \token_if_eq_charcode:NNTF). Spaces are respected by the test and the <i><token></i> is removed from the input stream if the test is true. The function then places either the <i><true code></i> or <i><false code></i> in the input stream (as appropriate to the result of the test).
<hr/> \peek_meaning:NTF <hr/>	\peek_meaning:NTF <i><test token></i> <i>{<true code>}</i> <i>{<false code>}</i>
Updated: 2011-07-02	Tests if the next <i><token></i> in the input stream has the same meaning as the <i><test token></i> (as defined by the test \token_if_eq_meaning:NNTF). Spaces are respected by the test and the <i><token></i> is left in the input stream after the <i><true code></i> or <i><false code></i> (as appropriate to the result of the test).
<hr/> \peek_meaning_remove:NTF <hr/>	\peek_meaning_remove:NTF <i><test token></i> <i>{<true code>}</i> <i>{<false code>}</i>
Updated: 2011-07-02	Tests if the next <i><token></i> in the input stream has the same meaning as the <i><test token></i> (as defined by the test \token_if_eq_meaning:NNTF). Spaces are respected by the test and the <i><token></i> is removed from the input stream if the test is true. The function then places either the <i><true code></i> or <i><false code></i> in the input stream (as appropriate to the result of the test).
<hr/> \peek_remove_spaces:n <hr/>	\peek_remove_spaces:n <i>{<code>}</i>
New: 2018-10-01	Peeks ahead and detect if the following token is a space (category code 10 and character code 32). If so, removes the token and checks the next token. Once a non-space token is found, the <i><code></i> will be inserted into the input stream. Typically this will contain a peek operation, but this is not required.

`\peek_remove_filler:n`

New: 2022-01-10

`\peek_remove_filler:n` `{\code}`

Peeks ahead and detect if the following token is a space (category code 10) or has meaning equal to `\scan_stop:`. If so, removes the token and checks the next token. If neither of these cases apply, expands the next token using f-type expansion, then checks the resulting leading token in the same way. If after expansion the next token is neither of the two test cases, the `\code` will be inserted into the input stream. Typically this will contain a `peek` operation, but this is not required.

T_EXhackers note: This is essentially a macro-based implementation of how T_EX handles the search for a left brace after for example `\everypar`, except that any non-expandable token cleanly ends the `\filler` (i.e. it does not lead to a T_EX error).

In contrast to T_EX's filler removal, a construct `\exp_not:N \foo` will be treated in the same way as `\foo`.

`\peek_N_type:TF`

Updated: 2012-12-20

`\peek_N_type:TF` `{\true code}{\false code}`

Tests if the next `\token` in the input stream can be safely grabbed as an N-type argument. The test is `\false` if the next `\token` is either an explicit or implicit begin-group or end-group token (with any character code), or an explicit or implicit space character (with character code 32 and category code 10), or an outer token (never used in L^AT_EX3) and `\true` in all other cases. Note that a `\true` result ensures that the next `\token` is a valid N-type argument. However, if the next `\token` is for instance `\c_space_token`, the test takes the `\false` branch, even though the next `\token` is in fact a valid N-type argument. The `\token` is left in the input stream after the `\true code` or `\false code` (as appropriate to the result of the test).

<code>\peek_analysis_map_inline:n</code>	<code>\peek_analysis_map_inline:n {<i><inline function></i>}</code>
--	---

New: 2020-12-03

Repeatedly removes one *<token>* from the input stream and applies the *<inline function>* to it, until `\peek_analysis_map_break:` is called. The *<inline function>* receives three arguments for each *<token>* in the input stream:

- *<tokens>*, which both o-expand and x-expand to the *<token>*. The detailed form of *<tokens>* may change in later releases.
- *<char code>*, a decimal representation of the character code of the *<token>*, -1 if it is a control sequence.
- *<catcode>*, a capital hexadecimal digit which denotes the category code of the *<token>* (0: control sequence, 1: begin-group, 2: end-group, 3: math shift, 4: alignment tab, 6: parameter, 7: superscript, 8: subscript, A: space, B: letter, C: other, D: active). This can be converted to an integer by writing "*<catcode>*".

These arguments are the same as for `\tl_analysis_map_inline:nn` defined in `l3tl-analysis`. The *<char code>* and *<catcode>* do not take the meaning of a control sequence or active character into account: for instance, upon encountering the token `\c_group_begin_token` in the input stream, `\peek_analysis_map_inline:n` calls the *<inline function>* with #1 being `\exp_not:n { \c_group_begin_token }` (with the current implementation), #2 being -1 , and #3 being 0, as for any other control sequence. In contrast, upon encountering an explicit begin-group token `{`, the *<inline function>* is called with arguments `\exp_after:wN { \if_false: } \fi:`, 123 and 1.

The mapping is done at the current group level, *i.e.* any local assignments made by the *<inline function>* remain in effect after the loop. Within the code, `\l_peek_token` is set equal (as a token, not a token list) to the token under consideration.

<code>\peek_analysis_map_break:</code>	<code>\peek_analysis_map_inline:n</code>
<code>\peek_analysis_map_break:n</code>	<code>{ ... \peek_analysis_map_break:n {<i><code></i>} }</code>

New: 2020-12-03

Stops the `\peek_analysis_map_inline:n` loop from seeking more tokens, and inserts *<code>* in the input stream (empty for `\peek_analysis_map_break:`).

<code>\peek_regex:nTF</code>	<code>\peek_regex:nTF {<i><regex></i>} {<i><true code></i>} {<i><false code></i>}</code>
<code>\peek_regex:NTF</code>	

New: 2020-12-03

Tests if the *<tokens>* that follow in the input stream match the *<regular expression>*. Any *<tokens>* that have been read are left in the input stream after the *<true code>* or *<false code>* (as appropriate to the result of the test). See `l3regex` for documentation of the syntax of regular expressions. The *<regular expression>* is implicitly anchored at the start, so for instance `\peek_regex:nTF { a }` is essentially equivalent to `\peek_charcode:NTF a`.

T_EXhackers note: Implicit character tokens are correctly considered by `\peek_regex:nTF` as control sequences, while functions that inspect individual tokens (for instance `\peek_charcode:NTF`) only take into account their meaning.

The `\peek_regex:nTF` function only inspects as few tokens as necessary to determine whether the regular expression matches. For instance `\peek_regex:nTF { abc | [a-z] } { } { }` `abc` will only inspect the first token `a` even though the first branch `abc` of the alternative is preferred in functions such as `\peek_regex_remove_once:n`. This may have an effect on tokenization if the input stream has not yet been tokenized and category codes are changed.

```
\peek_regex_remove_once:nTF \peek_regex_remove_once:nTF {\<regex>} {\<true code>} {\<false code>}
\peek_regex_remove_once:NTF
```

New: 2020-12-03

Tests if the $\langle tokens \rangle$ that follow in the input stream match the $\langle regex \rangle$. If the test is true, the $\langle tokens \rangle$ are removed from the input stream and the $\langle true code \rangle$ is inserted, while if the test is false, the $\langle false code \rangle$ is inserted followed by the $\langle tokens \rangle$ that were originally in the input stream. See `l3regex` for documentation of the syntax of regular expressions. The $\langle regular expression \rangle$ is implicitly anchored at the start, so for instance `\peek_regex_remove_once:nTF { a }` is essentially equivalent to `\peek_charcode_remove:NTF a`.

TeXhackers note: Implicit character tokens are correctly considered by `\peek_regex_remove_once:nTF` as control sequences, while functions that inspect individual tokens (for instance `\peek_charcode:NTF`) only take into account their meaning.

```
\peek_regex_replace_once:nn \peek_regex_replace_once:nnTF {\<regex>} {\<replacement>} {\<true code>}
\peek_regex_replace_once:nnTF {\<false code>}
\peek_regex_replace_once:Nn
\peek_regex_replace_once:NnTF
```

New: 2020-12-03

If the $\langle tokens \rangle$ that follow in the input stream match the $\langle regex \rangle$, replaces them according to the $\langle replacement \rangle$ as for `\regex_replace_once:nnN`, and leaves the result in the input stream, after the $\langle true code \rangle$. Otherwise, leaves $\langle false code \rangle$ followed by the $\langle tokens \rangle$ that were originally in the input stream, with no modifications. See `l3regex` for documentation of the syntax of regular expressions and of the $\langle replacement \rangle$: for instance `\0` in the $\langle replacement \rangle$ is replaced by the tokens that were matched in the input stream. The $\langle regular expression \rangle$ is implicitly anchored at the start. In contrast to `\regex_replace_once:nnN`, no error arises if the $\langle replacement \rangle$ leads to an unbalanced token list: the tokens are inserted into the input stream without issue.

TeXhackers note: Implicit character tokens are correctly considered by `\peek_regex_replace_once:nnTF` as control sequences, while functions that inspect individual tokens (for instance `\peek_charcode:NTF`) only take into account their meaning.

23.7 Description of all possible tokens

Let us end by reviewing every case that a given token can fall into. This section is quite technical and some details are only meant for completeness. We distinguish the meaning of the token, which controls the expansion of the token and its effect on TeX's state, and its shape, which is used when comparing token lists such as for delimited arguments. Two tokens of the same shape must have the same meaning, but the converse does not hold.

A token has one of the following shapes.

- A control sequence, characterized by the sequence of characters that constitute its name: for instance, `\use:n` is a five-letter control sequence.

- An active character token, characterized by its character code (between 0 and 1114111 for LuaTeX and XeTeX and less for other engines) and category code 13.
- A character token, characterized by its character code and category code (one of 1, 2, 3, 4, 6, 7, 8, 10, 11 or 12 whose meaning is described below).

There are also a few internal tokens. The following list may be incomplete in some engines.

- Expanding `\the\font` results in a token that looks identical to the command that was used to select the current font (such as `\tenrm`) but it differs from it in shape.
- A “frozen” `\relax`, which differs from the primitive in shape (but has the same meaning), is inserted when the closing `\fi` of a conditional is encountered before the conditional is evaluated.
- Expanding `\noexpand⟨token⟩` (when the `⟨token⟩` is expandable) results in an internal token, displayed (temporarily) as `\notexpanded:⟨token⟩`, whose shape coincides with the `⟨token⟩` and whose meaning differs from `\relax`.
- An `\outer endtemplate:` can be encountered when peeking ahead at the next token; this expands to another internal token, `end of alignment template`.
- Tricky programming might access a frozen `\endwrite`.
- Some frozen tokens can only be accessed in interactive sessions: `\cr`, `\right`, `\endgroup`, `\fi`, `\inaccessible`.
- In LuaTeX, there is also the strange case of “bytes” `~~~~~1100xy` where x, y are any two lowercase hexadecimal digits, so that the hexadecimal number ranges from `\text{110000}=1114112$` to `~$1100ff = 1114367`. These are used to output individual bytes to files, rather than UTF-8. For the purposes of token comparisons they behave like non-expandable primitive control sequences (*not characters*) whose `\meaning` is the `␣character␣` followed by the given byte. If this byte is in the range 80–ff this gives an “invalid utf-8 sequence” error: applying `\token_to_str:N` or `\token_to_meaning:N` to these tokens is unsafe. Unfortunately, they don’t seem to be detectable safely by any means except perhaps Lua code.

The meaning of a (non-active) character token is fixed by its category code (and character code) and cannot be changed. We call these tokens *explicit* character tokens. Category codes that a character token can have are listed below by giving a sample output of the TeX primitive `\meaning`, together with their L^AT_EX3 names and most common example:

- 1 begin-group character (`group_begin`, often `{`),
- 2 end-group character (`group_end`, often `}`),
- 3 math shift character (`math_toggle`, often `$`),
- 4 alignment tab character (`alignment`, often `&`),
- 6 macro parameter character (`parameter`, often `#`),
- 7 superscript character (`math_superscript`, often `^`),

- 8 subscript character (`math_subscript`, often `_`),
- 10 blank space (`space`, often character code 32),
- 11 the letter (`letter`, such as `A`),
- 12 the character (`other`, such as `0`).

Category code 13 (`active`) is discussed below. Input characters can also have several other category codes which do not lead to character tokens for later processing: 0 (`escape`), 5 (`end_line`), 9 (`ignore`), 14 (`comment`), and 15 (`invalid`).

The meaning of a control sequence or active character can be identical to that of any character token listed above (with any character code), and we call such tokens *implicit* character tokens. The meaning is otherwise in the following list:

- a macro, used in \LaTeX 3 for most functions and some variables (`tl`, `fp`, `seq`, ...),
- a primitive such as `\def` or `\topmark`, used in \LaTeX 3 for some functions,
- a register such as `\count123`, used in \LaTeX 3 for the implementation of some variables (`int`, `dim`, ...),
- a constant integer such as `\char"56` or `\mathchar"121`,
- a font selection command,
- undefined.

Macros can be `\protected` or not, `\long` or not (the opposite of what \LaTeX 3 calls `\nopar`), and `\outer` or not (unused in \LaTeX 3). Their `\meaning` takes the form

$\langle prefix \rangle \text{ macro:} \langle argument \rangle \rightarrow \langle replacement \rangle$

where $\langle prefix \rangle$ is among `\protected`, `\long`, `\outer`, $\langle argument \rangle$ describes parameters that the macro expects, such as `#1#2#3`, and $\langle replacement \rangle$ describes how the parameters are manipulated, such as `\int_eval:n{\#2+\#1*\#3}`.

Now is perhaps a good time to mention some subtleties relating to tokens with category code 10 (space). Any input character with this category code (normally, space and tab characters) becomes a normal space, with character code 32 and category code 10.

When a macro takes an undelimited argument, explicit space characters (with character code 32 and category code 10) are ignored. If the following token is an explicit character token with category code 1 (begin-group) and an arbitrary character code, then \TeX scans ahead to obtain an equal number of explicit character tokens with category code 1 (begin-group) and 2 (end-group), and the resulting list of tokens (with outer braces removed) becomes the argument. Otherwise, a single token is taken as the argument for the macro: we call such single tokens “N-type”, as they are suitable to be used as an argument for a function with the signature `:N`.

When a macro takes a delimited argument \TeX scans ahead until finding the delimiter (outside any pairs of begin-group/end-group explicit characters), and the resulting list of tokens (with outer braces removed) becomes the argument. Note that explicit space characters at the start of the argument are *not* ignored in this case (and they prevent brace-stripping).

Chapter 24

The l3prop package

Property lists

L^AT_EX3 implements a “property list” data type, which contain an unordered list of entries each of which consists of a $\langle key \rangle$ and an associated $\langle value \rangle$. The $\langle key \rangle$ and $\langle value \rangle$ may both be any $\langle balanced\ text \rangle$, the $\langle key \rangle$ is processed using `\tl_to_str:n`, meaning that category codes are ignored. It is possible to map functions to property lists such that the function is applied to every key–value pair within the list.

Each entry in a property list must have a unique $\langle key \rangle$: if an entry is added to a property list which already contains the $\langle key \rangle$ then the new entry overwrites the existing one. The $\langle keys \rangle$ are compared on a string basis, using the same method as `\str_if_eq:nn`.

Property lists are intended for storing key-based information for use within code. This is in contrast to key–value lists, which are a form of *input* parsed by the `l3keys` module.

24.1 Creating and initialising property lists

<code>\prop_new:N</code>	<code>\prop_new:N</code>
<code>\prop_new:c</code>	

$\langle property\ list \rangle$

Creates a new $\langle property\ list \rangle$ or raises an error if the name is already taken. The declaration is global. The $\langle property\ list \rangle$ initially contains no entries.

<code>\prop_clear:N</code>	<code>\prop_clear:N</code>
<code>\prop_clear:c</code>	
<code>\prop_gclear:N</code>	
<code>\prop_gclear:c</code>	

$\langle property\ list \rangle$

Clears all entries from the $\langle property\ list \rangle$.

<code>\prop_clear_new:N</code>	<code>\prop_clear_new:N</code>
<code>\prop_clear_new:c</code>	
<code>\prop_gclear_new:N</code>	
<code>\prop_gclear_new:c</code>	

$\langle property\ list \rangle$

Ensures that the $\langle property\ list \rangle$ exists globally by applying `\prop_new:N` if necessary, then applies `\prop_(g)clear:N` to leave the list empty.

```

\prop_set_eq:Nn
\prop_set_eq:(cN|Nc|cc)
\prop_gset_eq:Nn
\prop_gset_eq:(cN|Nc|cc)

```

```

\prop_set_eq:Nn <property list1> <property list2>

```

Sets the content of *<property list₁>* equal to that of *<property list₂>*.

```

\prop_set_from_keyval:Nn
\prop_set_from_keyval:cn
\prop_gset_from_keyval:Nn
\prop_gset_from_keyval:cn

```

New: 2017-11-28
Updated: 2021-11-07

```

\prop_set_from_keyval:Nn <prop var>
{
  <key1> = <value1> ,
  <key2> = <value2> , ...
}

```

Sets *<prop var>* to contain key–value pairs given in the second argument. If duplicate keys appear only the last of the values is kept.

Spaces are trimmed around every *<key>* and every *<value>*, and if the result of trimming spaces consists of a single brace group then a set of outer braces is removed. This enables both the *<key>* and the *<value>* to contain spaces, commas or equal signs. The *<key>* is then processed by `\tl_to_str:n`. This function correctly detects the = and , signs provided they have the standard category code 12 or they are active.

Notice that in contrast to most keyval lists (e.g. those in `l3keys`), each key here *must* be followed with an = sign.

```

\prop_const_from_keyval:Nn
\prop_const_from_keyval:cn

```

New: 2017-11-28
Updated: 2021-11-07

```

\prop_const_from_keyval:Nn <prop var>
{
  <key1> = <value1> ,
  <key2> = <value2> , ...
}

```

Creates a new constant *<prop var>* or raises an error if the name is already taken. The *<prop var>* is set globally to contain key–value pairs given in the second argument, processed in the way described for `\prop_set_from_keyval:Nn`. If duplicate keys appear only the last of the values is kept. This function correctly detects the = and , signs provided they have the standard category code 12 or they are active.

Notice that in contrast to most keyval lists (e.g. those in `l3keys`), each key here *must* be followed with an = sign.

24.2 Adding and updating property list entries

```

\prop_put:Nnn
\prop_put:(NnV|Nno|Nnx|NVn|NVV|NVx|Nvx|Non|Noo|Nxx|cnn|cnV|cno|cnx|cVn|cVV|cVx|cvx|con|coo|cxx)
\prop_gput:Nnn
\prop_gput:(NnV|Nno|Nnx|NVn|NVV|NVx|Nvx|Non|Noo|Nxx|cnn|cnV|cno|cnx|cVn|cVV|cVx|cvx|con|coo|cxx)

```

```

\prop_put:Nnn
  <property list>
  {<key>}
  {<value>}

```

Updated: 2012-07-09

Adds an entry to the *<property list>* which may be accessed using the *<key>* and which has *<value>*. If the *<key>* is already present in the *<property list>*, the existing entry is overwritten by the new *<value>*. Both the *<key>* and *<value>* may contain any *<balanced text>*. The *<key>* is stored after processing with `\tl_to_str:n`, meaning that category codes are ignored.

```

\prop_put_if_new:Nnn
\prop_put_if_new:cnn
\prop_gput_if_new:Nnn
\prop_gput_if_new:cnn

```

```
\prop_put_if_new:Nnn <property list> {{<key>}} {{<value>}}
```

If the $\langle key \rangle$ is present in the $\langle property list \rangle$ then no action is taken. Otherwise, a new entry is added as described for `\prop_put:Nnn`.

```

\prop_concat:NNN
\prop_concat:ccc
\prop_gconcat:NNN
\prop_gconcat:ccc

```

New: 2021-05-16

```
\prop_concat:NNN <prop var1> <prop var2> <prop var3>
```

Combines the key–value pairs of $\langle prop var2 \rangle$ and $\langle prop var3 \rangle$, and saves the result in $\langle prop var1 \rangle$. If a key appears in both $\langle prop var2 \rangle$ and $\langle prop var3 \rangle$ then the last value, namely the value in $\langle prop var3 \rangle$ is kept.

```

\prop_put_from_keyval:Nn
\prop_put_from_keyval:cn
\prop_gput_from_keyval:Nn
\prop_gput_from_keyval:cn

```

New: 2021-05-16

Updated: 2021-11-07

```

\prop_put_from_keyval:Nn <prop var>
{
  <key1> = <value1> ,
  <key2> = <value2> , ...
}

```

Updates the $\langle prop var \rangle$ by adding entries for each key–value pair given in the second argument. The addition is done through `\prop_put:Nnn`, hence if the $\langle prop var \rangle$ already contains some of the keys, the corresponding values are discarded and replaced by those given in the key–value list. If duplicate keys appear in the key–value list then only the last of the values is kept.

The function is equivalent to storing the key–value pairs in a temporary property variable using `\prop_set_from_keyval:Nn`, then combining $\langle prop var \rangle$ with the temporary variable using `\prop_concat:NNN`. In particular, the $\langle keys \rangle$ and $\langle values \rangle$ are space-trimmed and unbraced as described in `\prop_set_from_keyval:Nn`. This function correctly detects the = and , signs provided they have the standard category code 12 or they are active.

24.3 Recovering values from property lists

```

\prop_get:NnN
\prop_get:(NVN|NvN|NoN|cnN|cVN|cvN|coN)

```

Updated: 2011-08-28

```
\prop_get:NnN <property list> {{<key>}} <tl var>
```

Recovers the $\langle value \rangle$ stored with $\langle key \rangle$ from the $\langle property list \rangle$, and places this in the $\langle token list variable \rangle$. If the $\langle key \rangle$ is not found in the $\langle property list \rangle$ then the $\langle token list variable \rangle$ is set to the special marker `\q_no_value`. The $\langle token list variable \rangle$ is set within the current T_EX group. See also `\prop_get:NnNTF`.

```

\prop_pop:NnN
\prop_pop:(NoN|cnN|coN)

```

Updated: 2011-08-18

```
\prop_pop:NnN <property list> {{<key>}} <tl var>
```

Recovers the $\langle value \rangle$ stored with $\langle key \rangle$ from the $\langle property list \rangle$, and places this in the $\langle token list variable \rangle$. If the $\langle key \rangle$ is not found in the $\langle property list \rangle$ then the $\langle token list variable \rangle$ is set to the special marker `\q_no_value`. The $\langle key \rangle$ and $\langle value \rangle$ are then deleted from the property list. Both assignments are local. See also `\prop_pop:NnNTF`.

<code>\prop_gpop:NnN</code>
<code>\prop_gpop:(NoN cnN coN)</code>
Updated: 2011-08-18

`\prop_gpop:NnN` $\langle property list \rangle$ $\{\langle key \rangle\}$ $\langle tl var \rangle$

Recovers the $\langle value \rangle$ stored with $\langle key \rangle$ from the $\langle property list \rangle$, and places this in the $\langle token list variable \rangle$. If the $\langle key \rangle$ is not found in the $\langle property list \rangle$ then the $\langle token list variable \rangle$ is set to the special marker `\q_no_value`. The $\langle key \rangle$ and $\langle value \rangle$ are then deleted from the property list. The $\langle property list \rangle$ is modified globally, while the assignment of the $\langle token list variable \rangle$ is local. See also `\prop_gpop:NnNTF`.

<code>\prop_item:Nn</code> *
<code>\prop_item:cn</code> *
New: 2014-07-17

`\prop_item:Nn` $\langle property list \rangle$ $\{\langle key \rangle\}$

Expands to the $\langle value \rangle$ corresponding to the $\langle key \rangle$ in the $\langle property list \rangle$. If the $\langle key \rangle$ is missing, this has an empty expansion.

T_EXhackers note: This function is slower than the non-expandable analogue `\prop_get:NnN`. The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the $\langle value \rangle$ does not expand further when appearing in an **x**-type or **e**-type argument expansion.

<code>\prop_count:N</code> *
<code>\prop_count:c</code> *

`\prop_count:N` $\langle property list \rangle$

Leaves the number of key–value pairs in the $\langle property list \rangle$ in the input stream as an $\langle integer denotation \rangle$.

`\prop_to_keyval:N` *

`\prop_to_keyval:N` $\langle property list \rangle$

Expands to the $\langle property list \rangle$ in a key–value notation. Keep in mind that a $\langle property list \rangle$ is *unordered*, while key–value interfaces don’t necessarily are, so this can’t be used for arbitrary interfaces.

T_EXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the key–value list does not expand further when appearing in an **x**-type or **e**-type argument expansion. It also needs exactly two steps of expansion.

24.4 Modifying property lists

<code>\prop_remove:Nn</code>
<code>\prop_remove:(NV cn cV)</code>
<code>\prop_gremove:Nn</code>
<code>\prop_gremove:(NV cn cV)</code>
New: 2012-05-12

`\prop_remove:Nn` $\langle property list \rangle$ $\{\langle key \rangle\}$

Removes the entry listed under $\langle key \rangle$ from the $\langle property list \rangle$. If the $\langle key \rangle$ is not found in the $\langle property list \rangle$ no change occurs, *i.e* there is no need to test for the existence of a key before deleting it.

24.5 Property list conditionals

<code>\prop_if_exist_p:N</code> *
<code>\prop_if_exist_p:c</code> *
<code>\prop_if_exist:NTF</code> *
<code>\prop_if_exist:cTF</code> *
New: 2012-03-03

`\prop_if_exist_p:N` $\langle property list \rangle$

`\prop_if_exist:NTF` $\langle property list \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Tests whether the $\langle property list \rangle$ is currently defined. This does not check that the $\langle property list \rangle$ really is a property list variable.

<code>\prop_if_empty_p:N</code>	★	<code>\prop_if_empty_p:N</code> $\langle property\ list \rangle$
<code>\prop_if_empty_p:c</code>	★	<code>\prop_if_empty:N</code> TF $\langle property\ list \rangle$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$
<code>\prop_if_empty:N</code> TF	★	Tests if the $\langle property\ list \rangle$ is empty (containing no entries).
<code>\prop_if_empty:c</code> TF	★	

<code>\prop_if_in_p:Nn</code>	★	<code>\prop_if_in:Nn</code> TF $\langle property\ list \rangle$ $\{\langle key \rangle\}$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$
<code>\prop_if_in_p:(NV No cn cV co)</code>	★	
<code>\prop_if_in:Nn</code> TF	★	
<code>\prop_if_in:(NV No cn cV co)</code> TF	★	

Updated: 2011-09-15

Tests if the $\langle key \rangle$ is present in the $\langle property\ list \rangle$, making the comparison using the method described by `\str_if_eq:n`TF.

TeXhackers note: This function iterates through every key–value pair in the $\langle property\ list \rangle$ and is therefore slower than using the non-expandable `\prop_get:Nn`TF.

24.6 Recovering values from property lists with branching

The functions in this section combine tests for the presence of a key in a property list with recovery of the associated valued. This makes them useful for cases where different cases follow dependent on the presence or absence of a key in a property list. They offer increased readability and performance over separate testing and recovery phases.

<code>\prop_get:Nn</code> TF	<code>\prop_get:Nn</code> TF $\langle property\ list \rangle$ $\{\langle key \rangle\}$ $\langle token\ list\ variable \rangle$
<code>\prop_get:(NVN NvN NoN cnN cVN cvN coN)</code> TF	$\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$

Updated: 2012-05-19

If the $\langle key \rangle$ is not present in the $\langle property\ list \rangle$, leaves the $\langle false\ code \rangle$ in the input stream. The value of the $\langle token\ list\ variable \rangle$ is not defined in this case and should not be relied upon. If the $\langle key \rangle$ is present in the $\langle property\ list \rangle$, stores the corresponding $\langle value \rangle$ in the $\langle token\ list\ variable \rangle$ without removing it from the $\langle property\ list \rangle$, then leaves the $\langle true\ code \rangle$ in the input stream. The $\langle token\ list\ variable \rangle$ is assigned locally.

<code>\prop_pop:Nn</code> TF	<code>\prop_pop:Nn</code> TF $\langle property\ list \rangle$ $\{\langle key \rangle\}$ $\langle token\ list\ variable \rangle$ $\{\langle true\ code \rangle\}$
<code>\prop_pop:cn</code> TF	$\{\langle false\ code \rangle\}$

New: 2011-08-18
Updated: 2012-05-19

If the $\langle key \rangle$ is not present in the $\langle property\ list \rangle$, leaves the $\langle false\ code \rangle$ in the input stream. The value of the $\langle token\ list\ variable \rangle$ is not defined in this case and should not be relied upon. If the $\langle key \rangle$ is present in the $\langle property\ list \rangle$, pops the corresponding $\langle value \rangle$ in the $\langle token\ list\ variable \rangle$, *i.e.* removes the item from the $\langle property\ list \rangle$. Both the $\langle property\ list \rangle$ and the $\langle token\ list\ variable \rangle$ are assigned locally.

`\prop_gpop:NnNTF`
`\prop_gpop:cnNTF`

New: 2011-08-18
Updated: 2012-05-19

`\prop_gpop:NnNTF` $\langle\textit{property list}\rangle$ $\{\langle\textit{key}\rangle\}$ $\langle\textit{token list variable}\rangle$ $\{\langle\textit{true code}\rangle\}$
 $\{\langle\textit{false code}\rangle\}$

If the $\langle\textit{key}\rangle$ is not present in the $\langle\textit{property list}\rangle$, leaves the $\langle\textit{false code}\rangle$ in the input stream. The value of the $\langle\textit{token list variable}\rangle$ is not defined in this case and should not be relied upon. If the $\langle\textit{key}\rangle$ is present in the $\langle\textit{property list}\rangle$, pops the corresponding $\langle\textit{value}\rangle$ in the $\langle\textit{token list variable}\rangle$, *i.e.* removes the item from the $\langle\textit{property list}\rangle$. The $\langle\textit{property list}\rangle$ is modified globally, while the $\langle\textit{token list variable}\rangle$ is assigned locally.

24.7 Mapping over property lists

All mappings are done at the current group level, *i.e.* any local assignments made by the $\langle\textit{function}\rangle$ or $\langle\textit{code}\rangle$ discussed below remain in effect after the loop.

`\prop_map_function:NN` ☆
`\prop_map_function:cN` ☆

Updated: 2013-01-08

`\prop_map_function:NN` $\langle\textit{property list}\rangle$ $\langle\textit{function}\rangle$

Applies $\langle\textit{function}\rangle$ to every $\langle\textit{entry}\rangle$ stored in the $\langle\textit{property list}\rangle$. The $\langle\textit{function}\rangle$ receives two arguments for each iteration: the $\langle\textit{key}\rangle$ and associated $\langle\textit{value}\rangle$. The order in which $\langle\textit{entries}\rangle$ are returned is not defined and should not be relied upon. To pass further arguments to the $\langle\textit{function}\rangle$, see `\prop_map_tokens:Nn`.

`\prop_map_inline:Nn`
`\prop_map_inline:cn`

Updated: 2013-01-08

`\prop_map_inline:Nn` $\langle\textit{property list}\rangle$ $\{\langle\textit{inline function}\rangle\}$

Applies $\langle\textit{inline function}\rangle$ to every $\langle\textit{entry}\rangle$ stored within the $\langle\textit{property list}\rangle$. The $\langle\textit{inline function}\rangle$ should consist of code which receives the $\langle\textit{key}\rangle$ as #1 and the $\langle\textit{value}\rangle$ as #2. The order in which $\langle\textit{entries}\rangle$ are returned is not defined and should not be relied upon.

`\prop_map_tokens:Nn` ☆
`\prop_map_tokens:cn` ☆

`\prop_map_tokens:Nn` $\langle\textit{property list}\rangle$ $\{\langle\textit{code}\rangle\}$

Analogue of `\prop_map_function:NN` which maps several tokens instead of a single function. The $\langle\textit{code}\rangle$ receives each key–value pair in the $\langle\textit{property list}\rangle$ as two trailing brace groups. For instance,

`\prop_map_tokens:Nn \l_my_prop { \str_if_eq:nnT { mykey } }`

expands to the value corresponding to `mykey`: for each pair in `\l_my_prop` the function `\str_if_eq:nnT` receives `mykey`, the $\langle\textit{key}\rangle$ and the $\langle\textit{value}\rangle$ as its three arguments. For that specific task, `\prop_item:Nn` is faster.

\prop_map_break: ☆

Updated: 2012-06-29

\prop_map_break:

Used to terminate a `\prop_map...` function before all entries in the *⟨property list⟩* have been processed. This normally takes place within a conditional statement, for example

```
\prop_map_inline:Nn \l_my_prop
{
  \str_if_eq:nnTF { #1 } { bingo }
  { \prop_map_break: }
  {
    % Do something useful
  }
}
```

Use outside of a `\prop_map...` scenario leads to low level T_EX errors.

T_EXhackers note: When the mapping is broken, additional tokens may be inserted before further items are taken from the input stream. This depends on the design of the mapping function.

\prop_map_break:n ☆

Updated: 2012-06-29

\prop_map_break:n {⟨code⟩}

Used to terminate a `\prop_map...` function before all entries in the *⟨property list⟩* have been processed, inserting the *⟨code⟩* after the mapping has ended. This normally takes place within a conditional statement, for example

```
\prop_map_inline:Nn \l_my_prop
{
  \str_if_eq:nnTF { #1 } { bingo }
  { \prop_map_break:n { <code> } }
  {
    % Do something useful
  }
}
```

Use outside of a `\prop_map...` scenario leads to low level T_EX errors.

T_EXhackers note: When the mapping is broken, additional tokens may be inserted before the *⟨code⟩* is inserted into the input stream. This depends on the design of the mapping function.

24.8 Viewing property lists

\prop_show:N

\prop_show:c

Updated: 2021-04-29

\prop_show:N *⟨property list⟩*

Displays the entries in the *⟨property list⟩* in the terminal.

<code>\prop_log:N</code>	<code>\prop_log:N</code> \langle <i>property list</i> \rangle
<code>\prop_log:c</code>	Writes the entries in the \langle <i>property list</i> \rangle in the log file.
<small>New: 2014-08-12 Updated: 2021-04-29</small>	

24.9 Scratch property lists

<code>\l_tmpa_prop</code>	Scratch property lists for local assignment. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
<code>\l_tmpb_prop</code>	
<small>New: 2012-06-23</small>	

<code>\g_tmpa_prop</code>	Scratch property lists for global assignment. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
<code>\g_tmpb_prop</code>	
<small>New: 2012-06-23</small>	

24.10 Constants

<code>\c_empty_prop</code>	A permanently-empty property list used for internal comparisons.
----------------------------	--

Chapter 25

The l3skip package

Dimensions and skips

L^AT_EX3 provides two general length variables: `dim` and `skip`. Lengths stored as `dim` variables have a fixed length, whereas `skip` lengths have a rubber (stretch/shrink) component. In addition, the `muskip` type is available for use in math mode: this is a special form of `skip` where the lengths involved are determined by the current math font (in μ). There are common features in the creation and setting of length variables, but for clarity the functions are grouped by variable type.

25.1 Creating and initialising dim variables

<code>\dim_new:N</code>
<code>\dim_new:c</code>

<code>\dim_new:N</code>	<code>\dim_new:N</code> $\langle dimension \rangle$
-------------------------	---

Creates a new $\langle dimension \rangle$ or raises an error if the name is already taken. The declaration is global. The $\langle dimension \rangle$ is initially equal to 0pt.

<code>\dim_const:Nn</code>
<code>\dim_const:cn</code>

<code>\dim_const:Nn</code>	<code>\dim_const:Nn</code> $\langle dimension \rangle$ { $\langle dimension expression \rangle$ }
----------------------------	---

Creates a new constant $\langle dimension \rangle$ or raises an error if the name is already taken. The value of the $\langle dimension \rangle$ is set globally to the $\langle dimension expression \rangle$.

New: 2012-03-05

<code>\dim_zero:N</code>
<code>\dim_zero:c</code>
<code>\dim_gzero:N</code>
<code>\dim_gzero:c</code>

<code>\dim_zero:N</code>	<code>\dim_zero:N</code> $\langle dimension \rangle$
--------------------------	--

Sets $\langle dimension \rangle$ to 0pt.

<code>\dim_zero_new:N</code>
<code>\dim_zero_new:c</code>
<code>\dim_gzero_new:N</code>
<code>\dim_gzero_new:c</code>

<code>\dim_zero_new:N</code>	<code>\dim_zero_new:N</code> $\langle dimension \rangle$
------------------------------	--

Ensures that the $\langle dimension \rangle$ exists globally by applying `\dim_new:N` if necessary, then applies `\dim_(g)zero:N` to leave the $\langle dimension \rangle$ set to zero.

New: 2012-01-07

<code>\dim_if_exist_p:N</code> *	<code>\dim_if_exist_p:N</code> $\langle dimension \rangle$
<code>\dim_if_exist_p:c</code> *	<code>\dim_if_exist:NTF</code> $\langle dimension \rangle$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$
<code>\dim_if_exist:N\underline{TF}</code> *	Tests whether the $\langle dimension \rangle$ is currently defined. This does not check that the $\langle dimension \rangle$ really is a dimension variable.
<code>\dim_if_exist:c\underline{TF}</code> *	

New: 2012-03-03

25.2 Setting dim variables

<code>\dim_add:Nn</code>	<code>\dim_add:Nn</code> $\langle dimension \rangle$ $\{\langle dimension\ expression \rangle\}$
<code>\dim_add:cn</code>	Adds the result of the $\langle dimension\ expression \rangle$ to the current content of the $\langle dimension \rangle$.
<code>\dim_gadd:Nn</code>	
<code>\dim_gadd:cn</code>	

Updated: 2011-10-22

<code>\dim_set:Nn</code>	<code>\dim_set:Nn</code> $\langle dimension \rangle$ $\{\langle dimension\ expression \rangle\}$
<code>\dim_set:cn</code>	Sets $\langle dimension \rangle$ to the value of $\langle dimension\ expression \rangle$, which must evaluate to a length with units.
<code>\dim_gset:Nn</code>	
<code>\dim_gset:cn</code>	

Updated: 2011-10-22

<code>\dim_set_eq:NN</code>	<code>\dim_set_eq:NN</code> $\langle dimension_1 \rangle$ $\langle dimension_2 \rangle$
<code>\dim_set_eq:(cN Nc cc)</code>	Sets the content of $\langle dimension_1 \rangle$ equal to that of $\langle dimension_2 \rangle$.
<code>\dim_gset_eq:NN</code>	
<code>\dim_gset_eq:(cN Nc cc)</code>	

<code>\dim_sub:Nn</code>	<code>\dim_sub:Nn</code> $\langle dimension \rangle$ $\{\langle dimension\ expression \rangle\}$
<code>\dim_sub:cn</code>	Subtracts the result of the $\langle dimension\ expression \rangle$ from the current content of the $\langle dimension \rangle$.
<code>\dim_gsub:Nn</code>	
<code>\dim_gsub:cn</code>	

Updated: 2011-10-22

25.3 Utilities for dimension calculations

<code>\dim_abs:n</code> *	<code>\dim_abs:n</code> $\{\langle dimexpr \rangle\}$
---------------------------	---

Updated: 2012-09-26

Converts the $\langle dimexpr \rangle$ to its absolute value, leaving the result in the input stream as a $\langle dimension\ denotation \rangle$.

<code>\dim_max:nn</code> *	<code>\dim_max:nn</code> $\{\langle dimexpr_1 \rangle\}$ $\{\langle dimexpr_2 \rangle\}$
<code>\dim_min:nn</code> *	<code>\dim_min:nn</code> $\{\langle dimexpr_1 \rangle\}$ $\{\langle dimexpr_2 \rangle\}$

New: 2012-09-09

Updated: 2012-09-26

Evaluates the two $\langle dimension\ expressions \rangle$ and leaves either the maximum or minimum value in the input stream as appropriate, as a $\langle dimension\ denotation \rangle$.

`\dim_ratio:nn` ☆

Updated: 2011-10-22

`\dim_ratio:nn` { $\langle dimexpr_1 \rangle$ } { $\langle dimexpr_2 \rangle$ }

Parses the two $\langle dimension expressions \rangle$ and converts the ratio of the two to a form suitable for use inside a $\langle dimension expression \rangle$. This ratio is then left in the input stream, allowing syntax such as

```
\dim_set:Nn \l_my_dim
{ 10 pt * \dim_ratio:nn { 5 pt } { 10 pt } }
```

The output of `\dim_ratio:nn` on full expansion is a ratio expression between two integers, with all distances converted to scaled points. Thus

```
\tl_set:Nx \l_my_tl { \dim_ratio:nn { 5 pt } { 10 pt } }
\tl_show:N \l_my_tl
```

displays 327680/655360 on the terminal.

25.4 Dimension expression conditionals

`\dim_compare_p:nNn` ★

`\dim_compare:nNnTF` ★

`\dim_compare_p:nNn` { $\langle dimexpr_1 \rangle$ } $\langle relation \rangle$ { $\langle dimexpr_2 \rangle$ }

`\dim_compare:nNnTF`

{ $\langle dimexpr_1 \rangle$ } $\langle relation \rangle$ { $\langle dimexpr_2 \rangle$ }
{ $\langle true code \rangle$ } { $\langle false code \rangle$ }

This function first evaluates each of the $\langle dimension expressions \rangle$ as described for `\dim_eval:n`. The two results are then compared using the $\langle relation \rangle$:

Equal	=
Greater than	>
Less than	<

This function is less flexible than `\dim_compare:nTF` but around 5 times faster.

```

\dim_compare_p:n * \dim_compare_p:n
\dim_compare:nTF * {
    <dimexpr1> <relation1>
    ...
    <dimexprN> <relationN>
    <dimexprN+1>
}
\dim_compare:nTF
{
    <dimexpr1> <relation1>
    ...
    <dimexprN> <relationN>
    <dimexprN+1>
}
{{true code}} {{false code}}

```

Updated: 2013-01-13

This function evaluates the *<dimension expressions>* as described for `\dim_eval:n` and compares consecutive result using the corresponding *<relation>*, namely it compares *<dimexpr₁>* and *<dimexpr₂>* using the *<relation₁>*, then *<dimexpr₂>* and *<dimexpr₃>* using the *<relation₂>*, until finally comparing *<dimexpr_N>* and *<dimexpr_{N+1}>* using the *<relation_N>*. The test yields `true` if all comparisons are `true`. Each *<dimension expression>* is evaluated only once, and the evaluation is lazy, in the sense that if one comparison is `false`, then no other *<dimension expression>* is evaluated and no other comparison is performed. The *<relations>* can be any of the following:

Equal	= or ==
Greater than or equal to	>=
Greater than	>
Less than or equal to	<=
Less than	<
Not equal	!=

This function is more flexible than `\dim_compare:nNnTF` but around 5 times slower.

<code>\dim_case:nn</code> ☆	<code>\dim_case:nnTF {⟨test dimension expression⟩}</code>
<code>\dim_case:nnTF</code> ☆	<code>{</code>
New: 2013-07-24	<code>{⟨dimexpr case₁⟩} {⟨code case₁⟩}</code>
	<code>{⟨dimexpr case₂⟩} {⟨code case₂⟩}</code>
	<code>...</code>
	<code>{⟨dimexpr case_n⟩} {⟨code case_n⟩}</code>
	<code>}</code>
	<code>{⟨true code⟩}</code>
	<code>{⟨false code⟩}</code>

This function evaluates the *⟨test dimension expression⟩* and compares this in turn to each of the *⟨dimension expression cases⟩*. If the two are equal then the associated *⟨code⟩* is left in the input stream and other cases are discarded. If any of the cases are matched, the *⟨true code⟩* is also inserted into the input stream (after the code for the appropriate case), while if none match then the *⟨false code⟩* is inserted. The function `\dim_case:nn`, which does nothing if there is no match, is also available. For example

```

\dim_set:Nn \l_tmpa_dim { 5 pt }
\dim_case:nnF
{ 2 \l_tmpa_dim }
{
  { 5 pt }      { Small }
  { 4 pt + 6 pt } { Medium }
  { - 10 pt }   { Negative }
}
{ No idea! }
```

leaves “Medium” in the input stream.

25.5 Dimension expression loops

<code>\dim_do_until:nNnn</code> ☆	<code>\dim_do_until:nNnn {⟨dimexpr₁⟩} ⟨relation⟩ {⟨dimexpr₂⟩} {⟨code⟩}</code>
-----------------------------------	---

Places the *⟨code⟩* in the input stream for T_EX to process, and then evaluates the relationship between the two *⟨dimension expressions⟩* as described for `\dim_compare:nNnTF`. If the test is **false** then the *⟨code⟩* is inserted into the input stream again and a loop occurs until the *⟨relation⟩* is **true**.

<code>\dim_do_while:nNnn</code> ☆	<code>\dim_do_while:nNnn {⟨dimexpr₁⟩} ⟨relation⟩ {⟨dimexpr₂⟩} {⟨code⟩}</code>
-----------------------------------	---

Places the *⟨code⟩* in the input stream for T_EX to process, and then evaluates the relationship between the two *⟨dimension expressions⟩* as described for `\dim_compare:nNnTF`. If the test is **true** then the *⟨code⟩* is inserted into the input stream again and a loop occurs until the *⟨relation⟩* is **false**.

<code>\dim_until_do:nNnn</code> ☆	<code>\dim_until_do:nNnn {⟨dimexpr₁⟩} ⟨relation⟩ {⟨dimexpr₂⟩} {⟨code⟩}</code>
-----------------------------------	---

Evaluates the relationship between the two *⟨dimension expressions⟩* as described for `\dim_compare:nNnTF`, and then places the *⟨code⟩* in the input stream if the *⟨relation⟩* is **false**. After the *⟨code⟩* has been processed by T_EX the test is repeated, and a loop occurs until the test is **true**.

<hr/> <code>\dim_while_do:nNnn</code> ☆ <hr/>	<code>\dim_while_do:nNnn {<dimexpr₁>} <relation> {<dimexpr₂>} {<code>}</code>
	Evaluates the relationship between the two <i><dimension expressions></i> as described for <code>\dim_compare:nNnTF</code> , and then places the <i><code></i> in the input stream if the <i><relation></i> is true . After the <i><code></i> has been processed by T _E X the test is repeated, and a loop occurs until the test is false .
<hr/> <code>\dim_do_until:nn</code> ☆ <hr/> <div>Updated: 2013-01-13</div>	<code>\dim_do_until:nn {<dimension relation>} {<code>}</code>
	Places the <i><code></i> in the input stream for T _E X to process, and then evaluates the <i><dimension relation></i> as described for <code>\dim_compare:nTF</code> . If the test is false then the <i><code></i> is inserted into the input stream again and a loop occurs until the <i><relation></i> is true .
<hr/> <code>\dim_do_while:nn</code> ☆ <hr/> <div>Updated: 2013-01-13</div>	<code>\dim_do_while:nn {<dimension relation>} {<code>}</code>
	Places the <i><code></i> in the input stream for T _E X to process, and then evaluates the <i><dimension relation></i> as described for <code>\dim_compare:nTF</code> . If the test is true then the <i><code></i> is inserted into the input stream again and a loop occurs until the <i><relation></i> is false .
<hr/> <code>\dim_until_do:nn</code> ☆ <hr/> <div>Updated: 2013-01-13</div>	<code>\dim_until_do:nn {<dimension relation>} {<code>}</code>
	Evaluates the <i><dimension relation></i> as described for <code>\dim_compare:nTF</code> , and then places the <i><code></i> in the input stream if the <i><relation></i> is false . After the <i><code></i> has been processed by T _E X the test is repeated, and a loop occurs until the test is true .
<hr/> <code>\dim_while_do:nn</code> ☆ <hr/> <div>Updated: 2013-01-13</div>	<code>\dim_while_do:nn {<dimension relation>} {<code>}</code>
	Evaluates the <i><dimension relation></i> as described for <code>\dim_compare:nTF</code> , and then places the <i><code></i> in the input stream if the <i><relation></i> is true . After the <i><code></i> has been processed by T _E X the test is repeated, and a loop occurs until the test is false .

25.6 Dimension step functions

<hr/> <code>\dim_step_function:nnnN</code> ☆ <hr/> <div>New: 2018-02-18</div>	<code>\dim_step_function:nnnN {<initial value>} {<step>} {<final value>} <function></code>
	This function first evaluates the <i><initial value></i> , <i><step></i> and <i><final value></i> , all of which should be dimension expressions. The <i><function></i> is then placed in front of each <i><value></i> from the <i><initial value></i> to the <i><final value></i> in turn (using <i><step></i> between each <i><value></i>). The <i><step></i> must be non-zero. If the <i><step></i> is positive, the loop stops when the <i><value></i> becomes larger than the <i><final value></i> . If the <i><step></i> is negative, the loop stops when the <i><value></i> becomes smaller than the <i><final value></i> . The <i><function></i> should absorb one argument.
<hr/> <code>\dim_step_inline:nnnn</code> <hr/> <div>New: 2018-02-18</div>	<code>\dim_step_inline:nnnn {<initial value>} {<step>} {<final value>} {<code>}</code>
	This function first evaluates the <i><initial value></i> , <i><step></i> and <i><final value></i> , all of which should be dimension expressions. Then for each <i><value></i> from the <i><initial value></i> to the <i><final value></i> in turn (using <i><step></i> between each <i><value></i>), the <i><code></i> is inserted into the input stream with #1 replaced by the current <i><value></i> . Thus the <i><code></i> should define a function of one argument (#1).

`\dim_step_variable:nnnNn`
 New: 2018-02-18

`\dim_step_variable:nnnNn`
`{\langle initial value \rangle}{\langle step \rangle}{\langle final value \rangle}{\langle tl var \rangle}{\langle code \rangle}`

This function first evaluates the $\langle initial value \rangle$, $\langle step \rangle$ and $\langle final value \rangle$, all of which should be dimension expressions. Then for each $\langle value \rangle$ from the $\langle initial value \rangle$ to the $\langle final value \rangle$ in turn (using $\langle step \rangle$ between each $\langle value \rangle$), the $\langle code \rangle$ is inserted into the input stream, with the $\langle tl var \rangle$ defined as the current $\langle value \rangle$. Thus the $\langle code \rangle$ should make use of the $\langle tl var \rangle$.

25.7 Using dim expressions and variables

`\dim_eval:n` ★
 Updated: 2011-10-22

`\dim_eval:n` $\{\langle dimension expression \rangle\}$

Evaluates the $\langle dimension expression \rangle$, expanding any dimensions and token list variables within the $\langle expression \rangle$ to their content (without requiring `\dim_use:N/\tl_use:N`) and applying the standard mathematical rules. The result of the calculation is left in the input stream as a $\langle dimension denotation \rangle$ after two expansions. This is expressed in points (`pt`), and requires suitable termination if used in a TeX-style assignment as it is *not* an $\langle internal dimension \rangle$.

`\dim_sign:n` ★
 New: 2018-11-03

`\dim_sign:n` $\{\langle dimexpr \rangle\}$

Evaluates the $\langle dimexpr \rangle$ then leaves 1 or 0 or -1 in the input stream according to the sign of the result.

`\dim_use:N` ★
`\dim_use:c` ★

`\dim_use:N` $\langle dimension \rangle$

Recovers the content of a $\langle dimension \rangle$ and places it directly in the input stream. An error is raised if the variable does not exist or if it is invalid. Can be omitted in places where a $\langle dimension \rangle$ is required (such as in the argument of `\dim_eval:n`).

TeXhackers note: `\dim_use:N` is the TeX primitive `\the`: this is one of several L^ATeX3 names for this primitive.

`\dim_to_decimal:n` ★
 New: 2014-07-15

`\dim_to_decimal:n` $\{\langle dimexpr \rangle\}$

Evaluates the $\langle dimension expression \rangle$, and leaves the result, expressed in points (`pt`) in the input stream, with *no units*. The result is rounded by TeX to four or five decimal places. If the decimal part of the result is zero, it is omitted, together with the decimal marker.

For example

`\dim_to_decimal:n { 1bp }`

leaves 1.00374 in the input stream, *i.e.* the magnitude of one “big point” when converted to (TeX) points.

<hr/> <code>\dim_to_decimal_in_bp:n</code> ★ <hr/>	<code>\dim_to_decimal_in_bp:n {⟨dimexpr⟩}</code>
<hr/> New: 2014-07-15 <hr/>	Evaluates the <i>⟨dimension expression⟩</i> , and leaves the result, expressed in big points (bp) in the input stream, with <i>no units</i> . The result is rounded by T _E X to four or five decimal places. If the decimal part of the result is zero, it is omitted, together with the decimal marker.

For example

```
\dim_to_decimal_in_bp:n { 1pt }
```

leaves 0.99628 in the input stream, *i.e.* the magnitude of one (T_EX) point when converted to big points.

<hr/> <code>\dim_to_decimal_in_sp:n</code> ★ <hr/>	<code>\dim_to_decimal_in_sp:n {⟨dimexpr⟩}</code>
<hr/> New: 2015-05-18 <hr/>	Evaluates the <i>⟨dimension expression⟩</i> , and leaves the result, expressed in scaled points (sp) in the input stream, with <i>no units</i> . The result is necessarily an integer.

<hr/> <code>\dim_to_decimal_in_unit:nn</code> ★ <hr/>	<code>\dim_to_decimal_in_unit:nn {⟨dimexpr₁⟩} {⟨dimexpr₂⟩}</code>
<hr/> New: 2014-07-15 <hr/>	

Evaluates the *⟨dimension expressions⟩*, and leaves the value of *⟨dimexpr₁⟩*, expressed in a unit given by *⟨dimexpr₂⟩*, in the input stream. The result is a decimal number, rounded by T_EX to four or five decimal places. If the decimal part of the result is zero, it is omitted, together with the decimal marker.

For example

```
\dim_to_decimal_in_unit:nn { 1bp } { 1mm }
```

leaves 0.35277 in the input stream, *i.e.* the magnitude of one big point when converted to millimetres.

Note that this function is not optimised for any particular output and as such may give different results to `\dim_to_decimal_in_bp:n` or `\dim_to_decimal_in_sp:n`. In particular, the latter is able to take a wider range of input values as it is not limited by the ability to calculate a ratio using ε -T_EX primitives, which is required internally by `\dim_to_decimal_in_unit:nn`.

<hr/> <code>\dim_to_fp:n</code> ★ <hr/>	<code>\dim_to_fp:n {⟨dimexpr⟩}</code>
<hr/> New: 2012-05-08 <hr/>	Expands to an internal floating point number equal to the value of the <i>⟨dimexpr⟩</i> in pt. Since dimension expressions are evaluated much faster than their floating point equivalent, <code>\dim_to_fp:n</code> can be used to speed up parts of a computation where a low precision and a smaller range are acceptable.

25.8 Viewing dim variables

<hr/> <code>\dim_show:N</code> <hr/>	<code>\dim_show:N ⟨dimension⟩</code>
<code>\dim_show:c</code>	Displays the value of the <i>⟨dimension⟩</i> on the terminal.

<hr/> <code>\dim_show:n</code> <hr/>	<code>\dim_show:n {⟨<i>dimension expression</i>⟩}</code>
New: 2011-11-22 Updated: 2015-08-07	Displays the result of evaluating the $\langle dimension\ expression \rangle$ on the terminal.
<hr/> <code>\dim_log:N</code> <code>\dim_log:c</code> <hr/>	<code>\dim_log:N ⟨<i>dimension</i>⟩</code>
New: 2014-08-22 Updated: 2015-08-03	Writes the value of the $\langle dimension \rangle$ in the log file.
<hr/> <code>\dim_log:n</code> <hr/>	<code>\dim_log:n {⟨<i>dimension expression</i>⟩}</code>
New: 2014-08-22 Updated: 2015-08-07	Writes the result of evaluating the $\langle dimension\ expression \rangle$ in the log file.

25.9 Constant dimensions

<hr/> <code>\c_max_dim</code> <hr/>	The maximum value that can be stored as a dimension. This can also be used as a component of a skip.
<hr/> <code>\c_zero_dim</code> <hr/>	A zero length as a dimension. This can also be used as a component of a skip.

25.10 Scratch dimensions

<hr/> <code>\l_tmpa_dim</code> <code>\l_tmpb_dim</code> <hr/>	Scratch dimension for local assignment. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
<hr/> <code>\g_tmpa_dim</code> <code>\g_tmpb_dim</code> <hr/>	Scratch dimension for global assignment. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

25.11 Creating and initialising skip variables

<hr/> <code>\skip_new:N</code> <code>\skip_new:c</code> <hr/>	<code>\skip_new:N ⟨<i>skip</i>⟩</code>
	Creates a new $\langle skip \rangle$ or raises an error if the name is already taken. The declaration is global. The $\langle skip \rangle$ is initially equal to 0 pt.

<code>\skip_const:Nn</code>	<code>\skip_const:Nn <skip> {<skip expression>}</code>
<code>\skip_const:cn</code>	Creates a new constant <i><skip></i> or raises an error if the name is already taken. The value of the <i><skip></i> is set globally to the <i><skip expression></i> .
New: 2012-03-05	

<code>\skip_zero:N</code>	<code>\skip_zero:N <skip></code>
<code>\skip_zero:c</code>	Sets <i><skip></i> to 0pt.
<code>\skip_gzero:N</code>	
<code>\skip_gzero:c</code>	

<code>\skip_zero_new:N</code>	<code>\skip_zero_new:N <skip></code>
<code>\skip_zero_new:c</code>	Ensures that the <code><skip></code> exists globally by applying <code>\skip_new:N</code> if necessary, then applies <code>\skip_(g)zero:N</code> to leave the <code><skip></code> set to zero.
<code>\skip_gzero_new:N</code>	
<code>\skip_gzero_new:c</code>	
<hr/>	
New: 2012-01-07	

<code>\skip_if_exist_p:N *</code>	<code>\skip_if_exist_p:N $\langle skip \rangle$</code>
<code>\skip_if_exist_p:c *</code>	<code>\skip_if_exist:NTF $\langle skip \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$</code>
<code>\skip_if_exist:NTF *</code>	Tests whether the $\langle skip \rangle$ is currently defined. This does not check that the $\langle skip \rangle$ really is a skip variable.
<code>\skip_if_exist:cTF *</code>	

New: 2012-03-03

25.12 Setting skip variables

<code>\skip_add:Nn</code>	<code>\skip_add:Nn <skip> {<skip expression>}</code>
<code>\skip_add:cn</code>	Adds the result of the <i><skip expression></i> to the current content of the <i><skip></i> .
<code>\skip_gadd:Nn</code>	
<code>\skip_gadd:cn</code>	
<hr/>	
Updated: 2011-10-22	

<code>\skip_set:Nn</code>	<code>\skip_set:Nn <skip> {<skip expression>}</code>
<code>\skip_set:cn</code>	Sets <code><skip></code> to the value of <code><skip expression></code> , which must evaluate to a length with units and may include a rubber component (for example 1 cm plus 0.5 cm).
<code>\skip_gset:Nn</code>	
<code>\skip_gset:cn</code>	

Updated: 2011-10-22

<code>\skip_set_eq:NN</code>	<code>\skip_set_eq:NN <skip₁₂</code>
<code>\skip_set_eq:(cN Nc cc)</code>	Sets the content of <i><skip_{1 equal to that of <i><skip_{2.}</i>}</i>
<code>\skip_gset_eq:NN</code>	
<code>\skip_gset_eq:(cN Nc cc)</code>	

<code>\skip_sub:Nn</code>	<code>\skip_sub:Nn <skip> {<skip expression>}</code>
<code>\skip_sub:cn</code>	Subtracts the result of the <i><skip expression></i> from the current content of the <i><skip></i> .
<code>\skip_gsub:Nn</code>	
<code>\skip_gsub:cn</code>	
<hr/>	
Updated: 2011-10-22	

25.13 Skip expression conditionals

<code>\skip_if_eq_p:nn</code> *	<code>\skip_if_eq_p:nn {\langle skipexpr_1 \rangle} {\langle skipexpr_2 \rangle}</code>
<code>\skip_if_eq:nnTF</code> *	<code>\skip_if_eq:nnTF</code> <code>{\langle skipexpr_1 \rangle} {\langle skipexpr_2 \rangle}</code> <code>{\langle true code \rangle} {\langle false code \rangle}</code>

This function first evaluates each of the $\langle skip\ expressions \rangle$ as described for `\skip_eval:n`. The two results are then compared for exact equality, *i.e.* both the fixed and rubber components must be the same for the test to be true.

<code>\skip_if_finite_p:n</code> *	<code>\skip_if_finite_p:n {\langle skipexpr \rangle}</code>
<code>\skip_if_finite:nTF</code> *	<code>\skip_if_finite:nTF {\langle skipexpr \rangle} {\langle true code \rangle} {\langle false code \rangle}</code>

New: 2012-03-05

Evaluates the $\langle skip\ expression \rangle$ as described for `\skip_eval:n`, and then tests if all of its components are finite.

25.14 Using skip expressions and variables

<code>\skip_eval:n</code> *	<code>\skip_eval:n {\langle skip expression \rangle}</code>
-----------------------------	---

Updated: 2011-10-22

Evaluates the $\langle skip\ expression \rangle$, expanding any skips and token list variables within the $\langle expression \rangle$ to their content (without requiring `\skip_use:N/\tl_use:N`) and applying the standard mathematical rules. The result of the calculation is left in the input stream as a $\langle glue\ denotation \rangle$ after two expansions. This is expressed in points (`pt`), and requires suitable termination if used in a \TeX -style assignment as it is *not* an $\langle internal\ glue \rangle$.

<code>\skip_use:N</code> *	<code>\skip_use:N \langle skip \rangle</code>
<code>\skip_use:c</code> *	

Recovers the content of a $\langle skip \rangle$ and places it directly in the input stream. An error is raised if the variable does not exist or if it is invalid. Can be omitted in places where a $\langle dimension \rangle$ or $\langle skip \rangle$ is required (such as in the argument of `\skip_eval:n`).

\TeX hackers note: `\skip_use:N` is the \TeX primitive `\the`: this is one of several \LaTeX 3 names for this primitive.

25.15 Viewing skip variables

<code>\skip_show:N</code>	<code>\skip_show:N \langle skip \rangle</code>
<code>\skip_show:c</code>	

Updated: 2015-08-03

Displays the value of the $\langle skip \rangle$ on the terminal.

<code>\skip_show:n</code>	<code>\skip_show:n {\langle skip expression \rangle}</code>
---------------------------	---

New: 2011-11-22
Updated: 2015-08-07

Displays the result of evaluating the $\langle skip\ expression \rangle$ on the terminal.

<code>\skip_log:N</code>	<code>\skip_log:N <skip></code>
<code>\skip_log:c</code>	Writes the value of the $\langle skip \rangle$ in the log file.
New: 2014-08-22	
Updated: 2015-08-03	

<code>\skip_log:n</code>	<code>\skip_log:n {\langle skip expression \rangle}</code>
	Writes the result of evaluating the $\langle skip expression \rangle$ in the log file.
New: 2014-08-22	
Updated: 2015-08-07	

25.16 Constant skips

<code>\c_max_skip</code>	The maximum value that can be stored as a skip (equal to <code>\c_max_dim</code> in length), with no stretch nor shrink component.
Updated: 2012-11-02	

<code>\c_zero_skip</code>	A zero length as a skip, with no stretch nor shrink component.
Updated: 2012-11-01	

25.17 Scratch skips

<code>\l_tmpa_skip</code> <code>\l_tmpb_skip</code>	Scratch skip for local assignment. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
--	--

<code>\g_tmpa_skip</code> <code>\g_tmpb_skip</code>	Scratch skip for global assignment. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
--	---

25.18 Inserting skips into the output

<code>\skip_horizontal:N</code>	<code>\skip_horizontal:N <skip></code>
<code>\skip_horizontal:c</code>	<code>\skip_horizontal:n {\langle skipexpr \rangle}</code>
<code>\skip_horizontal:n</code>	Inserts a horizontal $\langle skip \rangle$ into the current list. The argument can also be a $\langle dim \rangle$.
Updated: 2011-10-22	
T_EXhackers note: <code>\skip_horizontal:N</code> is the T _E X primitive <code>\hskip</code> renamed.	

<hr/> <code>\skip_vertical:N</code>	<code>\skip_vertical:N <skip></code>
<code>\skip_vertical:c</code>	<code>\skip_vertical:n {<skipexpr>}</code>
<code>\skip_vertical:n</code>	Inserts a vertical <code><skip></code> into the current list. The argument can also be a <code><dim></code> .
<hr/> Updated: 2011-10-22 <hr/>	

T_EXhackers note: `\skip_vertical:N` is the T_EX primitive `\vskip` renamed.

25.19 Creating and initialising muskip variables

<hr/> <code>\muskip_new:N</code>	<code>\muskip_new:N <muskip></code>
<code>\muskip_new:c</code>	Creates a new <code><muskip></code> or raises an error if the name is already taken. The declaration is global. The <code><muskip></code> is initially equal to 0 mu.

<hr/> <code>\muskip_const:Nn</code>	<code>\muskip_const:Nn <muskip> {<muskip expression>}</code>
<code>\muskip_const:cn</code>	Creates a new constant <code><muskip></code> or raises an error if the name is already taken. The value of the <code><muskip></code> is set globally to the <code><muskip expression></code> .
<hr/> New: 2012-03-05 <hr/>	

<hr/> <code>\muskip_zero:N</code>	<code>\skip_zero:N <muskip></code>
<code>\muskip_zero:c</code>	Sets <code><muskip></code> to 0 mu.
<code>\muskip_gzero:N</code>	
<code>\muskip_gzero:c</code>	

<code>\muskip_zero_new:N</code>	<code>\muskip_zero_new:N <muskip></code>
<code>\muskip_zero_new:c</code>	Ensures that the <code><muskip></code> exists globally by applying <code>\muskip_new:N</code> if necessary, then applies <code>\muskip_(g)zero:N</code> to leave the <code><muskip></code> set to zero.
<code>\muskip_gzero_new:N</code>	
<code>\muskip_gzero_new:c</code>	
<hr/>	
New: 2012-01-07	

<code>\muskip_if_exist_p:N</code> *	<code>\muskip_if_exist_p:N</code> $\langle muskip \rangle$
<code>\muskip_if_exist_p:c</code> *	<code>\muskip_if_exist:NTF</code> $\langle muskip \rangle$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$
<code>\muskip_if_exist:NTF</code> *	Tests whether the $\langle muskip \rangle$ is currently defined. This does not check that the $\langle muskip \rangle$ really is a muskip variable.
<code>\muskip_if_exist:cTF</code> *	
<hr/>	
New: 2012-03-03	

25.20 Setting muskip variables

<code>\muskip_add:Nn</code>	<code>\muskip_add:Nn <muskip> {<muskip expression>}</code>
<code>\muskip_add:cn</code>	Adds the result of the <i><muskip expression></i> to the current content of the <i><muskip></i> .
<code>\muskip_gadd:Nn</code>	
<code>\muskip_gadd:cn</code>	
<hr/>	
Updated: 2011-10-22	

<hr/> <code>\muskip_set:Nn</code> <hr/>	<code>\muskip_set:Nn <muskip> {<muskip expression>}</code>
<code>\muskip_set:cn</code>	
<code>\muskip_gset:Nn</code>	Sets $\langle muskip \rangle$ to the value of $\langle muskip expression \rangle$, which must evaluate to a math length with units and may include a rubber component (for example 1 mu plus 0.5 mu).
<code>\muskip_gset:cn</code> <hr/>	
Updated: 2011-10-22 <hr/>	

<hr/> <code>\muskip_set_eq:NN</code> <hr/>	<code>\muskip_set_eq:NN <muskip₁> <muskip₂></code>
<code>\muskip_set_eq:(cN Nc cc)</code>	
<code>\muskip_gset_eq:NN</code>	Sets the content of $\langle muskip_1 \rangle$ equal to that of $\langle muskip_2 \rangle$.
<code>\muskip_gset_eq:(cN Nc cc)</code> <hr/>	

<hr/> <code>\muskip_sub:Nn</code> <hr/>	<code>\muskip_sub:Nn <muskip> {<muskip expression>}</code>
<code>\muskip_sub:cn</code>	
<code>\muskip_gsub:Nn</code>	Subtracts the result of the $\langle muskip expression \rangle$ from the current content of the $\langle muskip \rangle$.
<code>\muskip_gsub:cn</code> <hr/>	
Updated: 2011-10-22 <hr/>	

25.21 Using muskip expressions and variables

<hr/> <code>\muskip_eval:n *</code> <hr/>	<code>\muskip_eval:n {<muskip expression>}</code>
Updated: 2011-10-22 <hr/>	Evaluates the $\langle muskip expression \rangle$, expanding any skips and token list variables within the $\langle expression \rangle$ to their content (without requiring <code>\muskip_use:N/\tl_use:N</code>) and applying the standard mathematical rules. The result of the calculation is left in the input stream as a $\langle muglue denotation \rangle$ after two expansions. This is expressed in mu, and requires suitable termination if used in a TeX-style assignment as it is <i>not</i> an $\langle internal muglue \rangle$.

<hr/> <code>\muskip_use:N *</code> <hr/>	<code>\muskip_use:N <muskip></code>
<code>\muskip_use:c *</code> <hr/>	Recovers the content of a $\langle skip \rangle$ and places it directly in the input stream. An error is raised if the variable does not exist or if it is invalid. Can be omitted in places where a $\langle dimension \rangle$ is required (such as in the argument of <code>\muskip_eval:n</code>).

TeXhackers note: `\muskip_use:N` is the TeX primitive `\the`: this is one of several L^AT_EX3 names for this primitive.

25.22 Viewing muskip variables

<hr/> <code>\muskip_show:N</code> <hr/>	<code>\muskip_show:N <muskip></code>
<code>\muskip_show:c</code> <hr/>	Displays the value of the $\langle muskip \rangle$ on the terminal.
Updated: 2015-08-03 <hr/>	

<hr/> <code>\muskip_show:n</code> <hr/>	<code>\muskip_show:n {⟨<i>muskip expression</i>⟩}</code>
New: 2011-11-22 Updated: 2015-08-07	Displays the result of evaluating the $\langle muskip expression \rangle$ on the terminal.
<hr/> <code>\muskip_log:N</code> <code>\muskip_log:c</code> <hr/>	<code>\muskip_log:N ⟨<i>muskip</i>⟩</code>
New: 2014-08-22 Updated: 2015-08-03	Writes the value of the $\langle muskip \rangle$ in the log file.
<hr/> <code>\muskip_log:n</code> <hr/>	<code>\muskip_log:n {⟨<i>muskip expression</i>⟩}</code>
New: 2014-08-22 Updated: 2015-08-07	Writes the result of evaluating the $\langle muskip expression \rangle$ in the log file.

25.23 Constant muskips

<hr/> <code>\c_max_muskip</code> <hr/>	The maximum value that can be stored as a muskip, with no stretch nor shrink component.
<hr/> <code>\c_zero_muskip</code> <hr/>	A zero length as a muskip, with no stretch nor shrink component.

25.24 Scratch muskips

<hr/> <code>\l_tmpa_muskip</code> <code>\l_tmpb_muskip</code> <hr/>	Scratch muskip for local assignment. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
<hr/> <code>\g_tmpa_muskip</code> <code>\g_tmpb_muskip</code> <hr/>	Scratch muskip for global assignment. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

25.25 Primitive conditional

<hr/> <code>\if_dim:w ★</code> <hr/>	<code>\if_dim:w ⟨<i>dimen</i>₁⟩ ⟨<i>relation</i>⟩ ⟨<i>dimen</i>₂⟩</code> <code> ⟨<i>true code</i>⟩</code> <code> \else:</code> <code> ⟨<i>false</i>⟩</code> <code> \fi:</code>
	Compare two dimensions. The $\langle relation \rangle$ is one of $<$, $=$ or $>$ with category code 12.

T_EXhackers note: This is the T_EX primitive `\ifdim`.

Chapter 26

The l3keys package

Key–value interfaces

The key–value method is a popular system for creating large numbers of settings for controlling function or package behaviour. The system normally results in input of the form

```
\MyModuleSetup{
  key-one = value one,
  key-two = value two
}
```

or

```
\MyModuleMacro[
  key-one = value one,
  key-two = value two
]{argument}
```

for the user.

The high level functions here are intended as a method to create key–value controls. Keys are themselves created using a key–value interface, minimising the number of functions and arguments required. Each key is created by setting one or more *properties* of the key:

```
\keys_define:nn { mymodule }
{
  key-one .code:n    = code including parameter #1,
  key-two .tl_set:N = \l_mymodule_store_tl
}
```

These values can then be set as with other key–value approaches:

```
\keys_set:nn { mymodule }
{
  key-one = value one,
  key-two = value two
}
```

As illustrated, keys are created inside a $\langle module \rangle$: a set of related keys, typically those for a single module/L^AT_EX 2 ϵ package. See Section for suggestions on how to divide large numbers of keys for a single module.

At a document level, `\keys_set:nn` is used within a document function, for example

```
\DeclareDocumentCommand \MyModuleSetup { m }
  { \keys_set:nn { mymodule } { #1 } }
\DeclareDocumentCommand \MyModuleMacro { o m }
{
  \group_begin:
    \keys_set:nn { mymodule } { #1 }
    % Main code for \MyModuleMacro
  \group_end:
}
```

Key names may contain any tokens, as they are handled internally using `\tl_to_str:n`. As discussed in section 26.2, it is suggested that the character `/` is reserved for sub-division of keys into logical groups. Functions and variables are *not* expanded when creating key names, and so

```
\tl_set:Nn \l_mymodule_tmp_tl { key }
\keys_define:nn { mymodule }
{
  \l_mymodule_tmp_tl .code:n = code
}
```

creates a key called `\l_mymodule_tmp_tl`, and not one called `key`.

26.1 Creating keys

`\keys_define:nn`

Updated: 2017-11-14

`\keys_define:nn { $\langle module \rangle$ } { $\langle keyval list \rangle$ }`

Parses the $\langle keyval list \rangle$ and defines the keys listed there for $\langle module \rangle$. The $\langle module \rangle$ name is treated as a string. In practice the $\langle module \rangle$ should be chosen to be unique to the module in question (unless deliberately adding keys to an existing module).

The $\langle keyval list \rangle$ should consist of one or more key names along with an associated key *property*. The properties of a key determine how it acts. The individual properties are described in the following text; a typical use of `\keys_define:nn` might read

```
\keys_define:nn { mymodule }
{
  keyname .code:n = Some-code~using~#1,
  keyname .value_required:n = true
}
```

where the properties of the key begin from the `.` after the key name.

The various properties available take either no arguments at all, or require one or more arguments. This is indicated in the name of the property using an argument specification. In the following discussion, each property is illustrated attached to an arbitrary $\langle key \rangle$, which when used may be supplied with a $\langle value \rangle$. All key *definitions* are local.

Key properties are applied in the reading order and so the ordering is significant. Key properties which define “actions”, such as `.code:n`, `.tl_set:N`, *etc.*, override one another. Some other properties are mutually exclusive, notably `.value_required:n` and `.value_forbidden:n`, and so they replace one another. However, properties covering non-exclusive behaviours may be given in any order. Thus for example the following definitions are equivalent.

```
\keys_define:nn { mymodule }
{
  keyname .code:n          = Some~code~using~#1,
  keyname .value_required:n = true
}
\keys_define:nn { mymodule }
{
  keyname .value_required:n = true,
  keyname .code:n          = Some~code~using~#1
}
```

Note that with the exception of the special `.undefine:` property, all key properties define the key within the current \TeX scope.

`.bool_set:N`
`.bool_set:c`
`.bool_gset:N`
`.bool_gset:c`

Updated: 2013-07-08

$\langle key \rangle$ `.bool_set:N` = $\langle boolean\ variable \rangle$
 Defines $\langle key \rangle$ to set $\langle boolean\ variable \rangle$ to $\langle value \rangle$ (which must be either “true” or “false”). If the variable does not exist, it will be created globally at the point that the key is set up.

`.bool_set_inverse:N`
`.bool_set_inverse:c`
`.bool_gset_inverse:N`
`.bool_gset_inverse:c`

New: 2011-08-28
 Updated: 2013-07-08

$\langle key \rangle$ `.bool_set_inverse:N` = $\langle boolean\ variable \rangle$
 Defines $\langle key \rangle$ to set $\langle boolean\ variable \rangle$ to the logical inverse of $\langle value \rangle$ (which must be either “true” or “false”). If the $\langle boolean\ variable \rangle$ does not exist, it will be created globally at the point that the key is set up.

`.choice:`

$\langle key \rangle$ `.choice:`
 Sets $\langle key \rangle$ to act as a choice key. Each valid choice for $\langle key \rangle$ must then be created, as discussed in section 26.3.

`.choices:nn`
`.choices:(Vn|on|xn)`

New: 2011-08-21
 Updated: 2013-07-10

$\langle key \rangle$ `.choices:nn` = $\{ \langle choices \rangle \} \{ \langle code \rangle \}$
 Sets $\langle key \rangle$ to act as a choice key, and defines a series $\langle choices \rangle$ which are implemented using the $\langle code \rangle$. Inside $\langle code \rangle$, `\l_keys_choice_tl` will be the name of the choice made, and `\l_keys_choice_int` will be the position of the choice in the list of $\langle choices \rangle$ (indexed from 1). Choices are discussed in detail in section 26.3.

`.clist_set:N`
`.clist_set:c`
`.clist_gset:N`
`.clist_gset:c`

New: 2011-09-11

$\langle key \rangle$ `.clist_set:N` = $\langle comma\ list\ variable \rangle$
 Defines $\langle key \rangle$ to set $\langle comma\ list\ variable \rangle$ to $\langle value \rangle$. Spaces around commas and empty items will be stripped. If the variable does not exist, it is created globally at the point that the key is set up.

<hr/> <code>.code:n</code> <hr/>	<code><key> .code:n = {<code>}</code>
<code>Updated: 2013-07-10</code> <hr/>	Stores the <code><code></code> for execution when <code><key></code> is used. The <code><code></code> can include one parameter (<code>#1</code>), which will be the <code><value></code> given for the <code><key></code> .
<hr/> <code>.cs_set:Np</code> <code>.cs_set:cp</code> <code>.cs_set_protected:Np</code> <code>.cs_set_protected:cp</code> <code>.cs_gset:Np</code> <code>.cs_gset:cp</code> <code>.cs_gset_protected:Np</code> <code>.cs_gset_protected:cp</code> <hr/>	<code><key> .cs_set:Np = <control sequence> <arg. spec.></code> Defines <code><key></code> to set <code><control sequence></code> to have <code><arg. spec.></code> and replacement text <code><value></code> .
<code>New: 2020-01-11</code> <hr/>	
<hr/> <code>.default:n</code> <code>.default:(V o x)</code> <hr/>	<code><key> .default:n = {<default>}</code>
<code>Updated: 2013-07-09</code> <hr/>	Creates a <code><default></code> value for <code><key></code> , which is used if no value is given. This will be used if only the key name is given, but not if a blank <code><value></code> is given:
	<pre> \keys_define:nn { mymodule } { key .code:n = Hello~#1, key .default:n = World } \keys_set:nn { mymodule } { key = Fred, % Prints 'Hello Fred' key, % Prints 'Hello World' key = , % Prints 'Hello ' } </pre>
	The default does not affect keys where values are required or forbidden. Thus a required value cannot be supplied by a default value, and giving a default value for a key which cannot take a value does not trigger an error.
<hr/> <code>.dim_set:N</code> <code>.dim_set:c</code> <code>.dim_gset:N</code> <code>.dim_gset:c</code> <hr/>	<code><key> .dim_set:N = <dimension></code>
<code>Updated: 2020-01-17</code> <hr/>	Defines <code><key></code> to set <code><dimension></code> to <code><value></code> (which must a dimension expression). If the variable does not exist, it is created globally at the point that the key is set up. The key will require a value at point-of-use unless a default is set.
<hr/> <code>.fp_set:N</code> <code>.fp_set:c</code> <code>.fp_gset:N</code> <code>.fp_gset:c</code> <hr/>	<code><key> .fp_set:N = <floating point></code>
<code>Updated: 2020-01-17</code> <hr/>	Defines <code><key></code> to set <code><floating point></code> to <code><value></code> (which must a floating point expression). If the variable does not exist, it is created globally at the point that the key is set up. The key will require a value at point-of-use unless a default is set.
<hr/> <code>.groups:n</code> <hr/>	<code><key> .groups:n = {<groups>}</code>
<code>New: 2013-07-14</code> <hr/>	Defines <code><key></code> as belonging to the <code><groups></code> declared. Groups provide a “secondary axis” for selectively setting keys, and are described in Section 26.7 .

<code>.inherit:n</code>	<code><key> .inherit:n = {<parents>}</code>
-------------------------	---

New: 2016-11-22 Specifies that the `<key>` path should inherit the keys listed as any of the `<parents>` (a comma list), which can be a module or a subgroup. For example, after setting

```
\keys_define:nn { foo } { test .code:n = \tl_show:n {#1} }
\keys_define:nn { } { bar .inherit:n = foo }
```

setting

```
\keys_set:nn { bar } { test = a }
```

will be equivalent to

```
\keys_set:nn { foo } { test = a }
```

Inheritance applies at point of use, not at definition, thus keys may be added to the `<parent>` after the use of `.inherit:n` and will be active. If more than one `<parent>` is specified, the presence of the `<key>` will be tested for each in turn, with the first successful hit taking priority.

<code>.initial:n</code>	<code><key> .initial:n = {<value>}</code>
-------------------------	---

`.initial:(V|o|x)`

Initialises the `<key>` with the `<value>`, equivalent to

Updated: 2013-07-09

```
\keys_set:nn {<module>} { <key> = <value> }
```

<code>.int_set:N</code>	<code><key> .int_set:N = <integer></code>
-------------------------	---

`.int_set:c`

`.int_gset:N`

`.int_gset:c`

Defines `<key>` to set `<integer>` to `<value>` (which must be an integer expression). If the variable does not exist, it is created globally at the point that the key is set up. The key will require a value at point-of-use unless a default is set.

Updated: 2020-01-17

<code>.legacy_if_set:n</code>	<code><key> .legacy_if_set:n = <switch></code>
-------------------------------	--

`.legacy_if_gset:n`

`.legacy_if_set_inverse:n`

`.legacy_if_gset_inverse:n`

Defines `<key>` to set legacy `\if <switch>` to `<value>` (which must be either “true” or “false”). The `<switch>` is the name of the switch *without the leading \if*.

The *inverse* versions will set the `<switch>` to the logical opposite of the `<value>`.

Updated: 2022-01-15

<code>.meta:n</code>	<code><key> .meta:n = {<keyval list>}</code>
----------------------	--

Updated: 2013-07-10

Makes `<key>` a meta-key, which will set `<keyval list>` in one go. The `<keyval list>` can refer as `#1` to the value given at the time the `<key>` is used (or, if no value is given, the `<key>`’s default value).

<code>.meta:nn</code>	<code><key> .meta:nn = {<path>} {<keyval list>}</code>
-----------------------	--

New: 2013-07-10

Makes `<key>` a meta-key, which will set `<keyval list>` in one go using the `<path>` in place of the current one. The `<keyval list>` can refer as `#1` to the value given at the time the `<key>` is used (or, if no value is given, the `<key>`’s default value).

<hr/> <code>.multichoice:</code> <hr/>	<code><key> .multichoice:</code>
<code>New: 2011-08-21</code> <hr/>	Sets <code><key></code> to act as a multiple choice key. Each valid choice for <code><key></code> must then be created, as discussed in section 26.3 .
<hr/> <code>.multichoices:nn</code> <code>.multichoices:(Vn on xn)</code> <hr/>	<code><key> .multichoices:nn {<choices>} {<code>}</code>
<code>New: 2011-08-21</code> <code>Updated: 2013-07-10</code> <hr/>	Sets <code><key></code> to act as a multiple choice key, and defines a series <code><choices></code> which are implemented using the <code><code></code> . Inside <code><code></code> , <code>\l_keys_choice_tl</code> will be the name of the choice made, and <code>\l_keys_choice_int</code> will be the position of the choice in the list of <code><choices></code> (indexed from 1). Choices are discussed in detail in section 26.3 .
<hr/> <code>.muskip_set:N</code> <code>.muskip_set:c</code> <code>.muskip_gset:N</code> <code>.muskip_gset:c</code> <hr/>	<code><key> .muskip_set:N = <muskip></code>
<code>New: 2019-05-05</code> <code>Updated: 2020-01-17</code> <hr/>	Defines <code><key></code> to set <code><muskip></code> to <code><value></code> (which must be a muskip expression). If the variable does not exist, it is created globally at the point that the key is set up. The key will require a value at point-of-use unless a default is set.
<hr/> <code>.prop_put:N</code> <code>.prop_put:c</code> <code>.prop_gput:N</code> <code>.prop_gput:c</code> <hr/>	<code><key> .prop_put:N = <property list></code>
<code>New: 2019-01-31</code> <hr/>	Defines <code><key></code> to put the <code><value></code> onto the <code><property list></code> stored under the <code><key></code> . If the variable does not exist, it is created globally at the point that the key is set up.
<hr/> <code>.skip_set:N</code> <code>.skip_set:c</code> <code>.skip_gset:N</code> <code>.skip_gset:c</code> <hr/>	<code><key> .skip_set:N = <skip></code>
<code>Updated: 2020-01-17</code> <hr/>	Defines <code><key></code> to set <code><skip></code> to <code><value></code> (which must be a skip expression). If the variable does not exist, it is created globally at the point that the key is set up. The key will require a value at point-of-use unless a default is set.
<hr/> <code>.str_set:N</code> <code>.str_set:c</code> <code>.str_gset:N</code> <code>.str_gset:c</code> <hr/>	<code><key> .str_set:N = <string variable></code>
<code>New: 2021-10-30</code> <hr/>	Defines <code><key></code> to set <code><string variable></code> to <code><value></code> . If the variable does not exist, it is created globally at the point that the key is set up.
<hr/> <code>.str_set_x:N</code> <code>.str_set_x:c</code> <code>.str_gset_x:N</code> <code>.str_gset_x:c</code> <hr/>	<code><key> .str_set_x:N = <string variable></code>
<code>New: 2021-10-30</code> <hr/>	Defines <code><key></code> to set <code><string variable></code> to <code><value></code> , which will be subjected to an <code>x</code> -type expansion (<i>i.e.</i> using <code>\str_set:Nx</code>). If the variable does not exist, it is created globally at the point that the key is set up.
<hr/> <code>.tl_set:N</code> <code>.tl_set:c</code> <code>.tl_gset:N</code> <code>.tl_gset:c</code> <hr/>	<code><key> .tl_set:N = <token list variable></code>
	Defines <code><key></code> to set <code><token list variable></code> to <code><value></code> . If the variable does not exist, it is created globally at the point that the key is set up.

<code>.tl_set_x:N</code>	<code><key> .tl_set_x:N = <token list variable></code>
<code>.tl_set_x:c</code>	
<code>.tl_gset_x:N</code>	Defines <code><key></code> to set <code><token list variable></code> to <code><value></code> , which will be subjected to an <code>x</code> -type expansion (<i>i.e.</i> using <code>\tl_set:Nx</code>). If the variable does not exist, it is created globally at the point that the key is set up.
<code>.tl_gset_x:c</code>	

<code>.undefine:</code>	<code><key> .undefine:</code>
New: 2015-07-14	Removes the definition of the <code><key></code> within the current scope.

<code>.value_forbidden:n</code>	<code><key> .value_forbidden:n = true false</code>
New: 2015-07-14	Specifies that <code><key></code> cannot receive a <code><value></code> when used. If a <code><value></code> is given then an error will be issued. Setting the property “ false ” cancels the restriction.

<code>.value_required:n</code>	<code><key> .value_required:n = true false</code>
New: 2015-07-14	Specifies that <code><key></code> must receive a <code><value></code> when used. If a <code><value></code> is not given then an error will be issued. Setting the property “ false ” cancels the restriction.

26.2 Sub-dividing keys

When creating large numbers of keys, it may be desirable to divide them into several sub-groups for a given module. This can be achieved either by adding a sub-division to the module name:

```
\keys_define:nn { mymodule / subgroup }
  { key .code:n = code }
```

or to the key name:

```
\keys_define:nn { mymodule }
  { subgroup / key .code:n = code }
```

As illustrated, the best choice of token for sub-dividing keys in this way is `/`. This is because of the method that is used to represent keys internally. Both of the above code fragments set the same key, which has full name `mymodule/subgroup/key`.

As illustrated in the next section, this subdivision is particularly relevant to making multiple choices.

26.3 Choice and multiple choice keys

The `l3keys` system supports two types of choice key, in which a series of pre-defined input values are linked to varying implementations. Choice keys are usually created so that the various values are mutually-exclusive: only one can apply at any one time. “Multiple” choice keys are also supported: these allow a selection of values to be chosen at the same time.

Mutually-exclusive choices are created by setting the `.choice:` property:

```
\keys_define:nn { mymodule }
  { key .choice: }
```

For keys which are set up as choices, the valid choices are generated by creating sub-keys of the choice key. This can be carried out in two ways.

In many cases, choices execute similar code which is dependant only on the name of the choice or the position of the choice in the list of all possibilities. Here, the keys can share the same code, and can be rapidly created using the `.choices:nn` property.

```
\keys_define:nn { mymodule }
{
  key .choices:nn =
    { choice-a, choice-b, choice-c }
    {
      You~gave~choice~'\tl_use:N \l_keys_choice_tl',~
      which~is~in~position~\int_use:N \l_keys_choice_int \c_space_tl
      in~the~list.
    }
}
```

The index `\l_keys_choice_int` in the list of choices starts at 1.

`\l_keys_choice_int`
`\l_keys_choice_tl`

Inside the code block for a choice generated using `.choices:nn`, the variables `\l_keys_choice_tl` and `\l_keys_choice_int` are available to indicate the name of the current choice, and its position in the comma list. The position is indexed from 1. Note that, as with standard key code generated using `.code:n`, the value passed to the key (i.e. the choice name) is also available as `#1`.

On the other hand, it is sometimes useful to create choices which use entirely different code from one another. This can be achieved by setting the `.choice:` property of a key, then manually defining sub-keys.

```
\keys_define:nn { mymodule }
{
  key .choice:,
  key / choice-a .code:n = code-a,
  key / choice-b .code:n = code-b,
  key / choice-c .code:n = code-c,
}
```

It is possible to mix the two methods, but manually-created choices should *not* use `\l_keys_choice_tl` or `\l_keys_choice_int`. These variables do not have defined behaviour when used outside of code created using `.choices:nn` (i.e. anything might happen).

It is possible to allow choice keys to take values which have not previously been defined by adding code for the special **unknown** choice. The general behavior of the **unknown** key is described in Section 26.6. A typical example in the case of a choice would be to issue a custom error message:

```
\keys_define:nn { mymodule }
{
  key .choice:,
  key / choice-a .code:n = code-a,
  key / choice-b .code:n = code-b,
  key / choice-c .code:n = code-c,
```

```

key / unknown .code:n =
  \msg_error:nnxxx { mymodule } { unknown-choice }
  { key } % Name of choice key
  { choice-a , choice-b , choice-c } % Valid choices
  { \exp_not:n {#1} } % Invalid choice given
%
%
}

```

Multiple choices are created in a very similar manner to mutually-exclusive choices, using the properties `.multichoice:` and `.multichoices:nn`. As with mutually exclusive choices, multiple choices are define as sub-keys. Thus both

```

\keys_define:nn { mymodule }
{
  key .multichoices:nn =
    { choice-a, choice-b, choice-c }
    {
      You~gave~choice~'\tl_use:N \l_keys_choice_tl',~
      which~is~in~position~
      \int_use:N \l_keys_choice_int \c_space_tl
      in~the~list.
    }
}

```

and

```

\keys_define:nn { mymodule }
{
  key .multichoice:,
  key / choice-a .code:n = code-a,
  key / choice-b .code:n = code-b,
  key / choice-c .code:n = code-c,
}

```

are valid.

When a multiple choice key is set

```

\keys_set:nn { mymodule }
{
  key = { a , b , c } % 'key' defined as a multiple choice
}

```

each choice is applied in turn, equivalent to a `clist` mapping or to applying each value individually:

```

\keys_set:nn { mymodule }
{
  key = a ,
  key = b ,
  key = c ,
}

```

Thus each separate choice will have passed to it the `\l_keys_choice_tl` and `\l_keys_choice_int` in exactly the same way as described for `.choices:nn`.

26.4 Key usage scope

Some keys will be used as settings which have a strictly limited scope of usage. Some will be only available once, others will only be valid until typesetting begins. To allow formats to support this in a structured way, `l3keys` allows this information to be specified using the `.usage:n` property.

<code>.usage:n</code>	<code><key> .usage:n = <scope></code>
New: 2022-01-10	Defines the <code><key></code> to have usage within the <code><scope></code> , which should be one of general , preamble or load .

<code>\l_keys_usage_load_prop</code>
<code>\l_keys_usage_preamble_prop</code>
New: 2022-01-10

`l3keys` itself does *not* attempt to redefine keys based on the usage scope. Rather, this information is made available with these two property lists. These hold an entry for each module (prefix); the value of each entry is a comma-separated list of the usage-restricted key(s).

26.5 Setting keys

<code>\keys_set:nn</code>	<code>\keys_set:nn {<module>} {<keyval list>}</code>
<code>\keys_set:(nV nv no)</code>	Parses the <code><keyval list></code> , and sets those keys which are defined for <code><module></code> . The behaviour on finding an unknown key can be set by defining a special unknown key: this is illustrated later.
Updated: 2017-11-14	

<code>\l_keys_key_str</code>	For each key processed, information of the full <i>path</i> of the key, the <i>name</i> of the key and the <i>value</i> of the key is available within three token list variables. These may be used within the code of the key.
<code>\l_keys_path_str</code>	
<code>\l_keys_value_tl</code>	

Updated: 2020-02-08

The *value* is everything after the `=`, which may be empty if no value was given. This is stored in `\l_keys_value_tl`, and is not processed in any way by `\keys_set:nn`.

The *path* of the key is a “full” description of the key, and is unique for each key. It consists of the module and full key name, thus for example

```
\keys_set:nn { mymodule } { key-a = some-value }
```

has path `mymodule/key-a` while

```
\keys_set:nn { mymodule } { subset / key-a = some-value }
```

has path `mymodule/subset/key-a`. This information is stored in `\l_keys_path_str`.

The *name* of the key is the part of the path after the last `/`, and thus is not unique. In the preceding examples, both keys have name `key-a` despite having different paths. This information is stored in `\l_keys_key_str`.

26.6 Handling of unknown keys

If a key has not previously been defined (is unknown), `\keys_set:nn` looks for a special `unknown` key for the same module, and if this is not defined raises an error indicating that the key name was unknown. This mechanism can be used for example to issue custom error texts.

```
\keys_define:nn { mymodule }
{
  unknown .code:n =
    You~tried~to~set~key~'\l_keys_key_str'~to~'#1'.
}
```

<code>\keys_set_known:nn</code>	<code>\keys_set_known:nn {<module>} {<keyval list>}</code>
<code>\keys_set_known:(nV nv no)</code>	<code>\keys_set_known:nnN {<module>} {<keyval list>} <tl></code>
<code>\keys_set_known:nnN</code>	<code>\keys_set_known:nnnN {<module>} {<keyval list>} {<root>} <tl></code>
<code>\keys_set_known:(nVN nvN noN)</code>	
<code>\keys_set_known:nnnN</code>	
<code>\keys_set_known:(nVnN nvN nonN)</code>	

New: 2011-08-23

Updated: 2019-01-29

These functions set keys which are known for the `<module>`, and simply ignore other keys. The `\keys_set_known:nn` function parses the `<keyval list>`, and sets those keys which are defined for `<module>`. Any keys which are unknown are not processed further by the parser. In addition, `\keys_set_known:nnN` stores the key–value pairs in the `<tl>` in comma-separated form (*i.e.* an edited version of the `<keyval list>`). When a `<root>` is given (`\keys_set_known:nnnN`), the key–value entries are returned relative to this point in the key tree. When it is absent, only the key name and value are provided. The correct list is returned by nested calls.

26.7 Selective key setting

In some cases it may be useful to be able to select only some keys for setting, even though these keys have the same path. For example, with a set of keys defined using

```
\keys define:nn { mymodule }
{
  key-one .code:n = { \my_func:n {#1} } ,
  key-two .tl_set:N = \l_my_a_tl ,
  key-three .tl_set:N = \l_my_b_tl ,
  key-four .fp_set:N = \l_my_a_fp ,
}
```

the use of `\keys_set:nn` attempts to set all four keys. However, in some contexts it may only be sensible to set some keys, or to control the order of setting. To do this, keys may be assigned to *groups*: arbitrary sets which are independent of the key tree. Thus modifying the example to read

```

\keys_define:nn { mymodule }
{
  key-one   .code:n   = { \my_func:n {#1} } ,
  key-one   .groups:n = { first }           ,
  key-two   .tl_set:N = \l_my_a_tl          ,
  key-two   .groups:n = { first }           ,
  key-three .tl_set:N = \l_my_b_tl          ,
  key-three .groups:n = { second }          ,
  key-four  .fp_set:N = \l_my_a_fp         ,
}

```

assigns `key-one` and `key-two` to group `first`, `key-three` to group `second`, while `key-four` is not assigned to a group.

Selective key setting may be achieved either by selecting one or more groups to be made “active”, or by marking one or more groups to be ignored in key setting.

<code>\keys_set_filter:nnn</code>	<code>\keys_set_filter:nnn {<module>} {<groups>} {<keyval list>}</code>
<code>\keys_set_filter:(nnV nnv nno)</code>	<code>\keys_set_filter:nnnN {<module>} {<groups>} {<keyval list>} <tl></code>
<code>\keys_set_filter:nnnN</code>	<code>\keys_set_filter:nnnnN {<module>} {<groups>} {<keyval list>} <root></code>
<code>\keys_set_filter:(nnVN nnvN nnoN)</code>	<code><tl></code>
<code>\keys_set_filter:nnnnN</code>	
<code>\keys_set_filter:(nnVnN nnvnN nnonN)</code>	

New: 2013-07-14

Updated: 2019-01-29

Activates key filtering in an “opt-out” sense: keys assigned to any of the *<groups>* specified are ignored. The *<groups>* are given as a comma-separated list. Unknown keys are not assigned to any group and are thus always set. The key–value pairs for each key which is filtered out are stored in the *<tl>* in a comma-separated form (*i.e.* an edited version of the *<keyval list>*). The `\keys_set_filter:nnn` version skips this stage.

Use of `\keys_set_filter:nnnN` can be nested, with the correct residual *<keyval list>* returned at each stage. In the version which takes a *<root>* argument, the key list is returned relative to that point in the key tree. In the cases without a *<root>* argument, only the key names and values are returned.

<code>\keys_set_groups:nnn</code>	<code>\keys_set_groups:nnn {<module>} {<groups>} {<keyval list>}</code>
<code>\keys_set_groups:(nnV nnv nno)</code>	

New: 2013-07-14

Updated: 2017-05-27

Activates key filtering in an “opt-in” sense: only keys assigned to one or more of the *<groups>* specified are set. The *<groups>* are given as a comma-separated list. Unknown keys are not assigned to any group and are thus never set.

26.8 Utility functions for keys

<code>\keys_if_exist_p:nn</code>	★	<code>\keys_if_exist_p:nn {<module>} {<key>}</code>
<code>\keys_if_exist_p:ne</code>	★	<code>\keys_if_exist:nnTF {<module>} {<key>} {<true code>} {<false code>}</code>
<code>\keys_if_exist:nnTF</code>	★	Tests if the <i><key></i> exists for <i><module></i> , <i>i.e.</i> if any code has been defined for <i><key></i> .
<code>\keys_if_exist:neTF</code>	★	

Updated: 2022-01-10

<code>\keys_if_choice_exist_p:nnn</code>	★	<code>\keys_if_choice_exist_p:nnn {<module>} {<key>} {<choice>}</code>
<code>\keys_if_choice_exist:nnnTF</code>	★	<code>\keys_if_choice_exist:nnnTF {<module>} {<key>} {<choice>} {<true code>} {<false code>}</code>

New: 2011-08-21

Updated: 2017-11-14

Tests if the *<choice>* is defined for the *<key>* within the *<module>*, *i.e.* if any code has been defined for *<key>/<choice>*. The test is **false** if the *<key>* itself is not defined.

<code>\keys_show:nn</code>	<code>\keys_show:nn {<module>} {<key>}</code>
----------------------------	---

Updated: 2015-08-09

Displays in the terminal the information associated to the *<key>* for a *<module>*, including the function which is used to actually implement it.

<code>\keys_log:nn</code>	<code>\keys_log:nn {<module>} {<key>}</code>
---------------------------	--

New: 2014-08-22

Updated: 2015-08-09

Writes in the log file the information associated to the *<key>* for a *<module>*. See also `\keys_show:nn` which displays the result in the terminal.

26.9 Low-level interface for parsing key–val lists

To re-cap from earlier, a key–value list is input of the form

```
KeyOne = ValueOne ,
KeyTwo = ValueTwo ,
KeyThree
```

where each key–value pair is separated by a comma from the rest of the list, and each key–value pair does not necessarily contain an equals sign or a value! Processing this type of input correctly requires a number of careful steps, to correctly account for braces, spaces and the category codes of separators.

While the functions described earlier are used as a high-level interface for processing such input, in special circumstances you may wish to use a lower-level approach. The low-level parsing system converts a *<key–value list>* into *<keys>* and associated *<values>*. After the parsing phase is completed, the resulting keys and values (or keys alone) are available for further processing. This processing is not carried out by the low-level parser itself, and so the parser requires the names of two functions along with the key–value list. One function is needed to process key–value pairs (it receives two arguments), and a second function is required for keys given without any value (it is called with a single argument).

The parser does not double **#** tokens or expand any input. Active tokens **=** and **,** appearing at the outer level of braces are converted to category “other” (12) so that the

parser does not “miss” any due to category code changes. Spaces are removed from the ends of the keys and values. Keys and values which are given in braces have exactly one set removed (after space trimming), thus

```
key = {value here},
```

and

```
key = value here,
```

are treated identically.

`\keyval_parse:nnn` ☆

New: 2020-12-19
Updated: 2021-05-10

```
\keyval_parse:nnn {<code1>} {<code2>} {<key-value list>}
```

Parses the *<key-value list>* into a series of *<keys>* and associated *<values>*, or keys alone (if no *<value>* was given). *<code₁>* receives each *<key>* (with no *<value>*) as a trailing brace group, whereas *<code₂>* is appended by two brace groups, the *<key>* and *<value>*. The order of the *<keys>* in the *<key-value list>* is preserved. Thus

```
\keyval_parse:nnn
{ \use_none:nn { code 1 } }
{ \use_none:nnn { code 2 } }
{ key1 = value1 , key2 = value2, key3 = , key4 }
```

is converted into an input stream

```
\use_none:nnn { code 2 } { key1 } { value1 }
\use_none:nnn { code 2 } { key2 } { value2 }
\use_none:nnn { code 2 } { key3 } { }
\use_none:nn { code 1 } { key4 }
```

Note that there is a difference between an empty value (an equals sign followed by nothing) and a missing value (no equals sign at all). Spaces are trimmed from the ends of the *<key>* and *<value>*, then one *outer* set of braces is removed from the *<key>* and *<value>* as part of the processing. If you need exactly the output shown above, you’ll need to either **x**-type or **e**-type expand the function.

TeXhackers note: The result of each list element is returned within `\exp_not:n`, which means that the converted input stream does not expand further when appearing in an **x**-type or **e**-type argument expansion.

`\keyval_parse:NNn` ☆

Updated: 2021-05-10

`\keyval_parse:NNn` $\langle function_1 \rangle$ $\langle function_2 \rangle$ { $\langle key-value list \rangle$ }

Parses the $\langle key-value list \rangle$ into a series of $\langle keys \rangle$ and associated $\langle values \rangle$, or keys alone (if no $\langle value \rangle$ was given). $\langle function_1 \rangle$ should take one argument, while $\langle function_2 \rangle$ should absorb two arguments. After `\keyval_parse:NNn` has parsed the $\langle key-value list \rangle$, $\langle function_1 \rangle$ is used to process keys given with no value and $\langle function_2 \rangle$ is used to process keys given with a value. The order of the $\langle keys \rangle$ in the $\langle key-value list \rangle$ is preserved. Thus

```
\keyval_parse:NNn \function:n \function:nn
{ key1 = value1 , key2 = value2, key3 = , key4 }
```

is converted into an input stream

```
\function:nn { key1 } { value1 }
\function:nn { key2 } { value2 }
\function:nn { key3 } { }
\function:n { key4 }
```

Note that there is a difference between an empty value (an equals sign followed by nothing) and a missing value (no equals sign at all). Spaces are trimmed from the ends of the $\langle key \rangle$ and $\langle value \rangle$, then one *outer* set of braces is removed from the $\langle key \rangle$ and $\langle value \rangle$ as part of the processing.

This shares the implementation of `\keyval_parse:nnn`, the difference is only semantically.

TeXhackers note: The result is returned within `\exp_not:n`, which means that the converted input stream does not expand further when appearing in an **x**-type or **e**-type argument expansion.

Chapter 27

The l3intarray package: fast global integer arrays

27.1 l3intarray documentation

For applications requiring heavy use of integers, this module provides arrays which can be accessed in constant time (contrast l3seq, where access time is linear). These arrays have several important features

- The size of the array is fixed and must be given at point of initialisation
- The absolute value of each entry has maximum $2^{30} - 1$ (*i.e.* one power lower than the usual `\c_max_int` ceiling of $2^{31} - 1$)

The use of `intarray` data is therefore recommended for cases where the need for fast access is of paramount importance.

`\intarray_new:Nn`
`\intarray_new:cn`

New: 2018-03-29

`\intarray_new:Nn` $\langle \textit{intarray var} \rangle$ $\{\langle \textit{size} \rangle\}$

Evaluates the integer expression $\langle \textit{size} \rangle$ and allocates an $\langle \textit{integer array variable} \rangle$ with that number of (zero) entries. The variable name should start with `\g_` because assignments are always global.

`\intarray_count:N` ★
`\intarray_count:c` ★

New: 2018-03-29

`\intarray_count:N` $\langle \textit{intarray var} \rangle$

Expands to the number of entries in the $\langle \textit{integer array variable} \rangle$. Contrarily to `\seq_count:N` this is performed in constant time.

`\intarray_gset:Nnn`
`\intarray_gset:cnn`

New: 2018-03-29

`\intarray_gset:Nnn` $\langle \textit{intarray var} \rangle$ $\{\langle \textit{position} \rangle\}$ $\{\langle \textit{value} \rangle\}$

Stores the result of evaluating the integer expression $\langle \textit{value} \rangle$ into the $\langle \textit{integer array variable} \rangle$ at the (integer expression) $\langle \textit{position} \rangle$. If the $\langle \textit{position} \rangle$ is not between 1 and the `\intarray_count:N`, or the $\langle \textit{value} \rangle$'s absolute value is bigger than $2^{30} - 1$, an error occurs. Assignments are always global.

<code>\intarray_const_from_clist:Nn</code>	<code>\intarray_const_from_clist:Nn <intarray var> <intexpr clist></code>
<code>\intarray_const_from_clist:cn</code>	

New: 2018-05-04

Creates a new constant *<integer array variable>* or raises an error if the name is already taken. The *<integer array variable>* is set (globally) to contain as its items the results of evaluating each *<integer expression>* in the *<comma list>*.

<code>\intarray_gzero:N</code>	<code>\intarray_gzero:N <intarray var></code>
<code>\intarray_gzero:c</code>	

New: 2018-05-04

Sets all entries of the *<integer array variable>* to zero. Assignments are always global.

<code>\intarray_item:Nn *</code>	<code>\intarray_item:Nn <intarray var> {<position>}</code>
<code>\intarray_item:cn *</code>	

New: 2018-03-29

Expands to the integer entry stored at the (integer expression) *<position>* in the *<integer array variable>*. If the *<position>* is not between 1 and the `\intarray_count:N`, an error occurs.

<code>\intarray_rand_item:N *</code>	<code>\intarray_rand_item:N <intarray var></code>
<code>\intarray_rand_item:c *</code>	

New: 2018-05-05

Selects a pseudo-random item of the *<integer array>*. If the *<integer array>* is empty, produce an error.

<code>\intarray_show:N</code>	<code>\intarray_show:N <intarray var></code>
<code>\intarray_show:c</code>	<code>\intarray_log:N <intarray var></code>
<code>\intarray_log:N</code>	
<code>\intarray_log:c</code>	

New: 2018-05-04

Displays the items in the *<integer array variable>* in the terminal or writes them in the log file.

27.1.1 Implementation notes

It is a wrapper around the `\fontdimen` primitive, used to store arrays of integers (with a restricted range: absolute value at most $2^{30} - 1$). In contrast to `l3seq` sequences the access to individual entries is done in constant time rather than linear time, but only integers can be stored. More precisely, the primitive `\fontdimen` stores dimensions but the `l3intarray` package transparently converts these from/to integers. Assignments are always global.

While LuaTeX's memory is extensible, other engines can “only” deal with a bit less than 4×10^6 entries in all `\fontdimen` arrays combined (with default TeX Live settings).

Chapter 28

The **l3fp** package: Floating points

A decimal floating point number is one which is stored as a significand and a separate exponent. The module implements expandably a wide set of arithmetic, trigonometric, and other operations on decimal floating point numbers, to be used within floating point expressions. Floating point expressions support the following operations with their usual precedence.

- Basic arithmetic: addition $x + y$, subtraction $x - y$, multiplication $x * y$, division x / y , square root \sqrt{x} , and parentheses.
 - Comparison operators: $x < y$, $x \leq y$, $x > y$, $x \neq y$ etc.
 - Boolean logic: sign $\text{sign } x$, negation $!x$, conjunction $x \&\& y$, disjunction $x || y$, ternary operator $x ? y : z$.
 - Exponentials: $\exp x$, $\ln x$, x^y , $\log b x$.
 - Integer factorial: $\text{fact } x$.
 - Trigonometry: $\sin x$, $\cos x$, $\tan x$, $\cot x$, $\sec x$, $\csc x$ expecting their arguments in radians, and $\text{sind } x$, $\text{cosd } x$, $\text{tand } x$, $\text{cotd } x$, $\text{secd } x$, $\text{cscd } x$ expecting their arguments in degrees.
 - Inverse trigonometric functions: $\text{asin } x$, $\text{acos } x$, $\text{atan } x$, $\text{acot } x$, $\text{asec } x$, $\text{acsc } x$ giving a result in radians, and $\text{asind } x$, $\text{acosd } x$, $\text{atand } x$, $\text{acotd } x$, $\text{asecd } x$, $\text{acscd } x$ giving a result in degrees.
- (not yet) Hyperbolic functions and their inverse functions: $\sinh x$, $\cosh x$, $\tanh x$, $\coth x$, $\text{sech } x$, $\text{csch } x$, and $\text{asinh } x$, $\text{acosh } x$, $\text{atanh } x$, $\text{acoth } x$, $\text{asech } x$, $\text{acsch } x$.
- Extrema: $\max(x_1, x_2, \dots)$, $\min(x_1, x_2, \dots)$, $\text{abs}(x)$.
 - Rounding functions, controlled by two optional values, n (number of places, 0 by default) and t (behavior on a tie, **nan** by default):
 - $\text{trunc}(x, n)$ rounds towards zero,
 - $\text{floor}(x, n)$ rounds towards $-\infty$,

- `ceil(x, n)` rounds towards $+\infty$,
- `round(x, n, t)` rounds to the closest value, with ties rounded to an even value by default, towards zero if $t = 0$, towards $+\infty$ if $t > 0$ and towards $-\infty$ if $t < 0$.

And (*not yet*) modulo, and “quantize”.

- Random numbers: `rand()`, `randint(m, n)`.
- Constants: `pi`, `deg` (one degree in radians).
- Dimensions, automatically expressed in points, *e.g.*, `pc` is 12.
- Automatic conversion (no need for `\langle type \rangle_use:N`) of integer, dimension, and skip variables to floating point numbers, expressing dimensions in points and ignoring the stretch and shrink components of skips.
- Tuples: (x_1, \dots, x_n) that can be stored in variables, added together, multiplied or divided by a floating point number, and nested.

Floating point numbers can be given either explicitly (in a form such as `1.234e-34`, or `-.0001`), or as a stored floating point variable, which is automatically replaced by its current value. A “floating point” is a floating point number or a tuple thereof. See section 28.9.1 for a description of what a floating point is, section 28.9.2 for details about how an expression is parsed, and section 28.9.3 to know what the various operations do. Some operations may raise exceptions (error messages), described in section 28.7.

An example of use could be the following.

```
\LaTeX{} can now compute: $ \frac{\sin (3.5)}{2} + 2\cdot 10^{-3}
= \ExplSyntaxOn \fp_to_decimal:n {\sin(3.5)/2 + 2e-3} $.
```

The operation `round` can be used to limit the result’s precision. Adding `+0` avoids the possibly undesirable output `-0`, replacing it by `+0`. However, the `l3fp` module is mostly meant as an underlying tool for higher-level commands. For example, one could provide a function to typeset nicely the result of floating point computations.

```
\documentclass{article}
\usepackage{xparse, siunitx}
\ExplSyntaxOn
\NewDocumentCommand { \calcnun } { m }
{ \num { \fp_to_scientific:n {#1} } }
\ExplSyntaxOff
\begin{document}
\calcnun { 2 pi * sin ( 2.3 ^ 5 ) }
\end{document}
```

See the documentation of `siunitx` for various options of `\num`.

28.1 Creating and initialising floating point variables

<hr/> <code>\fp_new:N</code> <code>\fp_new:c</code> <hr/> <small>Updated: 2012-05-08</small>	<code>\fp_new:N <fp var></code> Creates a new <i><fp var></i> or raises an error if the name is already taken. The declaration is global. The <i><fp var></i> is initially +0.
<hr/> <code>\fp_const:Nn</code> <code>\fp_const:cn</code> <hr/> <small>Updated: 2012-05-08</small>	<code>\fp_const:Nn <fp var> {<floating point expression>}</code> Creates a new constant <i><fp var></i> or raises an error if the name is already taken. The <i><fp var></i> is set globally equal to the result of evaluating the <i><floating point expression></i> .
<hr/> <code>\fp_zero:N</code> <code>\fp_zero:c</code> <code>\fp_gzero:N</code> <code>\fp_gzero:c</code> <hr/> <small>Updated: 2012-05-08</small>	<code>\fp_zero:N <fp var></code> Sets the <i><fp var></i> to +0.
<hr/> <code>\fp_zero_new:N</code> <code>\fp_zero_new:c</code> <code>\fp_gzero_new:N</code> <code>\fp_gzero_new:c</code> <hr/> <small>Updated: 2012-05-08</small>	<code>\fp_zero_new:N <fp var></code> Ensures that the <i><fp var></i> exists globally by applying <code>\fp_new:N</code> if necessary, then applies <code>\fp_(g)zero:N</code> to leave the <i><fp var></i> set to +0.

28.2 Setting floating point variables

<hr/> <code>\fp_set:Nn</code> <code>\fp_set:cn</code> <code>\fp_gset:Nn</code> <code>\fp_gset:cn</code> <hr/> <small>Updated: 2012-05-08</small>	<code>\fp_set:Nn <fp var> {<floating point expression>}</code> Sets <i><fp var></i> equal to the result of computing the <i><floating point expression></i> .
<hr/> <code>\fp_set_eq:NN</code> <code>\fp_set_eq:(cN Nc cc)</code> <code>\fp_gset_eq:NN</code> <code>\fp_gset_eq:(cN Nc cc)</code> <hr/> <small>Updated: 2012-05-08</small>	<code>\fp_set_eq:NN <fp var_{12 Sets the floating point variable <i><fp var_{1 equal to the current value of <i><fp var_{2.}</i>}</i>}</code>
<hr/> <code>\fp_add:Nn</code> <code>\fp_add:cn</code> <code>\fp_gadd:Nn</code> <code>\fp_gadd:cn</code> <hr/> <small>Updated: 2012-05-08</small>	<code>\fp_add:Nn <fp var> {<floating point expression>}</code> Adds the result of computing the <i><floating point expression></i> to the <i><fp var></i> . This also applies if <i><fp var></i> and <i><floating point expression></i> evaluate to tuples of the same size.

<code>\fp_sub:Nn</code>	<code>\fp_sub:Nn <fp var> {<floating point expression>}</code>
<code>\fp_sub:cn</code>	
<code>\fp_gsub:Nn</code>	Subtracts the result of computing the <i><floating point expression></i> from the <i><fp var></i> . This
<code>\fp_gsub:cn</code>	also applies if <i><fp var></i> and <i><floating point expression></i> evaluate to tuples of the same size.

Updated: 2012-05-08

28.3 Using floating points

<code>\fp_eval:n</code> *	<code>\fp_eval:n {<floating point expression>}</code>
New: 2012-05-08	
Updated: 2012-07-08	

Evaluates the *<floating point expression>* and expresses the result as a decimal number with no exponent. Leading or trailing zeros may be inserted to compensate for the exponent. Non-significant trailing zeros are trimmed, and integers are expressed without a decimal separator. The values $\pm\infty$ and `nan` trigger an “invalid operation” exception. For a tuple, each item is converted using `\fp_eval:n` and they are combined as $(\langle fp_1 \rangle, \sqcup \langle fp_2 \rangle, \sqcup \dots \langle fp_n \rangle)$ if $n > 1$ and $(\langle fp_1 \rangle,)$ or $()$ for fewer items. This function is identical to `\fp_to_decimal:n`.

<code>\fp_sign:n</code> *	<code>\fp_sign:n {<fpexpr>}</code>
New: 2018-11-03	

Evaluates the *<fpexpr>* and leaves its sign in the input stream using `\fp_eval:n {sign(<result>)}`: $+1$ for positive numbers and for $+\infty$, -1 for negative numbers and for $-\infty$, ± 0 for ± 0 . If the operand is a tuple or is `nan`, then “invalid operation” occurs and the result is 0.

<code>\fp_to_decimal:N</code> *	<code>\fp_to_decimal:N <fp var></code>
<code>\fp_to_decimal:c</code> *	<code>\fp_to_decimal:n {<floating point expression>}</code>
<code>\fp_to_decimal:n</code> *	
New: 2012-05-08	
Updated: 2012-07-08	

Evaluates the *<floating point expression>* and expresses the result as a decimal number with no exponent. Leading or trailing zeros may be inserted to compensate for the exponent. Non-significant trailing zeros are trimmed, and integers are expressed without a decimal separator. The values $\pm\infty$ and `nan` trigger an “invalid operation” exception. For a tuple, each item is converted using `\fp_to_decimal:n` and they are combined as $(\langle fp_1 \rangle, \sqcup \langle fp_2 \rangle, \sqcup \dots \langle fp_n \rangle)$ if $n > 1$ and $(\langle fp_1 \rangle,)$ or $()$ for fewer items.

<code>\fp_to_dim:N</code> *	<code>\fp_to_dim:N <fp var></code>
<code>\fp_to_dim:c</code> *	<code>\fp_to_dim:n {<floating point expression>}</code>
<code>\fp_to_dim:n</code> *	
Updated: 2016-03-22	

Evaluates the *<floating point expression>* and expresses the result as a dimension (in `pt`) suitable for use in dimension expressions. The output is identical to `\fp_to_decimal:n`, with an additional trailing `pt` (both letter tokens). In particular, the result may be outside the range $[-2^{14} + 2^{-17}, 2^{14} - 2^{-17}]$ of valid `TeX` dimensions, leading to overflow errors if used as a dimension. Tuples, as well as the values $\pm\infty$ and `nan`, trigger an “invalid operation” exception.

<code>\fp_to_int:N</code> *	<code>\fp_to_int:N <fp var></code>
<code>\fp_to_int:c</code> *	<code>\fp_to_int:n {<floating point expression>}</code>
<code>\fp_to_int:n</code> *	
Updated: 2012-07-08	

Evaluates the *<floating point expression>*, and rounds the result to the closest integer, rounding exact ties to an even integer. The result may be outside the range $[-2^{31} + 1, 2^{31} - 1]$ of valid `TeX` integers, leading to overflow errors if used in an integer expression. Tuples, as well as the values $\pm\infty$ and `nan`, trigger an “invalid operation” exception.

<code>\fp_to_scientific:N</code> *	<code>\fp_to_scientific:N <fp var></code>
<code>\fp_to_scientific:c</code> *	<code>\fp_to_scientific:n {<floating point expression>}</code>
<code>\fp_to_scientific:n</code> *	Evaluates the <i><floating point expression></i> and expresses the result in scientific notation:
	$\langle optional - \rangle \langle digit \rangle . \langle 15 digits \rangle e \langle optional sign \rangle \langle exponent \rangle$

New: 2012-05-08
Updated: 2016-03-22

The leading *<digit>* is non-zero except in the case of ± 0 . The values $\pm\infty$ and `nan` trigger an “invalid operation” exception. Normal category codes apply: thus the `e` is category code 11 (a letter). For a tuple, each item is converted using `\fp_to_scientific:n` and they are combined as $(\langle fp_1 \rangle, \sqcup \langle fp_2 \rangle, \sqcup \dots \langle fp_n \rangle)$ if $n > 1$ and $(\langle fp_1 \rangle,)$ or $()$ for fewer items.

<code>\fp_to_tl:N</code> *	<code>\fp_to_tl:N <fp var></code>
<code>\fp_to_tl:c</code> *	<code>\fp_to_tl:n {<floating point expression>}</code>
<code>\fp_to_tl:n</code> *	Evaluates the <i><floating point expression></i> and expresses the result in (almost) the shortest possible form. Numbers in the ranges $(0, 10^{-3})$ and $[10^{16}, \infty)$ are expressed in scientific notation with trailing zeros trimmed and no decimal separator when there is a single significant digit (this differs from <code>\fp_to_scientific:n</code>). Numbers in the range $[10^{-3}, 10^{16})$ are expressed in a decimal notation without exponent, with trailing zeros trimmed, and no decimal separator for integer values (see <code>\fp_to_decimal:n</code>). Negative numbers start with <code>-</code> . The special values ± 0 , $\pm\infty$ and <code>nan</code> are rendered as <code>0</code> , <code>-0</code> , <code>inf</code> , <code>-inf</code> , and <code>nan</code> respectively. Normal category codes apply and thus <code>inf</code> or <code>nan</code> , if produced, are made up of letters. For a tuple, each item is converted using <code>\fp_to_tl:n</code> and they are combined as $(\langle fp_1 \rangle, \sqcup \langle fp_2 \rangle, \sqcup \dots \langle fp_n \rangle)$ if $n > 1$ and $(\langle fp_1 \rangle,)$ or $()$ for fewer items.

Updated: 2016-03-22

<code>\fp_use:N</code> *	<code>\fp_use:N <fp var></code>
<code>\fp_use:c</code> *	Inserts the value of the <i><fp var></i> into the input stream as a decimal number with no exponent. Leading or trailing zeros may be inserted to compensate for the exponent. Non-significant trailing zeros are trimmed. Integers are expressed without a decimal separator. The values $\pm\infty$ and <code>nan</code> trigger an “invalid operation” exception. For a tuple, each item is converted using <code>\fp_to_decimal:n</code> and they are combined as $(\langle fp_1 \rangle, \sqcup \langle fp_2 \rangle, \sqcup \dots \langle fp_n \rangle)$ if $n > 1$ and $(\langle fp_1 \rangle,)$ or $()$ for fewer items. This function is identical to <code>\fp_to_decimal:N</code> .

Updated: 2012-07-08

28.4 Floating point conditionals

<code>\fp_if_exist_p:N</code> *	<code>\fp_if_exist_p:N <fp var></code>
<code>\fp_if_exist_p:c</code> *	<code>\fp_if_exist:NTF <fp var> {<true code>} {<false code>}</code>
<code>\fp_if_exist:NTF</code> *	Tests whether the <i><fp var></i> is currently defined. This does not check that the <i><fp var></i> really is a floating point variable.
<code>\fp_if_exist:cTF</code> *	

Updated: 2012-05-08

<code>\fp_compare_p:nNn</code> \star <code>\fp_compare:nNnTF</code> \star	<code>\fp_compare_p:nNn {<fpexpr₁>} <relation> {<fpexpr₂>}</code> <code>\fp_compare:nNnTF {<fpexpr₁>} <relation> {<fpexpr₂>} {<true code>} {<false code>}</code>
--	---

Updated: 2012-05-08

Compares the $\langle fpexpr_1 \rangle$ and the $\langle fpexpr_2 \rangle$, and returns `true` if the $\langle relation \rangle$ is obeyed. Two floating points x and y may obey four mutually exclusive relations: $x < y$, $x = y$, $x > y$, or $x?y$ (“not ordered”). The last case occurs exactly if one or both operands is `nan` or is a tuple, unless they are equal tuples. Note that a `nan` is distinct from any value, even another `nan`, hence $x = x$ is not true for a `nan`. To test if a value is `nan`, compare it to an arbitrary number with the “not ordered” relation.

```

\fp_compare:nNnTF { <value> } ? { 0 }
{ } % <value> is nan
{ } % <value> is not nan

```

Tuples are equal if they have the same number of items and items compare equal (in particular there must be no `nan`). At present any other comparison with tuples yields ? (not ordered). This is experimental.

This function is less flexible than `\fp_compare:nTF` but slightly faster. It is provided for consistency with `\int_compare:nNnTF` and `\dim_compare:nNnTF`.

<code>\fp_compare_p:n</code> ☆	<code>\fp_compare_p:n</code>
<code>\fp_compare:nTF</code> ☆	{
Updated: 2013-12-14	$\langle fpexpr_1 \rangle$ $\langle relation_1 \rangle$
	...
	$\langle fpexpr_N \rangle$ $\langle relation_N \rangle$
	$\langle fpexpr_{N+1} \rangle$
	}
	<code>\fp_compare:nTF</code>
	{
	$\langle fpexpr_1 \rangle$ $\langle relation_1 \rangle$
	...
	$\langle fpexpr_N \rangle$ $\langle relation_N \rangle$
	$\langle fpexpr_{N+1} \rangle$
	}
	{ $\langle true\ code \rangle$ } { $\langle false\ code \rangle$ }

Evaluates the $\langle floating\ point\ expressions \rangle$ as described for `\fp_eval:n` and compares consecutive result using the corresponding $\langle relation \rangle$, namely it compares $\langle fpexpr_1 \rangle$ and $\langle fpexpr_2 \rangle$ using the $\langle relation_1 \rangle$, then $\langle fpexpr_2 \rangle$ and $\langle fpexpr_3 \rangle$ using the $\langle relation_2 \rangle$, until finally comparing $\langle fpexpr_N \rangle$ and $\langle fpexpr_{N+1} \rangle$ using the $\langle relation_N \rangle$. The test yields **true** if all comparisons are **true**. Each $\langle floating\ point\ expression \rangle$ is evaluated only once. Contrarily to `\int_compare:nTF`, all $\langle floating\ point\ expressions \rangle$ are computed, even if one comparison is **false**. Two floating points x and y may obey four mutually exclusive relations: $x < y$, $x = y$, $x > y$, or $x?y$ (“not ordered”). The last case occurs exactly if one or both operands is **nan** or is a tuple, unless they are equal tuples. Each $\langle relation \rangle$ can be any (non-empty) combination of $<$, $=$, $>$, and $?$, plus an optional leading $!$ (which negates the $\langle relation \rangle$), with the restriction that the $\langle relation \rangle$ may not start with $?$, as this symbol has a different meaning (in combination with $:$) within floating point expressions. The comparison $x \langle relation \rangle y$ is then **true** if the $\langle relation \rangle$ does not start with $!$ and the actual relation ($<$, $=$, $>$, or $?$) between x and y appears within the $\langle relation \rangle$, or on the contrary if the $\langle relation \rangle$ starts with $!$ and the relation between x and y does not appear within the $\langle relation \rangle$. Common choices of $\langle relation \rangle$ include \geq (greater or equal), \neq (not equal), $!?$ or \leq (comparable).

This function is more flexible than `\fp_compare:nNnTF` and only slightly slower.

28.5 Floating point expression loops

<code>\fp_do_until:nNnn</code> ☆	<code>\fp_do_until:nNnn {$\langle fpexpr_1 \rangle$} $\langle relation \rangle$ {$\langle fpexpr_2 \rangle$} {$\langle code \rangle$}</code>
New: 2012-08-16	Places the $\langle code \rangle$ in the input stream for \TeX to process, and then evaluates the relationship between the two $\langle floating\ point\ expressions \rangle$ as described for <code>\fp_compare:nNnTF</code> . If the test is false then the $\langle code \rangle$ is inserted into the input stream again and a loop occurs until the $\langle relation \rangle$ is true .
<code>\fp_do_while:nNnn</code> ☆	<code>\fp_do_while:nNnn {$\langle fpexpr_1 \rangle$} $\langle relation \rangle$ {$\langle fpexpr_2 \rangle$} {$\langle code \rangle$}</code>
New: 2012-08-16	Places the $\langle code \rangle$ in the input stream for \TeX to process, and then evaluates the relationship between the two $\langle floating\ point\ expressions \rangle$ as described for <code>\fp_compare:nNnTF</code> . If the test is true then the $\langle code \rangle$ is inserted into the input stream again and a loop occurs until the $\langle relation \rangle$ is false .

<hr/> <code>\fp_until_do:nNnn</code> ☆ <hr/>	<code>\fp_until_do:nNnn {<fpexpr1>} <relation> {<fpexpr2>} {<code>}</code>
New: 2012-08-16	Evaluates the relationship between the two <i><floating point expressions></i> as described for <code>\fp_compare:nNnTF</code> , and then places the <i><code></i> in the input stream if the <i><relation></i> is false. After the <i><code></i> has been processed by T _E X the test is repeated, and a loop occurs until the test is true .
<hr/> <code>\fp_while_do:nNnn</code> ☆ <hr/>	<code>\fp_while_do:nNnn {<fpexpr1>} <relation> {<fpexpr2>} {<code>}</code>
New: 2012-08-16	Evaluates the relationship between the two <i><floating point expressions></i> as described for <code>\fp_compare:nNnTF</code> , and then places the <i><code></i> in the input stream if the <i><relation></i> is true. After the <i><code></i> has been processed by T _E X the test is repeated, and a loop occurs until the test is false .
<hr/> <code>\fp_do_until:nn</code> ☆ <hr/>	<code>\fp_do_until:nn { <fpexpr1> <relation> <fpexpr2> } {<code>}</code>
New: 2012-08-16 Updated: 2013-12-14	Places the <i><code></i> in the input stream for T _E X to process, and then evaluates the relationship between the two <i><floating point expressions></i> as described for <code>\fp_compare:nTF</code> . If the test is false then the <i><code></i> is inserted into the input stream again and a loop occurs until the <i><relation></i> is true .
<hr/> <code>\fp_do_while:nn</code> ☆ <hr/>	<code>\fp_do_while:nn { <fpexpr1> <relation> <fpexpr2> } {<code>}</code>
New: 2012-08-16 Updated: 2013-12-14	Places the <i><code></i> in the input stream for T _E X to process, and then evaluates the relationship between the two <i><floating point expressions></i> as described for <code>\fp_compare:nTF</code> . If the test is true then the <i><code></i> is inserted into the input stream again and a loop occurs until the <i><relation></i> is false .
<hr/> <code>\fp_until_do:nn</code> ☆ <hr/>	<code>\fp_until_do:nn { <fpexpr1> <relation> <fpexpr2> } {<code>}</code>
New: 2012-08-16 Updated: 2013-12-14	Evaluates the relationship between the two <i><floating point expressions></i> as described for <code>\fp_compare:nTF</code> , and then places the <i><code></i> in the input stream if the <i><relation></i> is false. After the <i><code></i> has been processed by T _E X the test is repeated, and a loop occurs until the test is true .
<hr/> <code>\fp_while_do:nn</code> ☆ <hr/>	<code>\fp_while_do:nn { <fpexpr1> <relation> <fpexpr2> } {<code>}</code>
New: 2012-08-16 Updated: 2013-12-14	Evaluates the relationship between the two <i><floating point expressions></i> as described for <code>\fp_compare:nTF</code> , and then places the <i><code></i> in the input stream if the <i><relation></i> is true. After the <i><code></i> has been processed by T _E X the test is repeated, and a loop occurs until the test is false .

`\fp_step_function:nnnN` ☆
`\fp_step_function:nnnc` ☆

New: 2016-11-21
Updated: 2016-12-06

`\fp_step_function:nnnN` { $\langle initial\ value \rangle$ } { $\langle step \rangle$ } { $\langle final\ value \rangle$ } $\langle function \rangle$

This function first evaluates the $\langle initial\ value \rangle$, $\langle step \rangle$ and $\langle final\ value \rangle$, each of which should be a floating point expression evaluating to a floating point number, not a tuple. The $\langle function \rangle$ is then placed in front of each $\langle value \rangle$ from the $\langle initial\ value \rangle$ to the $\langle final\ value \rangle$ in turn (using $\langle step \rangle$ between each $\langle value \rangle$). The $\langle step \rangle$ must be non-zero. If the $\langle step \rangle$ is positive, the loop stops when the $\langle value \rangle$ becomes larger than the $\langle final\ value \rangle$. If the $\langle step \rangle$ is negative, the loop stops when the $\langle value \rangle$ becomes smaller than the $\langle final\ value \rangle$. The $\langle function \rangle$ should absorb one numerical argument. For example

```
\cs_set:Npn \my_func:n #1 { [I~saw~#1] \quad }
\fp_step_function:nnnN { 1.0 } { 0.1 } { 1.5 } \my_func:n
```

would print

```
[I saw 1.0]   [I saw 1.1]   [I saw 1.2]   [I saw 1.3]   [I saw 1.4]   [I saw 1.5]
```

TpXhackers note: Due to rounding, it may happen that adding the $\langle step \rangle$ to the $\langle value \rangle$ does not change the $\langle value \rangle$; such cases give an error, as they would otherwise lead to an infinite loop.

`\fp_step_inline:nnnn`

New: 2016-11-21
Updated: 2016-12-06

`\fp_step_inline:nnnn` { $\langle initial\ value \rangle$ } { $\langle step \rangle$ } { $\langle final\ value \rangle$ } { $\langle code \rangle$ }

This function first evaluates the $\langle initial\ value \rangle$, $\langle step \rangle$ and $\langle final\ value \rangle$, all of which should be floating point expressions evaluating to a floating point number, not a tuple. Then for each $\langle value \rangle$ from the $\langle initial\ value \rangle$ to the $\langle final\ value \rangle$ in turn (using $\langle step \rangle$ between each $\langle value \rangle$), the $\langle code \rangle$ is inserted into the input stream with `#1` replaced by the current $\langle value \rangle$. Thus the $\langle code \rangle$ should define a function of one argument (`#1`).

`\fp_step_variable:nnnNn`

New: 2017-04-12

`\fp_step_variable:nnnNn`
{ $\langle initial\ value \rangle$ } { $\langle step \rangle$ } { $\langle final\ value \rangle$ } $\langle tl\ var \rangle$ { $\langle code \rangle$ }

This function first evaluates the $\langle initial\ value \rangle$, $\langle step \rangle$ and $\langle final\ value \rangle$, all of which should be floating point expressions evaluating to a floating point number, not a tuple. Then for each $\langle value \rangle$ from the $\langle initial\ value \rangle$ to the $\langle final\ value \rangle$ in turn (using $\langle step \rangle$ between each $\langle value \rangle$), the $\langle code \rangle$ is inserted into the input stream, with the $\langle tl\ var \rangle$ defined as the current $\langle value \rangle$. Thus the $\langle code \rangle$ should make use of the $\langle tl\ var \rangle$.

28.6 Some useful constants, and scratch variables

`\c_zero_fp`
`\c_minus_zero_fp`

New: 2012-05-08

Zero, with either sign.

`\c_one_fp`

New: 2012-05-08

One as an fp: useful for comparisons in some places.

<hr/> <code>\c_inf_fp</code> <code>\c_minus_inf_fp</code> <hr/> New: 2012-05-08 <hr/>	<p>Infinity, with either sign. These can be input directly in a floating point expression as <code>inf</code> and <code>-inf</code>.</p>
<hr/> <code>\c_e_fp</code> <hr/> Updated: 2012-05-08 <hr/>	<p>The value of the base of the natural logarithm, $e = \exp(1)$.</p>
<hr/> <code>\c_pi_fp</code> <hr/> Updated: 2013-11-17 <hr/>	<p>The value of π. This can be input directly in a floating point expression as <code>pi</code>.</p>
<hr/> <code>\c_one_degree_fp</code> <hr/> New: 2012-05-08 Updated: 2013-11-17 <hr/>	<p>The value of 1° in radians. Multiply an angle given in degrees by this value to obtain a result in radians. Note that trigonometric functions expecting an argument in radians or in degrees are both available. Within floating point expressions, this can be accessed as <code>deg</code>.</p>
<hr/> <code>\l_tmpa_fp</code> <code>\l_tmpb_fp</code> <hr/>	<p>Scratch floating points for local assignment. These are never used by the kernel code, and so are safe for use with any L^AT_EX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.</p>
<hr/> <code>\g_tmpa_fp</code> <code>\g_tmpb_fp</code> <hr/>	<p>Scratch floating points for global assignment. These are never used by the kernel code, and so are safe for use with any L^AT_EX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.</p>

28.7 Floating point exceptions

The functions defined in this section are experimental, and their functionality may be altered or removed altogether.

“Exceptions” may occur when performing some floating point operations, such as $0 / 0$, or $10 ** 1e9999$. The relevant IEEE standard defines 5 types of exceptions, of which we implement 4.

- *Overflow* occurs whenever the result of an operation is too large to be represented as a normal floating point number. This results in $\pm\infty$.
- *Underflow* occurs whenever the result of an operation is too close to 0 to be represented as a normal floating point number. This results in ± 0 .
- *Invalid operation* occurs for operations with no defined outcome, for instance $0/0$ or $\sin(\infty)$, and results in a `nan`. It also occurs for conversion functions whose target type does not have the appropriate infinite or `nan` value (e.g., `\fp_to_dim:n`).
- *Division by zero* occurs when dividing a non-zero number by 0, or when evaluating functions at poles, e.g., $\ln(0)$ or $\cot(0)$. This results in $\pm\infty$.

(*not yet*) *Inexact* occurs whenever the result of a computation is not exact, in other words, almost always. At the moment, this exception is entirely ignored in L^AT_EX3.

To each exception we associate a “flag”: `fp_overflow`, `fp_underflow`, `fp_invalid_operation` and `fp_division_by_zero`. The state of these flags can be tested and modified with commands from `l3flag`

By default, the “invalid operation” exception triggers an (expandable) error, and raises the corresponding flag. Other exceptions raise the corresponding flag but do not trigger an error. The behaviour when an exception occurs can be modified (using `\fp_trap:nn`) to either produce an error and raise the flag, or only raise the flag, or do nothing at all.

<code>\fp_trap:nn</code>	<code>\fp_trap:nn {<exception>} {<trap type>}</code>
New: 2012-07-19 Updated: 2017-02-13	All occurrences of the <code><exception></code> (<code>overflow</code> , <code>underflow</code> , <code>invalid_operation</code> or <code>division_by_zero</code>) within the current group are treated as <code><trap type></code> , which can be <ul style="list-style-type: none"> • none: the <code><exception></code> will be entirely ignored, and leave no trace; • flag: the <code><exception></code> will turn the corresponding flag on when it occurs; • error: additionally, the <code><exception></code> will halt the T_EX run and display some information about the current operation in the terminal.

This function is experimental, and may be altered or removed.

`flag_fp_overflow`
`flag_fp_underflow`
`flag_fp_invalid_operation`
`flag_fp_division_by_zero`

Flags denoting the occurrence of various floating-point exceptions.

28.8 Viewing floating points

<code>\fp_show:N</code> <code>\fp_show:c</code> <code>\fp_show:n</code>	<code>\fp_show:N <fp var></code> <code>\fp_show:n {<floating point expression>}</code> Evaluates the <code><floating point expression></code> and displays the result in the terminal.
New: 2012-05-08 Updated: 2021-04-29	
<code>\fp_log:N</code> <code>\fp_log:c</code> <code>\fp_log:n</code>	<code>\fp_log:N <fp var></code> <code>\fp_log:n {<floating point expression>}</code> Evaluates the <code><floating point expression></code> and writes the result in the log file.
New: 2014-08-22 Updated: 2021-04-29	

28.9 Floating point expressions

28.9.1 Input of floating point numbers

We support four types of floating point numbers:

- $\pm m \cdot 10^n$, a floating point number, with integer $1 \leq m \leq 10^{16}$, and $-10000 \leq n \leq 10000$;
- ± 0 , zero, with a given sign;
- $\pm \infty$, infinity, with a given sign;
- **nan**, is “not a number”, and can be either quiet or signalling (*not yet*: this distinction is currently unsupported);

Normal floating point numbers are stored in base 10, with up to 16 significant figures.

On input, a normal floating point number consists of:

- $\langle sign \rangle$: a possibly empty string of + and - characters;
- $\langle significand \rangle$: a non-empty string of digits together with zero or one dot;
- $\langle exponent \rangle$ optionally: the character **e** or **E**, followed by a possibly empty string of + and - tokens, and a non-empty string of digits.

The sign of the resulting number is + if $\langle sign \rangle$ contains an even number of -, and - otherwise, hence, an empty $\langle sign \rangle$ denotes a non-negative input. The stored significand is obtained from $\langle significand \rangle$ by omitting the decimal separator and leading zeros, and rounding to 16 significant digits, filling with trailing zeros if necessary. In particular, the value stored is exact if the input $\langle significand \rangle$ has at most 16 digits. The stored $\langle exponent \rangle$ is obtained by combining the input $\langle exponent \rangle$ (0 if absent) with a shift depending on the position of the significand and the number of leading zeros.

A special case arises if the resulting $\langle exponent \rangle$ is either too large or too small for the floating point number to be represented. This results either in an overflow (the number is then replaced by $\pm \infty$), or an underflow (resulting in ± 0).

The result is thus ± 0 if and only if $\langle significand \rangle$ contains no non-zero digit (*i.e.*, consists only in characters 0, and an optional period), or if there is an underflow. Note that a single dot is currently a valid floating point number, equal to +0, but that is not guaranteed to remain true.

The $\langle significand \rangle$ must be non-empty, so **e1** and **e-1** are not valid floating point numbers. Note that the latter could be mistaken with the difference of “**e**” and 1. To avoid confusions, the base of natural logarithms cannot be input as **e** and should be input as **exp(1)** or **\c_e_fp** (which is faster).

Special numbers are input as follows:

- **inf** represents $+\infty$, and can be preceded by any $\langle sign \rangle$, yielding $\pm \infty$ as appropriate.
- **nan** represents a (quiet) non-number. It can be preceded by any sign, but that sign is ignored.
- Any unrecognizable string triggers an error, and produces a **nan**.
- Note that commands such as **\infy**, **\pi**, or **\sin** *do not* work in floating point expressions. They may silently be interpreted as completely unexpected numbers, because integer constants (allowed in expressions) are commonly stored as mathematical characters.

28.9.2 Precedence of operators

We list here all the operations supported in floating point expressions, in order of decreasing precedence: operations listed earlier bind more tightly than operations listed below them.

- Function calls (`sin`, `ln`, *etc.*).
- Binary `**` and `^` (right associative).
- Unary `+`, `-`, `!`.
- Implicit multiplication by juxtaposition (`2pi`) when neither factor is in parentheses.
- Binary `*` and `/`, implicit multiplication by juxtaposition with parentheses (for instance `3(4+5)`).
- Binary `+` and `-`.
- Comparisons `>=`, `!=`, `<?`, *etc.*
- Logical `and`, denoted by `&&`.
- Logical `or`, denoted by `||`.
- Ternary operator `?:` (right associative).
- Comma (to build tuples).

The precedence of operations can be overridden using parentheses. In particular, the precedence of juxtaposition implies that

$$\begin{aligned}1/2\text{pi} &= 1/(2\pi), \\1/2\text{pi}(\text{pi} + \text{pi}) &= (2\pi)^{-1}(\pi + \pi) \simeq 1, \\ \sin 2\text{pi} &= \sin(2)\pi \neq 0, \\ 2^{\text{2max}(3,5)} &= 2^{\max(3,5)} = 20, \\ 1\text{in}/1\text{cm} &= (1\text{in})/(1\text{cm}) = 2.54.\end{aligned}$$

Functions are called on the value of their argument, contrarily to `TeX` macros.

28.9.3 Operations

We now present the various operations allowed in floating point expressions, from the lowest precedence to the highest. When used as a truth value, a floating point expression is `false` if it is ± 0 , and `true` otherwise, including when it is `nan` or a tuple such as `(0, 0)`. Tuples are only supported to some extent by operations that work with truth values (`?:`, `||`, `&&`, `!`), by comparisons (`!<=>?`), and by `+`, `-`, `*`, `/`. Unless otherwise specified, providing a tuple as an argument of any other operation yields the “invalid operation” exception and a `nan` result.

```
?: \fp_eval:n { <operand1> ? <operand2> : <operand3> }
```

The ternary operator `?:` results in $\langle operand_2 \rangle$ if $\langle operand_1 \rangle$ is true (not ± 0), and $\langle operand_3 \rangle$ if $\langle operand_1 \rangle$ is false (± 0). All three $\langle operands \rangle$ are evaluated in all cases; they may be tuples. The operator is right associative, hence

```
\fp_eval:n
{
  1 + 3 > 4 ? 1 :
  2 + 4 > 5 ? 2 :
  3 + 5 > 6 ? 3 : 4
}
```

first tests whether $1 + 3 > 4$; since this isn't true, the branch following `:` is taken, and $2 + 4 > 5$ is compared; since this is true, the branch before `:` is taken, and everything else is (evaluated then) ignored. That allows testing for various cases in a concise manner, with the drawback that all computations are made in all cases.

```
|| \fp_eval:n { <operand1> || <operand2> }
```

If $\langle operand_1 \rangle$ is true (not ± 0), use that value, otherwise the value of $\langle operand_2 \rangle$. Both $\langle operands \rangle$ are evaluated in all cases; they may be tuples. In $\langle operand_1 \rangle || \langle operand_2 \rangle || \dots || \langle operands_n \rangle$, the first true (nonzero) $\langle operand \rangle$ is used and if all are zero the last one (± 0) is used.

```
&& \fp_eval:n { <operand1> && <operand2> }
```

If $\langle operand_1 \rangle$ is false (equal to ± 0), use that value, otherwise the value of $\langle operand_2 \rangle$. Both $\langle operands \rangle$ are evaluated in all cases; they may be tuples. In $\langle operand_1 \rangle \&\& \langle operand_2 \rangle \&\& \dots \&\& \langle operands_n \rangle$, the first false (± 0) $\langle operand \rangle$ is used and if none is zero the last one is used.

```
< \fp_eval:n
= {
>   <operand1> <relation1>
?   ...
    <operand_N> <relation_N>
    <operand_{N+1}>
}
```

Updated: 2013-12-14

Each $\langle relation \rangle$ consists of a non-empty string of `<`, `=`, `>`, and `?`, optionally preceded by `!`, and may not start with `?`. This evaluates to $+1$ if all comparisons $\langle operand_i \rangle \langle relation_i \rangle \langle operand_{i+1} \rangle$ are true, and $+0$ otherwise. All $\langle operands \rangle$ are evaluated (once) in all cases. See `\fp_compare:nTF` for details.

```
+ \fp_eval:n { <operand1> + <operand2> }
- \fp_eval:n { <operand1> - <operand2> }
```

Computes the sum or the difference of its two $\langle operands \rangle$. The “invalid operation” exception occurs for $\infty - \infty$. “Underflow” and “overflow” occur when appropriate. These operations supports the itemwise addition or subtraction of two tuples, but if they have a different number of items the “invalid operation” exception occurs and the result is **nan**.

```

* \fp_eval:n { <operand1> * <operand2> }
/ \fp_eval:n { <operand1> / <operand2> }

```

Computes the product or the ratio of its two $\langle operand \rangle$. The “invalid operation” exception occurs for ∞/∞ , $0/0$, or $0 * \infty$. “Division by zero” occurs when dividing a finite non-zero number by ± 0 . “Underflow” and “overflow” occur when appropriate. When $\langle operand_1 \rangle$ is a tuple and $\langle operand_2 \rangle$ is a floating point number, each item of $\langle operand_1 \rangle$ is multiplied or divided by $\langle operand_2 \rangle$. Multiplication also supports the case where $\langle operand_1 \rangle$ is a floating point number and $\langle operand_2 \rangle$ a tuple. Other combinations yield an “invalid operation” exception and a **nan** result.

```

+ \fp_eval:n { + <operand> }
- \fp_eval:n { - <operand> }
! \fp_eval:n { ! <operand> }

```

The unary $+$ does nothing, the unary $-$ changes the sign of the $\langle operand \rangle$ (for a tuple, of all its components), and $!$ $\langle operand \rangle$ evaluates to 1 if $\langle operand \rangle$ is false (is ± 0) and 0 otherwise (this is the **not** boolean function). Those operations never raise exceptions.

```

** \fp_eval:n { <operand1> ** <operand2> }
^ \fp_eval:n { <operand1> ^ <operand2> }

```

Raises $\langle operand_1 \rangle$ to the power $\langle operand_2 \rangle$. This operation is right associative, hence $2^{**} 2^{**} 3$ equals $2^{2^3} = 256$. If $\langle operand_1 \rangle$ is negative or -0 then: the result’s sign is $+$ if the $\langle operand_2 \rangle$ is infinite and $(-1)^p$ if the $\langle operand_2 \rangle$ is $p/5^q$ with p, q integers; the result is $+0$ if $\text{abs}(\langle operand_1 \rangle)^{\langle operand_2 \rangle}$ evaluates to zero; in other cases the “invalid operation” exception occurs because the sign cannot be determined. “Division by zero” occurs when raising ± 0 to a finite strictly negative power. “Underflow” and “overflow” occur when appropriate. If either operand is a tuple, “invalid operation” occurs.

```

abs \fp_eval:n { abs( <fpexpr> ) }

```

Computes the absolute value of the $\langle fpexpr \rangle$. If the operand is a tuple, “invalid operation” occurs. This operation does not raise exceptions in other cases. See also `\fp_abs:n`.

```

exp \fp_eval:n { exp( <fpexpr> ) }

```

Computes the exponential of the $\langle fpexpr \rangle$. “Underflow” and “overflow” occur when appropriate. If the operand is a tuple, “invalid operation” occurs.

```

fact \fp_eval:n { fact( <fpexpr> ) }

```

Computes the factorial of the $\langle fpexpr \rangle$. If the $\langle fpexpr \rangle$ is an integer between -0 and 3248 included, the result is finite and correctly rounded. Larger positive integers give $+\infty$ with “overflow”, while $\text{fact}(+\infty) = +\infty$ and $\text{fact}(\text{nan}) = \text{nan}$ with no exception. All other inputs give **nan** with the “invalid operation” exception.

```

ln \fp_eval:n { ln( <fpexpr> ) }

```

Computes the natural logarithm of the $\langle fpexpr \rangle$. Negative numbers have no (real) logarithm, hence the “invalid operation” is raised in that case, including for $\ln(-0)$. “Division by zero” occurs when evaluating $\ln(+0) = -\infty$. “Underflow” and “overflow” occur when appropriate. If the operand is a tuple, “invalid operation” occurs.

<hr/> logb <hr/>	★	<code>\fp_eval:n { logb(<fpexpr>) }</code>	
<hr/> New: 2018-11-03 <hr/>			Determines the exponent of the $\langle fpexpr \rangle$, namely the floor of the base-10 logarithm of its absolute value. “Division by zero” occurs when evaluating $\logb(\pm 0) = -\infty$. Other special values are $\logb(\pm\infty) = +\infty$ and $\logb(\mathbf{nan}) = \mathbf{nan}$. If the operand is a tuple or is \mathbf{nan} , then “invalid operation” occurs and the result is \mathbf{nan} .
<hr/> max <hr/>		<code>\fp_eval:n { max(<fpexpr1> , <fpexpr2> , ...) }</code>	
<hr/> min <hr/>		<code>\fp_eval:n { min(<fpexpr1> , <fpexpr2> , ...) }</code>	
			Evaluates each $\langle fpexpr \rangle$ and computes the largest (smallest) of those. If any of the $\langle fpexpr \rangle$ is a \mathbf{nan} or tuple, the result is \mathbf{nan} . If any operand is a tuple, “invalid operation” occurs; these operations do not raise exceptions in other cases.
<hr/> round <hr/>		<code>\fp_eval:n { round (<fpexpr>) }</code>	
trunc		<code>\fp_eval:n { round (<fpexpr1> , <fpexpr2>) }</code>	
ceil		<code>\fp_eval:n { round (<fpexpr1> , <fpexpr2> , <fpexpr3>) }</code>	
floor			
<hr/> New: 2013-12-14 <hr/>			Only round accepts a third argument. Evaluates $\langle fpexpr_1 \rangle = x$ and $\langle fpexpr_2 \rangle = n$ and $\langle fpexpr_3 \rangle = t$ then rounds x to n places. If n is an integer, this rounds x to a multiple of 10^{-n} ; if $n = +\infty$, this always yields x ; if $n = -\infty$, this yields one of ± 0 , $\pm\infty$, or \mathbf{nan} ; if $n = \mathbf{nan}$, this yields \mathbf{nan} ; if n is neither $\pm\infty$ nor an integer, then an “invalid operation” exception is raised. When $\langle fpexpr_2 \rangle$ is omitted, $n = 0$, <i>i.e.</i> , $\langle fpexpr_1 \rangle$ is rounded to an integer. The rounding direction depends on the function.
<hr/> Updated: 2015-08-08 <hr/>			
			<ul style="list-style-type: none"> • round yields the multiple of 10^{-n} closest to x, with ties (x half-way between two such multiples) rounded as follows. If t is \mathbf{nan} (or not given) the even multiple is chosen (“ties to even”), if $t = \pm 0$ the multiple closest to 0 is chosen (“ties to zero”), if t is positive/negative the multiple closest to $\infty/-\infty$ is chosen (“ties towards positive/negative infinity”). • floor yields the largest multiple of 10^{-n} smaller or equal to x (“round towards negative infinity”); • ceil yields the smallest multiple of 10^{-n} greater or equal to x (“round towards positive infinity”); • trunc yields a multiple of 10^{-n} with the same sign as x and with the largest absolute value less than that of x (“round towards zero”).
			“Overflow” occurs if x is finite and the result is infinite (this can only happen if $\langle fpexpr_2 \rangle < -9984$). If any operand is a tuple, “invalid operation” occurs.
<hr/> sign <hr/>		<code>\fp_eval:n { sign(<fpexpr>) }</code>	
			Evaluates the $\langle fpexpr \rangle$ and determines its sign: $+1$ for positive numbers and for $+\infty$, -1 for negative numbers and for $-\infty$, ± 0 for ± 0 , and \mathbf{nan} for \mathbf{nan} . If the operand is a tuple, “invalid operation” occurs. This operation does not raise exceptions in other cases.

<code>sin</code>	<code>\fp_eval:n { sin(<fpexpr>) }</code>
<code>cos</code>	<code>\fp_eval:n { cos(<fpexpr>) }</code>
<code>tan</code>	<code>\fp_eval:n { tan(<fpexpr>) }</code>
<code>cot</code>	<code>\fp_eval:n { cot(<fpexpr>) }</code>
<code>csc</code>	<code>\fp_eval:n { csc(<fpexpr>) }</code>
<code>sec</code>	<code>\fp_eval:n { sec(<fpexpr>) }</code>

Updated: 2013-11-17

Computes the sine, cosine, tangent, cotangent, cosecant, or secant of the $\langle fpexpr \rangle$ given in radians. For arguments given in degrees, see `sind`, `cosd`, *etc.* Note that since π is irrational, `sin(8pi)` is not quite zero, while its analogue `sind(8 × 180)` is exactly zero. The trigonometric functions are undefined for an argument of $\pm\infty$, leading to the “invalid operation” exception. Additionally, evaluating tangent, cotangent, cosecant, or secant at one of their poles leads to a “division by zero” exception. “Underflow” and “overflow” occur when appropriate. If the operand is a tuple, “invalid operation” occurs.

<code>sind</code>	<code>\fp_eval:n { sind(<fpexpr>) }</code>
<code>cosd</code>	<code>\fp_eval:n { cosd(<fpexpr>) }</code>
<code>tand</code>	<code>\fp_eval:n { tand(<fpexpr>) }</code>
<code>cotd</code>	<code>\fp_eval:n { cotd(<fpexpr>) }</code>
<code>cscd</code>	<code>\fp_eval:n { cscd(<fpexpr>) }</code>
<code>secd</code>	<code>\fp_eval:n { secd(<fpexpr>) }</code>

New: 2013-11-02

Computes the sine, cosine, tangent, cotangent, cosecant, or secant of the $\langle fpexpr \rangle$ given in degrees. For arguments given in radians, see `sin`, `cos`, *etc.* Note that since π is irrational, `sin(8pi)` is not quite zero, while its analogue `sind(8 × 180)` is exactly zero. The trigonometric functions are undefined for an argument of $\pm\infty$, leading to the “invalid operation” exception. Additionally, evaluating tangent, cotangent, cosecant, or secant at one of their poles leads to a “division by zero” exception. “Underflow” and “overflow” occur when appropriate. If the operand is a tuple, “invalid operation” occurs.

<code>asin</code>	<code>\fp_eval:n { asin(<fpexpr>) }</code>
<code>acos</code>	<code>\fp_eval:n { acos(<fpexpr>) }</code>
<code>acsc</code>	<code>\fp_eval:n { acsc(<fpexpr>) }</code>
<code>asec</code>	<code>\fp_eval:n { asec(<fpexpr>) }</code>

New: 2013-11-02

Computes the arcsine, arccosine, arccosecant, or arcsecant of the $\langle fpexpr \rangle$ and returns the result in radians, in the range $[-\pi/2, \pi/2]$ for `asin` and `acsc` and $[0, \pi]$ for `acos` and `asec`. For a result in degrees, use `asind`, *etc.* If the argument of `asin` or `acos` lies outside the range $[-1, 1]$, or the argument of `acsc` or `asec` inside the range $(-1, 1)$, an “invalid operation” exception is raised. “Underflow” and “overflow” occur when appropriate. If the operand is a tuple, “invalid operation” occurs.

<code>asind</code>	<code>\fp_eval:n { asind(<fpexpr>) }</code>
<code>acosd</code>	<code>\fp_eval:n { acosd(<fpexpr>) }</code>
<code>acscd</code>	<code>\fp_eval:n { acscd(<fpexpr>) }</code>
<code>asecd</code>	<code>\fp_eval:n { asecd(<fpexpr>) }</code>

New: 2013-11-02

Computes the arcsine, arccosine, arccosecant, or arcsecant of the $\langle fpexpr \rangle$ and returns the result in degrees, in the range $[-90, 90]$ for `asin` and `acsc` and $[0, 180]$ for `acos` and `asec`. For a result in radians, use `asin`, *etc.* If the argument of `asin` or `acos` lies outside the range $[-1, 1]$, or the argument of `acsc` or `asec` inside the range $(-1, 1)$, an “invalid operation” exception is raised. “Underflow” and “overflow” occur when appropriate. If the operand is a tuple, “invalid operation” occurs.

atan	<code>\fp_eval:n { atan(<fpexpr>) }</code>
acot	<code>\fp_eval:n { atan(<fpexpr₁> , <fpexpr₂>) }</code>
<hr/>	
New: 2013-11-02	<code>\fp_eval:n { acot(<fpexpr>) }</code>
	<code>\fp_eval:n { acot(<fpexpr₁> , <fpexpr₂>) }</code>

Those functions yield an angle in radians: **atand** and **acotd** are their analogs in degrees. The one-argument versions compute the arctangent or arccotangent of the $\langle fpexpr \rangle$: arctangent takes values in the range $[-\pi/2, \pi/2]$, and arccotangent in the range $[0, \pi]$. The two-argument arctangent computes the angle in polar coordinates of the point with Cartesian coordinates $(\langle fpexpr_2 \rangle, \langle fpexpr_1 \rangle)$: this is the arctangent of $\langle fpexpr_1 \rangle / \langle fpexpr_2 \rangle$, possibly shifted by π depending on the signs of $\langle fpexpr_1 \rangle$ and $\langle fpexpr_2 \rangle$. The two-argument arccotangent computes the angle in polar coordinates of the point $(\langle fpexpr_1 \rangle, \langle fpexpr_2 \rangle)$, equal to the arccotangent of $\langle fpexpr_1 \rangle / \langle fpexpr_2 \rangle$, possibly shifted by π . Both two-argument functions take values in the wider range $[-\pi, \pi]$. The ratio $\langle fpexpr_1 \rangle / \langle fpexpr_2 \rangle$ need not be defined for the two-argument arctangent: when both expressions yield ± 0 , or when both yield $\pm \infty$, the resulting angle is one of $\{\pm\pi/4, \pm 3\pi/4\}$ depending on signs. The “underflow” exception can occur. If any operand is a tuple, “invalid operation” occurs.

atand	<code>\fp_eval:n { atand(<fpexpr>) }</code>
acotd	<code>\fp_eval:n { atand(<fpexpr₁> , <fpexpr₂>) }</code>
<hr/>	
New: 2013-11-02	<code>\fp_eval:n { acotd(<fpexpr>) }</code>
	<code>\fp_eval:n { acotd(<fpexpr₁> , <fpexpr₂>) }</code>

Those functions yield an angle in degrees: **atand** and **acotd** are their analogs in radians. The one-argument versions compute the arctangent or arccotangent of the $\langle fpexpr \rangle$: arctangent takes values in the range $[-90, 90]$, and arccotangent in the range $[0, 180]$. The two-argument arctangent computes the angle in polar coordinates of the point with Cartesian coordinates $(\langle fpexpr_2 \rangle, \langle fpexpr_1 \rangle)$: this is the arctangent of $\langle fpexpr_1 \rangle / \langle fpexpr_2 \rangle$, possibly shifted by 180 depending on the signs of $\langle fpexpr_1 \rangle$ and $\langle fpexpr_2 \rangle$. The two-argument arccotangent computes the angle in polar coordinates of the point $(\langle fpexpr_1 \rangle, \langle fpexpr_2 \rangle)$, equal to the arccotangent of $\langle fpexpr_1 \rangle / \langle fpexpr_2 \rangle$, possibly shifted by 180. Both two-argument functions take values in the wider range $[-180, 180]$. The ratio $\langle fpexpr_1 \rangle / \langle fpexpr_2 \rangle$ need not be defined for the two-argument arctangent: when both expressions yield ± 0 , or when both yield $\pm \infty$, the resulting angle is one of $\{\pm 45, \pm 135\}$ depending on signs. The “underflow” exception can occur. If any operand is a tuple, “invalid operation” occurs.

sqrt	<code>\fp_eval:n { sqrt(<fpexpr>) }</code>
-------------	--

New: 2013-12-14 Computes the square root of the $\langle fpexpr \rangle$. The “invalid operation” is raised when the $\langle fpexpr \rangle$ is negative or is a tuple; no other exception can occur. Special values yield $\sqrt{-0} = -0$, $\sqrt{+0} = +0$, $\sqrt{+\infty} = +\infty$ and $\sqrt{\text{nan}} = \text{nan}$.

<hr/> rand <hr/>	<code>\fp_eval:n { rand() }</code>
<hr/> <small>New: 2016-12-05</small> <hr/>	<p>Produces a pseudo-random floating-point number (multiple of 10^{-16}) between 0 included and 1 excluded. This is not available in older versions of $\text{X}\text{_}\text{T}\text{E}\text{X}$. The random seed can be queried using <code>\sys_rand_seed:</code> and set using <code>\sys_gset_rand_seed:n</code>.</p> <p>TEXhackers note: This is based on pseudo-random numbers provided by the engine’s primitive <code>\pdfuniformdeviate</code> in $\text{pdf}\text{_}\text{T}\text{E}\text{X}$, $\text{p}\text{T}\text{E}\text{X}$, $\text{up}\text{T}\text{E}\text{X}$ and <code>\uniformdeviate</code> in $\text{Lua}\text{_}\text{T}\text{E}\text{X}$ and $\text{X}\text{_}\text{T}\text{E}\text{X}$. The underlying code is based on Metapost, which follows an additive scheme recommended in Section 3.6 of “The Art of Computer Programming, Volume 2”.</p> <p>While we are more careful than <code>\uniformdeviate</code> to preserve uniformity of the underlying stream of 28-bit pseudo-random integers, these pseudo-random numbers should of course not be relied upon for serious numerical computations nor cryptography.</p>
<hr/> randint <hr/>	<code>\fp_eval:n { randint(<fpexpr>) }</code>
<hr/> <small>New: 2016-12-05</small> <hr/>	<code>\fp_eval:n { randint(<fpexpr₁> , <fpexpr₂>) }</code> <p>Produces a pseudo-random integer between 1 and $\langle fpexpr \rangle$ or between $\langle fpexpr_1 \rangle$ and $\langle fpexpr_2 \rangle$ inclusive. The bounds must be integers in the range $(-10^{16}, 10^{16})$ and the first must be smaller or equal to the second. See rand for important comments on how these pseudo-random numbers are generated.</p>
<hr/> inf nan <hr/>	The special values $+\infty$, $-\infty$, and nan are represented as inf , -inf and nan (see <code>\c_minus_inf_fp</code> , <code>\c_minus_inf_fp</code> and <code>\c_nan_fp</code>).
<hr/> pi <hr/>	The value of π (see <code>\c_pi_fp</code>).
<hr/> deg <hr/>	The value of 1° in radians (see <code>\c_one_degree_fp</code>).

<hr/>	Those units of measurement are equal to their values in <code>pt</code> , namely
<code>em</code>	
<code>ex</code>	
<code>in</code>	$1\text{ in} = 72.27\text{ pt}$
<code>pt</code>	$1\text{ pt} = 1\text{ pt}$
<code>pc</code>	
<code>cm</code>	$1\text{ pc} = 12\text{ pt}$
<code>mm</code>	
<code>dd</code>	$1\text{ cm} = \frac{1}{2.54}\text{ in} = 28.45275590551181\text{ pt}$
<code>cc</code>	
<code>nd</code>	$1\text{ mm} = \frac{1}{25.4}\text{ in} = 2.845275590551181\text{ pt}$
<code>nc</code>	
<code>bp</code>	$1\text{ dd} = 0.376065\text{ mm} = 1.07000856496063\text{ pt}$
<code>sp</code>	$1\text{ cc} = 12\text{ dd} = 12.84010277952756\text{ pt}$
<hr/>	$1\text{ nd} = 0.375\text{ mm} = 1.066978346456693\text{ pt}$
	$1\text{ nc} = 12\text{ nd} = 12.80374015748031\text{ pt}$
	$1\text{ bp} = \frac{1}{72}\text{ in} = 1.00375\text{ pt}$
	$1\text{ sp} = 2^{-16}\text{ pt} = 1.52587890625 \times 10^{-5}\text{ pt}.$

The values of the (font-dependent) units `em` and `ex` are gathered from \TeX when the surrounding floating point expression is evaluated.

<hr/>	
<code>true</code>	Other names for 1 and +0.
<code>false</code>	
<hr/>	

<hr/>	
<code>\fp_abs:n</code> *	<code>\fp_abs:n {<floating point expression>}</code>
<hr/>	
New: 2012-05-14	Evaluates the <i><floating point expression></i> as described for <code>\fp_eval:n</code> and leaves the
Updated: 2012-07-08	absolute value of the result in the input stream. If the argument is $\pm\infty$, <code>nan</code> or a tuple,
<hr/>	“invalid operation” occurs. Within floating point expressions, <code>abs()</code> can be used; it
	accepts $\pm\infty$ and <code>nan</code> as arguments.

<hr/>	
<code>\fp_max:nn</code> *	<code>\fp_max:nn {<fp expression 1>} {<fp expression 2>}</code>
<code>\fp_min:nn</code> *	
<hr/>	
New: 2012-09-26	Evaluates the <i><floating point expressions></i> as described for <code>\fp_eval:n</code> and leaves the
	resulting larger (<code>max</code>) or smaller (<code>min</code>) value in the input stream. If the argument is a
	tuple, “invalid operation” occurs, but no other case raises exceptions. Within floating
	point expressions, <code>max()</code> and <code>min()</code> can be used.

28.10 Disclaimer and roadmap

The package may break down if the escape character is among `0123456789_+`, or if it receives a \TeX primitive conditional affected by `\exp_not:N`.

The following need to be done. I’ll try to time-order the items.

- Function to count items in a tuple (and to determine if something is a tuple).
- Decide what exponent range to consider.

- Support signalling `nan`.
- Modulo and remainder, and rounding function `quantize` (and its friends analogous to `trunc`, `ceil`, `floor`).
- `\fp_format:nn` $\{\langle fpexpr \rangle\}$ $\{\langle format \rangle\}$, but what should $\langle format \rangle$ be? More general pretty printing?
- Add `and`, `or`, `xor`? Perhaps under the names `all`, `any`, and `xor`?
- Add $\log(x, b)$ for logarithm of x in base b .
- `hypot` (Euclidean length). Cartesian-to-polar transform.
- Hyperbolic functions `cosh`, `sinh`, `tanh`.
- Inverse hyperbolics.
- Base conversion, input such as `0xAB.CDEF`.
- Factorial (not with `!`), gamma function.
- Improve coefficients of the `sin` and `tan` series.
- Treat upper and lower case letters identically in identifiers, and ignore underscores.
- Add an `array(1,2,3)` and `i=complex(0,1)`.
- Provide an experimental `map` function? Perhaps easier to implement if it is a single character, `@sin(1,2)`?
- Provide an `isnan` function analogue of `\fp_if_nan:nTF`?
- Support keyword arguments?

`Pgfmath` also provides box-measurements (depth, height, width), but boxes are not possible expandably.

Bugs, and tests to add.

- Check that functions are monotonic when they should.
- Add exceptions to `?:`, `!<=>?`, `&&`, `||`, and `!`.
- Logarithms of numbers very close to 1 are inaccurate.
- When rounding towards $-\infty$, `\dim_to_fp:n` $\{\text{Opt}\}$ should return -0 , not $+0$.
- The result of $(\pm 0) + (\pm 0)$, of $x + (-x)$, and of $(-x) + x$ should depend on the rounding mode.
- `0e9999999999` gives a $\text{T}_{\text{E}}\text{X}$ “number too large” error.
- Subnormals are not implemented.

Possible optimizations/improvements.

- Document that `l3trial/l3fp-types` introduces tools for adding new types.
- In subsection [28.9.1](#), write a grammar.

- It would be nice if the `parse` auxiliaries for each operation were set up in the corresponding module, rather than centralizing in `l3fp-parse`.
- Some functions should get an `_o` ending to indicate that they expand after their result.
- More care should be given to distinguish expandable/restricted expandable (auxiliary and internal) functions.
- The code for the `ternary` set of functions is ugly.
- There are many `~` missing in the doc to avoid bad line-breaks.
- The algorithm for computing the logarithm of the significand could be made to use a 5 terms Taylor series instead of 10 terms by taking $c = 2000/([200x]+1) \in [10, 95]$ instead of $c \in [1, 10]$. Also, it would then be possible to simplify the computation of t . However, we would then have to hard-code the logarithms of 44 small integers instead of 9.
- Improve notations in the explanations of the division algorithm (`l3fp-basics`).
- Understand and document `_fp_basics_pack_weird_low:NNNNw` and `_fp_basics_pack_weird_high:NNNNNNNNw` better. Move the other `basics_pack` auxiliaries to `l3fp-aux` under a better name.
- Find out if underflow can really occur for trigonometric functions, and redoc as appropriate.
- Add bibliography. Some of Kahan’s articles, some previous T_EX fp packages, the international standards,...
- Also take into account the “inexact” exception?
- Support multi-character prefix operators (*e.g.*, `@/` or whatever)?

Chapter 29

The l3farray package: fast global floating point arrays

29.1 l3farray documentation

For applications requiring heavy use of floating points, this module provides arrays which can be accessed in constant time (contrast l3seq, where access time is linear). The interface is very close to that of l3intarray. The size of the array is fixed and must be given at point of initialisation

<code>\farray_new:Nn</code>	<code>\farray_new:Nn <farray var> {<size>}</code>
-----------------------------	---

New: 2018-05-05

Evaluates the integer expression *<size>* and allocates an *<floating point array variable>* with that number of (zero) entries. The variable name should start with `\g_` because assignments are always global.

<code>\farray_count:N *</code>	<code>\farray_count:N <farray var></code>
--------------------------------	---

New: 2018-05-05

Expands to the number of entries in the *<floating point array variable>*. This is performed in constant time.

<code>\farray_gset:Nnn</code>	<code>\farray_gset:Nnn <farray var> {<position>} {<value>}</code>
-------------------------------	---

New: 2018-05-05

Stores the result of evaluating the floating point expression *<value>* into the *<floating point array variable>* at the (integer expression) *<position>*. If the *<position>* is not between 1 and the `\farray_count:N`, an error occurs. Assignments are always global.

<code>\farray_gzero:N</code>	<code>\farray_gzero:N <farray var></code>
------------------------------	---

New: 2018-05-05

Sets all entries of the *<floating point array variable>* to +0. Assignments are always global.

<code>\farray_item:Nn *</code>	<code>\farray_item:Nn <farray var> {<position>}</code>
--------------------------------	--

`\farray_item_to_tl:Nn *`

New: 2018-05-05

Applies `\fp_use:N` or `\fp_to_tl:N` (respectively) to the floating point entry stored at the (integer expression) *<position>* in the *<floating point array variable>*. If the *<position>* is not between 1 and the `\farray_count:N`, an error occurs.

Chapter 30

The `\lccstab` package

Category code tables

A category code table enables rapid switching of all category codes in one operation. For `LuaTeX`, this is possible over the entire Unicode range. For other engines, only the 8-bit range (0–255) is covered by such tables.

30.1 Creating and initialising category code tables

<code>\lccstab_new:N</code>	<code>\lccstab_new:N</code> <i><category code table></i>
<code>\lccstab_new:c</code>	Creates a new <i><category code table></i> variable or raises an error if the name is already taken. The declaration is global. The <i><category code table></i> is initialised with the codes as used by <code>iniTeX</code> .

Updated: 2020-07-02

<code>\lccstab_const:Nn</code>	<code>\lccstab_const:Nn</code> <i><category code table></i> <i>{<category code set up>}</i>
<code>\lccstab_const:cn</code>	Creates a new <i><category code table></i> , applies (in a group) the <i><category code set up></i> on top of <code>iniTeX</code> settings, then saves them globally as a constant table. The <i><category code set up></i> can include a call to <code>\lccstab_select:N</code> .

Updated: 2020-07-07

<code>\lccstab_gset:Nn</code>	<code>\lccstab_gset:Nn</code> <i><category code table></i> <i>{<category code set up>}</i>
<code>\lccstab_gset:cn</code>	Starting from the <code>iniTeX</code> category codes, applies (in a group) the <i><category code set up></i> , then saves them globally in the <i><category code table></i> . The <i><category code set up></i> can include a call to <code>\lccstab_select:N</code> .

Updated: 2020-07-07

30.2 Using category code tables

<code>\lccstab_begin:N</code>	<code>\lccstab_begin:N</code> <i><category code table></i>
<code>\lccstab_begin:c</code>	Switches locally the category codes in force to those stored in the <i><category code table></i> . The prevailing codes before the function is called are added to a stack, for use with <code>\lccstab_end:.</code> This function does not start a <code>TeX</code> group.

Updated: 2020-07-02

<hr/> <code>\cctab_end:</code> <hr/>	<code>\cctab_end:</code>
<hr/> Updated: 2020-07-02 <hr/>	Ends the scope of a <i>category code table</i> started using <code>\cctab_begin:N</code> , returning the codes to those in force before the matching <code>\cctab_begin:N</code> was used. This must be used within the same T _E X group (and at the same T _E X group level) as the matching <code>\cctab_begin:N</code> .
<hr/> <code>\cctab_select:N</code> <code>\cctab_select:c</code> <hr/>	<code>\cctab_select:N</code> <i>category code table</i>
<hr/> New: 2020-05-19 Updated: 2020-07-02 <hr/>	Selects the <i>category code table</i> for the scope of the current group. This is in particular useful in the <i>setup</i> arguments of <code>\tl_set_rescan:Nnn</code> , <code>\tl_rescan:nn</code> , <code>\cctab_const:Nn</code> , and <code>\cctab_gset:Nn</code> .
<hr/> <code>\cctab_item:Nn</code> ★ <code>\cctab_item:cn</code> ★ <hr/>	<code>\cctab_item:Nn</code> <i>category code table</i> { <i>integer expression</i> }
<hr/> New: 2021-05-10 <hr/>	Determines the <i>character</i> with character code given by the <i>integer expression</i> and expands to its category code specified by the <i>category code table</i> .

30.3 Category code table conditionals

<hr/> <code>\cctab_if_exist_p:N</code> ★	<code>\cctab_if_exist_p:N</code> <i>category code table</i>
<code>\cctab_if_exist_p:c</code> ★	<code>\cctab_if_exist:NTF</code> <i>category code table</i> { <i>true code</i> } { <i>false code</i> }
<code>\cctab_if_exist:NTF</code> ★	Tests whether the <i>category code table</i> is currently defined. This does not check that the
<code>\cctab_if_exist:cTF</code> ★	<i>category code table</i> really is a category code table.

30.4 Constant category code tables

<hr/> <code>\c_code_cctab</code> <hr/>	Category code table for the <code>expl3</code> code environment; this does <i>not</i> include <code>@</code> , which is retained as an “other” character.
<hr/> Updated: 2020-07-10 <hr/>	
<hr/> <code>\c_document_cctab</code> <hr/>	Category code table for a standard L ^A T _E X document, as set by the L ^A T _E X kernel. In particular, the upper-half of the 8-bit range will be set to “active” with pdfT _E X <i>only</i> . No <code>babel</code> shorthands will be activated.
<hr/> Updated: 2020-07-08 <hr/>	
<hr/> <code>\c_initex_cctab</code> <hr/>	Category code table as set up by iniT _E X.
<hr/> Updated: 2020-07-02 <hr/>	
<hr/> <code>\c_other_cctab</code> <hr/>	Category code table where all characters have category code 12 (other).
<hr/> Updated: 2020-07-02 <hr/>	
<hr/> <code>\c_str_cctab</code> <hr/>	Category code table where all characters have category code 12 (other) with the exception of spaces, which have category code 10 (space).
<hr/> Updated: 2020-07-02 <hr/>	

Part V

Text manipulation

Chapter 31

The `l3unicode` package: Unicode support functions

This module provides Unicode-specific functions along with loading data from a range of Unicode Consortium files. At present, it provides no public functions.

Chapter 32

The l3text package: text processing

This module deals with manipulation of (formatted) text; such material is comprised of a restricted set of token list content. The functions provided here concern conversion of textual content for example in case changing, generation of bookmarks and extraction to tags. All of the major functions operate by expansion. Begin-group and end-group tokens in the $\langle text \rangle$ are normalized and become { and }, respectively.

32.1 Expanding text

`\text_expand:n` ★

New: 2020-01-02

`\text_expand:n` { $\langle text \rangle$ }

Takes user input $\langle text \rangle$ and expands the content. Protected commands (typically formatting) are left in place, and no processing takes place of math mode material (as delimited by pairs given in `\l_text_math_delims_tl` or as the argument to commands listed in `\l_text_math_arg_tl`). Commands which are neither engine- nor L^AT_EX protected are expanded exhaustively. Any commands listed in `\l_text_expand_exclude_tl`, `\l_text_accents_tl` and `\l_text_letterlike_tl` are excluded from expansion.

`\text_declare_expand_equivalent:Nn` `\text_declare_expand_equivalent:Nn` $\langle cmd \rangle$ { $\langle replacement \rangle$ }

`\text_declare_expand_equivalent:cn`

New: 2020-01-22

Declares that the $\langle replacement \rangle$ tokens should be used whenever the $\langle cmd \rangle$ (a single token) is encountered. The $\langle replacement \rangle$ tokens should be expandable.

32.2 Case changing

<code>\text_lowercase:n</code>	★	<code>\text_uppercase:n</code>	<code>{\tokens}</code>
<code>\text_uppercase:n</code>	★	<code>\text_uppercase:nn</code>	<code>{\language}</code> <code>{\tokens}</code>
<code>\text_titlecase:n</code>	★	Takes user input <code>\text</code> first applies <code>\text_expand</code> , then transforms the case of character tokens as specified by the function name. The category code of letters are not changed by this process (at least where they can be represented by the engine as a single token: 8-bit engines may require active characters).	
<code>\text_titlecase_first:n</code>	★		
<code>\text_lowercase:nn</code>	★		
<code>\text_uppercase:nn</code>	★		
<code>\text_titlecase:nn</code>	★		
<code>\text_titlecase_first:nn</code>	★		

New: 2019-11-20

Updated: 2020-02-24

Upper- and lowercase have the obvious meanings. Titlecasing may be regarded informally as converting the first character of the `\tokens` to uppercase and the rest to lowercase. However, the process is more complex than this as there are some situations where a single lowercase character maps to a special form, for example `ij` in Dutch which becomes `IJ`. The `\titlecase_first` variant does not attempt any case changing at all after the first letter has been processed.

Importantly, notice that these functions are intended for working with user *text for typesetting*. For case changing programmatic data see the `\l3str` module and discussion there of `\str_lowercase:n`, `\str_uppercase:n` and `\str_foldcase:n`.

Case changing does not take place within math mode material so for example

```
\text_uppercase:n { Some~text~$y = mx + c$~with~{Braces} }
```

becomes

```
SOME TEXT $y = mx + c$ WITH {BRACES}
```

The arguments of commands listed in `\l_text_case_exclude_arg_tl` are excluded from case changing; the latter are entirely non-textual content (such as labels).

As is generally true for `expl3`, these functions are designed to work with Unicode input only. As such, UTF-8 input is assumed for *all* engines. When used with `XYTeX` or `LuaTeX` a full range of Unicode transformations are enabled. Specifically, the standard mappings here follow those defined by the [Unicode Consortium](#) in `UnicodeData.txt` and `SpecialCasing.txt`. In the case of 8-bit engines, mappings are provided for characters which can be represented in output typeset using the `T1`, `T2` and `LGR` font encodings. Thus for example `ä` can be case-changed using `pdfTeX`. For `pTeX` only the ASCII range is covered as the engine treats input outside of this range as east Asian.

Language-sensitive conversions are enabled using the `\language` argument, and follow Unicode Consortium guidelines. Currently, the languages recognised for special handling are as follows.

- Azeri and Turkish (`az` and `tr`). The case pairs `I/i-dotless` and `I-dot/i` are activated for these languages. The combining dot mark is removed when lowercasing `I-dot` and introduced when upper casing `i-dotless`.
- German (`de-alt`). An alternative mapping for German in which the lowercase *Eszett* maps to a *großes Eszett*. Since there is a `T1` slot for the *großes Eszett* in `T1`, this tailoring *is* available with `pdfTeX` as well as in the Unicode `TeX` engines.

- Greek (`e1`). Removes accents from Greek letters when uppercasing; titlecasing leaves accents in place. (At present this is implemented only for Unicode engines.)
- Lithuanian (`1t`). The lowercase letters `i` and `j` should retain a dot above when the accents grave, acute or tilde are present. This is implemented for lowercasing of the relevant uppercase letters both when input as single Unicode codepoints and when using combining accents. The combining dot is removed when uppercasing in these cases. Note that *only* the accents used in Lithuanian are covered: the behaviour of other accents are not modified.
- Dutch (`n1`). Capitalisation of `ij` at the beginning of titlecased input produces `IJ` rather than `Ij`. The output retains two separate letters, thus this transformation *is* available using pdfTeX.

For titlecasing, note that there are two functions available. The function `\text_titlecase:n` applies (broadly) uppercasing to the first letter of the input, then lowercasing to the remainder. In contrast, `\text_titlecase_first:n` *only* carries out the uppercasing operation, and leaves the balance of the input unchanged. Determining whether non-letter characters at the start of text should switch from upper- to lowercasing is controllable. When `\l_text_titlecase_check_letter_bool` is `true`, characters which are not letters (category code 11) are left unchanged and “skipped”: the first *letter* is uppercased. (With 8-bit engines, this is extended to active characters which form part of a multi-byte letter codepoint.) When `\l_text_titlecase_check_letter_bool` is `false`, the first character is uppercased, and the rest lowercased, irrespective of the nature of the character.

32.3 Removing formatting from text

<code>\text_purify:n</code> ★	<code>\text_purify:n {<text>}</code>
-------------------------------	--

New: 2020-03-05
Updated: 2020-05-14

Takes user input `<text>` and expands as described for `\text_expand:n`, then removes all functions from the resulting text. Math mode material (as delimited by pairs given in `\l_text_math_delims_tl` or as the argument to commands listed in `\l_text_math_arg_tl`) is left contained in a pair of `$` delimiters. Non-expandable functions present in the `<text>` must either have a defined equivalent (see `\text_declare_purify_equivalent:Nn`) or will be removed from the result. Implicit tokens are converted to their explicit equivalent.

<code>\text_declare_purify_equivalent:Nn</code> <code>\text_declare_purify_equivalent:Nx</code>	<code>\text_declare_purify_equivalent:Nn <cmd> {<replacement>}</code>
--	---

New: 2020-03-05

Declares that the `<replacement>` tokens should be used whenever the `<cmd>` (a single token) is encountered. The `<replacement>` tokens should be expandable.

32.4 Control variables

<code>\l_text_accents_tl</code>	
---------------------------------	--

Lists commands which represent accents, and which are left unchanged by expansion. (Defined only for the L^AT_EX 2_ε package.)

<u><u><code>\l_text_letterlike_tl</code></u></u>	Lists commands which represent letters; these are left unchanged by expansion. (Defined only for the L ^A T _E X 2 _ε package.)
<u><u><code>\l_text_math_arg_tl</code></u></u>	Lists commands present in the $\langle text \rangle$ where the argument of the command should be treated as math mode material. The treatment here is similar to <code>\l_text_math_delims_tl</code> but for a command rather than paired delimiters.
<u><u><code>\l_text_math_delims_tl</code></u></u>	Lists pairs of tokens which delimit (in-line) math mode content; such content <i>may</i> be excluded from processing.
<u><u><code>\l_text_case_exclude_arg_tl</code></u></u>	Lists commands which are excluded from case changing. This protection includes everything up to and including their first braced argument.
<u><u><code>\l_text_expand_exclude_tl</code></u></u>	Lists commands which are excluded from expansion. This protection includes everything up to and including their first braced argument.
<u><u><code>\l_text_titlecase_check_letter_bool</code></u></u>	Controls how the start of titlecasing is handled: when <code>true</code> , the first <i>letter</i> in text is considered. The standard setting is <code>true</code> .

Part VI

Typesetting

Chapter 33

The **l3box** package

Boxes

Box variables contain typeset material that can be inserted on the page or in other boxes. Their contents cannot be converted back to lists of tokens. There are three kinds of box operations: horizontal mode denoted with prefix `\hbox_`, vertical mode with prefix `\vbox_`, and the generic operations working in both modes with prefix `\box_`. For instance, a new box variable containing the words “Hello, world!” (in a horizontal box) can be obtained by the following code.

```
\box_new:N \l_hello_box
\hbox_set:Nn \l_hello_box { Hello, ~ world! }
```

The argument is typeset inside a \TeX group so that any variables assigned during the construction of this box restores its value afterwards.

Box variables from **l3box** are compatible with those of $\text{\LaTeX 2}_{\epsilon}$ and plain \TeX and can be used interchangeably. The **l3box** commands to construct boxes, such as `\hbox:n` or `\hbox_set:Nn`, are “color-safe”, meaning that

```
\hbox:n { \color_select:n { blue } Hello, } ~ world!
```

will result in “Hello,” taking the color blue, but “world!” remaining with the prevailing color outside the box.

33.1 Creating and initialising boxes

```
\box_new:N \box_new:N <box>
```

```
\box_new:c
```

Creates a new $\langle box \rangle$ or raises an error if the name is already taken. The declaration is global. The $\langle box \rangle$ is initially void.

```
\box_clear:N \box_clear:N <box>
```

```
\box_clear:c
```

```
\box_gclear:N
```

```
\box_gclear:c
```

Clears the content of the $\langle box \rangle$ by setting the box equal to `\c_empty_box`.

<hr/>	
<code>\box_clear_new:N</code>	<code>\box_clear_new:N <box></code>
<code>\box_clear_new:c</code>	
<code>\box_gclear_new:N</code>	Ensures that the $\langle box \rangle$ exists globally by applying <code>\box_new:N</code> if necessary, then applies
<code>\box_gclear_new:c</code>	<code>\box_(g)clear:N</code> to leave the $\langle box \rangle$ empty.
<hr/>	
<code>\box_set_eq:NN</code>	<code>\box_set_eq:NN <box₁> <box₂></code>
<code>\box_set_eq:(cN Nc cc)</code>	Sets the content of $\langle box_1 \rangle$ equal to that of $\langle box_2 \rangle$.
<code>\box_gset_eq:NN</code>	
<code>\box_gset_eq:(cN Nc cc)</code>	
<hr/>	
<code>\box_if_exist_p:N *</code>	<code>\box_if_exist_p:N <box></code>
<code>\box_if_exist_p:c *</code>	<code>\box_if_exist:NTF <box> {\true code} {\false code}</code>
<code>\box_if_exist:NTF *</code>	
<code>\box_if_exist:cTF *</code>	Tests whether the $\langle box \rangle$ is currently defined. This does not check that the $\langle box \rangle$ really is a box.
<hr/>	
New: 2012-03-03	
<hr/>	

33.2 Using boxes

<hr/>	
<code>\box_use:N</code>	<code>\box_use:N <box></code>
<code>\box_use:c</code>	
<hr/>	
	Inserts the current content of the $\langle box \rangle$ onto the current list for typesetting. An error is raised if the variable does not exist or if it is invalid.
 T_EXhackers note: This is the T _E X primitive <code>\copy</code> .	
<hr/>	
<code>\box_move_right:nn</code>	<code>\box_move_right:nn {\dimexpr} {\box function}</code>
<code>\box_move_left:nn</code>	
<hr/>	
	This function operates in vertical mode, and inserts the material specified by the $\langle box \text{ function} \rangle$ such that its reference point is displaced horizontally by the given $\langle dimexpr \rangle$ from the reference point for typesetting, to the right or left as appropriate. The $\langle box \text{ function} \rangle$ should be a box operation such as <code>\box_use:N \<box></code> or a “raw” box specification such as <code>\vbox:n { xyz }</code> .
<hr/>	
<code>\box_move_up:nn</code>	<code>\box_move_up:nn {\dimexpr} {\box function}</code>
<code>\box_move_down:nn</code>	
<hr/>	
	This function operates in horizontal mode, and inserts the material specified by the $\langle box \text{ function} \rangle$ such that its reference point is displaced vertically by the given $\langle dimexpr \rangle$ from the reference point for typesetting, up or down as appropriate. The $\langle box \text{ function} \rangle$ should be a box operation such as <code>\box_use:N \<box></code> or a “raw” box specification such as <code>\vbox:n { xyz }</code> .

33.3 Measuring and setting box dimensions

<code>\box_dp:N</code>	<code>\box_dp:N</code> $\langle box \rangle$
<code>\box_dp:c</code>	Calculates the depth (below the baseline) of the $\langle box \rangle$ in a form suitable for use in a $\langle dimension expression \rangle$.

T_EXhackers note: This is the T_EX primitive `\dp`.

<code>\box_ht:N</code>	<code>\box_ht:N</code> $\langle box \rangle$
<code>\box_ht:c</code>	Calculates the height (above the baseline) of the $\langle box \rangle$ in a form suitable for use in a $\langle dimension expression \rangle$.

T_EXhackers note: This is the T_EX primitive `\ht`.

<code>\box_wd:N</code>	<code>\box_wd:N</code> $\langle box \rangle$
<code>\box_wd:c</code>	Calculates the width of the $\langle box \rangle$ in a form suitable for use in a $\langle dimension expression \rangle$.

T_EXhackers note: This is the T_EX primitive `\wd`.

<code>\box_ht_plus_dp:N</code>	<code>\box_ht_plus_dp:N</code> $\langle box \rangle$
<code>\box_ht_plus_dp:c</code>	Calculates the total vertical size (height plus depth) of the $\langle box \rangle$ in a form suitable for use in a $\langle dimension expression \rangle$.

New: 2021-05-05

<code>\box_set_dp:Nn</code>	<code>\box_set_dp:Nn</code> $\langle box \rangle$ $\{ \langle dimension expression \rangle \}$
<code>\box_set_dp:cn</code>	
<code>\box_gset_dp:Nn</code>	Set the depth (below the baseline) of the $\langle box \rangle$ to the value of the $\{ \langle dimension expression \rangle \}$.
<code>\box_gset_dp:cn</code>	

Updated: 2019-01-22

<code>\box_set_ht:Nn</code>	<code>\box_set_ht:Nn</code> $\langle box \rangle$ $\{ \langle dimension expression \rangle \}$
<code>\box_set_ht:cn</code>	
<code>\box_gset_ht:Nn</code>	Set the height (above the baseline) of the $\langle box \rangle$ to the value of the $\{ \langle dimension expression \rangle \}$.
<code>\box_gset_ht:cn</code>	

Updated: 2019-01-22

<code>\box_set_wd:Nn</code>	<code>\box_set_wd:Nn</code> $\langle box \rangle$ $\{ \langle dimension expression \rangle \}$
<code>\box_set_wd:cn</code>	
<code>\box_gset_wd:Nn</code>	Set the width of the $\langle box \rangle$ to the value of the $\{ \langle dimension expression \rangle \}$.
<code>\box_gset_wd:cn</code>	

Updated: 2019-01-22

33.4 Box conditionals

<code>\box_if_empty_p:N</code>	★	<code>\box_if_empty_p:N</code> $\langle box \rangle$
<code>\box_if_empty_p:c</code>	★	<code>\box_if_empty:NTF</code> $\langle box \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$
<code>\box_if_empty:NTF</code>	★	Tests if $\langle box \rangle$ is a empty (equal to <code>\c_empty_box</code>).
<code>\box_if_empty:cTF</code>	★	

<code>\box_if_horizontal_p:N</code>	★	<code>\box_if_horizontal_p:N</code> $\langle box \rangle$
<code>\box_if_horizontal_p:c</code>	★	<code>\box_if_horizontal:NTF</code> $\langle box \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$
<code>\box_if_horizontal:NTF</code>	★	Tests if $\langle box \rangle$ is a horizontal box.
<code>\box_if_horizontal:cTF</code>	★	

<code>\box_if_vertical_p:N</code>	★	<code>\box_if_vertical_p:N</code> $\langle box \rangle$
<code>\box_if_vertical_p:c</code>	★	<code>\box_if_vertical:NTF</code> $\langle box \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$
<code>\box_if_vertical:NTF</code>	★	Tests if $\langle box \rangle$ is a vertical box.
<code>\box_if_vertical:cTF</code>	★	

33.5 The last box inserted

<code>\box_set_to_last:N</code>	<code>\box_set_to_last:N</code> $\langle box \rangle$
<code>\box_set_to_last:c</code>	Sets the $\langle box \rangle$ equal to the last item (box) added to the current partial list, removing the item from the list at the same time. When applied to the main vertical list, the $\langle box \rangle$ is always void as it is not possible to recover the last added item.
<code>\box_gset_to_last:N</code>	
<code>\box_gset_to_last:c</code>	

33.6 Constant boxes

<code>\c_empty_box</code>	This is a permanently empty box, which is neither set as horizontal nor vertical.
Updated: 2012-11-04	TeXhackers note: At the TeX level this is a void box.

33.7 Scratch boxes

<code>\l_tmpa_box</code>	Scratch boxes for local assignment. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
<code>\l_tmpb_box</code>	
Updated: 2012-11-04	

<code>\g_tmpa_box</code>	Scratch boxes for global assignment. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
<code>\g_tmpb_box</code>	

33.8 Viewing box contents

<hr/> <code>\box_show:N</code> <hr/>	<code>\box_show:N <box></code>
<code>\box_show:c</code> <hr/>	Shows full details of the content of the $\langle box \rangle$ in the terminal.
<hr/> <code>Updated: 2012-05-11</code> <hr/>	
<hr/> <code>\box_show:Nnn</code> <hr/>	<code>\box_show:Nnn <box> {\intexpr_1} {\intexpr_2}</code>
<code>\box_show:cnn</code> <hr/>	Display the contents of $\langle box \rangle$ in the terminal, showing the first $\langle intexpr_1 \rangle$ items of the box, and descending into $\langle intexpr_2 \rangle$ group levels.
<hr/> <code>New: 2012-05-11</code> <hr/>	
<hr/> <code>\box_log:N</code> <hr/>	<code>\box_log:N <box></code>
<code>\box_log:c</code> <hr/>	Writes full details of the content of the $\langle box \rangle$ to the log.
<hr/> <code>New: 2012-05-11</code> <hr/>	
<hr/> <code>\box_log:Nnn</code> <hr/>	<code>\box_log:Nnn <box> {\intexpr_1} {\intexpr_2}</code>
<code>\box_log:cnn</code> <hr/>	Writes the contents of $\langle box \rangle$ to the log, showing the first $\langle intexpr_1 \rangle$ items of the box, and descending into $\langle intexpr_2 \rangle$ group levels.
<hr/> <code>New: 2012-05-11</code> <hr/>	

33.9 Boxes and color

All L^AT_EX3 boxes are “color safe”: a color set inside the box stops applying after the end of the box has occurred.

33.10 Horizontal mode boxes

<hr/> <code>\hbox:n</code> <hr/>	<code>\hbox:n {\contents}</code>
<code>Updated: 2017-04-05</code> <hr/>	Typesets the $\langle contents \rangle$ into a horizontal box of natural width and then includes this box in the current list for typesetting.
<hr/> <code>\hbox_to_wd:nn</code> <hr/>	<code>\hbox_to_wd:nn {\dimexpr} {\contents}</code>
<code>Updated: 2017-04-05</code> <hr/>	Typesets the $\langle contents \rangle$ into a horizontal box of width $\langle dimexpr \rangle$ and then includes this box in the current list for typesetting.
<hr/> <code>\hbox_to_zero:n</code> <hr/>	<code>\hbox_to_zero:n {\contents}</code>
<code>Updated: 2017-04-05</code> <hr/>	Typesets the $\langle contents \rangle$ into a horizontal box of zero width and then includes this box in the current list for typesetting.
<hr/> <code>\hbox_set:Nn</code> <hr/>	<code>\hbox_set:Nn <box> {\contents}</code>
<code>\hbox_set:cn</code> <hr/>	Typesets the $\langle contents \rangle$ at natural width and then stores the result inside the $\langle box \rangle$.
<code>\hbox_gset:Nn</code> <hr/>	
<code>\hbox_gset:cn</code> <hr/>	
<hr/> <code>Updated: 2017-04-05</code> <hr/>	

```
\hbox_set_to_wd:Nnn
\hbox_set_to_wd:cnn
\hbox_gset_to_wd:Nnn
\hbox_gset_to_wd:cnn
```

Updated: 2017-04-05

```
\hbox_set_to_wd:Nnn <box> {<dimexpr>} {<contents>}
```

Typesets the $\langle contents \rangle$ to the width given by the $\langle dimexpr \rangle$ and then stores the result inside the $\langle box \rangle$.

```
\hbox_overlap_center:n
```

New: 2020-08-25

```
\hbox_overlap_center:n {<contents>}
```

Typesets the $\langle contents \rangle$ into a horizontal box of zero width such that material protrudes equally to both sides of the insertion point.

```
\hbox_overlap_right:n
```

Updated: 2017-04-05

```
\hbox_overlap_right:n {<contents>}
```

Typesets the $\langle contents \rangle$ into a horizontal box of zero width such that material protrudes to the right of the insertion point.

```
\hbox_overlap_left:n
```

Updated: 2017-04-05

```
\hbox_overlap_left:n {<contents>}
```

Typesets the $\langle contents \rangle$ into a horizontal box of zero width such that material protrudes to the left of the insertion point.

```
\hbox_set:Nw
\hbox_set:cw
\hbox_set_end:
\hbox_gset:Nw
\hbox_gset:cw
\hbox_gset_end:
```

Updated: 2017-04-05

```
\hbox_set:Nw <box> <contents> \hbox_set_end:
```

Typesets the $\langle contents \rangle$ at natural width and then stores the result inside the $\langle box \rangle$. In contrast to `\hbox_set:Nn` this function does not absorb the argument when finding the $\langle content \rangle$, and so can be used in circumstances where the $\langle content \rangle$ may not be a simple argument.

```
\hbox_set_to_wd:Nnw
\hbox_set_to_wd:cnw
\hbox_gset_to_wd:Nnw
\hbox_gset_to_wd:cnw
```

New: 2017-06-08

```
\hbox_set_to_wd:Nnw <box> {<dimexpr>} <contents> \hbox_set_end:
```

Typesets the $\langle contents \rangle$ to the width given by the $\langle dimexpr \rangle$ and then stores the result inside the $\langle box \rangle$. In contrast to `\hbox_set_to_wd:Nnn` this function does not absorb the argument when finding the $\langle content \rangle$, and so can be used in circumstances where the $\langle content \rangle$ may not be a simple argument

```
\hbox_unpack:N
\hbox_unpack:c
```

```
\hbox_unpack:N <box>
```

Unpacks the content of the horizontal $\langle box \rangle$, retaining any stretching or shrinking applied when the $\langle box \rangle$ was set.

T_EXhackers note: This is the T_EX primitive `\unhcopy`.

33.11 Vertical mode boxes

Vertical boxes inherit their baseline from their contents. The standard case is that the baseline of the box is at the same position as that of the last item added to the box. This means that the box has no depth unless the last item added to it had depth. As a result most vertical boxes have a large height value and small or zero depth. The exception are

_top boxes, where the reference point is that of the first item added. These tend to have a large depth and small height, although the latter is typically non-zero.

<hr/> <code>\vbox:n</code> <hr/>	<code>\vbox:n {<contents>}</code>
Updated: 2017-04-05 <hr/>	Typesets the $\langle contents \rangle$ into a vertical box of natural height and includes this box in the current list for typesetting.
<hr/> <code>\vbox_top:n</code> <hr/>	<code>\vbox_top:n {<contents>}</code>
Updated: 2017-04-05 <hr/>	Typesets the $\langle contents \rangle$ into a vertical box of natural height and includes this box in the current list for typesetting. The baseline of the box is equal to that of the <i>first</i> item added to the box.
<hr/> <code>\vbox_to_ht:nn</code> <hr/>	<code>\vbox_to_ht:nn {<dimexpr>} {<contents>}</code>
Updated: 2017-04-05 <hr/>	Typesets the $\langle contents \rangle$ into a vertical box of height $\langle dimexpr \rangle$ and then includes this box in the current list for typesetting.
<hr/> <code>\vbox_to_zero:n</code> <hr/>	<code>\vbox_to_zero:n {<contents>}</code>
Updated: 2017-04-05 <hr/>	Typesets the $\langle contents \rangle$ into a vertical box of zero height and then includes this box in the current list for typesetting.
<hr/> <code>\vbox_set:Nn</code> <code>\vbox_set:cn</code> <code>\vbox_gset:Nn</code> <code>\vbox_gset:cn</code> <hr/>	<code>\vbox_set:Nn <box> {<contents>}</code> Typesets the $\langle contents \rangle$ at natural height and then stores the result inside the $\langle box \rangle$.
Updated: 2017-04-05 <hr/>	
<hr/> <code>\vbox_set_top:Nn</code> <code>\vbox_set_top:cn</code> <code>\vbox_gset_top:Nn</code> <code>\vbox_gset_top:cn</code> <hr/>	<code>\vbox_set_top:Nn <box> {<contents>}</code> Typesets the $\langle contents \rangle$ at natural height and then stores the result inside the $\langle box \rangle$. The baseline of the box is equal to that of the <i>first</i> item added to the box.
Updated: 2017-04-05 <hr/>	
<hr/> <code>\vbox_set_to_ht:Nnn</code> <code>\vbox_set_to_ht:cnn</code> <code>\vbox_gset_to_ht:Nnn</code> <code>\vbox_gset_to_ht:cnn</code> <hr/>	<code>\vbox_set_to_ht:Nnn <box> {<dimexpr>} {<contents>}</code> Typesets the $\langle contents \rangle$ to the height given by the $\langle dimexpr \rangle$ and then stores the result inside the $\langle box \rangle$.
Updated: 2017-04-05 <hr/>	
<hr/> <code>\vbox_set:Nw</code> <code>\vbox_set:cw</code> <code>\vbox_set_end:</code> <code>\vbox_gset:Nw</code> <code>\vbox_gset:cw</code> <code>\vbox_gset_end:</code> <hr/>	<code>\vbox_set:Nw <box> <contents> \vbox_set_end:</code> Typesets the $\langle contents \rangle$ at natural height and then stores the result inside the $\langle box \rangle$. In contrast to <code>\vbox_set:Nn</code> this function does not absorb the argument when finding the $\langle content \rangle$, and so can be used in circumstances where the $\langle content \rangle$ may not be a simple argument.
Updated: 2017-04-05 <hr/>	

<hr/> <code>\vbox_set_to_ht:Nnw</code>	<code>\vbox_set_to_ht:Nnw <box> {<dimexpr>} <contents> \vbox_set_end:</code>
<code>\vbox_set_to_ht:cnw</code>	
<code>\vbox_gset_to_ht:Nnw</code>	Typesets the $\langle contents \rangle$ to the height given by the $\langle dimexpr \rangle$ and then stores the result inside the $\langle box \rangle$. In contrast to <code>\vbox_set_to_ht:Nnn</code> this function does not absorb the argument when finding the $\langle content \rangle$, and so can be used in circumstances where the $\langle content \rangle$ may not be a simple argument
<code>\vbox_gset_to_ht:cnw</code>	
<hr/> New: 2017-06-08 <hr/>	

<hr/> <code>\vbox_set_split_to_ht:NNn</code>	<code>\vbox_set_split_to_ht:NNn <box₁> <box₂> {<dimexpr>}</code>
<code>\vbox_set_split_to_ht:(cNn Ncn ccn)</code>	
<code>\vbox_gset_split_to_ht:NNn</code>	
<code>\vbox_gset_split_to_ht:(cNn Ncn ccn)</code>	
<hr/> Updated: 2018-12-29 <hr/>	

Sets $\langle box_1 \rangle$ to contain material to the height given by the $\langle dimexpr \rangle$ by removing content from the top of $\langle box_2 \rangle$ (which must be a vertical box).

<hr/> <code>\vbox_unpack:N</code>	<code>\vbox_unpack:N <box></code>
<code>\vbox_unpack:c</code>	Unpacks the content of the vertical $\langle box \rangle$, retaining any stretching or shrinking applied when the $\langle box \rangle$ was set.

TeXhackers note: This is the TeX primitive `\unvcopy`.

33.12 Using boxes efficiently

The functions above for using box contents work in exactly the same way as for any other `expl3` variable. However, for efficiency reasons, it is also useful to have functions which *drop* box contents on use. When a box is dropped, the box becomes empty at the group level *where the box was originally set* rather than necessarily *at the current group level*. For example, with

```
\hbox_set:Nn \l_tmpa_box { A }
\group_begin:
  \hbox_set:Nn \l_tmpa_box { B }
  \group_begin:
    \box_use_drop:N \l_tmpa_box
  \group_end:
  \box_show:N \l_tmpa_box
\group_end:
\box_show:N \l_tmpa_box
```

the first use of `\box_show:N` will show an entirely cleared (void) box, and the second will show the letter A in the box.

These functions should be preferred when the content of the box is no longer required after use. Note that due to the unusual scoping behaviour of **drop** functions they may be applied to both local and global boxes: the latter will naturally be set and thus cleared at a global level.

`\box_use_drop:N`
`\box_use_drop:c`

`\box_use_drop:N` $\langle box \rangle$

Inserts the current content of the $\langle box \rangle$ onto the current list for typesetting then drops the box content. An error is raised if the variable does not exist or if it is invalid. This function may be applied to local or global boxes.

TeXhackers note: This is the `\box` primitive.

`\box_set_eq_drop:NN`
`\box_set_eq_drop:(cN|Nc|cc)`

New: 2019-01-17

`\box_set_eq_drop:NN` $\langle box_1 \rangle$ $\langle box_2 \rangle$

Sets the content of $\langle box_1 \rangle$ equal to that of $\langle box_2 \rangle$, then drops $\langle box_2 \rangle$.

`\box_gset_eq_drop:NN`
`\box_gset_eq_drop:(cN|Nc|cc)`

New: 2019-01-17

`\box_gset_eq_drop:NN` $\langle box_1 \rangle$ $\langle box_2 \rangle$

Sets the content of $\langle box_1 \rangle$ globally equal to that of $\langle box_2 \rangle$, then drops $\langle box_2 \rangle$.

`\hbox_unpack_drop:N`
`\hbox_unpack_drop:c`

New: 2019-01-17

`\hbox_unpack_drop:N` $\langle box \rangle$

Unpacks the content of the horizontal $\langle box \rangle$, retaining any stretching or shrinking applied when the $\langle box \rangle$ was set. The original $\langle box \rangle$ is then dropped.

TeXhackers note: This is the TeX primitive `\unhbox`.

`\vbox_unpack_drop:N`
`\vbox_unpack_drop:c`

New: 2019-01-17

`\vbox_unpack_drop:N` $\langle box \rangle$

Unpacks the content of the vertical $\langle box \rangle$, retaining any stretching or shrinking applied when the $\langle box \rangle$ was set. The original $\langle box \rangle$ is then dropped.

TeXhackers note: This is the TeX primitive `\unvbox`.

33.13 Affine transformations

Affine transformations are changes which (informally) preserve straight lines. Simple translations are affine transformations, but are better handled in TeX by doing the translation first, then inserting an unmodified box. On the other hand, rotation and resizing of boxed material can best be handled by modifying boxes. These transformations are described here.

<code>\box_autosize_to_wd_and_ht:Nnn</code>	<code>\box_autosize_to_wd_and_ht:Nnn <box> {<x-size>} {<y-size>}</code>
<code>\box_autosize_to_wd_and_ht:cnn</code>	
<code>\box_gautosize_to_wd_and_ht:Nnn</code>	
<code>\box_gautosize_to_wd_and_ht:cnn</code>	

New: 2017-04-04

Updated: 2019-01-22

Resizes the $\langle box \rangle$ to fit within the given $\langle x-size \rangle$ (horizontally) and $\langle y-size \rangle$ (vertically); both of the sizes are dimension expressions. The $\langle y-size \rangle$ is the height only: it does not include any depth. The updated $\langle box \rangle$ is an `hbox`, irrespective of the nature of the $\langle box \rangle$ before the resizing is applied. The final size of the $\langle box \rangle$ is the smaller of $\{ \langle x-size \rangle \}$ and $\{ \langle y-size \rangle \}$, *i.e.* the result fits within the dimensions specified. Negative sizes cause the material in the $\langle box \rangle$ to be reversed in direction, but the reference point of the $\langle box \rangle$ is unchanged. Thus a negative $\langle y-size \rangle$ results in the $\langle box \rangle$ having a depth dependent on the height of the original and *vice versa*.

<code>\box_autosize_to_wd_and_ht_plus_dp:Nnn</code>	<code>\box_autosize_to_wd_and_ht_plus_dp:Nnn <box> {<x-size>}</code>
<code>\box_autosize_to_wd_and_ht_plus_dp:cnn</code>	<code>{<y-size>}</code>
<code>\box_gautosize_to_wd_and_ht_plus_dp:Nnn</code>	
<code>\box_gautosize_to_wd_and_ht_plus_dp:cnn</code>	

New: 2017-04-04

Updated: 2019-01-22

Resizes the $\langle box \rangle$ to fit within the given $\langle x-size \rangle$ (horizontally) and $\langle y-size \rangle$ (vertically); both of the sizes are dimension expressions. The $\langle y-size \rangle$ is the total vertical size (height plus depth). The updated $\langle box \rangle$ is an `hbox`, irrespective of the nature of the $\langle box \rangle$ before the resizing is applied. The final size of the $\langle box \rangle$ is the smaller of $\{ \langle x-size \rangle \}$ and $\{ \langle y-size \rangle \}$, *i.e.* the result fits within the dimensions specified. Negative sizes cause the material in the $\langle box \rangle$ to be reversed in direction, but the reference point of the $\langle box \rangle$ is unchanged. Thus a negative $\langle y-size \rangle$ results in the $\langle box \rangle$ having a depth dependent on the height of the original and *vice versa*.

<code>\box_resize_to_ht:Nn</code>	<code>\box_resize_to_ht:Nn <box> {<y-size>}</code>
<code>\box_resize_to_ht:cn</code>	
<code>\box_gresize_to_ht:Nn</code>	
<code>\box_gresize_to_ht:cn</code>	

Updated: 2019-01-22

Resizes the $\langle box \rangle$ to $\langle y-size \rangle$ (vertically), scaling the horizontal size by the same amount; $\langle y-size \rangle$ is a dimension expression. The $\langle y-size \rangle$ is the height only: it does not include any depth. The updated $\langle box \rangle$ is an `hbox`, irrespective of the nature of the $\langle box \rangle$ before the resizing is applied. A negative $\langle y-size \rangle$ causes the material in the $\langle box \rangle$ to be reversed in direction, but the reference point of the $\langle box \rangle$ is unchanged. Thus a negative $\langle y-size \rangle$ results in the $\langle box \rangle$ having a depth dependent on the height of the original and *vice versa*.

<code>\box_resize_to_ht_plus_dp:Nn</code>	<code>\box_resize_to_ht_plus_dp:Nn <box> {<y-size>}</code>
<code>\box_resize_to_ht_plus_dp:cn</code>	
<code>\box_gresize_to_ht_plus_dp:Nn</code>	
<code>\box_gresize_to_ht_plus_dp:cn</code>	

Updated: 2019-01-22

Resizes the $\langle box \rangle$ to $\langle y-size \rangle$ (vertically), scaling the horizontal size by the same amount; $\langle y-size \rangle$ is a dimension expression. The $\langle y-size \rangle$ is the total vertical size (height plus depth). The updated $\langle box \rangle$ is an `hbox`, irrespective of the nature of the $\langle box \rangle$ before the resizing is applied. A negative $\langle y-size \rangle$ causes the material in the $\langle box \rangle$ to be reversed in direction, but the reference point of the $\langle box \rangle$ is unchanged. Thus a negative $\langle y-size \rangle$ results in the $\langle box \rangle$ having a depth dependent on the height of the original and *vice versa*.

<code>\box_resize_to_wd:Nn</code>	<code>\box_resize_to_wd:Nn <box> {<x-size>}</code>
<code>\box_resize_to_wd:cn</code>	
<code>\box_gresize_to_wd:Nn</code>	
<code>\box_gresize_to_wd:cn</code>	

Updated: 2019-01-22

Resizes the $\langle box \rangle$ to $\langle x-size \rangle$ (horizontally), scaling the vertical size by the same amount; $\langle x-size \rangle$ is a dimension expression. The updated $\langle box \rangle$ is an `hbox`, irrespective of the nature of the $\langle box \rangle$ before the resizing is applied. A negative $\langle x-size \rangle$ causes the material in the $\langle box \rangle$ to be reversed in direction, but the reference point of the $\langle box \rangle$ is unchanged. Thus a negative $\langle x-size \rangle$ results in the $\langle box \rangle$ having a depth dependent on the height of the original and *vice versa*.

<code>\box_resize_to_wd_and_ht:Nnn</code>	<code>\box_resize_to_wd_and_ht:Nnn <box> {<x-size>} {<y-size>}</code>
<code>\box_resize_to_wd_and_ht:cnn</code>	
<code>\box_gresize_to_wd_and_ht:Nnn</code>	
<code>\box_gresize_to_wd_and_ht:cnn</code>	

New: 2014-07-03

Updated: 2019-01-22

Resizes the $\langle box \rangle$ to $\langle x-size \rangle$ (horizontally) and $\langle y-size \rangle$ (vertically): both of the sizes are dimension expressions. The $\langle y-size \rangle$ is the height only and does not include any depth. The updated $\langle box \rangle$ is an `hbox`, irrespective of the nature of the $\langle box \rangle$ before the resizing is applied. Negative sizes cause the material in the $\langle box \rangle$ to be reversed in direction, but the reference point of the $\langle box \rangle$ is unchanged. Thus a negative $\langle y-size \rangle$ results in the $\langle box \rangle$ having a depth dependent on the height of the original and *vice versa*.

<code>\box_resize_to_wd_and_ht_plus_dp:Nnn</code>	<code>\box_resize_to_wd_and_ht_plus_dp:Nnn <box> {<x-size>} {<y-size>}</code>
<code>\box_resize_to_wd_and_ht_plus_dp:cnn</code>	
<code>\box_gresize_to_wd_and_ht_plus_dp:Nnn</code>	
<code>\box_gresize_to_wd_and_ht_plus_dp:cnn</code>	

New: 2017-04-06

Updated: 2019-01-22

Resizes the $\langle box \rangle$ to $\langle x-size \rangle$ (horizontally) and $\langle y-size \rangle$ (vertically): both of the sizes are dimension expressions. The $\langle y-size \rangle$ is the total vertical size (height plus depth). The updated $\langle box \rangle$ is an `hbox`, irrespective of the nature of the $\langle box \rangle$ before the resizing is applied. Negative sizes cause the material in the $\langle box \rangle$ to be reversed in direction, but the reference point of the $\langle box \rangle$ is unchanged. Thus a negative $\langle y-size \rangle$ results in the $\langle box \rangle$ having a depth dependent on the height of the original and *vice versa*.

<code>\box_rotate:Nn</code>	<code>\box_rotate:Nn <box> {<angle>}</code>
<code>\box_rotate:cn</code>	
<code>\box_grotate:Nn</code>	Rotates the $\langle box \rangle$ by $\langle angle \rangle$ (in degrees) anti-clockwise about its reference point. The reference point of the updated box is moved horizontally such that it is at the left side of the smallest rectangle enclosing the rotated material. The updated $\langle box \rangle$ is an <code>hbox</code> , irrespective of the nature of the $\langle box \rangle$ before the rotation is applied.
<code>\box_grotate:cn</code>	
<hr/> Updated: 2019-01-22 <hr/>	

<code>\box_scale:Nnn</code>	<code>\box_scale:Nnn <box> {<x-scale>} {<y-scale>}</code>
<code>\box_scale:cnn</code>	
<code>\box_gscale:Nnn</code>	Scales the $\langle box \rangle$ by factors $\langle x-scale \rangle$ and $\langle y-scale \rangle$ in the horizontal and vertical directions, respectively (both scales are integer expressions). The updated $\langle box \rangle$ is an <code>hbox</code> , irrespective of the nature of the $\langle box \rangle$ before the scaling is applied. Negative scalings cause the material in the $\langle box \rangle$ to be reversed in direction, but the reference point of the $\langle box \rangle$ is unchanged. Thus a negative $\langle y-scale \rangle$ results in the $\langle box \rangle$ having a depth dependent on the height of the original and <i>vice versa</i> .
<code>\box_gscale:cnn</code>	
<hr/> Updated: 2019-01-22 <hr/>	

33.14 Primitive box conditionals

<code>\if_hbox:N *</code>	<code>\if_hbox:N <box></code> <code> <true code></code> <code>\else:</code> <code> <false code></code> <code>\fi:</code>
	Tests if $\langle box \rangle$ is a horizontal box.

T_EXhackers note: This is the T_EX primitive `\ifhbox`.

<code>\if_vbox:N *</code>	<code>\if_vbox:N <box></code> <code> <true code></code> <code>\else:</code> <code> <false code></code> <code>\fi:</code>
	Tests if $\langle box \rangle$ is a vertical box.

T_EXhackers note: This is the T_EX primitive `\ifvbox`.

<code>\if_box_empty:N *</code>	<code>\if_box_empty:N <box></code> <code> <true code></code> <code>\else:</code> <code> <false code></code> <code>\fi:</code>
	Tests if $\langle box \rangle$ is an empty (void) box.

T_EXhackers note: This is the T_EX primitive `\ifvoid`.

Chapter 34

The l3coffins package

Coffin code layer

The material in this module provides the low-level support system for coffins. For details about the design concept of a coffin, see the xcoffins module (in the l3experimental bundle).

34.1 Creating and initialising coffins

<code>\coffin_new:N</code>
<code>\coffin_new:c</code>
<code>New: 2011-08-17</code>

`\coffin_new:N` $\langle coffin \rangle$

Creates a new $\langle coffin \rangle$ or raises an error if the name is already taken. The declaration is global. The $\langle coffin \rangle$ is initially empty.

<code>\coffin_clear:N</code>
<code>\coffin_clear:c</code>
<code>\coffin_gclear:N</code>
<code>\coffin_gclear:c</code>
<code>New: 2011-08-17</code>
<code>Updated: 2019-01-21</code>

`\coffin_clear:N` $\langle coffin \rangle$

Clears the content of the $\langle coffin \rangle$.

<code>\coffin_set_eq:NN</code>
<code>\coffin_set_eq:(Nc cN cc)</code>
<code>\coffin_gset_eq:NN</code>
<code>\coffin_gset_eq:(Nc cN cc)</code>
<code>New: 2011-08-17</code>
<code>Updated: 2019-01-21</code>

`\coffin_set_eq:NN` $\langle coffin_1 \rangle$ $\langle coffin_2 \rangle$

Sets both the content and poles of $\langle coffin_1 \rangle$ equal to those of $\langle coffin_2 \rangle$.

<code>\coffin_if_exist_p:N *</code>
<code>\coffin_if_exist_p:c *</code>
<code>\coffin_if_exist:NTF *</code>
<code>\coffin_if_exist:cTF *</code>
<code>New: 2012-06-20</code>

`\coffin_if_exist_p:N` $\langle coffin \rangle$

`\coffin_if_exist:NTF` $\langle coffin \rangle$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$

Tests whether the $\langle coffin \rangle$ is currently defined.

34.2 Setting coffin content and poles

```
\hcoffin_set:Nn
\hcoffin_set:cn
\hcoffin_gset:Nn
\hcoffin_gset:cn
```

New: 2011-08-17
Updated: 2019-01-21

```
\hcoffin_set:Nn <coffin> {\material}
```

Typesets the $\langle material \rangle$ in horizontal mode, storing the result in the $\langle coffin \rangle$. The standard poles for the $\langle coffin \rangle$ are then set up based on the size of the typeset material.

```
\hcoffin_set:Nw
\hcoffin_set:cw
\hcoffin_set_end:
\hcoffin_gset:Nw
\hcoffin_gset:cw
\hcoffin_gset_end:
```

New: 2011-09-10
Updated: 2019-01-21

```
\hcoffin_set:Nw <coffin> <material> \hcoffin_set_end:
```

Typesets the $\langle material \rangle$ in horizontal mode, storing the result in the $\langle coffin \rangle$. The standard poles for the $\langle coffin \rangle$ are then set up based on the size of the typeset material. These functions are useful for setting the entire contents of an environment in a coffin.

```
\vcoffin_set:Nnn
\vcoffin_set:cnn
\vcoffin_gset:Nnn
\vcoffin_gset:cnn
```

New: 2011-08-17
Updated: 2019-01-21

```
\vcoffin_set:Nnn <coffin> {\width} {\material}
```

Typesets the $\langle material \rangle$ in vertical mode constrained to the given $\langle width \rangle$ and stores the result in the $\langle coffin \rangle$. The standard poles for the $\langle coffin \rangle$ are then set up based on the size of the typeset material.

```
\vcoffin_set:Nnw
\vcoffin_set:cnw
\vcoffin_set_end:
\vcoffin_gset:Nnw
\vcoffin_gset:cnw
\vcoffin_gset_end:
```

New: 2011-09-10
Updated: 2019-01-21

```
\vcoffin_set:Nnw <coffin> {\width} <material> \vcoffin_set_end:
```

Typesets the $\langle material \rangle$ in vertical mode constrained to the given $\langle width \rangle$ and stores the result in the $\langle coffin \rangle$. The standard poles for the $\langle coffin \rangle$ are then set up based on the size of the typeset material. These functions are useful for setting the entire contents of an environment in a coffin.

```
\coffin_set_horizontal_pole:Nnn
\coffin_set_horizontal_pole:cnn
\coffin_gset_horizontal_pole:Nnn
\coffin_gset_horizontal_pole:cnn
```

New: 2012-07-20
Updated: 2019-01-21

```
\coffin_set_horizontal_pole:Nnn <coffin>
{\pole} {\offset}
```

Sets the $\langle pole \rangle$ to run horizontally through the $\langle coffin \rangle$. The $\langle pole \rangle$ is placed at the $\langle offset \rangle$ from the baseline of the $\langle coffin \rangle$. The $\langle offset \rangle$ should be given as a dimension expression.

<code>\coffin_set_vertical_pole:Nnn</code>	<code>\coffin_set_vertical_pole:Nnn <coffin> {<pole>} {<offset>}</code>
<code>\coffin_set_vertical_pole:cnn</code>	
<code>\coffin_gset_vertical_pole:Nnn</code>	
<code>\coffin_gset_vertical_pole:cnn</code>	

New: 2012-07-20

Updated: 2019-01-21

Sets the $\langle pole \rangle$ to run vertically through the $\langle coffin \rangle$. The $\langle pole \rangle$ is placed at the $\langle offset \rangle$ from the left-hand edge of the bounding box of the $\langle coffin \rangle$. The $\langle offset \rangle$ should be given as a dimension expression.

34.3 Coffin affine transformations

<code>\coffin_resize:Nnn</code>	<code>\coffin_resize:Nnn <coffin> {<width>} {<total-height>}</code>
<code>\coffin_resize:cnn</code>	
<code>\coffin_gresize:Nnn</code>	
<code>\coffin_gresize:cnn</code>	

Updated: 2019-01-23

Resized the $\langle coffin \rangle$ to $\langle width \rangle$ and $\langle total-height \rangle$, both of which should be given as dimension expressions.

<code>\coffin_rotate:Nn</code>	<code>\coffin_rotate:Nn <coffin> {<angle>}</code>
<code>\coffin_rotate:cn</code>	
<code>\coffin_grotate:Nn</code>	
<code>\coffin_grotate:cn</code>	

Rotates the $\langle coffin \rangle$ by the given $\langle angle \rangle$ (given in degrees counter-clockwise). This process rotates both the coffin content and poles. Multiple rotations do not result in the bounding box of the coffin growing unnecessarily.

<code>\coffin_scale:Nnn</code>	<code>\coffin_scale:Nnn <coffin> {<x-scale>} {<y-scale>}</code>
<code>\coffin_scale:cnn</code>	
<code>\coffin_gscale:Nnn</code>	
<code>\coffin_gscale:cnn</code>	

Updated: 2019-01-23

Scales the $\langle coffin \rangle$ by a factors $\langle x-scale \rangle$ and $\langle y-scale \rangle$ in the horizontal and vertical directions, respectively. The two scale factors should be given as real numbers.

34.4 Joining and using coffins

<code>\coffin_attach:NnnNnnnn</code>	<code>\coffin_attach:NnnNnnnn</code>
<code>\coffin_attach:(cnnNnnnn Nnnncnnnn cnnncnnnn)</code>	<code><coffin₁> {<coffin₁-pole₁>} {<coffin₁-pole₂>}</code>
<code>\coffin_gattach:NnnNnnnn</code>	<code><coffin₂> {<coffin₂-pole₁>} {<coffin₂-pole₂>}</code>
<code>\coffin_gattach:(cnnNnnnn Nnnncnnnn cnnncnnnn)</code>	<code>{<x-offset>} {<y-offset>}</code>

Updated: 2019-01-22

This function attaches $\langle coffin_2 \rangle$ to $\langle coffin_1 \rangle$ such that the bounding box of $\langle coffin_1 \rangle$ is not altered, *i.e.* $\langle coffin_2 \rangle$ can protrude outside of the bounding box of the coffin. The alignment is carried out by first calculating $\langle handle_1 \rangle$, the point of intersection of $\langle coffin_1-pole_1 \rangle$ and $\langle coffin_1-pole_2 \rangle$, and $\langle handle_2 \rangle$, the point of intersection of $\langle coffin_2-pole_1 \rangle$ and $\langle coffin_2-pole_2 \rangle$. $\langle coffin_2 \rangle$ is then attached to $\langle coffin_1 \rangle$ such that the relationship between $\langle handle_1 \rangle$ and $\langle handle_2 \rangle$ is described by the $\langle x-offset \rangle$ and $\langle y-offset \rangle$. The two offsets should be given as dimension expressions.

<hr/>	
<code>\coffin_join:NnnNnnnn</code>	<code>\coffin_join:NnnNnnnn</code>
<code>\coffin_join:(cnnNnnnn Nnnncnnnn cnnncnnnn)</code>	<code>\coffin_1 {<coffin_1-pole_1>} {<coffin_1-pole_2>}</code>
<code>\coffin_gjoin:NnnNnnnn</code>	<code>\coffin_2 {<coffin_2-pole_1>} {<coffin_2-pole_2>}</code>
<code>\coffin_gjoin:(cnnNnnnn Nnnncnnnn cnnncnnnn)</code>	<code>{<x-offset>} {<y-offset>}</code>

Updated: 2019-01-22

This function joins $\langle coffin_2 \rangle$ to $\langle coffin_1 \rangle$ such that the bounding box of $\langle coffin_1 \rangle$ may expand. The new bounding box covers the area containing the bounding boxes of the two original coffins. The alignment is carried out by first calculating $\langle handle_1 \rangle$, the point of intersection of $\langle coffin_1-pole_1 \rangle$ and $\langle coffin_1-pole_2 \rangle$, and $\langle handle_2 \rangle$, the point of intersection of $\langle coffin_2-pole_1 \rangle$ and $\langle coffin_2-pole_2 \rangle$. $\langle coffin_2 \rangle$ is then attached to $\langle coffin_1 \rangle$ such that the relationship between $\langle handle_1 \rangle$ and $\langle handle_2 \rangle$ is described by the $\langle x-offset \rangle$ and $\langle y-offset \rangle$. The two offsets should be given as dimension expressions.

<hr/>	
<code>\coffin_typeset:Nnnnn</code>	<code>\coffin_typeset:Nnnnn <coffin> {<pole_1>} {<pole_2>}</code>
<code>\coffin_typeset:cnnnn</code>	<code>{<x-offset>} {<y-offset>}</code>

Updated: 2012-07-20

Typesetting is carried out by first calculating $\langle handle \rangle$, the point of intersection of $\langle pole_1 \rangle$ and $\langle pole_2 \rangle$. The coffin is then typeset in horizontal mode such that the relationship between the current reference point in the document and the $\langle handle \rangle$ is described by the $\langle x-offset \rangle$ and $\langle y-offset \rangle$. The two offsets should be given as dimension expressions. Typesetting a coffin is therefore analogous to carrying out an alignment where the “parent” coffin is the current insertion point.

34.5 Measuring coffins

<hr/>	
<code>\coffin_dp:N</code>	<code>\coffin_dp:N <coffin></code>
<code>\coffin_dp:c</code>	Calculates the depth (below the baseline) of the $\langle coffin \rangle$ in a form suitable for use in a $\langle dimension expression \rangle$.
<hr/>	
<code>\coffin_ht:N</code>	<code>\coffin_ht:N <coffin></code>
<code>\coffin_ht:c</code>	Calculates the height (above the baseline) of the $\langle coffin \rangle$ in a form suitable for use in a $\langle dimension expression \rangle$.
<hr/>	
<code>\coffin_wd:N</code>	<code>\coffin_wd:N <coffin></code>
<code>\coffin_wd:c</code>	Calculates the width of the $\langle coffin \rangle$ in a form suitable for use in a $\langle dimension expression \rangle$.

34.6 Coffin diagnostics

<hr/>	
<code>\coffin_display_handles:Nn</code>	<code>\coffin_display_handles:Nn <coffin> {<color>}</code>
<code>\coffin_display_handles:cn</code>	

Updated: 2011-09-02

This function first calculates the intersections between all of the $\langle poles \rangle$ of the $\langle coffin \rangle$ to give a set of $\langle handles \rangle$. It then prints the $\langle coffin \rangle$ at the current location in the source, with the position of the $\langle handles \rangle$ marked on the coffin. The $\langle handles \rangle$ are labelled as part of this process: the locations of the $\langle handles \rangle$ and the labels are both printed in the $\langle color \rangle$ specified.

<hr/> <code>\coffin_mark_handle:Nnnn</code> <code>\coffin_mark_handle:cnnn</code> <hr/> Updated: 2011-09-02 <hr/>	<code>\coffin_mark_handle:Nnnn <coffin> {<pole₁>} {<pole₂>} {<color>}</code> <p>This function first calculates the $\langle handle \rangle$ for the $\langle coffin \rangle$ as defined by the intersection of $\langle pole_1 \rangle$ and $\langle pole_2 \rangle$. It then marks the position of the $\langle handle \rangle$ on the $\langle coffin \rangle$. The $\langle handle \rangle$ are labelled as part of this process: the location of the $\langle handle \rangle$ and the label are both printed in the $\langle color \rangle$ specified.</p>
<hr/> <code>\coffin_show_structure:N</code> <code>\coffin_show_structure:c</code> <hr/> Updated: 2015-08-01 <hr/>	<code>\coffin_show_structure:N <coffin></code> <p>This function shows the structural information about the $\langle coffin \rangle$ in the terminal. The width, height and depth of the typeset material are given, along with the location of all of the poles of the coffin.</p> <p>Notice that the poles of a coffin are defined by four values: the x and y co-ordinates of a point that the pole passes through and the x- and y-components of a vector denoting the direction of the pole. It is the ratio between the later, rather than the absolute values, which determines the direction of the pole.</p>
<hr/> <code>\coffin_log_structure:N</code> <code>\coffin_log_structure:c</code> <hr/> New: 2014-08-22 Updated: 2015-08-01 <hr/>	<code>\coffin_log_structure:N <coffin></code> <p>This function writes the structural information about the $\langle coffin \rangle$ in the log file. See also <code>\coffin_show_structure:N</code> which displays the result in the terminal.</p>
<hr/> <code>\coffin_show:N</code> <code>\coffin_show:c</code> <code>\coffin_log:N</code> <code>\coffin_log:c</code> <hr/> New: 2021-05-11 <hr/>	<code>\coffin_show:N <coffin></code> <code>\coffin_log:N <coffin></code> <p>Shows full details of poles and contents of the $\langle coffin \rangle$ in the terminal or log file. See <code>\coffin_show_structure:N</code> and <code>\box_show:N</code> to show separately the pole structure and the contents.</p>
<hr/> <code>\coffin_show:Nnn</code> <code>\coffin_show:cnn</code> <code>\coffin_log:Nnn</code> <code>\coffin_log:cnn</code> <hr/> New: 2021-05-11 <hr/>	<code>\coffin_show:Nnn <coffin> {<intexpr₁>} {<intexpr₂>}</code> <code>\coffin_log:Nnn <coffin> {<intexpr₁>} {<intexpr₂>}</code> <p>Shows poles and contents of the $\langle coffin \rangle$ in the terminal or log file, showing the first $\langle intexpr_1 \rangle$ items in the coffin, and descending into $\langle intexpr_2 \rangle$ group levels. See <code>\coffin_show_structure:N</code> and <code>\box_show:Nnn</code> to show separately the pole structure and the contents.</p>

34.7 Constants and variables

<hr/> <code>\c_empty_coffin</code> <hr/>	A permanently empty coffin.
<hr/> <code>\l_tmpa_coffin</code> <code>\l_tmpb_coffin</code> <hr/> New: 2012-06-19 <hr/>	Scratch coffins for local assignment. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

`\g_tmpa_coffin`
`\g_tmpb_coffin`

New: 2019-01-24

Scratch coffins for global assignment. These are never used by the kernel code, and so are safe for use with any L^AT_EX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

Chapter 35

The l3color package

Color support

35.1 Color in boxes

Controlling the color of text in boxes requires a small number of control functions, so that the boxed material uses the color at the point where it is set, rather than where it is used.

```
\color_group_begin:  
\color_group_end:
```

New: 2011-09-03

```
\color_group_begin:  
...  
\color_group_end:
```

Creates a color group: one used to “trap” color settings. This grouping is built in to for example `\hbox_set:Nn`.

```
\color_ensure_current:
```

New: 2011-09-03

```
\color_ensure_current:
```

Ensures that material inside a box uses the foreground color at the point where the box is set, rather than that in force when the box is used. This function should usually be used within a `\color_group_begin: ... \color_group_end: group`.

35.2 Color models

A color *model* is a way to represent sets of colors. Different models are particularly suitable for different output methods, *e.g.* screen or print. Parameter-based models can describe a very large number of unique colors, and have a varying number of *axes* which define a color space. In contrast, various proprietary models are available which define *spot* colors (more formally separations).

Core models are used to pass color information to output; these are “native” to l3color. Core models use real numbers in the range $[0, 1]$ to represent values. The core models supported here are

- **gray** Grayscale color, with a single axis running from 0 (fully black) to 1 (fully white)
- **rgb** Red-green-blue color, with three axes, one for each of the components

- **cmk** Cyan-magenta-yellow-black color, with four axes, one for each of the components

There are also interface models: these are convenient for users but have to be manipulated before storing/passing to the backend. Interface models are primarily integer-based: see below for more detail. The supported interface models are

- **Gray** Grayscale color, with a single axis running from 0 (fully black) to 15 (fully white)
- **hsb** Hue-saturation-brightness color, with three axes, all real values in the range $[0, 1]$ for hue saturation and brightness
- **Hsb** Hue-saturation-brightness color, with three axes, integer in the range $[0, 360]$ for hue, real values in the range $[0, 1]$ for saturation and brightness
- **HSB** Hue-saturation-brightness color, with three axes, integers in the range $[0, 240]$ for hue, saturation and brightness
- **HTML** HTML format representation of RGB color given as a single six-digit hexadecimal number
- **RGB** Red-green-blue color, with three axes, one for each of the components, values as integers from 0 to 255
- **wave** Light wavelength, a real number in the range 380 to 780 (nanometres)

All interface models are internally stored as **rgb**.

To allow parsing of data from **xcolor**, any leading model up the first `:` will be discarded; the approach of selecting an internal form for data is *not* used in **l3color**.

Additional models may be created to allow mixing of separation colors with each other or with those from other models. See Section 35.9 for more detail of color support for additional models.

When color is selected by model, the *⟨values⟩* given are specified as a comma-separated list. The length of the list will therefore be determined by the detail of the model involved.

Color models (and interconversion) are complex, and more details are given in the manual to the L^AT_EX 2_ε **xcolor** package and in the *PostScript Language Reference Manual*, published by Addison–Wesley.

35.3 Color expressions

In addition to allowing specification of color by model and values, **l3color** also supports color expressions. These are created by combining one or more color names, with the amount of each specified as a percentage. The latter is given between `!` symbols in the expression. Thus for example

```
red!50!green
```

is a mixture of 50 % red and 50 % green. A trailing percentage is interpreted as implicitly followed by **white**, and so

```
red!25
```

specifies 25 % red mixed with 75 % white.

Where the models for the mixed colors are different, the model of the first color is used. Thus

```
red!50!cyan
```

will result in a color specification using the `rgb` model, made up of 50 % red and 50 % of cyan *expressed in `rgb`*. This may be important as color model interconversion is not exact.

The one exception to the above is where the first model in an expression is `gray`. In this case, the order of mixing is “swapped” internally, so that for example

```
black!50!red
```

has the same result as

```
red!50!black
```

(the predefined colors `black` and `white` use the `gray` model).

Where more than two colors are mixed in an expression, evaluation takes place in a stepwise fashion. Thus in

```
cyan!50!magenta!10!yellow
```

the sub-expression

```
cyan!50!magenta
```

is first evaluated to give an intermediate color specification, before the second step

```
<intermediate>!10!yellow
```

where `<intermediate>` represents this transitory calculated value.

Within a color expression, `.` may be used to represent the color active for typesetting (the current color). This allows for example

```
.!50
```

to mean a mixture of 50 % of current color with white.

(Color expressions supported here are a subset of those provided by the L^AT_EX 2_ε `xcolor` package. At present, only such features as are clearly useful have been added here.)

35.4 Named colors

Color names are stored in a single namespace, which makes them accessible as part of color expressions. Whilst they are not reserved in a technical sense, the names `black`, `white`, `red`, `green`, `blue`, `cyan`, `magenta` and `yellow` have special meaning and should not be redefined. Color names should be made up of letters, numbers and spaces only: other characters are reserved for use in color expressions. In particular, `.` represents the current color at the start of a color expression.

<code>\color_set:nn</code>	<code>\color_set:nn {<name>} {<color expression>}</code>
----------------------------	--

Evaluates the *<color expression>* and stores the resulting color specification as the *<name>*.

<hr/> <hr/>	<code>\color_set:nnn</code>	<code>{\langle name \rangle}{\langle model(s) \rangle}{\langle value(s) \rangle}</code>
		Stores the color specification equivalent to the $\langle model(s) \rangle$ and $\langle values \rangle$ as the $\langle name \rangle$.
<hr/> <hr/>	<code>\color_set_eq:nn</code>	<code>{\langle name1 \rangle}{\langle name2 \rangle}</code>
		Copies the color specification in $\langle name2 \rangle$ to $\langle name1 \rangle$. The special name <code>.</code> may be used to represent the current color, allowing it to be saved to a name.
<hr/> <hr/>	<code>\color_show:n</code> <code>\color_log:n</code>	<code>{\langle name \rangle}</code>
<hr/> <hr/>	New: 2021-05-11	Displays the color specification stored in the $\langle name \rangle$ on the terminal or log file.

35.5 Selecting colors

General selection of color is safe when split across pages: a stack is used to ensure that the correct color is re-selected on the new page.

These commands set the current color (`.`): other more specialised functions such as fill and stroke selectors do *not* adjust this value.

<hr/> <hr/>	<code>\color_select:n</code>	<code>{\langle color expression \rangle}</code>
		Parses the $\langle color expression \rangle$ and then activates the resulting color specification for typeset material.
<hr/> <hr/>	<code>\color_select:nn</code>	<code>{\langle model(s) \rangle}{\langle value(s) \rangle}</code>
		Activates the color specification equivalent to the $\langle model(s) \rangle$ and $\langle value(s) \rangle$ for typeset material.
<hr/> <hr/>	<code>\l_color_fixed_model_tl</code>	
		When this is set to a non-empty value, colors will be converted to the specified model when they are selected. Note that included images and similar are not influenced by this setting.

35.6 Colors for fills and strokes

Colors for drawing operations and so forth are split into strokes and fills (the latter may also be referred to as non-stroke color). The fill color is used for text under normal circumstances. Depending on the backend, stroke color may use a *stack*, in which case it exhibits the same page breaking behavior as general color. However, `dvips`/`dvipsw` do not support this, and so color will need to be contained within a scope, such as `\draw_begin:/\draw_end:.`

<hr/> <hr/>	<code>\color_fill:n</code> <code>\color_stroke:n</code>	<code>{\langle color expression \rangle}</code>
		Parses the $\langle color expression \rangle$ and then activates the resulting color specification for filling or stroking.

<code>\color_fill:nn</code>	<code>\color_fill:nn {<model(s)>} {<value(s)>}</code>
<code>\color_stroke:nn</code>	Activates the color specification equivalent to the <code><model(s)></code> and <code><value(s)></code> for filling or stroking.

<code>color.sc</code>	When using <code>dvips</code> , this PostScript variables hold the stroke color.
-----------------------	--

35.6.1 Coloring math mode material

Coloring math mode material using `\color_select:nn(n)` has some restrictions and often leads to spacing issues and/or poor input syntax. Avoiding generating `\mathord` atoms whilst coloring only those parts of the input which are required needs careful handling. The functionality here covers this important use case.

<code>\color_math:nn</code>	<code>\color_math:nn {<color expression>} {<content>}</code>
<code>\color_math:nnn</code>	<code>\color_math:nnn {<model(s)>} {<value(s)>} {<content>}</code>
New: 2022-01-26	Works as for <code>\color_select:n(n)</code> but applies color only to the math mode <code><content></code> . The function does not generate a group and the <code><content></code> therefore retains its math atom states. Sub/superscripts are also properly handled.

<code>\l_color_math_active_tl</code>	This list controls which tokens are considered as math active and should therefore be replaced by their definition during searching for sub/superscripts.
New: 2022-01-26	

35.7 Multiple color models

When selecting or setting a color with an explicit model, it is possible to give values for more than one model at one time. This is particularly useful where automated conversion between models does not give the desired outcome. To do this, the list of models and list of values are both subdivided using `/` characters (as for the similar function in `xcolor`). For example, to save a color with explicit `cmymk` and `rgb` values, one could use

```
\color_set:nnn { foo } { cmyk / rgb }
{ 0.1 , 0.2 , 0.3 , 0.4 / 0.1, 0.2 , 0.3 }
```

The manually-specified conversion will be used in preference to automated calculation whenever the model(s) listed are used: both in expressions and when a fixed model is active.

Similarly, the same syntax can be applied to directly selecting a color.

```
\color_select:nn { cmyk / rgb }
{ 0.1 , 0.2 , 0.3 , 0.4 / 0.1, 0.2 , 0.3 }
```

Again, this list is used when a fixed model is active: the first entry is used unless there is a fixed model matching one of the other entries.

35.8 Exporting color specifications

The major use of color expressions is in setting typesetting output, but there are other places in which some form of color information is required. These may need data in a different format or using a different model to the internal representation. Thus a set of functions are available to export colors in different formats.

Valid export targets are

- **backend** Two brace groups: the first containing the model, the second containing space-separated values appropriate for the model; this is the format required by backend functions of `expl3`
- **comma-sep-cmyk** Comma-separated cyan-magenta-yellow-black values
- **comma-sep-rgb** Comma-separated red-green-blue values suitable for use as a PDF annotation color
- **HTML** Uppercase two-digit hexadecimal values, expressing a red-green-blue color; the digits are *not* separated
- **space-sep-cmyk** Space-separated cyan-magenta-yellow-black values
- **space-sep-rgb** Space-separated red-green-blue values suitable for use as a PDF annotation color

`\color_export:nnN`

`\color_export:nnN` $\{\langle color\ expression \rangle\}$ $\{\langle format \rangle\}$ $\{\langle tl \rangle\}$

Parses the $\langle color\ expression \rangle$ as described earlier, then converts to the $\langle format \rangle$ specified and assigns the data to the $\langle tl \rangle$.

`\color_export:nnnnN`

`\color_export:nnnnN` $\{\langle model \rangle\}$ $\{\langle value(s) \rangle\}$ $\{\langle format \rangle\}$ $\{\langle tl \rangle\}$

Expresses the combination of $\langle model \rangle$ and $\langle value(s) \rangle$ in an internal representation, then converts to the $\langle format \rangle$ specified and assigns the data to the $\langle tl \rangle$.

35.9 Creating new color models

Additional color models are required to support specialist workflows, for example those involving separations (see <https://helpx.adobe.com/indesign/using/spot-process-colors.html> for details of the use of separations in print). Color models may be split into families; for the standard device-based color models (`DeviceCMYK`, `DeviceRGB`, `DeviceGray`), these are synonymous. This is not generally the case: see the PDF reference for more details. (Note that `l3color` uses the shorter names `cmyk`, etc.)

<code>\color_model_new:nnn</code>	<code>\color_model_new:nnn {<model>} {<family>} {<params>}</code>
-----------------------------------	---

Creates a new *<model>* which is derived from the color model *<family>*. The latter should be one of

- **DeviceN**
- **ICCBased**
- **Separation**

(The *<family>* may be given in mixed case as-in the PDF reference: internally, case of these strings is folded.) Depending on the *<family>*, one or more *<params>* are mandatory or optional.

For a **Separation** space, there are three *compulsory* keys.

- **name** The name of the Separation, for example the formal name of a spot color ink. Such a *<name>* may contain spaces, etc., which are not permitted in the *<model>*.
- **alternative-model** An alternative device colorspace, one of **cmym**, **rgb**, **gray** or **CIELAB**. The three parameter-based models work as described above; see below for details of CIELAB colors.
- **alternative-values** A comma-separated list of values appropriate to the **alternative-model**. This information is used by the PDF application if the **Separation** is not available.

CIELAB color separations are created using the **alternative-model = CIELAB** setting. These colors must also have an **illuminant** key, one of **a**, **c**, **e**, **d50**, **d55**, **d65** or **d75**. The **alternative-values** in this case are the three parameters *L**, *a** and *b** of the CIELAB model. Full details of this device-independent color approach are given in the documentation to the **colorspace** package.

CIELAB colors *cannot* be converted into other device-dependent color spaces, and as such, mixing can only occur if colors set up using the CIELAB model are also given with an alternative parameter-based model. If that is not the case, **l3color** will fallback to using black as the colorant in any mixing.

For a **DeviceN** space, there is one *compulsory* key.

- **names** The names of the components of the **DeviceN** space. Each should be either the *<name>* of a **Separation** model, a process color name (**cyan**, etc.) or the special name **none**.

For a **ICCBased** space, there is one *compulsory* key.

- **file** The name of the file containing the profile.

35.9.1 Color profiles

Color profiles are used to ensure color accuracy by linking to collaboration. Applying a profile can be used to standardise color which is otherwise device-dependence.

<code>\color_profile_apply:nn</code>	<code>\color_profile_apply:nn {<profile>} {<model>}</code>
--------------------------------------	--

New: 2021-02-23

This function applies a *<profile>* to one of the device *<models>*. The profile will then apply to all color of the selected *<model>*. The *<profile>* should specify an ICC profile file. The *<model>* has to be one the standard device models: **cmym**, **gray** or **rgb**.

Chapter 36

The l3pdf package

Core PDF support

36.1 Objects

<code>\pdf_object_new:nn</code>	<code>\pdf_object_new:nn {<object>} {<type>}</code>
---------------------------------	---

New: 2021-02-10

Declares `<object>` as a PDF object of `<type>`, which should be one of

- `array`
- `dict`
- `fstream`
- `stream`

The object may be referenced from this point on, and written later using `\pdf_object_write:nn`.

<code>\pdf_object_if_exist_p:n *</code>	<code>\pdf_object_if_exist_p:n {<object>}</code>
<code>\pdf_object_if_exist:nTF *</code>	<code>\pdf_object_if_exist:nTF {<object>}</code>

New: 2020-05-15

Tests whether an object with name `{<object>}` has been defined.

<code>\pdf_object_write:nn</code>	<code>\pdf_object_write:nn {<object>} {<content>}</code>
<code>\pdf_object_write:nx</code>	

New: 2021-02-10

Writes the `<content>` as content of the `<object>`. Depending on the `<type>` declared for the object, the format required for the `<data>` will vary

`array` A space-separated list of values

`dict` Key-value pairs in the form `/<key> <value>`

`fstream` Two brace groups: `<file name>` and `<file content>`

`stream` Two brace groups: `<attributes (dictionary)>` and `<stream contents>`

<hr/> <code>\pdf_object_ref:n</code> *	<code>\pdf_object_ref:n {<object>}</code>
<hr/> New: 2021-02-10	Inserts the appropriate information to reference the <i><object></i> in for example page resource allocation

<hr/> <code>\pdf_object_unnamed_write:nn</code>	<code>\pdf_object_unnamed_write:nn {<type>} {<content>}</code>
<hr/> <code>\pdf_object_unnamed_write:nx</code>	
<hr/> New: 2021-02-10	

Writes the *<content>* as content of an anonymous object. Depending on the *<type>*, the format required for the *<data>* will vary

array A space-separated list of values

dict Key-value pairs in the form */<key> <value>*

fstream Two brace groups: *<attributes (dictionary)>* and *<file name>*

stream Two brace groups: *<attributes (dictionary)>* and *<stream contents>*

<hr/> <code>\pdf_object_ref_last:</code> *	<code>\pdf_object_ref_last:</code>
<hr/> New: 2021-02-10	Inserts the appropriate information to reference the last <i><object></i> created. This is particularly useful for anonymous objects.

<hr/> <code>\pdf_pageobject_ref:n</code> *	<code>\pdf_pagobject_ref:n {<pageobject>}</code>
<hr/> New: 2021-02-10	Inserts the appropriate information to reference the <i><pageobject></i> .

36.2 Version

<hr/> <code>\pdf_version_compare_p:Nn</code> *	<code>\pdf_version_compare:NnTF <comparator> {<version>} {<true code>} {<false code>}</code>
<hr/> <code>\pdf_version_compare:NnTF</code> *	
<hr/> New: 2021-02-10	

Compares the version of the PDF being created with the *<version>* string specified, using the *<comparator>*. Either the *<true code>* or *<false code>* will be left in the output stream.

<hr/> <code>\pdf_version_gset:n</code>	<code>\pdf_version_gset:n {<version>}</code>
<hr/> <code>\pdf_version_min_gset:n</code>	Sets the <i><version></i> of the PDF being created. The <i>min</i> version will not alter the output version unless it is currently lower than the <i><version></i> requested.
<hr/> New: 2021-02-10	

This function may only be used up to the point where the PDF file is initialised. With dvips it sets `\pdf_version_major:` and `\pdf_version_minor:` and allows to compare the values with `\pdf_version_compare:Nn`, but the PDF version itself still has to be set with the command line option `-dCompatibilityLevel` of `ps2pdf`.

<hr/> <code>\pdf_version:</code> *	<code>\pdf_version:</code>
<hr/> <code>\pdf_version_major:</code> *	Expands to the currently-active PDF version.
<hr/> <code>\pdf_version_minor:</code> *	
<hr/> New: 2021-02-10	

36.3 Compression

`\pdf_uncompress:`

New: 2021-02-10

`\pdf_uncompress:`

Disables any compression of the PDF, where possible.

This function may only be used up to the point where the PDF file is initialised.

36.4 Destinations

Destinations are the places a link jumped too. Unlike the name may suggest they don't described an exact location in the PDF. Instead a destination contains a reference to a page along with an instruction how to display this page. The normally used “XYZ *top left zoom*” for example instructs the viewer to show the page with the given *zoom* and the top left corner at the *top left* coordinates—which then gives the impression that there is an anchor at this position.

If an instruction takes a coordinate, it is calculated by the following commands relative to the location the command is issued. So to get a specific coordinate one has to move the command to the right place.

`\pdf_destination:nn`

New: 2021-01-03

`\pdf_destination:nn {<name>} {<type or integer>}`

This creates a destination. `{<type or integer>}` can be one of `fit`, `fith`, `fitv`, `fitb`, `fitbh`, `fitbv`, `fitr`, `xyz` or an integer representing a scale factor in percent. `fitr` here gives only a lightweight version of `/FitR`: The backend code defines `fitr` so that it will with pdfL^AT_EX and LuaL^AT_EX use the coordinates of the surrounding box, with dvips and dvipdfmx it falls back to `fit`. For full control use `\pdf_destination:nnnn`.

The keywords match to the PDF names as described in the following tabular.

Keyword	PDF	Remarks
<code>fit</code>	<code>/Fit</code>	Fits the page to the window
<code>fith</code>	<code>/FitH top</code>	Fits the width of the page to the window
<code>fitv</code>	<code>/FitV left</code>	Fits the height of the page to the window
<code>fitb</code>	<code>/FitB</code>	Fits the page bounding box to the window
<code>fitbh</code>	<code>/FitBH top</code>	Fits the width of the page bounding box to the window.
<code>fitbv</code>	<code>/FitBV left</code>	Fits the height of the page bounding box to the window.
<code>fitr</code>	<code>/FitR left bottom right top</code>	Fits the rectangle specified by the four coordinates to the window (see above for the restrictions)
<code>xyz</code>	<code>/XYZ left top null</code>	Sets a coordinate but doesn't change the zoom.
<code>{<integer>}</code>	<code>/XYZ left top zoom</code>	Sets a coordinate and a zoom meaning <code>{<integer>}%</code> .

`\pdf_destination:nnnn`

New: 2021-01-17

`\pdf_destination:nnnn {<name>} {<width>} {<height>} {<depth>}`

This creates a destination with `/FitR` type with the given dimensions relative to the current location. The destination is in a box of size zero, but it doesn't switch to horizontal mode.

Part VII
Additions and removals

Chapter 37

The l3candidates package Experimental additions to l3kernel

37.1 Important notice

This module provides a space in which functions can be added to l3kernel (expl3) while still being experimental.

As such, the functions here may not remain in their current form, or indeed at all, in l3kernel in the future.

In contrast to the material in l3experimental, the functions here are all *small* additions to the kernel. We encourage programmers to test them out and report back on the LaTeX-L mailing list.

Thus, if you intend to use any of these functions from the candidate module in a public package offered to others for productive use (e.g., being placed on CTAN) please consider the following points carefully:

- Be prepared that your public packages might require updating when such functions are being finalized.
- Consider informing us that you use a particular function in your public package, e.g., by discussing this on the LaTeX-L mailing list. This way it becomes easier to coordinate any updates necessary without issues for the users of your package.
- Discussing and understanding use cases for a particular addition or concept also helps to ensure that we provide the right interfaces in the final version so please give us feedback if you consider a certain candidate function useful (or not).

We only add functions in this space if we consider them being serious candidates for a final inclusion into the kernel. However, real use sometimes leads to better ideas, so functions from this module are **not necessarily stable** and we may have to adjust them!

37.2 Additions to l3box

```
\box_clip:N
\box_clip:c
\box_gclip:N
\box_gclip:c
```

Updated: 2019-01-23

`\box_clip:N` $\langle box \rangle$

Clips the $\langle box \rangle$ in the output so that only material inside the bounding box is displayed in the output. The updated $\langle box \rangle$ is an hbox, irrespective of the nature of the $\langle box \rangle$ before the clipping is applied.

These functions require the L^AT_EX3 native drivers: they do not work with the L^AT_EX 2_ε graphics drivers!

T_EXhackers note: Clipping is implemented by the driver, and as such the full content of the box is placed in the output file. Thus clipping does not remove any information from the raw output, and hidden material can therefore be viewed by direct examination of the file.

```
\box_set_trim:Nnnnn
\box_set_trim:cnnnn
\box_gset_trim:Nnnnn
\box_gset_trim:cnnnn
```

New: 2019-01-23

`\box_set_trim:Nnnnn` $\langle box \rangle$ $\{\langle left \rangle\}$ $\{\langle bottom \rangle\}$ $\{\langle right \rangle\}$ $\{\langle top \rangle\}$

Adjusts the bounding box of the $\langle box \rangle$ $\langle left \rangle$ is removed from the left-hand edge of the bounding box, $\langle right \rangle$ from the right-hand edge and so fourth. All adjustments are *dimension expressions*. Material outside of the bounding box is still displayed in the output unless `\box_clip:N` is subsequently applied. The updated $\langle box \rangle$ is an hbox, irrespective of the nature of the $\langle box \rangle$ before the trim operation is applied. The behavior of the operation where the trims requested is greater than the size of the box is undefined.

```
\box_set_viewport:Nnnnn
\box_set_viewport:cnnnn
\box_gset_viewport:Nnnnn
\box_gset_viewport:cnnnn
```

New: 2019-01-23

`\box_set_viewport:Nnnnn` $\langle box \rangle$ $\{\langle llx \rangle\}$ $\{\langle lly \rangle\}$ $\{\langle urx \rangle\}$ $\{\langle ury \rangle\}$

Adjusts the bounding box of the $\langle box \rangle$ such that it has lower-left co-ordinates ($\langle llx \rangle$, $\langle lly \rangle$) and upper-right co-ordinates ($\langle urx \rangle$, $\langle ury \rangle$). All four co-ordinate positions are *dimension expressions*. Material outside of the bounding box is still displayed in the output unless `\box_clip:N` is subsequently applied. The updated $\langle box \rangle$ is an hbox, irrespective of the nature of the $\langle box \rangle$ before the viewport operation is applied.

37.3 Additions to l3expan

```
\exp_args_generate:n
```

New: 2018-04-04
Updated: 2019-02-08

`\exp_args_generate:n` $\{\langle variant argument specifiers \rangle\}$

Defines `\exp_args:N` $\langle variant \rangle$ functions for each $\langle variant \rangle$ given in the comma list $\{\langle variant argument specifiers \rangle\}$. Each $\langle variant \rangle$ should consist of the letters N, c, n, V, v, o, f, e, x, p and the resulting function is protected if the letter x appears in the $\langle variant \rangle$. This is only useful for cases where `\cs_generate_variant:Nn` is not applicable.

37.4 Additions to l3fp

```
\fp_if_nan_p:n ★
\fp_if_nan:nTF ★
```

New: 2019-08-25

`\fp_if_nan:n` $\{\langle fpexpr \rangle\}$

Evaluates the $\langle fpexpr \rangle$ and tests whether the result is exactly **nan**. The test returns **false** for any other result, even a tuple containing **nan**.

37.5 Additions to l3file

<hr/> <code>\iow_allow_break:</code> <hr/>	<code>\iow_allow_break:</code>
<hr/> New: 2018-12-29 <hr/>	In the first argument of <code>\iow_wrap:nnnN</code> (for instance in messages), inserts a break-point that allows a line break. In other words this is a zero-width breaking space.
<hr/> <code>\ior_get_term:nN</code> <code>\ior_str_get_term:nN</code> <hr/>	<code>\ior_get_term:nN</code> $\langle prompt \rangle$ $\langle token\ list\ variable \rangle$
<hr/> New: 2019-03-23 <hr/>	Function that reads one or more lines (until an equal number of left and right braces are found) from the terminal and stores the result locally in the $\langle token\ list \rangle$ variable. Tokenization occurs as described for <code>\ior_get:NN</code> or <code>\ior_str_get:NN</code> , respectively. When the $\langle prompt \rangle$ is empty, T _E X will wait for input without any other indication: typically the programmer will have provided a suitable text using e.g. <code>\iow_term:n</code> . Where the $\langle prompt \rangle$ is given, it will appear in the terminal followed by an =, e.g. <pre>prompt=</pre>
<hr/> <code>\ior_shell_open:Nn</code> <hr/>	<code>\ior_shell_open:Nn</code> $\langle stream \rangle$ $\{ \langle shell\ command \rangle \}$
<hr/> New: 2019-05-08 <hr/>	Opens the <i>pseudo</i> -file created by the output of the $\langle shell\ command \rangle$ for reading using $\langle stream \rangle$ as the control sequence for access. If the $\langle stream \rangle$ was already open it is closed before the new operation begins. The $\langle stream \rangle$ is available for access immediately and will remain allocated to $\langle shell\ command \rangle$ until a <code>\ior_close:N</code> instruction is given or the T _E X run ends. If piped system calls are disabled an error is raised. For details of handling of the $\langle shell\ command \rangle$, see <code>\sys_get_shell:nnNTF</code> .

37.6 Additions to l3flag

<hr/> <code>\flag_raise_if_clear:n</code> ☆ <hr/>	<code>\flag_raise_if_clear:n</code> $\{ \langle flag\ name \rangle \}$
<hr/> New: 2018-04-02 <hr/>	Ensures the $\langle flag \rangle$ is raised by making its height at least 1, locally.

37.7 Additions to l3intarray

<hr/> <code>\intarray_gset_rand:Nnn</code> <code>\intarray_gset_rand:cn</code> <code>\intarray_gset_rand:Nn</code> <code>\intarray_gset_rand:cn</code> <hr/>	<code>\intarray_gset_rand:Nnn</code> $\langle intarray\ var \rangle$ $\{ \langle minimum \rangle \}$ $\{ \langle maximum \rangle \}$ <code>\intarray_gset_rand:Nn</code> $\langle intarray\ var \rangle$ $\{ \langle maximum \rangle \}$
<hr/> New: 2018-05-05 <hr/>	Evaluates the integer expressions $\langle minimum \rangle$ and $\langle maximum \rangle$ then sets each entry (independently) of the $\langle integer\ array\ variable \rangle$ to a pseudo-random number between the two (with bounds included). If the absolute value of either bound is bigger than $2^{30} - 1$, an error occurs. Entries are generated in the same way as repeated calls to <code>\int_rand:nn</code> or <code>\int_rand:n</code> respectively, in particular for the second function the $\langle minimum \rangle$ is 1. Assignments are always global. This is not available in older versions of X _Y T _E X.
<hr/> <code>\intarray_to_clist:N</code> ☆ <hr/>	<code>\intarray_to_clist:N</code> $\langle intarray\ var \rangle$
<hr/> New: 2018-05-04 <hr/>	Converts the $\langle intarray \rangle$ to integer denotations separated by commas. All tokens have category code other. If the $\langle intarray \rangle$ has no entry the result is empty; otherwise the result has one fewer comma than the number of items.

37.8 Additions to l3msg

`\msg_show_eval:Nn`
`\msg_log_eval:Nn`

New: 2017-12-04

`\msg_show_eval:Nn` $\langle function \rangle$ $\{\langle expression \rangle\}$

Shows or logs the $\langle expression \rangle$ (turned into a string), an equal sign, and the result of applying the $\langle function \rangle$ to the $\{\langle expression \rangle\}$ (with f-expansion). For instance, if the $\langle function \rangle$ is `\int_eval:n` and the $\langle expression \rangle$ is `1+2` then this logs `> 1+2=3`.

<code>\msg_show_item:n</code>	★	<code>\seq_map_function:NN</code> $\langle seq \rangle$ <code>\msg_show_item:n</code>
<code>\msg_show_item_unbraced:n</code>	★	<code>\prop_map_function:NN</code> $\langle prop \rangle$ <code>\msg_show_item:nn</code>
<code>\msg_show_item:nn</code>	★	
<code>\msg_show_item_unbraced:nn</code>	★	

New: 2017-12-04

Used in the text of messages for `\msg_show:nnxxxx` to show or log a list of items or key-value pairs. The one-argument functions are used for sequences, clist or token lists and the others for property lists. These functions turn their arguments to strings.

37.9 Additions to l3prg

`\bool_set_inverse:N`
`\bool_set_inverse:c`
`\bool_gset_inverse:N`
`\bool_gset_inverse:c`

New: 2018-05-10

`\bool_set_inverse:N` $\langle boolean \rangle$

Toggles the $\langle boolean \rangle$ from `true` to `false` and conversely: sets it to the inverse of its current value.

<code>\bool_case_true:n</code>	★	<code>\bool_case_true:nTF</code>
<code>\bool_case_true:nTF</code>	★	{
<code>\bool_case_false:n</code>	★	{⟨ <i>boolexpr case₁</i> ⟩} {⟨ <i>code case₁</i> ⟩}
<code>\bool_case_false:nTF</code>	★	{⟨ <i>boolexpr case₂</i> ⟩} {⟨ <i>code case₂</i> ⟩}
<hr/>		...
New: 2019-02-10		{⟨ <i>boolexpr case_n</i> ⟩} {⟨ <i>code case_n</i> ⟩}
		}
		{⟨ <i>true code</i> ⟩}
		{⟨ <i>false code</i> ⟩}

Evaluates in turn each of the *⟨boolean expression cases⟩* until the first one that evaluates to true or to false, for `\bool_case_true:n` and `\bool_case_false:n`, respectively. The *⟨code⟩* associated to this first case is left in the input stream, followed by the *⟨true code⟩*, and other cases are discarded. If none of the cases match then only the *⟨false code⟩* is inserted. The functions `\bool_case_true:n` and `\bool_case_false:n`, which do nothing if there is no match, are also available. For example

```

\bool_case_true:nF
{
  { \dim_compare_p:n { \l__mypkg_wd_dim <= 10pt } }
    { Fits }
  { \int_compare_p:n { \l__mypkg_total_int >= 10 } }
    { Many }
  { \l__mypkg_special_bool }
    { Special }
}
{ No idea! }

```

leaves “Fits” or “Many” or “Special” or “No idea!” in the input stream, in a way similar to some other language’s “if ... elseif ... elseif ... else ...”.

37.10 Additions to l3prop

<code>\prop_rand_key_value:N</code>	★	<code>\prop_rand_key_value:N</code>	⟨ <i>prop var</i> ⟩
<code>\prop_rand_key_value:c</code>	★	Selects a pseudo-random key–value pair from the <i>⟨property list⟩</i> and returns {⟨ <i>key</i> ⟩} and {⟨ <i>value</i> ⟩}. If the <i>⟨property list⟩</i> is empty the result is empty. This is not available in older versions of X _Y TeX.	
<hr/>		New: 2016-12-06	

TeXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the *⟨value⟩* does not expand further when appearing in an x-type or e-type argument expansion.

37.11 Additions to l3seq

<code>\seq_mapthread_function:NNN</code>	☆	<code>\seq_mapthread_function:NNN <seq₁> <seq₂> <function></code>
<code>\seq_mapthread_function:(NcN cNN ccN)</code>	☆	

Applies *<function>* to every pair of items *<seq₁-item>*–*<seq₂-item>* from the two sequences, returning items from both sequences from left to right. The *<function>* receives two *n*-type arguments for each iteration. The mapping terminates when the end of either sequence is reached (*i.e.* whichever sequence has fewer items determines how many iterations occur).

<code>\seq_set_filter:NNn</code>	<code>\seq_set_filter:NNn <sequence₁> <sequence₂> {<inline boolexpr>}</code>
<code>\seq_gset_filter:NNn</code>	

Evaluates the *<inline boolexpr>* for every *<item>* stored within the *<sequence₂>*. The *<inline boolexpr>* receives the *<item>* as #1. The sequence of all *<items>* for which the *<inline boolexpr>* evaluated to `true` is assigned to *<sequence₁>*.

TeXhackers note: Contrarily to other mapping functions, `\seq_map_break:` cannot be used in this function, and would lead to low-level TeX errors.

<code>\seq_set_from_function:NnN</code>	<code>\seq_set_from_function:NnN <seq var> {<loop code>} <function></code>
<code>\seq_gset_from_function:NnN</code>	

New: 2018-04-06

Sets the *<seq var>* equal to a sequence whose items are obtained by *x*-expanding *<loop code>* *<function>*. This expansion must result in successive calls to the *<function>* with no nonexpandable tokens in between. More precisely the *<function>* is replaced by a wrapper function that inserts the appropriate separators between items in the sequence. The *<loop code>* must be expandable; it can be for example `\tl_map_function:NN <tl var>` or `\clist_map_function:nN {<clist>}` or `\int_step_function:nnnN {<initial value>} {<step>} {<final value>}`.

<code>\seq_set_from_inline_x:Nnn</code>	<code>\seq_set_from_inline_x:Nnn <seq var> {<loop code>} {<inline code>}</code>
<code>\seq_gset_from_inline_x:Nnn</code>	

New: 2018-04-06

Sets the *<seq var>* equal to a sequence whose items are obtained by *x*-expanding *<loop code>* applied to a *<function>* derived from the *<inline code>*. A *<function>* is defined, that takes one argument, *x*-expands the *<inline code>* with that argument as #1, then adds appropriate separators to turn the result into an item of the sequence. The *x*-expansion of *<loop code>* *<function>* must result in successive calls to the *<function>* with no nonexpandable tokens in between. The *<loop code>* must be expandable; it can be for example `\tl_map_function:NN <tl var>` or `\clist_map_function:nN {<clist>}` or `\int_step_function:nnnN {<initial value>} {<step>} {<final value>}`, but not the analogous “inline” mappings.

```

\seq_set_item:Nnn
\seq_set_item:cnN
\seq_set_item:NnnTF
\seq_set_item:cnNTF
\seq_gset_item:Nnn
\seq_gset_item:cnN
\seq_gset_item:NnnTF
\seq_gset_item:cnNTF

```

New: 2021-04-29

```

\seq_pop_item:NnN
\seq_pop_item:cnN
\seq_pop_item:NnNTF
\seq_pop_item:cnNNTF
\seq_gpop_item:NnN
\seq_gpop_item:cnN
\seq_gpop_item:NnNTF
\seq_gpop_item:cnNNTF

```

New: 2021-04-28

```

\seq_set_item:Nnn <seq var> {<intexpr>} {<item>}
\seq_set_item:NnnTF <seq var> {<intexpr>} {<item>} {<true code>} {<false code>}

```

Removes the item of *<sequence>* at the position given by evaluating the *<integer expression>* and replaces it by *<item>*. Items are indexed from 1 on the left/top of the *<sequence>*, or from -1 on the right/bottom. If the *<integer expression>* is zero or is larger (in absolute value) than the number of items in the sequence, the *<sequence>* is not modified. In these cases, `\seq_set_item:Nnn` raises an error while `\seq_set_item:NnnTF` runs the *<false code>*. In cases where the assignment was successful, *<true code>* is run afterwards.

```

\seq_pop_item:NnN <seq var> {<intexpr>} {<tl var>}
\seq_pop_item:NnNTF <seq var> {<intexpr>} {<tl var>} {<true code>} {<false code>}

```

Removes the *<item>* at position *<integer expression>* in the *<sequence>*, and places it in the *<token list variable>*. Items are indexed from 1 on the left/top of the *<sequence>*, or from -1 on the right/bottom. If the position is zero or is larger (in absolute value) than the number of items in the sequence, the *<seq var>* is not modified, the *<token list>* is set to the special marker `\q_no_value`, and the *<false code>* is left in the input stream; otherwise the *<true code>* is. The *<token list>* assignment is local while the *<sequence>* is assigned locally for `pop` or globally for `gpop` functions.

37.12 Additions to l3sys

```

\c_sys_engine_version_str

```

New: 2018-05-02

The version string of the current engine, in the same form as given in the banner issued when running a job. For pdfTeX and LuaTeX this is of the form

<major>.<minor>.<revision>

For XeTeX, the form is

<major>.<minor>

For pTeX and upTeX, only releases since TeX Live 2018 make the data available, and the form is more complex, as it comprises the pTeX version, the upTeX version and the e-pTeX version.

p<major>.<minor>.<revision>-u<major>.<minor>-<epTeX>

where the *u* part is only present for upTeX.

```

\sys_if_rand_exist_p: *
\sys_if_rand_exist:TF *

```

New: 2017-05-27

```

\sys_if_rand_exist_p:
\sys_if_rand_exist:TF {<true code>} {<false code>}

```

Tests if the engine has a pseudo-random number generator. Currently this is the case in pdfTeX, LuaTeX, pTeX, upTeX and recent releases of XeTeX.

37.13 Additions to l3tl

<code>\tl_range_braced:Nnn</code>	★	<code>\tl_range_braced:Nnn <tl var> {<start index>} {<end index>}</code>
<code>\tl_range_braced:cnn</code>	★	<code>\tl_range_braced:nnn {<token list>} {<start index>} {<end index>}</code>
<code>\tl_range_braced:nnn</code>	★	<code>\tl_range_unbraced:Nnn <tl var> {<start index>} {<end index>}</code>
<code>\tl_range_unbraced:Nnn</code>	★	<code>\tl_range_unbraced:nnn {<token list>} {<start index>} {<end index>}</code>
<code>\tl_range_unbraced:cnn</code>	★	Leaves in the input stream the items from the <i><start index></i> to the <i><end index></i> inclusive, using the same indexing as <code>\tl_range:nnn</code> . Spaces are ignored. Regardless of whether items appear with or without braces in the <i><token list></i> , the <code>\tl_range_braced:nnn</code> function wraps each item in braces, while <code>\tl_range_unbraced:nnn</code> does not (overall it removes an outer set of braces). For instance,
<code>\tl_range_unbraced:nnn</code>	★	

New: 2017-07-15

```

\tl_iow_term:x { \tl_range_braced:nnn { abcd~{e}}f } { 2 } { 5 } }
\tl_iow_term:x { \tl_range_braced:nnn { abcd~{e}}f } { -4 } { -1 } }
\tl_iow_term:x { \tl_range_braced:nnn { abcd~{e}}f } { -2 } { -1 } }
\tl_iow_term:x { \tl_range_braced:nnn { abcd~{e}}f } { 0 } { -1 } }

```

prints `{b}{c}{d}{e}}`, `{c}{d}{e}}{f}`, `{e}}{f}`, and an empty line to the terminal, while

```

\tl_iow_term:x { \tl_range_unbraced:nnn { abcd~{e}}f } { 2 } { 5 } }
\tl_iow_term:x { \tl_range_unbraced:nnn { abcd~{e}}f } { -4 } { -1 } }
\tl_iow_term:x { \tl_range_unbraced:nnn { abcd~{e}}f } { -2 } { -1 } }
\tl_iow_term:x { \tl_range_unbraced:nnn { abcd~{e}}f } { 0 } { -1 } }

```

prints `bcde{}`, `cde{f}`, `e{f}`, and an empty line to the terminal. Because braces are removed, the result of `\tl_range_unbraced:nnn` may have a different number of items as for `\tl_range:nnn` or `\tl_range_braced:nnn`. In cases where preserving spaces is important, consider the slower function `\tl_range:nnn`.

T_EXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the *<item>* does not expand further when appearing in an *x*-type or *e*-type argument expansion.

<code>\tl_build_begin:N</code>	<code>\tl_build_begin:N <tl var></code>
<code>\tl_build_gbegin:N</code>	Clears the <i><tl var></i> and sets it up to support other <code>\tl_build_...</code> functions, which allow accumulating large numbers of tokens piece by piece much more efficiently than standard l3tl functions. Until <code>\tl_build_end:N <tl var></code> is called, applying any function from l3tl other than <code>\tl_build_...</code> will lead to incorrect results. The <code>begin</code> and <code>gbegin</code> functions must be used for local and global <i><tl var></i> respectively.

New: 2018-04-01

<code>\tl_build_clear:N</code>	<code>\tl_build_clear:N <tl var></code>
<code>\tl_build_gclear:N</code>	Clears the <i><tl var></i> and sets it up to support other <code>\tl_build_...</code> functions. The <code>clear</code> and <code>gclear</code> functions must be used for local and global <i><tl var></i> respectively.

New: 2018-04-01

```

\tl_build_put_left:Nn
\tl_build_put_left:Nx
\tl_build_gput_left:Nn
\tl_build_gput_left:Nx
\tl_build_put_right:Nn
\tl_build_put_right:Nx
\tl_build_gput_right:Nn
\tl_build_gput_right:Nx

```

New: 2018-04-01

```

\tl_build_get:NN

```

New: 2018-04-01

```

\tl_build_end:N
\tl_build_gend:N

```

New: 2018-04-01

```

\tl_build_put_left:Nn <tl var> {<tokens>}
\tl_build_put_right:Nn <tl var> {<tokens>}

```

Adds *<tokens>* to the left or right side of the current contents of *<tl var>*. The *<tl var>* must have been set up with `\tl_build_begin:N` or `\tl_build_gbegin:N`. The `put` and `gput` functions must be used for local and global *<tl var>* respectively. The `right` functions are about twice faster than the `left` functions.

```

\tl_build_get:N <tl var1> <tl var2>

```

Stores the contents of the *<tl var₁>* in the *<tl var₂>*. The *<tl var₁>* must have been set up with `\tl_build_begin:N` or `\tl_build_gbegin:N`. The *<tl var₂>* is a “normal” token list variable, assigned locally using `\tl_set:Nn`.

```

\tl_build_end:N <tl var>

```

Gets the contents of *<tl var>* and stores that into the *<tl var>* using `\tl_set:Nn` or `\tl_gset:Nn`. The *<tl var>* must have been set up with `\tl_build_begin:N` or `\tl_build_gbegin:N`. The `end` and `gend` functions must be used for local and global *<tl var>* respectively. These functions completely remove the setup code that enabled *<tl var>* to be used for other `\tl_build_...` functions.

37.14 Additions to l3token

```

\c_catcode_active_space_tl

```

New: 2017-08-07

```

\char_to_utfviii_bytes:n ★

```

New: 2020-01-09

Token list containing one character with category code 13, (“active”), and character code 32 (space).

```

\char_to_utfviii_bytes:n {<codepoint>}

```

Converts the (Unicode) *<codepoint>* to UTF-8 bytes. The expansion of this function comprises four brace groups, each of which will contain a hexadecimal value: the appropriate byte. As UTF-8 is a variable-length, one or more of the groups may be empty: the bytes read in the logical order, such that a two-byte codepoint will have groups **#1** and **#2** filled and **#3** and **#4** empty.

```

\char_to_nfd:N ☆

```

New: 2020-01-02

```

\char_to_nfd:N <char>

```

Converts the *<char>* to the Unicode Normalization Form Canonical Decomposition. The category code of the generated character is the same as the *<char>*. With 8-bit engines, no change is made to the character.

<code>\peek_catcode_collect_inline:Nn</code>	<code>\peek_catcode_collect_inline:Nn <test token> {(inline code)}</code>
<code>\peek_charcode_collect_inline:Nn</code>	<code>\peek_charcode_collect_inline:Nn <test token> {(inline code)}</code>
<code>\peek_meaning_collect_inline:Nn</code>	<code>\peek_meaning_collect_inline:Nn <test token> {(inline code)}</code>

New: 2018-09-23

Collects and removes tokens from the input stream until finding a token that does not match the *<test token>* (as defined by the test `\token_if_eq_catcode:NNTF` or `\token_if_eq_charcode:NNTF` or `\token_if_eq_meaning:NNTF`). The collected tokens are passed to the *<inline code>* as #1. When begin-group or end-group tokens (usually { or }) are collected they are replaced by implicit `\c_group_begin_token` and `\c_group_end_token`, and when spaces (including `\c_space_token`) are collected they are replaced by explicit spaces.

For example the following code prints “Hello” to the terminal and leave “, world!” in the input stream.

```
\peek_catcode_collect_inline:Nn A { \iow_term:n {#1} } Hello,~world!
```

Another example is that the following code tests if the next token is *, ignoring intervening spaces, but putting them back using #1 if there is no *.

```
\peek_meaning_collect_inline:Nn \c_space_token
{ \peek_charcode:NNTF * { star } { no~star #1 } }
```

Part VIII
Implementation

Chapter 38

l3bootstrap implementation

```
1 <*package>
2 <@@=kernel>
```

38.1 LuaTeX-specific code

Depending on the versions available, the L^AT_EX format may not have the raw `\Umath` primitive names available. We fix that globally: it should cause no issues. Older LuaTeX versions do not have a pre-built table of the primitive names here so sort one out ourselves. These end up globally-defined but at that is true with a newer format anyway and as they all start `\U` this should be reasonably safe.

```
3 \begingroup
4   \expandafter\ifx\csname directlua\endcsname\relax
5   \else
6     \directlua{%
7       local i
8       local t = { }
9       for _,i in pairs(tex.extraprimatives("luatex")) do
10         if string.match(i,"^U") then
11           if not string.match(i,"^Uchar$") then %$
12             table.insert(t,i)
13           end
14         end
15       end
16       tex.enableprimitives("", t)
17     }%
18   \fi
19 \endgroup
```

38.2 The `\pdfstrcmp` primitive in X_ƎT_EX

Only pdfT_EX has a primitive called `\pdfstrcmp`. The X_ƎT_EX version is just `\strcmp`, so there is some shuffling to do. As this is still a real primitive, using the pdfT_EX name is “safe”.

```
20 \begingroup\expandafter\expandafter\expandafter\endgroup
21 \expandafter\ifx\csname pdfstrcmp\endcsname\relax
22   \let\pdfstrcmp\strcmp
```

```
23 \fi
```

38.3 Loading support Lua code

When LuaTeX is used there are various pieces of Lua code which need to be loaded. The code itself is defined in `l3luatex` and is extracted into a separate file. Thus here the task is to load the Lua code both now and (if required) at the start of each job.

```
24 \begingroup\expandafter\expandafter\expandafter\endgroup
25 \expandafter\ifx\csname directlua\endcsname\relax
26 \else
27   \ifnum\luatexversion<110 %
28   \else
```

For LuaTeX we make sure the basic support is loaded: this is only necessary in plain.

```
29   \begingroup\expandafter\expandafter\expandafter\endgroup
30   \expandafter\ifx\csname newcatcodetable\endcsname\relax
31     \input{ltluatex}%
32   \fi
33   \begingroup\expandafter\expandafter\expandafter\endgroup
34   \expandafter\ifx\csname newluabytecode\endcsname\relax
35   \else
36     \newluabytecode\@expl@luadata@bytecode
37   \fi
38   \directlua{require("expl3")}%
```

As the user might be making a custom format, no assumption is made about matching package mode with only loading the Lua code once. Instead, a query to Lua reveals what mode is in operation.

```
39   \ifnum 0%
40     \directlua{
41       if status.ini_version then
42         tex.write("1")
43       end
44     }>0 %
45     \everyjob\expandafter{%
46       \the\expandafter\everyjob
47       \csname\detokenize{lua_now:n}\endcsname{require("expl3")}%
48     }%
49   \fi
50 \fi
51 \fi
```

38.4 Engine requirements

The code currently requires ϵ -TeX and functionality equivalent to `\pdfstrcmp`, and also driver and Unicode character support. This is available in a reasonably-wide range of engines.

For LuaTeX, we require at least Lua 5.3 and the `token.set_lua` function. This is available at least since LuaTeX 1.10.

```
52 \begingroup
53   \def\next{\endgroup}%
54   \def\ShortText{Required primitives not found}%
```

```

55 \def\LongText%
56 {%
57   The L3 programming layer requires the e-TeX primitives and additional
58   \LineBreak functionality as described in the README file.
59   \LineBreak
60   These are available in the engines\LineBreak
61   - pdfTeX v1.40.0\LineBreak
62   - XeTeX v0.99992\LineBreak
63   - LuaTeX v1.10\LineBreak
64   - e-(u)pTeX mid-2012\LineBreak
65   - Prote (2021)\LineBreak
66   or later.\LineBreak
67   \LineBreak
68 }%
69 \ifnum0%
70   \expandafter\ifx\csname expanded\endcsname\relax
71     \expandafter\ifx\csname pdfstrcmp\endcsname\relax\else 1\fi
72   \else
73     \expandafter\ifx\csname luatexversion\endcsname\relax
74       1%
75     \else
76       \ifnum\luatexversion<110 \else 1\fi
77     \fi
78   \fi
79   =0 %
80   \newlinechar'\^^J %
81   \def\LineBreak{\noexpand\MessageBreak}%
82   \expandafter\ifx\csname PackageError\endcsname\relax
83     \def\LineBreak{\^^J}%
84     \begingroup
85       \lccode'\~=' \lccode'\}=' %
86       \lccode'\T=' \T\lccode'\H=' \H%
87       \catcode'\ =11 %
88   \lowercase{\endgroup\def\PackageError#1#2#3{%
89     \begingroup\errorcontextlines=1\immediate\write0{}\errhelp{#3}\def%
90     \
91     Type H <return> for immediate help}\def~{\errmessage{%
92     \
93     \fi
94     \edef\next
95     {%
96       \noexpand\PackageError{expl3}{\ShortText}
97       {\LongText Loading of expl3 will abort!}%
98     \endgroup
99     \noexpand\endinput
100    }%
101  \fi
102 \next

```

38.5 Extending allocators

The ability to extend \TeX 's allocation routine to allow for $\varepsilon\text{-TeX}$ has been around since 1997 in the `etex` package. Loading this support is delayed until here as we are now sure

that the ε -TeX extensions and `\pdfstrcmp` or equivalent are available. Thus there is no danger of an “uncontrolled” error if the engine requirements are not met.

For L^AT_EX_{2 ε} we need to make sure that the extended pool is being used: `expl3` uses a lot of registers. For formats from 2015 onward there is nothing to do as this is automatic. For older formats, the `etex` package needs to be loaded to do the job. In that case, some inserts are reserved also as these have to be from the standard pool. Note that `\reserveinserts` is `\outer` and so is accessed here by `csname`. In earlier versions, loading `etex` was done directly and so `\reserveinserts` appeared in the code: this then required a `\relax` after `\RequirePackage` to prevent an error with “unsafe” definitions as seen for example with `capoptions`. The optional loading here is done using a group and `\ifx` test as we are not quite in the position to have a single name for `\pdfstrcmp` just yet.

```

103 \begingroup
104   \def\@tempa{LaTeX2e}%
105   \def\next{}%
106   \ifx\fmtname\@tempa
107     \expandafter\ifx\csname extrafloats\endcsname\relax
108       \def\next
109         {%
110           \RequirePackage{etex}%
111           \csname reserveinserts\endcsname{32}%
112         }%
113     \fi
114   \fi
115 \expandafter\endgroup
116 \next

```

38.6 The L^AT_EX₃ code environment

The code environment is now set up.

\ExplSyntaxOff Before changing any category codes, in package mode we need to save the situation before loading. Note the set up here means that once applied `\ExplSyntaxOff` becomes a “do nothing” command until `\ExplSyntaxOn` is used.

```

117 \protected\edef\ExplSyntaxOff
118   {%
119     \protected\def\noexpand\ExplSyntaxOff{}%
120     \catcode 9 = \the\catcode 9\relax
121     \catcode 32 = \the\catcode 32\relax
122     \catcode 34 = \the\catcode 34\relax
123     \catcode 38 = \the\catcode 38\relax
124     \catcode 58 = \the\catcode 58\relax
125     \catcode 94 = \the\catcode 94\relax
126     \catcode 95 = \the\catcode 95\relax
127     \catcode 124 = \the\catcode 124\relax
128     \catcode 126 = \the\catcode 126\relax
129     \endlinechar = \the\endlinechar\relax
130     \chardef\csname\detokenize{l__kernel_expl_bool}\endcsname = 0\relax
131   }%

```

(End definition for `\ExplSyntaxOff`. This function is documented on page 9.)

The code environment is now set up.

```

132 \catcode 9 = 9\relax
133 \catcode 32 = 9\relax
134 \catcode 34 = 12\relax
135 \catcode 38 = 4\relax
136 \catcode 58 = 11\relax
137 \catcode 94 = 7\relax
138 \catcode 95 = 11\relax
139 \catcode 124 = 12\relax
140 \catcode 126 = 10\relax
141 \endlinechar = 32\relax

```

\l__kernel_expl_bool The status for code syntax: this is on at present.

```

142 \chardef\l__kernel_expl_bool = 1\relax

```

(End definition for \l__kernel_expl_bool.)

\ExplSyntaxOn The idea here is that multiple \ExplSyntaxOn calls are not going to mess up category codes, and that multiple calls to \ExplSyntaxOff are also not wasting time. Applying \ExplSyntaxOn alters the definition of \ExplSyntaxOff and so in package mode this function should not be used until after the end of the loading process!

```

143 \protected \def \ExplSyntaxOn
144 {
145   \bool_if:NF \l__kernel_expl_bool
146   {
147     \cs_set_protected:Npx \ExplSyntaxOff
148     {
149       \char_set_catcode:nn { 9 } { \char_value_catcode:n { 9 } }
150       \char_set_catcode:nn { 32 } { \char_value_catcode:n { 32 } }
151       \char_set_catcode:nn { 34 } { \char_value_catcode:n { 34 } }
152       \char_set_catcode:nn { 38 } { \char_value_catcode:n { 38 } }
153       \char_set_catcode:nn { 58 } { \char_value_catcode:n { 58 } }
154       \char_set_catcode:nn { 94 } { \char_value_catcode:n { 94 } }
155       \char_set_catcode:nn { 95 } { \char_value_catcode:n { 95 } }
156       \char_set_catcode:nn { 124 } { \char_value_catcode:n { 124 } }
157       \char_set_catcode:nn { 126 } { \char_value_catcode:n { 126 } }
158       \tex_endlinechar:D =
159       \tex_the:D \tex_endlinechar:D \scan_stop:
160       \bool_set_false:N \l__kernel_expl_bool
161       \cs_set_protected:Npn \ExplSyntaxOff { }
162     }
163   }
164   \char_set_catcode_ignore:n { 9 } % tab
165   \char_set_catcode_ignore:n { 32 } % space
166   \char_set_catcode_other:n { 34 } % double quote
167   \char_set_catcode_alignment:n { 38 } % ampersand
168   \char_set_catcode_letter:n { 58 } % colon
169   \char_set_catcode_math_superscript:n { 94 } % circumflex
170   \char_set_catcode_letter:n { 95 } % underscore
171   \char_set_catcode_other:n { 124 } % pipe
172   \char_set_catcode_space:n { 126 } % tilde
173   \tex_endlinechar:D = 32 \scan_stop:
174   \bool_set_true:N \l__kernel_expl_bool
175 }

```

(End definition for \ExplSyntaxOn. This function is documented on page 9.)

176 `\endpackage`

Chapter 39

l3names implementation

```
177 <*package & tex>
```

The prefix here is `kernel`. A few places need `@@` to be left as is; this is obtained as `@@@`.

```
178 <@@=kernel>
```

The code here simply renames all of the primitives to new, internal, names.

The `\let` primitive is renamed by hand first as it is essential for the entire process to follow. This also uses `\global`, as that way we avoid leaving an unneeded `csname` in the hash table.

```
179 \let \tex_global:D \global
```

```
180 \let \tex_let:D \let
```

Everything is inside a (rather long) group, which keeps `__kernel_primitive:NN` trapped.

```
181 \begingroup
```

```
\__kernel_primitive:NN A temporary function to actually do the renaming.
```

```
182 \long \def \__kernel_primitive:NN #1#2
```

```
183 { \tex_global:D \tex_let:D #2 #1 }
```

(End definition for __kernel_primitive:NN.)

To allow extracting “just the names”, a bit of DocStrip fiddling.

```
184 </package & tex>
```

```
185 <*names | tex>
```

```
186 <*names | package>
```

In the current incarnation of this package, all TeX primitives are given a new name of the form `\tex_oldname:D`. But first three special cases which have symbolic original names. These are given modified new names, so that they may be entered without catcode tricks.

```
187 \__kernel_primitive:NN \tex_space:D
```

```
188 \__kernel_primitive:NN \tex_italiccorrection:D
```

```
189 \__kernel_primitive:NN \tex_hyphen:D
```

Now all the other primitives.

```
190 \__kernel_primitive:NN \tex_above:D
```

```
191 \__kernel_primitive:NN \tex_abovedisplayshortskip \tex_abovedisplayshortskip:D
```

```
192 \__kernel_primitive:NN \tex_abovedisplayskip \tex_abovedisplayskip:D
```

```
193 \__kernel_primitive:NN \tex_abovewithdelims \tex_abovewithdelims:D
```

194	_kernel_primitive:NN	\accent	\tex_accent:D
195	_kernel_primitive:NN	\adjdemerits	\tex_adjdemerits:D
196	_kernel_primitive:NN	\advance	\tex_advance:D
197	_kernel_primitive:NN	\afterassignment	\tex_afterassignment:D
198	_kernel_primitive:NN	\aftergroup	\tex_aftergroup:D
199	_kernel_primitive:NN	\atop	\tex_atop:D
200	_kernel_primitive:NN	\atopwithdelims	\tex_atopwithdelims:D
201	_kernel_primitive:NN	\badness	\tex_badness:D
202	_kernel_primitive:NN	\baselineskip	\tex_baselineskip:D
203	_kernel_primitive:NN	\batchmode	\tex_batchmode:D
204	_kernel_primitive:NN	\begingroup	\tex_begingroup:D
205	_kernel_primitive:NN	\belowdisplayshortskip	\tex_belowdisplayshortskip:D
206	_kernel_primitive:NN	\belowdisplayskip	\tex_belowdisplayskip:D
207	_kernel_primitive:NN	\binoppenalty	\tex_binoppenalty:D
208	_kernel_primitive:NN	\botmark	\tex_botmark:D
209	_kernel_primitive:NN	\box	\tex_box:D
210	_kernel_primitive:NN	\boxmaxdepth	\tex_boxmaxdepth:D
211	_kernel_primitive:NN	\brokenpenalty	\tex_brokenpenalty:D
212	_kernel_primitive:NN	\catcode	\tex_catcode:D
213	_kernel_primitive:NN	\char	\tex_char:D
214	_kernel_primitive:NN	\chardef	\tex_chardef:D
215	_kernel_primitive:NN	\cleaders	\tex_cleaders:D
216	_kernel_primitive:NN	\closein	\tex_closein:D
217	_kernel_primitive:NN	\closeout	\tex_closeout:D
218	_kernel_primitive:NN	\clubpenalty	\tex_clubpenalty:D
219	_kernel_primitive:NN	\copy	\tex_copy:D
220	_kernel_primitive:NN	\count	\tex_count:D
221	_kernel_primitive:NN	\countdef	\tex_countdef:D
222	_kernel_primitive:NN	\cr	\tex_cr:D
223	_kernel_primitive:NN	\crrcr	\tex_crrcr:D
224	_kernel_primitive:NN	\csname	\tex_csname:D
225	_kernel_primitive:NN	\day	\tex_day:D
226	_kernel_primitive:NN	\deadcycles	\tex_deadcycles:D
227	_kernel_primitive:NN	\def	\tex_def:D
228	_kernel_primitive:NN	\defaultthyphenchar	\tex_defaultthyphenchar:D
229	_kernel_primitive:NN	\defaultskewchar	\tex_defaultskewchar:D
230	_kernel_primitive:NN	\delcode	\tex_delcode:D
231	_kernel_primitive:NN	\delimiter	\tex_delimiter:D
232	_kernel_primitive:NN	\delimiterfactor	\tex_delimiterfactor:D
233	_kernel_primitive:NN	\delimitershortfall	\tex_delimitershortfall:D
234	_kernel_primitive:NN	\dimen	\tex_dimen:D
235	_kernel_primitive:NN	\dimendef	\tex_dimendef:D
236	_kernel_primitive:NN	\discretionary	\tex_discretionary:D
237	_kernel_primitive:NN	\displayindent	\tex_displayindent:D
238	_kernel_primitive:NN	\displaylimits	\tex_displaylimits:D
239	_kernel_primitive:NN	\displaystyle	\tex_displaystyle:D
240	_kernel_primitive:NN	\displaywidowpenalty	\tex_displaywidowpenalty:D
241	_kernel_primitive:NN	\displaywidth	\tex_displaywidth:D
242	_kernel_primitive:NN	\divide	\tex_divide:D
243	_kernel_primitive:NN	\doublehyphendemerits	\tex_doublehyphendemerits:D
244	_kernel_primitive:NN	\dp	\tex_dp:D
245	_kernel_primitive:NN	\dump	\tex_dump:D
246	_kernel_primitive:NN	\edef	\tex_edef:D
247	_kernel_primitive:NN	\else	\tex_else:D

248	<code>__kernel_primitive:NN \emergencystretch</code>	<code>\tex_emergencystretch:D</code>
249	<code>__kernel_primitive:NN \end</code>	<code>\tex_end:D</code>
250	<code>__kernel_primitive:NN \endcsname</code>	<code>\tex_endcsname:D</code>
251	<code>__kernel_primitive:NN \endgroup</code>	<code>\tex_endgroup:D</code>
252	<code>__kernel_primitive:NN \endinput</code>	<code>\tex_endinput:D</code>
253	<code>__kernel_primitive:NN \endlinechar</code>	<code>\tex_endlinechar:D</code>
254	<code>__kernel_primitive:NN \eqno</code>	<code>\tex_eqno:D</code>
255	<code>__kernel_primitive:NN \errhelp</code>	<code>\tex_errhelp:D</code>
256	<code>__kernel_primitive:NN \errmessage</code>	<code>\tex_errmessage:D</code>
257	<code>__kernel_primitive:NN \errorcontextlines</code>	<code>\tex_errorcontextlines:D</code>
258	<code>__kernel_primitive:NN \errorstopmode</code>	<code>\tex_errorstopmode:D</code>
259	<code>__kernel_primitive:NN \escapechar</code>	<code>\tex_escapechar:D</code>
260	<code>__kernel_primitive:NN \everycr</code>	<code>\tex_everycr:D</code>
261	<code>__kernel_primitive:NN \everydisplay</code>	<code>\tex_everydisplay:D</code>
262	<code>__kernel_primitive:NN \everyhbox</code>	<code>\tex_everyhbox:D</code>
263	<code>__kernel_primitive:NN \everyjob</code>	<code>\tex_everyjob:D</code>
264	<code>__kernel_primitive:NN \everymath</code>	<code>\tex_everymath:D</code>
265	<code>__kernel_primitive:NN \everypar</code>	<code>\tex_everypar:D</code>
266	<code>__kernel_primitive:NN \everyvbox</code>	<code>\tex_everyvbox:D</code>
267	<code>__kernel_primitive:NN \exhyphenpenalty</code>	<code>\tex_exhyphenpenalty:D</code>
268	<code>__kernel_primitive:NN \expandafter</code>	<code>\tex_expandafter:D</code>
269	<code>__kernel_primitive:NN \fam</code>	<code>\tex_fam:D</code>
270	<code>__kernel_primitive:NN \fi</code>	<code>\tex_fi:D</code>
271	<code>__kernel_primitive:NN \finalhyphendemerits</code>	<code>\tex_finalhyphendemerits:D</code>
272	<code>__kernel_primitive:NN \firstmark</code>	<code>\tex_firstmark:D</code>
273	<code>__kernel_primitive:NN \floatingpenalty</code>	<code>\tex_floatingpenalty:D</code>
274	<code>__kernel_primitive:NN \font</code>	<code>\tex_font:D</code>
275	<code>__kernel_primitive:NN \fontdimen</code>	<code>\tex_fontdimen:D</code>
276	<code>__kernel_primitive:NN \fontname</code>	<code>\tex_fontname:D</code>
277	<code>__kernel_primitive:NN \futurelet</code>	<code>\tex_futurelet:D</code>
278	<code>__kernel_primitive:NN \gdef</code>	<code>\tex_gdef:D</code>
279	<code>__kernel_primitive:NN \global</code>	<code>\tex_global:D</code>
280	<code>__kernel_primitive:NN \globaldefs</code>	<code>\tex_globaldefs:D</code>
281	<code>__kernel_primitive:NN \halign</code>	<code>\tex_halign:D</code>
282	<code>__kernel_primitive:NN \hangafter</code>	<code>\tex_hangafter:D</code>
283	<code>__kernel_primitive:NN \hangindent</code>	<code>\tex_hangindent:D</code>
284	<code>__kernel_primitive:NN \hbadness</code>	<code>\tex_hbadness:D</code>
285	<code>__kernel_primitive:NN \hbox</code>	<code>\tex_hbox:D</code>
286	<code>__kernel_primitive:NN \hfil</code>	<code>\tex_hfil:D</code>
287	<code>__kernel_primitive:NN \hfill</code>	<code>\tex_hfill:D</code>
288	<code>__kernel_primitive:NN \hfilneg</code>	<code>\tex_hfilneg:D</code>
289	<code>__kernel_primitive:NN \hfuzz</code>	<code>\tex_hfuzz:D</code>
290	<code>__kernel_primitive:NN \hoffset</code>	<code>\tex_hoffset:D</code>
291	<code>__kernel_primitive:NN \holdinginserts</code>	<code>\tex_holdinginserts:D</code>
292	<code>__kernel_primitive:NN \hrule</code>	<code>\tex_hrule:D</code>
293	<code>__kernel_primitive:NN \hsize</code>	<code>\tex_hsize:D</code>
294	<code>__kernel_primitive:NN \hskip</code>	<code>\tex_hskip:D</code>
295	<code>__kernel_primitive:NN \hss</code>	<code>\tex_hss:D</code>
296	<code>__kernel_primitive:NN \ht</code>	<code>\tex_ht:D</code>
297	<code>__kernel_primitive:NN \hyphenation</code>	<code>\tex_hyphenation:D</code>
298	<code>__kernel_primitive:NN \hyphenchar</code>	<code>\tex_hyphenchar:D</code>
299	<code>__kernel_primitive:NN \hyphenpenalty</code>	<code>\tex_hyphenpenalty:D</code>
300	<code>__kernel_primitive:NN \if</code>	<code>\tex_if:D</code>
301	<code>__kernel_primitive:NN \ifcase</code>	<code>\tex_ifcase:D</code>

302	<code>__kernel_primitive:NN \ifcat</code>	<code>\tex_ifcat:D</code>
303	<code>__kernel_primitive:NN \ifdim</code>	<code>\tex_ifdim:D</code>
304	<code>__kernel_primitive:NN \ifeof</code>	<code>\tex_ifeof:D</code>
305	<code>__kernel_primitive:NN \iffalse</code>	<code>\tex_iffalse:D</code>
306	<code>__kernel_primitive:NN \ifhbox</code>	<code>\tex_ifhbox:D</code>
307	<code>__kernel_primitive:NN \ifhmode</code>	<code>\tex_ifhmode:D</code>
308	<code>__kernel_primitive:NN \ifinner</code>	<code>\tex_ifinner:D</code>
309	<code>__kernel_primitive:NN \ifmmode</code>	<code>\tex_ifmmode:D</code>
310	<code>__kernel_primitive:NN \ifnum</code>	<code>\tex_ifnum:D</code>
311	<code>__kernel_primitive:NN \ifodd</code>	<code>\tex_ifodd:D</code>
312	<code>__kernel_primitive:NN \iftrue</code>	<code>\tex_iftrue:D</code>
313	<code>__kernel_primitive:NN \ifvbox</code>	<code>\tex_ifvbox:D</code>
314	<code>__kernel_primitive:NN \ifvmode</code>	<code>\tex_ifvmode:D</code>
315	<code>__kernel_primitive:NN \ifvoid</code>	<code>\tex_ifvoid:D</code>
316	<code>__kernel_primitive:NN \ifx</code>	<code>\tex_ifx:D</code>
317	<code>__kernel_primitive:NN \ignorespaces</code>	<code>\tex_ignorespaces:D</code>
318	<code>__kernel_primitive:NN \immediate</code>	<code>\tex_immediate:D</code>
319	<code>__kernel_primitive:NN \indent</code>	<code>\tex_indent:D</code>
320	<code>__kernel_primitive:NN \input</code>	<code>\tex_input:D</code>
321	<code>__kernel_primitive:NN \inputlineno</code>	<code>\tex_inputlineno:D</code>
322	<code>__kernel_primitive:NN \insert</code>	<code>\tex_insert:D</code>
323	<code>__kernel_primitive:NN \insertpenalties</code>	<code>\tex_insertpenalties:D</code>
324	<code>__kernel_primitive:NN \interlinepenalty</code>	<code>\tex_interlinepenalty:D</code>
325	<code>__kernel_primitive:NN \jobname</code>	<code>\tex_jobname:D</code>
326	<code>__kernel_primitive:NN \kern</code>	<code>\tex_kern:D</code>
327	<code>__kernel_primitive:NN \language</code>	<code>\tex_language:D</code>
328	<code>__kernel_primitive:NN \lastbox</code>	<code>\tex_lastbox:D</code>
329	<code>__kernel_primitive:NN \lastkern</code>	<code>\tex_lastkern:D</code>
330	<code>__kernel_primitive:NN \lastpenalty</code>	<code>\tex_lastpenalty:D</code>
331	<code>__kernel_primitive:NN \lastskip</code>	<code>\tex_lastskip:D</code>
332	<code>__kernel_primitive:NN \lccode</code>	<code>\tex_lccode:D</code>
333	<code>__kernel_primitive:NN \leaders</code>	<code>\tex_leaders:D</code>
334	<code>__kernel_primitive:NN \left</code>	<code>\tex_left:D</code>
335	<code>__kernel_primitive:NN \leftthyphenmin</code>	<code>\tex_leftthyphenmin:D</code>
336	<code>__kernel_primitive:NN \leftskip</code>	<code>\tex_leftskip:D</code>
337	<code>__kernel_primitive:NN \leqno</code>	<code>\tex_leqno:D</code>
338	<code>__kernel_primitive:NN \let</code>	<code>\tex_let:D</code>
339	<code>__kernel_primitive:NN \limits</code>	<code>\tex_limits:D</code>
340	<code>__kernel_primitive:NN \linepenalty</code>	<code>\tex_linepenalty:D</code>
341	<code>__kernel_primitive:NN \lineskip</code>	<code>\tex_lineskip:D</code>
342	<code>__kernel_primitive:NN \lineskiplimit</code>	<code>\tex_lineskiplimit:D</code>
343	<code>__kernel_primitive:NN \long</code>	<code>\tex_long:D</code>
344	<code>__kernel_primitive:NN \looseness</code>	<code>\tex_looseness:D</code>
345	<code>__kernel_primitive:NN \lower</code>	<code>\tex_lower:D</code>
346	<code>__kernel_primitive:NN \lowercase</code>	<code>\tex_lowercase:D</code>
347	<code>__kernel_primitive:NN \mag</code>	<code>\tex_mag:D</code>
348	<code>__kernel_primitive:NN \mark</code>	<code>\tex_mark:D</code>
349	<code>__kernel_primitive:NN \mathaccent</code>	<code>\tex_mathaccent:D</code>
350	<code>__kernel_primitive:NN \mathbin</code>	<code>\tex_mathbin:D</code>
351	<code>__kernel_primitive:NN \mathchar</code>	<code>\tex_mathchar:D</code>
352	<code>__kernel_primitive:NN \mathchardef</code>	<code>\tex_mathchardef:D</code>
353	<code>__kernel_primitive:NN \mathchoice</code>	<code>\tex_mathchoice:D</code>
354	<code>__kernel_primitive:NN \mathclose</code>	<code>\tex_mathclose:D</code>
355	<code>__kernel_primitive:NN \mathcode</code>	<code>\tex_mathcode:D</code>

356	_kernel_primitive:NN	\mathinner	\tex_mathinner:D
357	_kernel_primitive:NN	\mathop	\tex_mathop:D
358	_kernel_primitive:NN	\mathopen	\tex_mathopen:D
359	_kernel_primitive:NN	\mathord	\tex_mathord:D
360	_kernel_primitive:NN	\mathpunct	\tex_mathpunct:D
361	_kernel_primitive:NN	\mathrel	\tex_mathrel:D
362	_kernel_primitive:NN	\mathsurround	\tex_mathsurround:D
363	_kernel_primitive:NN	\maxdeadcycles	\tex_maxdeadcycles:D
364	_kernel_primitive:NN	\maxdepth	\tex_maxdepth:D
365	_kernel_primitive:NN	\meaning	\tex_meaning:D
366	_kernel_primitive:NN	\medmuskip	\tex_medmuskip:D
367	_kernel_primitive:NN	\message	\tex_message:D
368	_kernel_primitive:NN	\mkern	\tex_mkern:D
369	_kernel_primitive:NN	\month	\tex_month:D
370	_kernel_primitive:NN	\moveleft	\tex_moveleft:D
371	_kernel_primitive:NN	\moveright	\tex_moveright:D
372	_kernel_primitive:NN	\mskip	\tex_mskip:D
373	_kernel_primitive:NN	\multiply	\tex_multiply:D
374	_kernel_primitive:NN	\muskip	\tex_muskip:D
375	_kernel_primitive:NN	\muskipdef	\tex_muskipdef:D
376	_kernel_primitive:NN	\newlinechar	\tex_newlinechar:D
377	_kernel_primitive:NN	\noalign	\tex_noalign:D
378	_kernel_primitive:NN	\noboundary	\tex_noboundary:D
379	_kernel_primitive:NN	\noexpand	\tex_noexpand:D
380	_kernel_primitive:NN	\noindent	\tex_noindent:D
381	_kernel_primitive:NN	\nolimits	\tex_nolimits:D
382	_kernel_primitive:NN	\nonscript	\tex_nonscript:D
383	_kernel_primitive:NN	\nonstopmode	\tex_nonstopmode:D
384	_kernel_primitive:NN	\nulldelimiterspace	\tex_nulldelimiterspace:D
385	_kernel_primitive:NN	\nullfont	\tex_nullfont:D
386	_kernel_primitive:NN	\number	\tex_number:D
387	_kernel_primitive:NN	\omit	\tex_omit:D
388	_kernel_primitive:NN	\openin	\tex_openin:D
389	_kernel_primitive:NN	\openout	\tex_openout:D
390	_kernel_primitive:NN	\or	\tex_or:D
391	_kernel_primitive:NN	\outer	\tex_outer:D
392	_kernel_primitive:NN	\output	\tex_output:D
393	_kernel_primitive:NN	\outputpenalty	\tex_outputpenalty:D
394	_kernel_primitive:NN	\over	\tex_over:D
395	_kernel_primitive:NN	\overfullrule	\tex_overfullrule:D
396	_kernel_primitive:NN	\overline	\tex_overline:D
397	_kernel_primitive:NN	\overwithdelims	\tex_overwithdelims:D
398	_kernel_primitive:NN	\pagedepth	\tex_pagedepth:D
399	_kernel_primitive:NN	\pagefilllstretch	\tex_pagefilllstretch:D
400	_kernel_primitive:NN	\pagefillstretch	\tex_pagefillstretch:D
401	_kernel_primitive:NN	\pagefilstretch	\tex_pagefilstretch:D
402	_kernel_primitive:NN	\pagegoal	\tex_pagegoal:D
403	_kernel_primitive:NN	\pageshrink	\tex_pageshrink:D
404	_kernel_primitive:NN	\pagestretch	\tex_pagestretch:D
405	_kernel_primitive:NN	\pagetotal	\tex_pagetotal:D
406	_kernel_primitive:NN	\par	\tex_par:D
407	_kernel_primitive:NN	\parfillskip	\tex_parfillskip:D
408	_kernel_primitive:NN	\parindent	\tex_parindent:D
409	_kernel_primitive:NN	\parshape	\tex_parshape:D

410	_kernel_primitive:NN	\parskip	\tex_parskip:D
411	_kernel_primitive:NN	\patterns	\tex_patterns:D
412	_kernel_primitive:NN	\pausing	\tex_pausing:D
413	_kernel_primitive:NN	\penalty	\tex_penalty:D
414	_kernel_primitive:NN	\postdisplaypenalty	\tex_postdisplaypenalty:D
415	_kernel_primitive:NN	\predisdisplaypenalty	\tex_predisdisplaypenalty:D
416	_kernel_primitive:NN	\predisplaysize	\tex_predisplaysize:D
417	_kernel_primitive:NN	\pretolerance	\tex_pretolerance:D
418	_kernel_primitive:NN	\prevdepth	\tex_prevdepth:D
419	_kernel_primitive:NN	\prevgraf	\tex_prevgraf:D
420	_kernel_primitive:NN	\radical	\tex_radical:D
421	_kernel_primitive:NN	\raise	\tex_raise:D
422	_kernel_primitive:NN	\read	\tex_read:D
423	_kernel_primitive:NN	\relax	\tex_relax:D
424	_kernel_primitive:NN	\relpenalty	\tex_relpenalty:D
425	_kernel_primitive:NN	\right	\tex_right:D
426	_kernel_primitive:NN	\righthyphenmin	\tex_righthyphenmin:D
427	_kernel_primitive:NN	\rightskip	\tex_rightskip:D
428	_kernel_primitive:NN	\romannumeral	\tex_romannumeral:D
429	_kernel_primitive:NN	\scriptfont	\tex_scriptfont:D
430	_kernel_primitive:NN	\scriptscriptfont	\tex_scriptscriptfont:D
431	_kernel_primitive:NN	\scriptscriptstyle	\tex_scriptscriptstyle:D
432	_kernel_primitive:NN	\scriptspace	\tex_scriptspace:D
433	_kernel_primitive:NN	\scriptstyle	\tex_scriptstyle:D
434	_kernel_primitive:NN	\scrollmode	\tex_scrollmode:D
435	_kernel_primitive:NN	\setbox	\tex_setbox:D
436	_kernel_primitive:NN	\setlanguage	\tex_setlanguage:D
437	_kernel_primitive:NN	\sfcode	\tex_sfcode:D
438	_kernel_primitive:NN	\shipout	\tex_shipout:D
439	_kernel_primitive:NN	\show	\tex_show:D
440	_kernel_primitive:NN	\showbox	\tex_showbox:D
441	_kernel_primitive:NN	\showboxbreadth	\tex_showboxbreadth:D
442	_kernel_primitive:NN	\showboxdepth	\tex_showboxdepth:D
443	_kernel_primitive:NN	\showlists	\tex_showlists:D
444	_kernel_primitive:NN	\showthe	\tex_showthe:D
445	_kernel_primitive:NN	\skewchar	\tex_skewchar:D
446	_kernel_primitive:NN	\skip	\tex_skip:D
447	_kernel_primitive:NN	\skipdef	\tex_skipdef:D
448	_kernel_primitive:NN	\spacefactor	\tex_spacefactor:D
449	_kernel_primitive:NN	\spaceskip	\tex_spaceskip:D
450	_kernel_primitive:NN	\span	\tex_span:D
451	_kernel_primitive:NN	\special	\tex_special:D
452	_kernel_primitive:NN	\splitbotmark	\tex_splitbotmark:D
453	_kernel_primitive:NN	\splitfirstmark	\tex_splitfirstmark:D
454	_kernel_primitive:NN	\splitmaxdepth	\tex_splitmaxdepth:D
455	_kernel_primitive:NN	\splittopskip	\tex_splittopskip:D
456	_kernel_primitive:NN	\string	\tex_string:D
457	_kernel_primitive:NN	\tabskip	\tex_tabskip:D
458	_kernel_primitive:NN	\textfont	\tex_textfont:D
459	_kernel_primitive:NN	\textstyle	\tex_textstyle:D
460	_kernel_primitive:NN	\the	\tex_the:D
461	_kernel_primitive:NN	\thickmuskip	\tex_thickmuskip:D
462	_kernel_primitive:NN	\thinmuskip	\tex_thinmuskip:D
463	_kernel_primitive:NN	\time	\tex_time:D

464	<code>_kernel_primitive:NN \toks</code>	<code>\tex_toks:D</code>
465	<code>_kernel_primitive:NN \toksdef</code>	<code>\tex_toksdef:D</code>
466	<code>_kernel_primitive:NN \tolerance</code>	<code>\tex_tolerance:D</code>
467	<code>_kernel_primitive:NN \topmark</code>	<code>\tex_topmark:D</code>
468	<code>_kernel_primitive:NN \topskip</code>	<code>\tex_topskip:D</code>
469	<code>_kernel_primitive:NN \tracingcommands</code>	<code>\tex_tracingcommands:D</code>
470	<code>_kernel_primitive:NN \tracinglostchars</code>	<code>\tex_tracinglostchars:D</code>
471	<code>_kernel_primitive:NN \tracingmacros</code>	<code>\tex_tracingmacros:D</code>
472	<code>_kernel_primitive:NN \tracingonline</code>	<code>\tex_tracingonline:D</code>
473	<code>_kernel_primitive:NN \tracingoutput</code>	<code>\tex_tracingoutput:D</code>
474	<code>_kernel_primitive:NN \tracingpages</code>	<code>\tex_tracingpages:D</code>
475	<code>_kernel_primitive:NN \tracingparagraphs</code>	<code>\tex_tracingparagraphs:D</code>
476	<code>_kernel_primitive:NN \tracingrestores</code>	<code>\tex_tracingrestores:D</code>
477	<code>_kernel_primitive:NN \tracingstats</code>	<code>\tex_tracingstats:D</code>
478	<code>_kernel_primitive:NN \uccode</code>	<code>\tex_uccode:D</code>
479	<code>_kernel_primitive:NN \uchyph</code>	<code>\tex_uchyph:D</code>
480	<code>_kernel_primitive:NN \underline</code>	<code>\tex_underline:D</code>
481	<code>_kernel_primitive:NN \unhbox</code>	<code>\tex_unhbox:D</code>
482	<code>_kernel_primitive:NN \unhcopy</code>	<code>\tex_unhcopy:D</code>
483	<code>_kernel_primitive:NN \unkern</code>	<code>\tex_unkern:D</code>
484	<code>_kernel_primitive:NN \unpenalty</code>	<code>\tex_unpenalty:D</code>
485	<code>_kernel_primitive:NN \unskip</code>	<code>\tex_unskip:D</code>
486	<code>_kernel_primitive:NN \unvbox</code>	<code>\tex_unvbox:D</code>
487	<code>_kernel_primitive:NN \unvcopy</code>	<code>\tex_unvcopy:D</code>
488	<code>_kernel_primitive:NN \uppercase</code>	<code>\tex_uppercase:D</code>
489	<code>_kernel_primitive:NN \vadjust</code>	<code>\tex_vadjust:D</code>
490	<code>_kernel_primitive:NN \valign</code>	<code>\tex_valign:D</code>
491	<code>_kernel_primitive:NN \vbadness</code>	<code>\tex_vbadness:D</code>
492	<code>_kernel_primitive:NN \vbox</code>	<code>\tex_vbox:D</code>
493	<code>_kernel_primitive:NN \vcenter</code>	<code>\tex_vcenter:D</code>
494	<code>_kernel_primitive:NN \vfil</code>	<code>\tex_vfil:D</code>
495	<code>_kernel_primitive:NN \vfill</code>	<code>\tex_vfill:D</code>
496	<code>_kernel_primitive:NN \vfилneg</code>	<code>\tex_vfилneg:D</code>
497	<code>_kernel_primitive:NN \vfuzz</code>	<code>\tex_vfuzz:D</code>
498	<code>_kernel_primitive:NN \voffset</code>	<code>\tex_voffset:D</code>
499	<code>_kernel_primitive:NN \vrule</code>	<code>\tex_vrule:D</code>
500	<code>_kernel_primitive:NN \vsize</code>	<code>\tex_vsize:D</code>
501	<code>_kernel_primitive:NN \vskip</code>	<code>\tex_vskip:D</code>
502	<code>_kernel_primitive:NN \vsplit</code>	<code>\tex_vsplit:D</code>
503	<code>_kernel_primitive:NN \vss</code>	<code>\tex_vss:D</code>
504	<code>_kernel_primitive:NN \vtop</code>	<code>\tex_vtop:D</code>
505	<code>_kernel_primitive:NN \wd</code>	<code>\tex_wd:D</code>
506	<code>_kernel_primitive:NN \widowpenalty</code>	<code>\tex_widowpenalty:D</code>
507	<code>_kernel_primitive:NN \write</code>	<code>\tex_write:D</code>
508	<code>_kernel_primitive:NN \xdef</code>	<code>\tex_xdef:D</code>
509	<code>_kernel_primitive:NN \xleaders</code>	<code>\tex_xleaders:D</code>
510	<code>_kernel_primitive:NN \xspaceskip</code>	<code>\tex_xspaceskip:D</code>
511	<code>_kernel_primitive:NN \year</code>	<code>\tex_year:D</code>

Primitives introduced by ϵ -TeX.

512	<code>_kernel_primitive:NN \beginL</code>	<code>\tex_beginL:D</code>
513	<code>_kernel_primitive:NN \beginR</code>	<code>\tex_beginR:D</code>
514	<code>_kernel_primitive:NN \botmarks</code>	<code>\tex_botmarks:D</code>
515	<code>_kernel_primitive:NN \clubpenalties</code>	<code>\tex_clubpenalties:D</code>
516	<code>_kernel_primitive:NN \currentgrouplevel</code>	<code>\tex_currentgrouplevel:D</code>

517	<code>__kernel_primitive:NN \currentgrouptype</code>	<code>\tex_currentgrouptype:D</code>
518	<code>__kernel_primitive:NN \currentifbranch</code>	<code>\tex_currentifbranch:D</code>
519	<code>__kernel_primitive:NN \currentiflevel</code>	<code>\tex_currentiflevel:D</code>
520	<code>__kernel_primitive:NN \currentifttype</code>	<code>\tex_currentifttype:D</code>
521	<code>__kernel_primitive:NN \detokenize</code>	<code>\tex_detokenize:D</code>
522	<code>__kernel_primitive:NN \dimexpr</code>	<code>\tex_dimexpr:D</code>
523	<code>__kernel_primitive:NN \displaywidowpenalties</code>	<code>\tex_displaywidowpenalties:D</code>
524	<code>__kernel_primitive:NN \endL</code>	<code>\tex_endL:D</code>
525	<code>__kernel_primitive:NN \endR</code>	<code>\tex_endR:D</code>
526	<code>__kernel_primitive:NN \eTeXrevision</code>	<code>\tex_eTeXrevision:D</code>
527	<code>__kernel_primitive:NN \eTeXversion</code>	<code>\tex_eTeXversion:D</code>
528	<code>__kernel_primitive:NN \everyeof</code>	<code>\tex_everyeof:D</code>
529	<code>__kernel_primitive:NN \firstmarks</code>	<code>\tex_firstmarks:D</code>
530	<code>__kernel_primitive:NN \fontchardp</code>	<code>\tex_fontchardp:D</code>
531	<code>__kernel_primitive:NN \fontcharht</code>	<code>\tex_fontcharht:D</code>
532	<code>__kernel_primitive:NN \fontcharic</code>	<code>\tex_fontcharic:D</code>
533	<code>__kernel_primitive:NN \fontcharwd</code>	<code>\tex_fontcharwd:D</code>
534	<code>__kernel_primitive:NN \glueexpr</code>	<code>\tex_glueexpr:D</code>
535	<code>__kernel_primitive:NN \glueshrink</code>	<code>\tex_glueshrink:D</code>
536	<code>__kernel_primitive:NN \glueshrinkorder</code>	<code>\tex_glueshrinkorder:D</code>
537	<code>__kernel_primitive:NN \gluestretch</code>	<code>\tex_gluestretch:D</code>
538	<code>__kernel_primitive:NN \gluestretchorder</code>	<code>\tex_gluestretchorder:D</code>
539	<code>__kernel_primitive:NN \gluetomu</code>	<code>\tex_gluetomu:D</code>
540	<code>__kernel_primitive:NN \ifcsname</code>	<code>\tex_ifcsname:D</code>
541	<code>__kernel_primitive:NN \ifdefined</code>	<code>\tex_ifdefined:D</code>
542	<code>__kernel_primitive:NN \iffontchar</code>	<code>\tex_iffontchar:D</code>
543	<code>__kernel_primitive:NN \interactionmode</code>	<code>\tex_interactionmode:D</code>
544	<code>__kernel_primitive:NN \interlinepenalties</code>	<code>\tex_interlinepenalties:D</code>
545	<code>__kernel_primitive:NN \lastlinefit</code>	<code>\tex_lastlinefit:D</code>
546	<code>__kernel_primitive:NN \lastnodetype</code>	<code>\tex_lastnodetype:D</code>
547	<code>__kernel_primitive:NN \marks</code>	<code>\tex_marks:D</code>
548	<code>__kernel_primitive:NN \middle</code>	<code>\tex_middle:D</code>
549	<code>__kernel_primitive:NN \muexpr</code>	<code>\tex_muexpr:D</code>
550	<code>__kernel_primitive:NN \mutoglu</code>	<code>\tex_mutoglu:D</code>
551	<code>__kernel_primitive:NN \numexpr</code>	<code>\tex_numexpr:D</code>
552	<code>__kernel_primitive:NN \pagediscards</code>	<code>\tex_pagediscards:D</code>
553	<code>__kernel_primitive:NN \parshapedimen</code>	<code>\tex_parshapedimen:D</code>
554	<code>__kernel_primitive:NN \parshapeindent</code>	<code>\tex_parshapeindent:D</code>
555	<code>__kernel_primitive:NN \parshapelength</code>	<code>\tex_parshapelength:D</code>
556	<code>__kernel_primitive:NN \predisplaydirection</code>	<code>\tex_predisplaydirection:D</code>
557	<code>__kernel_primitive:NN \protected</code>	<code>\tex_protected:D</code>
558	<code>__kernel_primitive:NN \readline</code>	<code>\tex_readline:D</code>
559	<code>__kernel_primitive:NN \savinghyphcodes</code>	<code>\tex_savinghyphcodes:D</code>
560	<code>__kernel_primitive:NN \savingvdiscards</code>	<code>\tex_savingvdiscards:D</code>
561	<code>__kernel_primitive:NN \scantokens</code>	<code>\tex_scantokens:D</code>
562	<code>__kernel_primitive:NN \showgroups</code>	<code>\tex_showgroups:D</code>
563	<code>__kernel_primitive:NN \showifs</code>	<code>\tex_showifs:D</code>
564	<code>__kernel_primitive:NN \showtokens</code>	<code>\tex_showtokens:D</code>
565	<code>__kernel_primitive:NN \splitbotmarks</code>	<code>\tex_splitbotmarks:D</code>
566	<code>__kernel_primitive:NN \splitdiscards</code>	<code>\tex_splitdiscards:D</code>
567	<code>__kernel_primitive:NN \splitfirstmarks</code>	<code>\tex_splitfirstmarks:D</code>
568	<code>__kernel_primitive:NN \TeXXeTstate</code>	<code>\tex_TeXXeTstate:D</code>
569	<code>__kernel_primitive:NN \topmarks</code>	<code>\tex_topmarks:D</code>
570	<code>__kernel_primitive:NN \tracingassigns</code>	<code>\tex_tracingassigns:D</code>

571	<code>_kernel_primitive:NN</code>	<code>\tracinggroups</code>	<code>\tex_tracinggroups:D</code>
572	<code>_kernel_primitive:NN</code>	<code>\tracingifs</code>	<code>\tex_tracingifs:D</code>
573	<code>_kernel_primitive:NN</code>	<code>\tracingnesting</code>	<code>\tex_tracingnesting:D</code>
574	<code>_kernel_primitive:NN</code>	<code>\tracingscantokens</code>	<code>\tex_tracingscantokens:D</code>
575	<code>_kernel_primitive:NN</code>	<code>\unexpanded</code>	<code>\tex_unexpanded:D</code>
576	<code>_kernel_primitive:NN</code>	<code>\unless</code>	<code>\tex_unless:D</code>
577	<code>_kernel_primitive:NN</code>	<code>\widowpenalties</code>	<code>\tex_widowpenalties:D</code>

Post- ϵ -TeX primitives do not always end up with the same name in all engines, if indeed they are available cross-engine anyway. We therefore take the approach of preferring the shortest name that makes sense. First, we deal with the primitives introduced by pdfTeX which directly relate to PDF output: these are copied with the names unchanged.

578	<code>_kernel_primitive:NN</code>	<code>\pdfannot</code>	<code>\tex_pdfannot:D</code>
579	<code>_kernel_primitive:NN</code>	<code>\pdfcatalog</code>	<code>\tex_pdfcatalog:D</code>
580	<code>_kernel_primitive:NN</code>	<code>\pdfcompresslevel</code>	<code>\tex_pdfcompresslevel:D</code>
581	<code>_kernel_primitive:NN</code>	<code>\pdfcolorstack</code>	<code>\tex_pdfcolorstack:D</code>
582	<code>_kernel_primitive:NN</code>	<code>\pdfcolorstackinit</code>	<code>\tex_pdfcolorstackinit:D</code>
583	<code>_kernel_primitive:NN</code>	<code>\pdfcreationdate</code>	<code>\tex_pdfcreationdate:D</code>
584	<code>_kernel_primitive:NN</code>	<code>\pdfdecimaldigits</code>	<code>\tex_pdfdecimaldigits:D</code>
585	<code>_kernel_primitive:NN</code>	<code>\pdfdest</code>	<code>\tex_pdfdest:D</code>
586	<code>_kernel_primitive:NN</code>	<code>\pdfdestmargin</code>	<code>\tex_pdfdestmargin:D</code>
587	<code>_kernel_primitive:NN</code>	<code>\pdfendlink</code>	<code>\tex_pdfendlink:D</code>
588	<code>_kernel_primitive:NN</code>	<code>\pdfendthread</code>	<code>\tex_pdfendthread:D</code>
589	<code>_kernel_primitive:NN</code>	<code>\pdffontattr</code>	<code>\tex_pdffontattr:D</code>
590	<code>_kernel_primitive:NN</code>	<code>\pdffontname</code>	<code>\tex_pdffontname:D</code>
591	<code>_kernel_primitive:NN</code>	<code>\pdffontobjnum</code>	<code>\tex_pdffontobjnum:D</code>
592	<code>_kernel_primitive:NN</code>	<code>\pdfgamma</code>	<code>\tex_pdfgamma:D</code>
593	<code>_kernel_primitive:NN</code>	<code>\pdfimageapplygamma</code>	<code>\tex_pdfimageapplygamma:D</code>
594	<code>_kernel_primitive:NN</code>	<code>\pdfimagegamma</code>	<code>\tex_pdfimagegamma:D</code>
595	<code>_kernel_primitive:NN</code>	<code>\pdfgentounicode</code>	<code>\tex_pdfgentounicode:D</code>
596	<code>_kernel_primitive:NN</code>	<code>\pdfglyphtounicode</code>	<code>\tex_pdfglyphtounicode:D</code>
597	<code>_kernel_primitive:NN</code>	<code>\pdfhorigin</code>	<code>\tex_pdfhorigin:D</code>
598	<code>_kernel_primitive:NN</code>	<code>\pdfimagehicolor</code>	<code>\tex_pdfimagehicolor:D</code>
599	<code>_kernel_primitive:NN</code>	<code>\pdfimageresolution</code>	<code>\tex_pdfimageresolution:D</code>
600	<code>_kernel_primitive:NN</code>	<code>\pdfincludechars</code>	<code>\tex_pdfincludechars:D</code>
601	<code>_kernel_primitive:NN</code>	<code>\pdfinclusioncopyfonts</code>	<code>\tex_pdfinclusioncopyfonts:D</code>
602	<code>_kernel_primitive:NN</code>	<code>\pdfinclusionerrorlevel</code>	
603		<code>\tex_pdfinclusionerrorlevel:D</code>	
604	<code>_kernel_primitive:NN</code>	<code>\pdfinfo</code>	<code>\tex_pdfinfo:D</code>
605	<code>_kernel_primitive:NN</code>	<code>\pdflastannot</code>	<code>\tex_pdflastannot:D</code>
606	<code>_kernel_primitive:NN</code>	<code>\pdflastlink</code>	<code>\tex_pdflastlink:D</code>
607	<code>_kernel_primitive:NN</code>	<code>\pdflastobj</code>	<code>\tex_pdflastobj:D</code>
608	<code>_kernel_primitive:NN</code>	<code>\pdflastxform</code>	<code>\tex_pdflastxform:D</code>
609	<code>_kernel_primitive:NN</code>	<code>\pdflastximage</code>	<code>\tex_pdflastximage:D</code>
610	<code>_kernel_primitive:NN</code>	<code>\pdflastximagecolordepth</code>	
611		<code>\tex_pdflastximagecolordepth:D</code>	
612	<code>_kernel_primitive:NN</code>	<code>\pdflastximagepages</code>	<code>\tex_pdflastximagepages:D</code>
613	<code>_kernel_primitive:NN</code>	<code>\pdflinkmargin</code>	<code>\tex_pdflinkmargin:D</code>
614	<code>_kernel_primitive:NN</code>	<code>\pdfliteral</code>	<code>\tex_pdfliteral:D</code>
615	<code>_kernel_primitive:NN</code>	<code>\pdfmajorversion</code>	<code>\tex_pdfmajorversion:D</code>
616	<code>_kernel_primitive:NN</code>	<code>\pdfminorversion</code>	<code>\tex_pdfminorversion:D</code>
617	<code>_kernel_primitive:NN</code>	<code>\pdfnames</code>	<code>\tex_pdfnames:D</code>
618	<code>_kernel_primitive:NN</code>	<code>\pdfobj</code>	<code>\tex_pdfobj:D</code>
619	<code>_kernel_primitive:NN</code>	<code>\pdfobjcompresslevel</code>	<code>\tex_pdfobjcompresslevel:D</code>

620	<code>__kernel_primitive:NN \pdfoutline</code>	<code>\tex_pdfoutline:D</code>
621	<code>__kernel_primitive:NN \pdfoutput</code>	<code>\tex_pdfoutput:D</code>
622	<code>__kernel_primitive:NN \pdfpageattr</code>	<code>\tex_pdfpageattr:D</code>
623	<code>__kernel_primitive:NN \pdfpagesattr</code>	<code>\tex_pdfpagesattr:D</code>
624	<code>__kernel_primitive:NN \pdfpagebox</code>	<code>\tex_pdfpagebox:D</code>
625	<code>__kernel_primitive:NN \pdfpageref</code>	<code>\tex_pdfpageref:D</code>
626	<code>__kernel_primitive:NN \pdfpageresources</code>	<code>\tex_pdfpageresources:D</code>
627	<code>__kernel_primitive:NN \pdfpagesattr</code>	<code>\tex_pdfpagesattr:D</code>
628	<code>__kernel_primitive:NN \pdfrefobj</code>	<code>\tex_pdfrefobj:D</code>
629	<code>__kernel_primitive:NN \pdfrefxform</code>	<code>\tex_pdfrefxform:D</code>
630	<code>__kernel_primitive:NN \pdfrefximage</code>	<code>\tex_pdfrefximage:D</code>
631	<code>__kernel_primitive:NN \pdfrestore</code>	<code>\tex_pdfrestore:D</code>
632	<code>__kernel_primitive:NN \pdfretval</code>	<code>\tex_pdfretval:D</code>
633	<code>__kernel_primitive:NN \pdfsave</code>	<code>\tex_pdfsave:D</code>
634	<code>__kernel_primitive:NN \pdfsetmatrix</code>	<code>\tex_pdfsetmatrix:D</code>
635	<code>__kernel_primitive:NN \pdfstartlink</code>	<code>\tex_pdfstartlink:D</code>
636	<code>__kernel_primitive:NN \pdfstartthread</code>	<code>\tex_pdfstartthread:D</code>
637	<code>__kernel_primitive:NN \pdfsuppressptexinfo</code>	<code>\tex_pdfsuppressptexinfo:D</code>
638	<code>__kernel_primitive:NN \pdfthread</code>	<code>\tex_pdfthread:D</code>
639	<code>__kernel_primitive:NN \pdfthreadmargin</code>	<code>\tex_pdfthreadmargin:D</code>
640	<code>__kernel_primitive:NN \pdftrailer</code>	<code>\tex_pdftrailer:D</code>
641	<code>__kernel_primitive:NN \pdfuniqueresname</code>	<code>\tex_pdfuniqueresname:D</code>
642	<code>__kernel_primitive:NN \pdfvorigin</code>	<code>\tex_pdfvorigin:D</code>
643	<code>__kernel_primitive:NN \pdfxform</code>	<code>\tex_pdfxform:D</code>
644	<code>__kernel_primitive:NN \pdfxformname</code>	<code>\tex_pdfxformname:D</code>
645	<code>__kernel_primitive:NN \pdfximage</code>	<code>\tex_pdfximage:D</code>
646	<code>__kernel_primitive:NN \pdfximagebbox</code>	<code>\tex_pdfximagebbox:D</code>

These are not related to PDF output and either already appear in other engines without the `\pdf` prefix, or might reasonably do so at some future stage. We therefore drop the leading `pdf` here.

647	<code>__kernel_primitive:NN \ifpdfabsdim</code>	<code>\tex_ifabsdim:D</code>
648	<code>__kernel_primitive:NN \ifpdfabsnum</code>	<code>\tex_ifabsnum:D</code>
649	<code>__kernel_primitive:NN \ifpdfprimitive</code>	<code>\tex_ifprimitive:D</code>
650	<code>__kernel_primitive:NN \pdfadjustspacing</code>	<code>\tex_adjustspacing:D</code>
651	<code>__kernel_primitive:NN \pdfcopyfont</code>	<code>\tex_copyfont:D</code>
652	<code>__kernel_primitive:NN \pdfdraftmode</code>	<code>\tex_draftmode:D</code>
653	<code>__kernel_primitive:NN \pdfeachlinedepth</code>	<code>\tex_eachlinedepth:D</code>
654	<code>__kernel_primitive:NN \pdfeachlineheight</code>	<code>\tex_eachlineheight:D</code>
655	<code>__kernel_primitive:NN \pdfelapsedtime</code>	<code>\tex_elapsedtime:D</code>
656	<code>__kernel_primitive:NN \pdffirstlineheight</code>	<code>\tex_firstlineheight:D</code>
657	<code>__kernel_primitive:NN \pdffontexpand</code>	<code>\tex_fontexpand:D</code>
658	<code>__kernel_primitive:NN \pdffontsize</code>	<code>\tex_fontsize:D</code>
659	<code>__kernel_primitive:NN \pdfignoreddimen</code>	<code>\tex_ignoreddimen:D</code>
660	<code>__kernel_primitive:NN \pdfinserttht</code>	<code>\tex_inserttht:D</code>
661	<code>__kernel_primitive:NN \pdflastlinedepth</code>	<code>\tex_lastlinedepth:D</code>
662	<code>__kernel_primitive:NN \pdflastxpos</code>	<code>\tex_lastxpos:D</code>
663	<code>__kernel_primitive:NN \pdflastypos</code>	<code>\tex_lastypos:D</code>
664	<code>__kernel_primitive:NN \pdfmapfile</code>	<code>\tex_mapfile:D</code>
665	<code>__kernel_primitive:NN \pdfmapline</code>	<code>\tex_mapline:D</code>
666	<code>__kernel_primitive:NN \pdfnoligatures</code>	<code>\tex_noligatures:D</code>
667	<code>__kernel_primitive:NN \pdfnormaldeviate</code>	<code>\tex_normaldeviate:D</code>
668	<code>__kernel_primitive:NN \pdfpageheight</code>	<code>\tex_pageheight:D</code>
669	<code>__kernel_primitive:NN \pdfpagewidth</code>	<code>\tex_pagewidth:D</code>

```

670 \__kernel_primitive:NN \pdfpkmode \tex_pkmode:D
671 \__kernel_primitive:NN \pdfpkresolution \tex_pkresolution:D
672 \__kernel_primitive:NN \pdfprimitive \tex_primitive:D
673 \__kernel_primitive:NN \pdfprotrudechars \tex_protrudechars:D
674 \__kernel_primitive:NN \pdfpxdimen \tex_pxdimen:D
675 \__kernel_primitive:NN \pdfrandomseed \tex_randomseed:D
676 \__kernel_primitive:NN \pdfresettimer \tex_resettimer:D
677 \__kernel_primitive:NN \pdfsavepos \tex_savepos:D
678 \__kernel_primitive:NN \pdfsetrandomseed \tex_setrandomseed:D
679 \__kernel_primitive:NN \pdfshellescape \tex_shellescape:D
680 \__kernel_primitive:NN \pdftracingfonts \tex_tracingfonts:D
681 \__kernel_primitive:NN \pdfuniformdeviate \tex_uniformdeviate:D

```

The version primitives are not related to PDF mode but are pdfTeX-specific, so again are carried forward unchanged.

```

682 \__kernel_primitive:NN \pdfTeXbanner \tex_pdfTeXbanner:D
683 \__kernel_primitive:NN \pdfTeXrevision \tex_pdfTeXrevision:D
684 \__kernel_primitive:NN \pdfTeXversion \tex_pdfTeXversion:D

```

These ones appear in pdfTeX but don't have pdf in the name at all: no decisions to make.

```

685 \__kernel_primitive:NN \efcode \tex_efcode:D
686 \__kernel_primitive:NN \ifincsname \tex_ifincsname:D
687 \__kernel_primitive:NN \leftmarginkern \tex_leftmarginkern:D
688 \__kernel_primitive:NN \letterspacefont \tex_letterspacefont:D
689 \__kernel_primitive:NN \lpcode \tex_lpcode:D
690 \__kernel_primitive:NN \quitvmode \tex_quitvmode:D
691 \__kernel_primitive:NN \rightmarginkern \tex_rightmarginkern:D
692 \__kernel_primitive:NN \rprcode \tex_rprcode:D
693 \__kernel_primitive:NN \synctex \tex_synctex:D
694 \__kernel_primitive:NN \tagcode \tex_tagcode:D

```

Post pdfTeX primitive availability gets more complex. Both XeTeX and LuaTeX have varying names for some primitives from pdfTeX. Particularly for LuaTeX tracking all of that would be hard. Instead, we now check that we only save primitives if they actually exist.

```

695 </names | package>
696 <*package>
697 \tex_long:D \tex_def:D \use_ii:nn #1#2 {#2}
698 \tex_long:D \tex_def:D \use_none:n #1 { }
699 \tex_long:D \tex_def:D \__kernel_primitive:NN #1#2
700 {
701 \tex_ifdefined:D #1
702 \tex_expandafter:D \use_ii:nn
703 \tex_fi:D
704 \use_none:n { \tex_global:D \tex_let:D #2 #1 }
705 }
706 </package>
707 <*names | package>

```

Some pdfTeX primitives are handled here because they got dropped in LuaTeX but the corresponding internal names are emulated later. The Lua code is already loaded at this point, so we shouldn't overwrite them.

```

708 \__kernel_primitive:NN \pdfstrcmp \tex_strcmp:D
709 \__kernel_primitive:NN \pdffilesize \tex_filesize:D

```

```

710 \__kernel_primitive:NN \pdfmdfivesum \tex_mdfivesum:D
711 \__kernel_primitive:NN \pdffilemoddate \tex_filemoddate:D
712 \__kernel_primitive:NN \pdffiledump \tex_filedump:D

```

X_YTeX-specific primitives. Note that X_YTeX's `\strcmp` is handled earlier and is “rolled up” into `\pdfstrcmp`. A few cross-compatibility names which lack the pdf of the original are handled later.

```

713 \__kernel_primitive:NN \suppressfontnotfounderror
714 \tex_suppressfontnotfounderror:D
715 \__kernel_primitive:NN \XeTeXcharclass \tex_XeTeXcharclass:D
716 \__kernel_primitive:NN \XeTeXcharglyph \tex_XeTeXcharglyph:D
717 \__kernel_primitive:NN \XeTeXcountfeatures \tex_XeTeXcountfeatures:D
718 \__kernel_primitive:NN \XeTeXcountglyphs \tex_XeTeXcountglyphs:D
719 \__kernel_primitive:NN \XeTeXcountselectors \tex_XeTeXcountselectors:D
720 \__kernel_primitive:NN \XeTeXcountvariations \tex_XeTeXcountvariations:D
721 \__kernel_primitive:NN \XeTeXdefaultencoding \tex_XeTeXdefaultencoding:D
722 \__kernel_primitive:NN \XeTeXdashbreakstate \tex_XeTeXdashbreakstate:D
723 \__kernel_primitive:NN \XeTeXfeaturecode \tex_XeTeXfeaturecode:D
724 \__kernel_primitive:NN \XeTeXfeaturename \tex_XeTeXfeaturename:D
725 \__kernel_primitive:NN \XeTeXfindfeaturebyname
726 \tex_XeTeXfindfeaturebyname:D
727 \__kernel_primitive:NN \XeTeXfindselectorbyname
728 \tex_XeTeXfindselectorbyname:D
729 \__kernel_primitive:NN \XeTeXfindvariationbyname
730 \tex_XeTeXfindvariationbyname:D
731 \__kernel_primitive:NN \XeTeXfirstfontchar \tex_XeTeXfirstfontchar:D
732 \__kernel_primitive:NN \XeTeXfonttype \tex_XeTeXfonttype:D
733 \__kernel_primitive:NN \XeTeXgenerateactualtext
734 \tex_XeTeXgenerateactualtext:D
735 \__kernel_primitive:NN \XeTeXglyph \tex_XeTeXglyph:D
736 \__kernel_primitive:NN \XeTeXglyphbounds \tex_XeTeXglyphbounds:D
737 \__kernel_primitive:NN \XeTeXglyphindex \tex_XeTeXglyphindex:D
738 \__kernel_primitive:NN \XeTeXglyphname \tex_XeTeXglyphname:D
739 \__kernel_primitive:NN \XeTeXinputencoding \tex_XeTeXinputencoding:D
740 \__kernel_primitive:NN \XeTeXinputnormalization
741 \tex_XeTeXinputnormalization:D
742 \__kernel_primitive:NN \XeTeXinterchartokenstate
743 \tex_XeTeXinterchartokenstate:D
744 \__kernel_primitive:NN \XeTeXinterchartoks \tex_XeTeXinterchartoks:D
745 \__kernel_primitive:NN \XeTeXisdefaultselector
746 \tex_XeTeXisdefaultselector:D
747 \__kernel_primitive:NN \XeTeXisexclusivefeature
748 \tex_XeTeXisexclusivefeature:D
749 \__kernel_primitive:NN \XeTeXlastfontchar \tex_XeTeXlastfontchar:D
750 \__kernel_primitive:NN \XeTeXlinebreakskip \tex_XeTeXlinebreakskip:D
751 \__kernel_primitive:NN \XeTeXlinebreaklocale \tex_XeTeXlinebreaklocale:D
752 \__kernel_primitive:NN \XeTeXlinebreakpenalty \tex_XeTeXlinebreakpenalty:D
753 \__kernel_primitive:NN \XeTeXOTcountfeatures \tex_XeTeXOTcountfeatures:D
754 \__kernel_primitive:NN \XeTeXOTcountlanguages \tex_XeTeXOTcountlanguages:D
755 \__kernel_primitive:NN \XeTeXOTcountscripts \tex_XeTeXOTcountscripts:D
756 \__kernel_primitive:NN \XeTeXOTfeaturetag \tex_XeTeXOTfeaturetag:D
757 \__kernel_primitive:NN \XeTeXOTlanguagetag \tex_XeTeXOTlanguagetag:D
758 \__kernel_primitive:NN \XeTeXOTscripttag \tex_XeTeXOTscripttag:D
759 \__kernel_primitive:NN \XeTeXpdffile \tex_XeTeXpdffile:D

```

760	<code>__kernel_primitive:NN \XeTeXpdfpagecount</code>	<code>\tex_XeTeXpdfpagecount:D</code>
761	<code>__kernel_primitive:NN \XeTeXpicfile</code>	<code>\tex_XeTeXpicfile:D</code>
762	<code>__kernel_primitive:NN \XeTeXrevision</code>	<code>\tex_XeTeXrevision:D</code>
763	<code>__kernel_primitive:NN \XeTeXselectorname</code>	<code>\tex_XeTeXselectorname:D</code>
764	<code>__kernel_primitive:NN \XeTeXtracingfonts</code>	<code>\tex_XeTeXtracingfonts:D</code>
765	<code>__kernel_primitive:NN \XeTeXupwardsmode</code>	<code>\tex_XeTeXupwardsmode:D</code>
766	<code>__kernel_primitive:NN \XeTeXuseglyphmetrics</code>	<code>\tex_XeTeXuseglyphmetrics:D</code>
767	<code>__kernel_primitive:NN \XeTeXvariation</code>	<code>\tex_XeTeXvariation:D</code>
768	<code>__kernel_primitive:NN \XeTeXvariationdefault</code>	<code>\tex_XeTeXvariationdefault:D</code>
769	<code>__kernel_primitive:NN \XeTeXvariationmax</code>	<code>\tex_XeTeXvariationmax:D</code>
770	<code>__kernel_primitive:NN \XeTeXvariationmin</code>	<code>\tex_XeTeXvariationmin:D</code>
771	<code>__kernel_primitive:NN \XeTeXvariationname</code>	<code>\tex_XeTeXvariationname:D</code>
772	<code>__kernel_primitive:NN \XeTeXversion</code>	<code>\tex_XeTeXversion:D</code>

Primitives from pdfTeX that XeTeX renames: also helps with LuaTeX.

773	<code>__kernel_primitive:NN \creationdate</code>	<code>\tex_creationdate:D</code>
774	<code>__kernel_primitive:NN \elapsedtime</code>	<code>\tex_elapsedtime:D</code>
775	<code>__kernel_primitive:NN \filedump</code>	<code>\tex_filedump:D</code>
776	<code>__kernel_primitive:NN \filemoddate</code>	<code>\tex_filemoddate:D</code>
777	<code>__kernel_primitive:NN \filesize</code>	<code>\tex_filesize:D</code>
778	<code>__kernel_primitive:NN \mdfivesum</code>	<code>\tex_mdfivesum:D</code>
779	<code>__kernel_primitive:NN \ifprimitive</code>	<code>\tex_ifprimitive:D</code>
780	<code>__kernel_primitive:NN \primitive</code>	<code>\tex_primitive:D</code>
781	<code>__kernel_primitive:NN \resettimer</code>	<code>\tex_resettimer:D</code>
782	<code>__kernel_primitive:NN \shellescape</code>	<code>\tex_shellescape:D</code>

Primitives from LuaTeX, some of which have been ported back to XeTeX.

783	<code>__kernel_primitive:NN \alignmark</code>	<code>\tex_alignmark:D</code>
784	<code>__kernel_primitive:NN \aligntab</code>	<code>\tex_aligntab:D</code>
785	<code>__kernel_primitive:NN \attribute</code>	<code>\tex_attribute:D</code>
786	<code>__kernel_primitive:NN \attributedef</code>	<code>\tex_attributedef:D</code>
787	<code>__kernel_primitive:NN \automaticdiscretionary</code>	
788	<code>\tex_automaticdiscretionary:D</code>	
789	<code>__kernel_primitive:NN \automatichyphenmode</code>	<code>\tex_automatichyphenmode:D</code>
790	<code>__kernel_primitive:NN \automatichyphenpenalty</code>	
791	<code>\tex_automatichyphenpenalty:D</code>	
792	<code>__kernel_primitive:NN \beginsname</code>	<code>\tex_beginsname:D</code>
793	<code>__kernel_primitive:NN \bodydir</code>	<code>\tex_bodydir:D</code>
794	<code>__kernel_primitive:NN \bodydirection</code>	<code>\tex_bodydirection:D</code>
795	<code>__kernel_primitive:NN \boxdir</code>	<code>\tex_boxdir:D</code>
796	<code>__kernel_primitive:NN \boxdirection</code>	<code>\tex_boxdirection:D</code>
797	<code>__kernel_primitive:NN \breakafterdirmode</code>	<code>\tex_breakafterdirmode:D</code>
798	<code>__kernel_primitive:NN \catcodetable</code>	<code>\tex_catcodetable:D</code>
799	<code>__kernel_primitive:NN \clearmarks</code>	<code>\tex_clearmarks:D</code>
800	<code>__kernel_primitive:NN \crampeddisplaystyle</code>	<code>\tex_crampeddisplaystyle:D</code>
801	<code>__kernel_primitive:NN \crampedscriptscriptstyle</code>	
802	<code>\tex_crampedscriptscriptstyle:D</code>	
803	<code>__kernel_primitive:NN \crampedscriptstyle</code>	<code>\tex_crampedscriptstyle:D</code>
804	<code>__kernel_primitive:NN \crampedtextstyle</code>	<code>\tex_crampedtextstyle:D</code>
805	<code>__kernel_primitive:NN \csstring</code>	<code>\tex_csstring:D</code>
806	<code>__kernel_primitive:NN \directlua</code>	<code>\tex_directlua:D</code>
807	<code>__kernel_primitive:NN \dviextension</code>	<code>\tex_dviextension:D</code>
808	<code>__kernel_primitive:NN \dvifedback</code>	<code>\tex_dvifedback:D</code>
809	<code>__kernel_primitive:NN \dvivariable</code>	<code>\tex_dvivariable:D</code>
810	<code>__kernel_primitive:NN \eTeXglueshrinkorder</code>	<code>\tex_eTeXglueshrinkorder:D</code>

811	_kernel_primitive:NN	\eTeXgluestretchorder	\tex_eTeXgluestretchorder:D
812	_kernel_primitive:NN	\etoksapp	\tex_etoksapp:D
813	_kernel_primitive:NN	\etokspre	\tex_etokspre:D
814	_kernel_primitive:NN	\exceptionpenalty	\tex_exceptionpenalty:D
815	_kernel_primitive:NN	\explicithyphenpenalty	\tex_explicithyphenpenalty:D
816	_kernel_primitive:NN	\expanded	\tex_expanded:D
817	_kernel_primitive:NN	\explicitdiscretionary	\tex_explicitdiscretionary:D
818	_kernel_primitive:NN	\firstvalidlanguage	\tex_firstvalidlanguage:D
819	_kernel_primitive:NN	\fontid	\tex_fontid:D
820	_kernel_primitive:NN	\formatname	\tex_formatname:D
821	_kernel_primitive:NN	\hjcode	\tex_hjcode:D
822	_kernel_primitive:NN	\hpack	\tex_hpack:D
823	_kernel_primitive:NN	\hyphenationbounds	\tex_hyphenationbounds:D
824	_kernel_primitive:NN	\hyphenationmin	\tex_hyphenationmin:D
825	_kernel_primitive:NN	\hyphenpenaltymode	\tex_hyphenpenaltymode:D
826	_kernel_primitive:NN	\gleaders	\tex_gleaders:D
827	_kernel_primitive:NN	\ifcondition	\tex_ifcondition:D
828	_kernel_primitive:NN	\immediateassigned	\tex_immediateassigned:D
829	_kernel_primitive:NN	\immediateassignment	\tex_immediateassignment:D
830	_kernel_primitive:NN	\initcatcodetable	\tex_initcatcodetable:D
831	_kernel_primitive:NN	\lastnamedcs	\tex_lastnamedcs:D
832	_kernel_primitive:NN	\latelua	\tex_latelua:D
833	_kernel_primitive:NN	\lateluafunction	\tex_lateluafunction:D
834	_kernel_primitive:NN	\leftghost	\tex_leftghost:D
835	_kernel_primitive:NN	\letcharcode	\tex_letcharcode:D
836	_kernel_primitive:NN	\linedir	\tex_linedir:D
837	_kernel_primitive:NN	\linedirection	\tex_linedirection:D
838	_kernel_primitive:NN	\localbrokenpenalty	\tex_localbrokenpenalty:D
839	_kernel_primitive:NN	\localinterlinepenalty	\tex_localinterlinepenalty:D
840	_kernel_primitive:NN	\luabytecode	\tex_luabytecode:D
841	_kernel_primitive:NN	\luabytecodecall	\tex_luabytecodecall:D
842	_kernel_primitive:NN	\luacopyinputnodes	\tex_luacopyinputnodes:D
843	_kernel_primitive:NN	\luadef	\tex_luadef:D
844	_kernel_primitive:NN	\lcalleftbox	\tex_lcalleftbox:D
845	_kernel_primitive:NN	\lcalrightbox	\tex_lcalrightbox:D
846	_kernel_primitive:NN	\luaescapestring	\tex_luaescapestring:D
847	_kernel_primitive:NN	\luafunction	\tex_luafunction:D
848	_kernel_primitive:NN	\luafunctioncall	\tex_luafunctioncall:D
849	_kernel_primitive:NN	\luatexbanner	\tex_luatexbanner:D
850	_kernel_primitive:NN	\luatexrevision	\tex_luatexrevision:D
851	_kernel_primitive:NN	\luatexversion	\tex_luatexversion:D
852	_kernel_primitive:NN	\mathdelimitersmode	\tex_mathdelimitersmode:D
853	_kernel_primitive:NN	\mathdir	\tex_mathdir:D
854	_kernel_primitive:NN	\mathdirection	\tex_mathdirection:D
855	_kernel_primitive:NN	\mathdisplayskipmode	\tex_mathdisplayskipmode:D
856	_kernel_primitive:NN	\matheqnogapstep	\tex_matheqnogapstep:D
857	_kernel_primitive:NN	\mathnolimitsmode	\tex_mathnolimitsmode:D
858	_kernel_primitive:NN	\mathoption	\tex_mathoption:D
859	_kernel_primitive:NN	\mathpenaltiesmode	\tex_mathpenaltiesmode:D
860	_kernel_primitive:NN	\mathrulesfam	\tex_mathrulesfam:D
861	_kernel_primitive:NN	\mathscriptsmode	\tex_mathscriptsmode:D
862	_kernel_primitive:NN	\mathscriptboxmode	\tex_mathscriptboxmode:D
863	_kernel_primitive:NN	\mathscriptcharmode	\tex_mathscriptcharmode:D
864	_kernel_primitive:NN	\mathstyle	\tex_mathstyle:D

865	<code>__kernel_primitive:NN \mathsurroundmode</code>	<code>\tex_mathsurroundmode:D</code>
866	<code>__kernel_primitive:NN \mathsurroundskip</code>	<code>\tex_mathsurroundskip:D</code>
867	<code>__kernel_primitive:NN \nohrule</code>	<code>\tex_nohrule:D</code>
868	<code>__kernel_primitive:NN \nokerns</code>	<code>\tex_nokerns:D</code>
869	<code>__kernel_primitive:NN \noligs</code>	<code>\tex_noligs:D</code>
870	<code>__kernel_primitive:NN \nospaces</code>	<code>\tex_nospaces:D</code>
871	<code>__kernel_primitive:NN \novrule</code>	<code>\tex_novrule:D</code>
872	<code>__kernel_primitive:NN \outputbox</code>	<code>\tex_outputbox:D</code>
873	<code>__kernel_primitive:NN \pagebottomoffset</code>	<code>\tex_pagebottomoffset:D</code>
874	<code>__kernel_primitive:NN \pagedir</code>	<code>\tex_pagedir:D</code>
875	<code>__kernel_primitive:NN \pagedirection</code>	<code>\tex_pagedirection:D</code>
876	<code>__kernel_primitive:NN \pageleftoffset</code>	<code>\tex_pageleftoffset:D</code>
877	<code>__kernel_primitive:NN \pagerightoffset</code>	<code>\tex_pagerightoffset:D</code>
878	<code>__kernel_primitive:NN \pagetopoffset</code>	<code>\tex_pagetopoffset:D</code>
879	<code>__kernel_primitive:NN \pardir</code>	<code>\tex_pardir:D</code>
880	<code>__kernel_primitive:NN \pardirection</code>	<code>\tex_pardirection:D</code>
881	<code>__kernel_primitive:NN \pdfextension</code>	<code>\tex_pdfextension:D</code>
882	<code>__kernel_primitive:NN \pdffeedback</code>	<code>\tex_pdffeedback:D</code>
883	<code>__kernel_primitive:NN \pdfvariable</code>	<code>\tex_pdfvariable:D</code>
884	<code>__kernel_primitive:NN \postexhyphenchar</code>	<code>\tex_postexhyphenchar:D</code>
885	<code>__kernel_primitive:NN \posthyphenchar</code>	<code>\tex_posthyphenchar:D</code>
886	<code>__kernel_primitive:NN \prebinoppenalty</code>	<code>\tex_prebinoppenalty:D</code>
887	<code>__kernel_primitive:NN \predisplaygapfactor</code>	<code>\tex_predisplaygapfactor:D</code>
888	<code>__kernel_primitive:NN \preexhyphenchar</code>	<code>\tex_preexhyphenchar:D</code>
889	<code>__kernel_primitive:NN \prehyphenchar</code>	<code>\tex_prehyphenchar:D</code>
890	<code>__kernel_primitive:NN \prerelpenalty</code>	<code>\tex_prerelpenalty:D</code>
891	<code>__kernel_primitive:NN \rightghost</code>	<code>\tex_rightghost:D</code>
892	<code>__kernel_primitive:NN \savecatcodetable</code>	<code>\tex_savecatcodetable:D</code>
893	<code>__kernel_primitive:NN \scantextokens</code>	<code>\tex_scantextokens:D</code>
894	<code>__kernel_primitive:NN \setfontid</code>	<code>\tex_setfontid:D</code>
895	<code>__kernel_primitive:NN \shapemode</code>	<code>\tex_shapemode:D</code>
896	<code>__kernel_primitive:NN \suppressifcsnameerror</code>	<code>\tex_suppressifcsnameerror:D</code>
897	<code>__kernel_primitive:NN \suppresslongerror</code>	<code>\tex_suppresslongerror:D</code>
898	<code>__kernel_primitive:NN \suppressmathparerror</code>	<code>\tex_suppressmathparerror:D</code>
899	<code>__kernel_primitive:NN \suppressoutererror</code>	<code>\tex_suppressoutererror:D</code>
900	<code>__kernel_primitive:NN \suppressprimitiveerror</code>	
901	<code>\tex_suppressprimitiveerror:D</code>	
902	<code>__kernel_primitive:NN \textdir</code>	<code>\tex_textdir:D</code>
903	<code>__kernel_primitive:NN \textdirection</code>	<code>\tex_textdirection:D</code>
904	<code>__kernel_primitive:NN \toksapp</code>	<code>\tex_toksapp:D</code>
905	<code>__kernel_primitive:NN \tokspre</code>	<code>\tex_tokspre:D</code>
906	<code>__kernel_primitive:NN \tpack</code>	<code>\tex_tpack:D</code>
907	<code>__kernel_primitive:NN \vpack</code>	<code>\tex_vpack:D</code>

Primitives from pdfTeX that LuaTeX renames.

908	<code>__kernel_primitive:NN \adjustspacing</code>	<code>\tex_adjustspacing:D</code>
909	<code>__kernel_primitive:NN \copyfont</code>	<code>\tex_copyfont:D</code>
910	<code>__kernel_primitive:NN \draftmode</code>	<code>\tex_draftmode:D</code>
911	<code>__kernel_primitive:NN \expandglyphsinfont</code>	<code>\tex_fontexpand:D</code>
912	<code>__kernel_primitive:NN \ifabsdim</code>	<code>\tex_ifabsdim:D</code>
913	<code>__kernel_primitive:NN \ifabsnum</code>	<code>\tex_ifabsnum:D</code>
914	<code>__kernel_primitive:NN \ignoreligaturesinfont</code>	<code>\tex_ignoreligaturesinfont:D</code>
915	<code>__kernel_primitive:NN \insertht</code>	<code>\tex_insertht:D</code>
916	<code>__kernel_primitive:NN \lastsavedboxresourceindex</code>	
917	<code>\tex_pdflastxform:D</code>	

```

918 \__kernel_primitive:NN \lastsavedimageresourceindex
919 \tex_pdflastximage:D
920 \__kernel_primitive:NN \lastsavedimageresourcepages
921 \tex_pdflastximagepages:D
922 \__kernel_primitive:NN \lastxpos \tex_lastxpos:D
923 \__kernel_primitive:NN \lastypos \tex_lastypos:D
924 \__kernel_primitive:NN \normaldeviate \tex_normaldeviate:D
925 \__kernel_primitive:NN \outputmode \tex_pdfoutput:D
926 \__kernel_primitive:NN \pageheight \tex_pageheight:D
927 \__kernel_primitive:NN \pagewidth \tex_pagewidth:D
928 \__kernel_primitive:NN \protrudechars \tex_protrudechars:D
929 \__kernel_primitive:NN \pxdimen \tex_pxdimen:D
930 \__kernel_primitive:NN \randomseed \tex_randomseed:D
931 \__kernel_primitive:NN \useboxresource \tex_pdfrefxform:D
932 \__kernel_primitive:NN \useimageresource \tex_pdfrefximage:D
933 \__kernel_primitive:NN \savepos \tex_savepos:D
934 \__kernel_primitive:NN \saveboxresource \tex_pdfxform:D
935 \__kernel_primitive:NN \saveimageresource \tex_pdfximage:D
936 \__kernel_primitive:NN \setrandomseed \tex_setrandomseed:D
937 \__kernel_primitive:NN \tracingfonts \tex_tracingfonts:D
938 \__kernel_primitive:NN \uniformdeviate \tex_uniformdeviate:D

```

The set of Unicode math primitives were introduced by XeTeX and LuaTeX in a somewhat complex fashion: a few first as \XeTeX... which were then renamed with LuaTeX having a lot more. These names now all start \U... and mainly \Umath....

```

939 \__kernel_primitive:NN \Uchar \tex_Uchar:D
940 \__kernel_primitive:NN \Ucharcat \tex_Ucharcat:D
941 \__kernel_primitive:NN \Udelcode \tex_Udelcode:D
942 \__kernel_primitive:NN \Udelcodenum \tex_Udelcodenum:D
943 \__kernel_primitive:NN \Udelimiter \tex_Udelimiter:D
944 \__kernel_primitive:NN \Udelimiterover \tex_Udelimiterover:D
945 \__kernel_primitive:NN \Udelimiterunder \tex_Udelimiterunder:D
946 \__kernel_primitive:NN \Uxextensible \tex_Uxextensible:D
947 \__kernel_primitive:NN \Umathaccent \tex_Umathaccent:D
948 \__kernel_primitive:NN \Umathaxis \tex_Umathaxis:D
949 \__kernel_primitive:NN \Umathbinbinspacing \tex_Umathbinbinspacing:D
950 \__kernel_primitive:NN \Umathbinclonespacing \tex_Umathbinclonespacing:D
951 \__kernel_primitive:NN \Umathbininnerspacing \tex_Umathbininnerspacing:D
952 \__kernel_primitive:NN \Umathbinopenspacing \tex_Umathbinopenspacing:D
953 \__kernel_primitive:NN \Umathbinopspacing \tex_Umathbinopspacing:D
954 \__kernel_primitive:NN \Umathbinordspacing \tex_Umathbinordspacing:D
955 \__kernel_primitive:NN \Umathbinpunctspacing \tex_Umathbinpunctspacing:D
956 \__kernel_primitive:NN \Umathbinrelspacing \tex_Umathbinrelspacing:D
957 \__kernel_primitive:NN \Umathchar \tex_Umathchar:D
958 \__kernel_primitive:NN \Umathcharclass \tex_Umathcharclass:D
959 \__kernel_primitive:NN \Umathchardef \tex_Umathchardef:D
960 \__kernel_primitive:NN \Umathcharfam \tex_Umathcharfam:D
961 \__kernel_primitive:NN \Umathcharnum \tex_Umathcharnum:D
962 \__kernel_primitive:NN \Umathcharnumdef \tex_Umathcharnumdef:D
963 \__kernel_primitive:NN \Umathcharslot \tex_Umathcharslot:D
964 \__kernel_primitive:NN \Umathclosebinspacing \tex_Umathclosebinspacing:D
965 \__kernel_primitive:NN \Umathcloseclonespacing
966 \tex_Umathcloseclonespacing:D
967 \__kernel_primitive:NN \Umathcloseinnerspacing

```

```

968 \tex_Umathcloseinnerspacing:D
969 \__kernel_primitive:NN \Umathcloseopenspacing \tex_Umathcloseopenspacing:D
970 \__kernel_primitive:NN \Umathcloseopspacing \tex_Umathcloseopspacing:D
971 \__kernel_primitive:NN \Umathcloseordspacing \tex_Umathcloseordspacing:D
972 \__kernel_primitive:NN \Umathclosepunctspacing
973 \tex_Umathclosepunctspacing:D
974 \__kernel_primitive:NN \Umathcloserelspacing \tex_Umathcloserelspacing:D
975 \__kernel_primitive:NN \Umathcode \tex_Umathcode:D
976 \__kernel_primitive:NN \Umathcodenum \tex_Umathcodenum:D
977 \__kernel_primitive:NN \Umathconnectoroverlapmin
978 \tex_Umathconnectoroverlapmin:D
979 \__kernel_primitive:NN \Umathfractiondelsize \tex_Umathfractiondelsize:D
980 \__kernel_primitive:NN \Umathfractiondenomdown
981 \tex_Umathfractiondenomdown:D
982 \__kernel_primitive:NN \Umathfractiondenomvgap
983 \tex_Umathfractiondenomvgap:D
984 \__kernel_primitive:NN \Umathfractionnumup \tex_Umathfractionnumup:D
985 \__kernel_primitive:NN \Umathfractionnumvgap \tex_Umathfractionnumvgap:D
986 \__kernel_primitive:NN \Umathfractionrule \tex_Umathfractionrule:D
987 \__kernel_primitive:NN \Umathinnerbinspacing \tex_Umathinnerbinspacing:D
988 \__kernel_primitive:NN \Umathinnerclosespacing
989 \tex_Umathinnerclosespacing:D
990 \__kernel_primitive:NN \Umathinnerinnerspacing
991 \tex_Umathinnerinnerspacing:D
992 \__kernel_primitive:NN \Umathinneropenspacing \tex_Umathinneropenspacing:D
993 \__kernel_primitive:NN \Umathinneropspacing \tex_Umathinneropspacing:D
994 \__kernel_primitive:NN \Umathinnerordspacing \tex_Umathinnerordspacing:D
995 \__kernel_primitive:NN \Umathinnerpunctspacing
996 \tex_Umathinnerpunctspacing:D
997 \__kernel_primitive:NN \Umathinnerrelspacing \tex_Umathinnerrelspacing:D
998 \__kernel_primitive:NN \Umathlimitabovebgap \tex_Umathlimitabovebgap:D
999 \__kernel_primitive:NN \Umathlimitabovekern \tex_Umathlimitabovekern:D
1000 \__kernel_primitive:NN \Umathlimitabovevgap \tex_Umathlimitabovevgap:D
1001 \__kernel_primitive:NN \Umathlimitbelowbgap \tex_Umathlimitbelowbgap:D
1002 \__kernel_primitive:NN \Umathlimitbelowkern \tex_Umathlimitbelowkern:D
1003 \__kernel_primitive:NN \Umathlimitbelowvgap \tex_Umathlimitbelowvgap:D
1004 \__kernel_primitive:NN \Umathnolimitsubfactor \tex_Umathnolimitsubfactor:D
1005 \__kernel_primitive:NN \Umathnolimitsupfactor \tex_Umathnolimitsupfactor:D
1006 \__kernel_primitive:NN \Umathopbinspacing \tex_Umathopbinspacing:D
1007 \__kernel_primitive:NN \Umathopclosespacing \tex_Umathopclosespacing:D
1008 \__kernel_primitive:NN \Umathopenbinspacing \tex_Umathopenbinspacing:D
1009 \__kernel_primitive:NN \Umathopenenclosespacing \tex_Umathopenenclosespacing:D
1010 \__kernel_primitive:NN \Umathopeninnerspacing \tex_Umathopeninnerspacing:D
1011 \__kernel_primitive:NN \Umathopenopenspacing \tex_Umathopenopenspacing:D
1012 \__kernel_primitive:NN \Umathopenopspacing \tex_Umathopenopspacing:D
1013 \__kernel_primitive:NN \Umathopenordspacing \tex_Umathopenordspacing:D
1014 \__kernel_primitive:NN \Umathopenpunctspacing \tex_Umathopenpunctspacing:D
1015 \__kernel_primitive:NN \Umathopenrelspacing \tex_Umathopenrelspacing:D
1016 \__kernel_primitive:NN \Umathoperatorsizsize \tex_Umathoperatorsizsize:D
1017 \__kernel_primitive:NN \Umathopinnerspacing \tex_Umathopinnerspacing:D
1018 \__kernel_primitive:NN \Umathopopenspacing \tex_Umathopopenspacing:D
1019 \__kernel_primitive:NN \Umathopopspacing \tex_Umathopopspacing:D
1020 \__kernel_primitive:NN \Umathopordspacing \tex_Umathopordspacing:D
1021 \__kernel_primitive:NN \Umathoppunctspacing \tex_Umathoppunctspacing:D

```

```

1022 \__kernel_primitive:NN \Umathoprelspacing \tex_Umathoprelspacing:D
1023 \__kernel_primitive:NN \Umathordbinspacing \tex_Umathordbinspacing:D
1024 \__kernel_primitive:NN \Umathordclosespacing \tex_Umathordclosespacing:D
1025 \__kernel_primitive:NN \Umathordinnerspacing \tex_Umathordinnerspacing:D
1026 \__kernel_primitive:NN \Umathordopenspacing \tex_Umathordopenspacing:D
1027 \__kernel_primitive:NN \Umathordopspacing \tex_Umathordopspacing:D
1028 \__kernel_primitive:NN \Umathordordspacing \tex_Umathordordspacing:D
1029 \__kernel_primitive:NN \Umathordpunctspacing \tex_Umathordpunctspacing:D
1030 \__kernel_primitive:NN \Umathordrelspacing \tex_Umathordrelspacing:D
1031 \__kernel_primitive:NN \Umathoverbarkern \tex_Umathoverbarkern:D
1032 \__kernel_primitive:NN \Umathoverbarrule \tex_Umathoverbarrule:D
1033 \__kernel_primitive:NN \Umathoverbarvgap \tex_Umathoverbarvgap:D
1034 \__kernel_primitive:NN \Umathoverdelimiterbgap
1035 \tex_Umathoverdelimiterbgap:D
1036 \__kernel_primitive:NN \Umathoverdelimitervgap
1037 \tex_Umathoverdelimitervgap:D
1038 \__kernel_primitive:NN \Umathpunctbinspacing \tex_Umathpunctbinspacing:D
1039 \__kernel_primitive:NN \Umathpunctclosespacing
1040 \tex_Umathpunctclosespacing:D
1041 \__kernel_primitive:NN \Umathpunctinnerspacing
1042 \tex_Umathpunctinnerspacing:D
1043 \__kernel_primitive:NN \Umathpunctopenspacing \tex_Umathpunctopenspacing:D
1044 \__kernel_primitive:NN \Umathpuncttopspacing \tex_Umathpuncttopspacing:D
1045 \__kernel_primitive:NN \Umathpunctordspacing \tex_Umathpunctordspacing:D
1046 \__kernel_primitive:NN \Umathpunctpunctspacing
1047 \tex_Umathpunctpunctspacing:D
1048 \__kernel_primitive:NN \Umathpunctrelspacing \tex_Umathpunctrelspacing:D
1049 \__kernel_primitive:NN \Umathquad \tex_Umathquad:D
1050 \__kernel_primitive:NN \Umathradicaldegreeafter
1051 \tex_Umathradicaldegreeafter:D
1052 \__kernel_primitive:NN \Umathradicaldegreebefore
1053 \tex_Umathradicaldegreebefore:D
1054 \__kernel_primitive:NN \Umathradicaldegreeraise
1055 \tex_Umathradicaldegreeraise:D
1056 \__kernel_primitive:NN \Umathradicalkern \tex_Umathradicalkern:D
1057 \__kernel_primitive:NN \Umathradicalrule \tex_Umathradicalrule:D
1058 \__kernel_primitive:NN \Umathradicalvgap \tex_Umathradicalvgap:D
1059 \__kernel_primitive:NN \Umathrelbinspacing \tex_Umathrelbinspacing:D
1060 \__kernel_primitive:NN \Umathrelclosespacing \tex_Umathrelclosespacing:D
1061 \__kernel_primitive:NN \Umathrelinnerspacing \tex_Umathrelinnerspacing:D
1062 \__kernel_primitive:NN \Umathrelopenspacing \tex_Umathrelopenspacing:D
1063 \__kernel_primitive:NN \Umathreltopspacing \tex_Umathreltopspacing:D
1064 \__kernel_primitive:NN \Umathrelordspacing \tex_Umathrelordspacing:D
1065 \__kernel_primitive:NN \Umathrelpunctspacing \tex_Umathrelpunctspacing:D
1066 \__kernel_primitive:NN \Umathrelrelspacing \tex_Umathrelrelspacing:D
1067 \__kernel_primitive:NN \Umathskewedfractionhgap
1068 \tex_Umathskewedfractionhgap:D
1069 \__kernel_primitive:NN \Umathskewedfractionvgap
1070 \tex_Umathskewedfractionvgap:D
1071 \__kernel_primitive:NN \Umathspaceafterscript \tex_Umathspaceafterscript:D
1072 \__kernel_primitive:NN \Umathstackdenomdown \tex_Umathstackdenomdown:D
1073 \__kernel_primitive:NN \Umathstacknumup \tex_Umathstacknumup:D
1074 \__kernel_primitive:NN \Umathstackvgap \tex_Umathstackvgap:D
1075 \__kernel_primitive:NN \Umathsubshiftdown \tex_Umathsubshiftdown:D

```

1076	<code>__kernel_primitive:NN \Umathsubshiftdrop</code>	<code>\tex_Umathsubshiftdrop:D</code>
1077	<code>__kernel_primitive:NN \Umathsubsupshiftdown</code>	<code>\tex_Umathsubsupshiftdown:D</code>
1078	<code>__kernel_primitive:NN \Umathsubsupvgap</code>	<code>\tex_Umathsubsupvgap:D</code>
1079	<code>__kernel_primitive:NN \Umathsubtopmax</code>	<code>\tex_Umathsubtopmax:D</code>
1080	<code>__kernel_primitive:NN \Umathsupbottommin</code>	<code>\tex_Umathsupbottommin:D</code>
1081	<code>__kernel_primitive:NN \Umathsupshiftdrop</code>	<code>\tex_Umathsupshiftdrop:D</code>
1082	<code>__kernel_primitive:NN \Umathsupshiftup</code>	<code>\tex_Umathsupshiftup:D</code>
1083	<code>__kernel_primitive:NN \Umathsupsubbottommax</code>	<code>\tex_Umathsupsubbottommax:D</code>
1084	<code>__kernel_primitive:NN \Umathunderbarkern</code>	<code>\tex_Umathunderbarkern:D</code>
1085	<code>__kernel_primitive:NN \Umathunderbarrule</code>	<code>\tex_Umathunderbarrule:D</code>
1086	<code>__kernel_primitive:NN \Umathunderbarvgap</code>	<code>\tex_Umathunderbarvgap:D</code>
1087	<code>__kernel_primitive:NN \Umathunderdelimitervgap</code>	
1088	<code>\tex_Umathunderdelimitervgap:D</code>	
1089	<code>__kernel_primitive:NN \Umathunderdelimitervgap</code>	
1090	<code>\tex_Umathunderdelimitervgap:D</code>	
1091	<code>__kernel_primitive:NN \Unosubscript</code>	<code>\tex_Unosubscript:D</code>
1092	<code>__kernel_primitive:NN \Unosuperscript</code>	<code>\tex_Unosuperscript:D</code>
1093	<code>__kernel_primitive:NN \Uoverdelimiter</code>	<code>\tex_Uoverdelimiter:D</code>
1094	<code>__kernel_primitive:NN \Uradical</code>	<code>\tex_Uradical:D</code>
1095	<code>__kernel_primitive:NN \Uroot</code>	<code>\tex_Uroot:D</code>
1096	<code>__kernel_primitive:NN \Uskewed</code>	<code>\tex_Uskewed:D</code>
1097	<code>__kernel_primitive:NN \Uskewedwithdelims</code>	<code>\tex_Uskewedwithdelims:D</code>
1098	<code>__kernel_primitive:NN \Ustack</code>	<code>\tex_Ustack:D</code>
1099	<code>__kernel_primitive:NN \Ustartdisplaymath</code>	<code>\tex_Ustartdisplaymath:D</code>
1100	<code>__kernel_primitive:NN \Ustartmath</code>	<code>\tex_Ustartmath:D</code>
1101	<code>__kernel_primitive:NN \Ustopdisplaymath</code>	<code>\tex_Ustopdisplaymath:D</code>
1102	<code>__kernel_primitive:NN \Ustopmath</code>	<code>\tex_Ustopmath:D</code>
1103	<code>__kernel_primitive:NN \Usubscript</code>	<code>\tex_Usubscript:D</code>
1104	<code>__kernel_primitive:NN \Usuperscript</code>	<code>\tex_Usuperscript:D</code>
1105	<code>__kernel_primitive:NN \Uunderdelimiter</code>	<code>\tex_Uunderdelimiter:D</code>
1106	<code>__kernel_primitive:NN \Uvextensible</code>	<code>\tex_Uvextensible:D</code>

Primitives from pTeX.

1107	<code>__kernel_primitive:NN \autospaceing</code>	<code>\tex_autospaceing:D</code>
1108	<code>__kernel_primitive:NN \autoxspaceing</code>	<code>\tex_autoxspaceing:D</code>
1109	<code>__kernel_primitive:NN \currentcjktoken</code>	<code>\tex_currentcjktoken:D</code>
1110	<code>__kernel_primitive:NN \currentspacingmode</code>	<code>\tex_currentspacingmode:D</code>
1111	<code>__kernel_primitive:NN \currentxspacingmode</code>	<code>\tex_currentxspacingmode:D</code>
1112	<code>__kernel_primitive:NN \disinhibitglue</code>	<code>\tex_disinhibitglue:D</code>
1113	<code>__kernel_primitive:NN \dtou</code>	<code>\tex_dtou:D</code>
1114	<code>__kernel_primitive:NN \epTeXinputencoding</code>	<code>\tex_epTeXinputencoding:D</code>
1115	<code>__kernel_primitive:NN \epTeXversion</code>	<code>\tex_epTeXversion:D</code>
1116	<code>__kernel_primitive:NN \euc</code>	<code>\tex_euc:D</code>
1117	<code>__kernel_primitive:NN \hfi</code>	<code>\tex_hfi:D</code>
1118	<code>__kernel_primitive:NN \ifdbbox</code>	<code>\tex_ifdbbox:D</code>
1119	<code>__kernel_primitive:NN \ifddir</code>	<code>\tex_ifddir:D</code>
1120	<code>__kernel_primitive:NN \ifjfont</code>	<code>\tex_ifjfont:D</code>
1121	<code>__kernel_primitive:NN \ifmbox</code>	<code>\tex_ifmbox:D</code>
1122	<code>__kernel_primitive:NN \ifmdir</code>	<code>\tex_ifmdir:D</code>
1123	<code>__kernel_primitive:NN \iftbox</code>	<code>\tex_iftbox:D</code>
1124	<code>__kernel_primitive:NN \iftfont</code>	<code>\tex_iftfont:D</code>
1125	<code>__kernel_primitive:NN \iftdir</code>	<code>\tex_iftdir:D</code>
1126	<code>__kernel_primitive:NN \ifybox</code>	<code>\tex_ifybox:D</code>
1127	<code>__kernel_primitive:NN \ifydir</code>	<code>\tex_ifydir:D</code>
1128	<code>__kernel_primitive:NN \inhibitglue</code>	<code>\tex_inhibitglue:D</code>

1129	_kernel_primitive:NN	\inhibitxspcode	\tex_inhibitxspcode:D
1130	_kernel_primitive:NN	\jcharwidowpenalty	\tex_jcharwidowpenalty:D
1131	_kernel_primitive:NN	\jfam	\tex_jfam:D
1132	_kernel_primitive:NN	\jfont	\tex_jfont:D
1133	_kernel_primitive:NN	\jis	\tex_jis:D
1134	_kernel_primitive:NN	\kanjiskip	\tex_kanjiskip:D
1135	_kernel_primitive:NN	\kansuji	\tex_kansuji:D
1136	_kernel_primitive:NN	\kansujichar	\tex_kansujichar:D
1137	_kernel_primitive:NN	\kcatcode	\tex_kcatcode:D
1138	_kernel_primitive:NN	\kuten	\tex_kuten:D
1139	_kernel_primitive:NN	\lastnodechar	\tex_lastnodechar:D
1140	_kernel_primitive:NN	\lastnodesubtype	\tex_lastnodesubtype:D
1141	_kernel_primitive:NN	\noautospaceing	\tex_noautospaceing:D
1142	_kernel_primitive:NN	\noautoxspaceing	\tex_noautoxspaceing:D
1143	_kernel_primitive:NN	\pagefistretch	\tex_pagefistretch:D
1144	_kernel_primitive:NN	\postbreakpenalty	\tex_postbreakpenalty:D
1145	_kernel_primitive:NN	\prebreakpenalty	\tex_prebreakpenalty:D
1146	_kernel_primitive:NN	\ptexminorversion	\tex_ptexminorversion:D
1147	_kernel_primitive:NN	\ptexrevision	\tex_ptexrevision:D
1148	_kernel_primitive:NN	\ptexversion	\tex_ptexversion:D
1149	_kernel_primitive:NN	\readpapersizespecial	\tex_readpapersizespecial:D
1150	_kernel_primitive:NN	\scriptbaselineshiftfactor	
1151		\tex_scriptbaselineshiftfactor:D	
1152	_kernel_primitive:NN	\scriptscriptbaselineshiftfactor	
1153		\tex_scriptscriptbaselineshiftfactor:D	
1154	_kernel_primitive:NN	\showmode	\tex_showmode:D
1155	_kernel_primitive:NN	\sjis	\tex_sjis:D
1156	_kernel_primitive:NN	\tate	\tex_tate:D
1157	_kernel_primitive:NN	\tbaselineshift	\tex_tbaselineshift:D
1158	_kernel_primitive:NN	\textbaselineshiftfactor	
1159		\tex_textbaselineshiftfactor:D	
1160	_kernel_primitive:NN	\tfont	\tex_tfont:D
1161	_kernel_primitive:NN	\xkanjiskip	\tex_xkanjiskip:D
1162	_kernel_primitive:NN	\xspcode	\tex_xspcode:D
1163	_kernel_primitive:NN	\ybaselineshift	\tex_ybaselineshift:D
1164	_kernel_primitive:NN	\yoko	\tex_yoko:D
1165	_kernel_primitive:NN	\vfi	\tex_vfi:D

Primitives from upTeX.

1166	_kernel_primitive:NN	\currentcjktoken	\tex_currentcjktoken:D
1167	_kernel_primitive:NN	\disablecjktoken	\tex_disablecjktoken:D
1168	_kernel_primitive:NN	\enablecjktoken	\tex_enablecjktoken:D
1169	_kernel_primitive:NN	\forcecjktoken	\tex_forcecjktoken:D
1170	_kernel_primitive:NN	\kchar	\tex_kchar:D
1171	_kernel_primitive:NN	\kchardef	\tex_kchardef:D
1172	_kernel_primitive:NN	\kuten	\tex_kuten:D
1173	_kernel_primitive:NN	\ucs	\tex_ucs:D
1174	_kernel_primitive:NN	\uptexrevision	\tex_uptexrevision:D
1175	_kernel_primitive:NN	\uptexversion	\tex_uptexversion:D

Omega primitives provided by pTeX (listed separately mainly to allow understanding of their source).

1176	_kernel_primitive:NN	\odelcode	\tex_odelcode:D
1177	_kernel_primitive:NN	\odelimiter	\tex_odelimiter:D
1178	_kernel_primitive:NN	\omathaccent	\tex_omathaccent:D

```

1179 \__kernel_primitive:NN \omathchar \tex_omathchar:D
1180 \__kernel_primitive:NN \omathchardef \tex_omathchardef:D
1181 \__kernel_primitive:NN \omathcode \tex_omathcode:D
1182 \__kernel_primitive:NN \oradical \tex_oradical:D

```

Newer cross-engine primitives.

```

1183 \__kernel_primitive:NN \partokencontext \tex_partokencontext:D
1184 \__kernel_primitive:NN \partokenname \tex_partokenname:D
1185 \__kernel_primitive:NN \tracingstacklevels \tex_tracingstacklevels:D

```

End of the “just the names” part of the source.

```

1186 </names | package>
1187 </names | tex>
1188 <*package>
1189 <*tex>

```

The job is done: close the group (using the primitive renamed!).

```

1190 \tex_endgroup:D

```

L^AT_EX 2_ε moves a few primitives, so these are sorted out. In newer versions of L^AT_EX 2_ε, expl3 is loaded rather early, so only some primitives are already renamed, so we need two tests here. At the beginning of the L^AT_EX 2_ε format, the primitives `\end` and `\input` are renamed, and only later on the other ones.

```

1191 \tex_ifdefined:D \@@end
1192 \tex_let:D \tex_end:D \@@end
1193 \tex_let:D \tex_input:D \@@input
1194 \tex_fi:D

```

If `\@@@hyph` is defined, we are loading expl3 in a pre-2020/10/01 release of L^AT_EX 2_ε, so a few other primitives have to be tested as well.

```

1195 \tex_ifdefined:D \@@hyph
1196 \tex_let:D \tex_everydisplay:D \frozen@everydisplay
1197 \tex_let:D \tex_everymath:D \frozen@everymath
1198 \tex_let:D \tex_hyphen:D \@@hyph
1199 \tex_let:D \tex_italiccorrection:D \@@italiccorr
1200 \tex_let:D \tex_underline:D \@@underline

```

The `\shipout` primitive is particularly tricky as a number of packages want to hook in here. First, we see if a sufficiently-new kernel has saved a copy: if it has, just use that. Otherwise, we need to check each of the possible packages/classes that might move it: here, we are looking for those which do *not* delay action to the `\AtBeginDocument` hook. (We cannot use `\primitive` as that doesn’t allow us to make a direct copy of the primitive *itself*.) As we know that L^AT_EX 2_ε is in use, we use its `\@tfor` loop here.

```

1201 \tex_ifdefined:D \@@shipout
1202 \tex_let:D \tex_shipout:D \@@shipout
1203 \tex_fi:D
1204 \tex_begingroup:D
1205 \tex_edef:D \l_tmpa_tl { \tex_string:D \shipout }
1206 \tex_edef:D \l_tmpb_tl { \tex_meaning:D \shipout }
1207 \tex_ifx:D \l_tmpa_tl \l_tmpb_tl
1208 \tex_else:D
1209 \tex_expandafter:D \@tfor \tex_expandafter:D \@tempa \tex_string:D :=
1210 \CROP@shipout
1211 \dup@shipout
1212 \GPTorg@shipout
1213 \LL@shipout

```

```

1214     \mem@oldshipout
1215     \opem@shipout
1216     \pgfpages@originalshipout
1217     \pr@shipout
1218     \Shipout
1219     \verso@orig@shipout
1220     \do
1221     {
1222         \tex_edef:D \l_tmpb_tl
1223         { \tex_expandafter:D \tex_meaning:D \@tempa }
1224         \tex_ifx:D \l_tmpa_tl \l_tmpb_tl
1225         \tex_global:D \tex_expandafter:D \tex_let:D
1226         \tex_expandafter:D \tex_shipout:D \@tempa
1227         \tex_fi:D
1228     }
1229     \tex_fi:D
1230     \tex_endgroup:D

```

Some tidying up is needed for `\(pdf)tracingfonts`. Newer LuaTeX has this simply as `\tracingfonts`, but that is overwritten by the L^AT_EX 2_ε kernel. So any spurious definition has to be removed, then the real version saved either from the pdfTeX name or from LuaTeX. In the latter case, we leave `\@@tracingfonts` available: this might be useful and almost all L^AT_EX 2_ε users will have expl3 loaded by fontspec. (We follow the usual kernel convention that `@@` is used for saved primitives.)

```

1231     \tex_let:D \tex_tracingfonts:D \tex_undefined:D
1232     \tex_ifdefined:D \pdftracingfonts
1233     \tex_let:D \tex_tracingfonts:D \pdftracingfonts
1234     \tex_else:D
1235     \tex_ifdefined:D \tex_directlua:D
1236     \tex_directlua:D { tex.enableprimitives("@@", {"tracingfonts"}) }
1237     \tex_let:D \tex_tracingfonts:D \@@tracingfonts
1238     \tex_fi:D
1239     \tex_fi:D
1240     \tex_fi:D

```

That is also true for the LuaTeX primitives under L^AT_EX 2_ε (depending on the format-building date). There are a few primitives that get the right names anyway so are missing here!

```

1241     \tex_ifdefined:D \luatexsuppressfontnotfounderror
1242     \tex_let:D \tex_alignmark:D \luatexalignmark
1243     \tex_let:D \tex_aligntab:D \luatexaligntab
1244     \tex_let:D \tex_attribute:D \luatexattribute
1245     \tex_let:D \tex_attributedef:D \luatexattributedef
1246     \tex_let:D \tex_catcodetable:D \luatexcacodetable
1247     \tex_let:D \tex_clearmarks:D \luatexclearmarks
1248     \tex_let:D \tex_crampeddisplaystyle:D \luatexcrampeddisplaystyle
1249     \tex_let:D \tex_crampedscriptscriptstyle:D
1250     \luatexcrampedscriptscriptstyle
1251     \tex_let:D \tex_crampedscriptstyle:D \luatexcrampedscriptstyle
1252     \tex_let:D \tex_crampedtextstyle:D \luatexcrampedtextstyle
1253     \tex_let:D \tex_fontid:D \luatexfontid
1254     \tex_let:D \tex_formatname:D \luatexformatname
1255     \tex_let:D \tex_gleaders:D \luatexgleaders
1256     \tex_let:D \tex_initcatcodetable:D \luatexinitcatcodetable

```



```

1257 \tex_let:D \tex_latelua:D \luatexlatelua
1258 \tex_let:D \tex_luaescapestring:D \luatexluaescapestring
1259 \tex_let:D \tex_luafunction:D \luatexluafunction
1260 \tex_let:D \tex_mathstyle:D \luatexmathstyle
1261 \tex_let:D \tex_nokerns:D \luatexnokerns
1262 \tex_let:D \tex_noligs:D \luatexnoligs
1263 \tex_let:D \tex_outputbox:D \luatexoutputbox
1264 \tex_let:D \tex_pageleftoffset:D \luatexpageleftoffset
1265 \tex_let:D \tex_pagetopoffset:D \luatexpagetopoffset
1266 \tex_let:D \tex_postexhyphenchar:D \luatexpostexhyphenchar
1267 \tex_let:D \tex_posthyphenchar:D \luatexposthyphenchar
1268 \tex_let:D \tex_preexhyphenchar:D \luatexpreexhyphenchar
1269 \tex_let:D \tex_prehyphenchar:D \luatexprehyphenchar
1270 \tex_let:D \tex_savecatcodetable:D \luatexsavecatcodetable
1271 \tex_let:D \tex_scantextokens:D \luatexscantextokens
1272 \tex_let:D \tex_suppressifcsnameerror:D
1273 \luatexsuppressifcsnameerror
1274 \tex_let:D \tex_suppresslongerror:D \luatexsuppresslongerror
1275 \tex_let:D \tex_suppressmathparerror:D
1276 \luatexsuppressmathparerror
1277 \tex_let:D \tex_suppressoutererror:D \luatexsuppressoutererror
1278 \tex_let:D \tex_Uchar:D \luatexUchar
1279 \tex_let:D \tex_suppressfontnotfounderror:D
1280 \luatexsuppressfontnotfounderror

```

Which also covers those slightly odd ones.

```

1281 \tex_let:D \tex_bodydir:D \luatexbodydir
1282 \tex_let:D \tex_boxdir:D \luatexboxdir
1283 \tex_let:D \tex_leftghost:D \luatexleftghost
1284 \tex_let:D \tex_localbrokenpenalty:D \luatexlocalbrokenpenalty
1285 \tex_let:D \tex_localinterlinepenalty:D
1286 \luatexlocalinterlinepenalty
1287 \tex_let:D \tex_localleftbox:D \luatexlocalleftbox
1288 \tex_let:D \tex_localrightbox:D \luatexlocalrightbox
1289 \tex_let:D \tex_mathdir:D \luatexmathdir
1290 \tex_let:D \tex_pagebottomoffset:D \luatexpagebottomoffset
1291 \tex_let:D \tex_pagedir:D \luatexpagedir
1292 \tex_let:D \tex_pageheight:D \luatexpageheight
1293 \tex_let:D \tex_pagerightoffset:D \luatexpagerightoffset
1294 \tex_let:D \tex_pagewidth:D \luatexpagewidth
1295 \tex_let:D \tex_pardir:D \luatexpardir
1296 \tex_let:D \tex_rightghost:D \luatexrightghost
1297 \tex_let:D \tex_textdir:D \luatextextdir
1298 \tex_fi:D

```

Only pdfTeX and LuaTeX define \pdfmapfile and \pdfmapline: Tidy up the fact that some format-building processes leave a couple of questionable decisions about that!

```

1299 \tex_ifnum:D 0
1300 \tex_ifdefined:D \tex_pdftexversion:D 1 \tex_fi:D
1301 \tex_ifdefined:D \tex_luatexversion:D 1 \tex_fi:D
1302 = 0 %
1303 \tex_let:D \tex_mapfile:D \tex_undefined:D
1304 \tex_let:D \tex_mapline:D \tex_undefined:D
1305 \tex_fi:D

```

A few packages do unfortunate things to date-related primitives.

```

1306 \tex_begingroup:D
1307   \tex_edef:D \l_tmpa_tl { \tex_meaning:D \tex_time:D }
1308   \tex_edef:D \l_tmpb_tl { \tex_string:D \time }
1309   \tex_ifx:D \l_tmpa_tl \l_tmpb_tl
1310   \tex_else:D
1311     \tex_global:D \tex_let:D \tex_time:D \tex_undefined:D
1312   \tex_fi:D
1313   \tex_edef:D \l_tmpa_tl { \tex_meaning:D \tex_day:D }
1314   \tex_edef:D \l_tmpb_tl { \tex_string:D \day }
1315   \tex_ifx:D \l_tmpa_tl \l_tmpb_tl
1316   \tex_else:D
1317     \tex_global:D \tex_let:D \tex_day:D \tex_undefined:D
1318   \tex_fi:D
1319   \tex_edef:D \l_tmpa_tl { \tex_meaning:D \tex_month:D }
1320   \tex_edef:D \l_tmpb_tl { \tex_string:D \month }
1321   \tex_ifx:D \l_tmpa_tl \l_tmpb_tl
1322   \tex_else:D
1323     \tex_global:D \tex_let:D \tex_month:D \tex_undefined:D
1324   \tex_fi:D
1325   \tex_edef:D \l_tmpa_tl { \tex_meaning:D \tex_year:D }
1326   \tex_edef:D \l_tmpb_tl { \tex_string:D \year }
1327   \tex_ifx:D \l_tmpa_tl \l_tmpb_tl
1328   \tex_else:D
1329     \tex_global:D \tex_let:D \tex_year:D \tex_undefined:D
1330   \tex_fi:D
1331 \tex_endgroup:D

```

Up to v0.80, LuaTeX defines the pdfTeX version data: rather confusing. Removing them means that `\tex_pdftexversion:D` is a marker for pdfTeX alone: useful in engine-dependent code later.

```

1332 \tex_ifdefined:D \tex luatexversion:D
1333   \tex_let:D \tex_pdftexbanner:D \tex_undefined:D
1334   \tex_let:D \tex_pdftexrevision:D \tex_undefined:D
1335   \tex_let:D \tex_pdftexversion:D \tex_undefined:D
1336 \tex_fi:D

```

`cslatex` moves a couple of primitives which we recover here; as there is no other marker, we can only work by looking for the names.

```

1337 \tex_ifdefined:D \orieveryjob
1338   \tex_let:D \tex_everyjob:D \orieveryjob
1339 \tex_fi:D
1340 \tex_ifdefined:D \oripdfoutput
1341   \tex_let:D \tex_pdfoutput:D \oripdfoutput
1342 \tex_fi:D

```

For ConTeXt, two tests are needed. Both Mark II and Mark IV move several primitives: these are all covered by the first test, again using `\end` as a marker. For Mark IV, a few more primitives are moved: they are implemented using some Lua code in the current ConTeXt.

```

1343 \tex_ifdefined:D \normalend
1344   \tex_let:D \tex_end:D \normalend
1345   \tex_let:D \tex_everyjob:D \normaleveryjob
1346   \tex_let:D \tex_input:D \normalinput

```

```

1347 \tex_let:D \tex_language:D \normallanguage
1348 \tex_let:D \tex_mathop:D \normalmathop
1349 \tex_let:D \tex_month:D \normalmonth
1350 \tex_let:D \tex_outer:D \normalouter
1351 \tex_let:D \tex_over:D \normalover
1352 \tex_let:D \tex_vcenter:D \normalvcenter
1353 \tex_let:D \tex_unexpanded:D \normalunexpanded
1354 \tex_let:D \tex_expanded:D \normalexpanded
1355 \tex_fi:D
1356 \tex_ifdefined:D \normalitaliccorrection
1357 \tex_let:D \tex_hoffset:D \normalhoffset
1358 \tex_let:D \tex_italiccorrection:D \normalitaliccorrection
1359 \tex_let:D \tex_voffset:D \normalvoffset
1360 \tex_let:D \tex_showtokens:D \normalshowtokens
1361 \tex_let:D \tex_bodydir:D \spac_directions_normal_body_dir
1362 \tex_let:D \tex_pagedir:D \spac_directions_normal_page_dir
1363 \tex_fi:D
1364 \tex_ifdefined:D \normalleft
1365 \tex_let:D \tex_left:D \normalleft
1366 \tex_let:D \tex_middle:D \normalmiddle
1367 \tex_let:D \tex_right:D \normalright
1368 \tex_fi:D
1369 \</tex>

```

In LuaTeX, we additionally emulate some primitives using Lua code.

```

1370 \*lua)

```

`\tex_strcmp:D` Compare two strings, expanding to 0 if they are equal, -1 if the first one is smaller and 1 if the second one is smaller. Here “smaller” refers to codepoint order which does not correspond to the user expected order for most non-ASCII strings.

```

1371 local minus_tok = token.new(string.byte'-', 12)
1372 local zero_tok = token.new(string.byte'0', 12)
1373 local one_tok = token.new(string.byte'1', 12)
1374 luacmd('tex_strcmp:D', function()
1375     local first = scan_string()
1376     local second = scan_string()
1377     if first < second then
1378         put_next(minus_tok, one_tok)
1379     else
1380         put_next(first == second and zero_tok or one_tok)
1381     end
1382 end, 'global')

```

(End definition for \tex_strcmp:D. This function is documented on page ??.)

`\tex_Ucharcat:D` Creating arbitrary chars using `tex.cprint`. The alternative approach using `token.put_next(token.cre` would be about 10% slower.

```

1383 local cprint = tex.cprint
1384 luacmd('tex_Ucharcat:D', function()
1385     local charcode = scan_int()
1386     local catcode = scan_int()
1387     cprint(catcode, utf8_char(charcode))
1388 end, 'global')

```

(End definition for `\tex_Ucharcat:D`. This function is documented on page ??.)

`\tex_filesize:D` Wrap the function from `ltxutils`.

```
1389 luacmd('tex_filesize:D', function()
1390   local size = filesize(scan_string())
1391   if size then write(size) end
1392 end, 'global')
```

(End definition for `\tex_filesize:D`. This function is documented on page ??.)

`\tex_mdffivesum:D` There are two cases: Either hash a file or a string. Both are already implemented in `l3luatex` or built-in.

```
1393 luacmd('tex_mdffivesum:D', function()
1394   local hash
1395   if scan_keyword"file" then
1396     hash = filemd5sum(scan_string())
1397   else
1398     hash = md5_HEX(scan_string())
1399   end
1400   if hash then write(hash) end
1401 end, 'global')
```

(End definition for `\tex_mdffivesum:D`. This function is documented on page ??.)

`\tex_filemoddate:D` A primitive for getting the modification date of a file.

```
1402 luacmd('tex_filemoddate:D', function()
1403   local date = filemoddate(scan_string())
1404   if date then write(date) end
1405 end, 'global')
```

(End definition for `\tex_filemoddate:D`. This function is documented on page ??.)

`\tex_filedump:D` An emulated primitive for getting a hexdump from a (partial) file. The length has a default of 0. This is consistent with `pdfTeX`, but it effectively makes the primitive useless without an explicit `length`. Therefore we allow the keyword `whole` to be used instead of a length, indicating that the whole remaining file should be read.

```
1406 luacmd('tex_filedump:D', function()
1407   local offset = scan_keyword'offset' and scan_int() or nil
1408   local length = scan_keyword'length' and scan_int()
1409                 or not scan_keyword'whole' and 0 or nil
1410   local data = filedump(scan_string(), offset, length)
1411   if data then write(data) end
1412 end, 'global')
```

(End definition for `\tex_filedump:D`. This function is documented on page ??.)

```
1413 </lua>
1414 </package>
```

Chapter 40

13kernel-functions: kernel-reserved functions

40.1 Internal kernel functions

`_kernel_chk_cs_exist:N`
`_kernel_chk_cs_exist:c`

`_kernel_chk_cs_exist:N` $\langle cs \rangle$

This function is only created if debugging is enabled. It checks that $\langle cs \rangle$ exists according to the criteria for `\cs_if_exist_p:N`, and if not raises a kernel-level error.

`_kernel_chk_defined:NT`

`_kernel_chk_defined:NT` $\langle variable \rangle$ $\{\langle true\ code \rangle\}$

If $\langle variable \rangle$ is not defined (according to `\cs_if_exist:NTF`), this triggers an error, otherwise the $\langle true\ code \rangle$ is run.

`_kernel_chk_expr:nNnN`

`_kernel_chk_expr:nNnN` $\{\langle expr \rangle\}$ $\langle eval \rangle$ $\{\langle convert \rangle\}$ $\langle caller \rangle$

This function is only created if debugging is enabled. By default it is equivalent to `\use_i:nnnn`. When expression checking is enabled, it leaves in the input stream the result of `\tex_the:D` $\langle eval \rangle$ $\langle expr \rangle$ `\tex_relax:D` after checking that no token was left over. If any token was not taken as part of the expression, there is an error message displaying the result of the evaluation as well as the $\langle caller \rangle$. For instance $\langle eval \rangle$ can be `_int_eval:w` and $\langle caller \rangle$ can be `\int_eval:n` or `\int_set:Nn`. The argument $\langle convert \rangle$ is empty except for mu expressions where it is `\tex_mutoglue:D`, used for internal purposes.

`_kernel_chk_tl_type:NnnT`

`_kernel_chk_tl_type:NnnT` $\langle control\ sequence \rangle$ $\{\langle specific\ type \rangle\}$
 $\{\langle reconstruction \rangle\}$ $\{\langle true\ code \rangle\}$

Helper to test that the $\langle control\ sequence \rangle$ is a variable of the given $\langle specific\ type \rangle$ of token list. Produces suitable error messages if the $\langle control\ sequence \rangle$ does not exist, or if it is not a token list variable at all, or if the $\langle control\ sequence \rangle$ differs from the result of x-expanding $\langle reconstruction \rangle$. If all of these tests succeed then the $\langle true\ code \rangle$ is run.

<code>_kernel_cs_parm_from_arg_count:nnF</code>	<code>_kernel_cs_parm_from_arg_count:nnF {<follow-on>} {<args>}</code>
	<code>{<false code>}</code>

Evaluates the number of *<args>* and leaves the *<follow-on>* code followed by a brace group containing the required number of primitive parameter markers (*#1*, *etc.*). If the number of *<args>* is outside the range $[0, 9]$, the *<false code>* is inserted *instead* of the *<follow-on>*.

<code>_kernel_dependency_version_check:Nn</code>	<code>_kernel_dependency_version_check:Nn {<\date>} {<file>}</code>
<code>_kernel_dependency_version_check:nn</code>	<code>_kernel_dependency_version_check:nn {<date>} {<file>}</code>

Checks if the loaded version of the expl3 kernel is at least *<date>*, required by *<file>*. If the kernel date is older than *<date>*, the loading of *<file>* is aborted and an error is raised.

<code>_kernel_deprecation_code:nn</code>	<code>_kernel_deprecation_code:nn {<error code>} {<working code>}</code>
---	---

Stores both an *<error>* and *<working>* definition for given material such that they can be exchanged by `\debug_on:` and `\debug_off:`.

<code>_kernel_exp_not:w *</code>	<code>_kernel_exp_not:w <expandable tokens> {<content>}</code>
-----------------------------------	---

Carries out expansion on the *<expandable tokens>* before preventing further expansion of the *<content>* as for `\exp_not:n`. Typically, the *<expandable tokens>* will alter the nature of the *<content>*, *i.e.* allow it to be generated in some way.

<code>\l__kernel_expl_bool</code>	A boolean which records the current code syntax status: true if currently inside a code environment. This variable should only be set by <code>\ExplSyntaxOn/\ExplSyntaxOff</code> .
-----------------------------------	---

(End definition for `\l__kernel_expl_bool`.)

<code>\c__kernel_expl_date_tl</code>	A token list containing the release date of the l3kernel preloaded in L ^A T _E X 2 _ε used to check if dependencies match.
--------------------------------------	---

(End definition for `\c__kernel_expl_date_tl`.)

<code>_kernel_file_missing:n</code>	<code>_kernel_file_missing:n {<name>}</code>
--------------------------------------	---

Expands the *<name>* as per `_kernel_file_name_sanitiz:n` then produces an error message indicating that this file was not found.

<code>_kernel_file_name_sanitiz:n *</code>	<code>_kernel_file_name_sanitiz:n {<name>}</code>
---	--

Updated: 2021-04-17

Expands the file name using a `\csname`-based approach, and relies on active characters (for example from UTF-8 characters) being properly set up to expand to an expansion-safe version using `\ifcsname`. This is less conservative than the token-by-token approach used before, but it is much faster.

<code>_kernel_file_input_push:n</code>	<code>_kernel_file_input_push:n {<name>}</code>
<code>_kernel_file_input_pop:</code>	<code>_kernel_file_input_pop:</code>

Used to push and pop data from the internal file stack: needed only in package mode, where interfacing with the L^AT_EX 2_ε kernel is necessary.

`_kernel_int_add:nnn` \star `_kernel_int_add:nnn {⟨integer1⟩} {⟨integer2⟩} {⟨integer3⟩}`

Expands to the result of adding the three $\langle integers \rangle$ (which must be suitable input for $\backslash\text{int_eval:w}$), avoiding intermediate overflow. Overflow occurs only if the overall result is outside $[-2^{31}+1, 2^{31}-1]$. The $\langle integers \rangle$ may be of the form $\backslash\text{int_eval:w} \dots \backslash\text{scan_stop}$: but may be evaluated more than once.

`_kernel_intarray_gset:Nnn` `_kernel_intarray_gset:Nnn ⟨intarray var⟩ {⟨index⟩} {⟨value⟩}`

New: 2018-03-31

Faster version of $\backslash\text{intarray_gset:Nnn}$. Stores the $\langle value \rangle$ into the $\langle integer\ array\ variable \rangle$ at the $\langle position \rangle$. The $\langle index \rangle$ and $\langle value \rangle$ must be suitable for a direct assignment to a T_EX count register, for instance expanding to an integer denotation or obtained through the primitive $\backslash\text{numexpr}$ (which may be un-terminated). No bound checking is performed: the caller is responsible for ensuring that the $\langle position \rangle$ is between 1 and the $\backslash\text{intarray_count:N}$, and the $\langle value \rangle$'s absolute value is at most $2^{30}-1$. Assignments are always global.

`_kernel_intarray_item:Nn` \star `_kernel_intarray_item:Nn ⟨intarray var⟩ {⟨index⟩}`

New: 2018-03-31

Faster version of $\backslash\text{intarray_item:Nn}$. Expands to the integer entry stored at the $\langle index \rangle$ in the $\langle integer\ array\ variable \rangle$. The $\langle index \rangle$ must be suitable for a direct assignment to a T_EX count register and must be between 1 and the $\backslash\text{intarray_count:N}$, lest a low-level T_EX error occur.

`_kernel_intarray_range_to_clist:Nnn` \star `_kernel_intarray_range_to_clist:Nnn ⟨intarray var⟩ {⟨start index⟩} {⟨end index⟩}`

New: 2020-07-12

Converts to integer denotations separated by commas the entries of the $\langle intarray \rangle$ from positions $\langle start\ index \rangle$ to $\langle end\ index \rangle$ included. The $\langle start\ index \rangle$ and $\langle end\ index \rangle$ must be suitable for a direct assignment to a T_EX count register, must be between 1 and the $\backslash\text{intarray_count:N}$, and be suitably ordered. All tokens have category code other.

`_kernel_intarray_gset_range_from_clist:Nnn` `_kernel_intarray_gset_range_from_clist:Nnn
⟨intarray var⟩ {⟨start index⟩} {⟨integer clist⟩}`

New: 2020-07-12

Stores the entries of the $\langle clist \rangle$ as entries of the $\langle intarray\ var \rangle$ starting from the $\langle start\ index \rangle$, upwards. This is done without any bound checking. The $\langle start\ index \rangle$ and all entries of the $\langle integer\ comma\ list \rangle$ (which do not undergo space trimming and brace stripping as in normal clist mappings) must be suitable for a direct assignment to a T_EX count register. An empty entry may stop the loop.

`_kernel_ior_open:Nn` `_kernel_ior_open:Nn ⟨stream⟩ {⟨file name⟩}`
`_kernel_ior_open:No`

This function has identical syntax to the public version. However, it does not take precautions against active characters in the $\langle file\ name \rangle$, and it does not attempt to add a $\langle path \rangle$ to the $\langle file\ name \rangle$: it is therefore intended to be used by higher-level functions which have already fully expanded the $\langle file\ name \rangle$ and which need to perform multiple open or close operations. See for example the implementation of $\backslash\text{file_get_full_name:nN}$,

<u>_kernel_iow_with:Nnn</u>	<p><code>_kernel_iow_with:Nnn <integer> {<value>} {<code>}</code></p> <p>If the <i><integer></i> is equal to the <i><value></i> then this function simply runs the <i><code></i>. Otherwise it saves the current value of the <i><integer></i>, sets it to the <i><value></i>, runs the <i><code></i>, and restores the <i><integer></i> to its former value. This is used to ensure that the <code>\newlinechar</code> is 10 when writing to a stream, which lets <code>\iow_newline:</code> work, and that <code>\errorcontextlines</code> is <code>-1</code> when displaying a message.</p>
<code>_kernel_kern:n</code>	<p><code>_kernel_kern:n {<length>}</code></p> <p>Inserts a kern of the specified <i><length></i>, a dimension expression.</p> <p>(End definition for <code>_kernel_kern:n</code>.)</p>
<code>\g__kernel_prg_map_int</code>	<p>This integer is used by non-expandable mapping functions to track the level of nesting in force. The functions <code>\<type>_map_1:w</code>, <code>\<type>_map_2:w</code>, <i>etc.</i>, labelled by <code>\g__kernel_prg_map_int</code> hold functions to be mapped over various list datatypes in inline and variable mappings.</p> <p>(End definition for <code>\g__kernel_prg_map_int</code>.)</p>

`__kernel_quark_new_test:N`

`__kernel_quark_new_test:N \<name>:<arg spec>`

Defines a quark-test function `\<name>:<arg spec>` which tests if its argument is `\q__<namespace>_recursion_tail`, then acts accordingly, as described below for each possible `<arg spec>`.

The `<namespace>` is determined as the first (nonempty) `_`-delimited word in `<name>` and is used internally in the definition of auxiliaries. The function `__kernel_quark_new_test:N` does *not* define the `\q__<namespace>_recursion_tail` and `\q__<namespace>_recursion_stop` quarks. They should be manually defined with `\quark_new:N`.

There are 6 different types of quark-test functions. Which one is defined depends on the `<arg spec>`, which *must* be one of the options listed now. Four of them are modeled after `\quark_if_recursion_tail:(N|n)` and `\quark_if_recursion_tail_do:(N|n)n`.

`n` defines `\<name>:n` such that it checks if #1 contains only `\q__<namespace>_recursion_tail`, and if so consumes all tokens up to `\q__<namespace>_recursion_stop` (c.f. `\quark_if_recursion_tail_stop:n`).

`nn` defines `\<name>:nn` such that it checks if #1 contains only `\q__<namespace>_recursion_tail`, and if so consumes all tokens up to `\q__<namespace>_recursion_stop`, then executes the code #2 after that (c.f. `\quark_if_recursion_tail_stop_do:nn`).

`N` defines `\<name>:N` such that it checks if #1 is `\q__<namespace>_recursion_tail`, and if so consumes all tokens up to `\q__<namespace>_recursion_stop` (c.f. `\quark_if_recursion_tail_stop:N`).

`Nn` defines `\<name>:Nn` such that it checks if #1 is `\q__<namespace>_recursion_tail`, and if so consumes all tokens up to `\q__<namespace>_recursion_stop`, then executes the code #2 after that (c.f. `\quark_if_recursion_tail_stop_do:Nn`).

The last two are modeled after `\quark_if_recursion_tail_break:(n|N)N`, and in those cases the quark `\q__<namespace>_recursion_stop` is not used (and thus needs not be defined).

`nN` defines `\<name>:nN` such that it checks if #1 contains only `\q__<namespace>_recursion_tail`, and if so uses the `\<type>_map_break:` function #2.

`NN` defines `\<name>:NN` such that it checks if #1 is `\q__<namespace>_recursion_tail`, and if so uses the `\<type>_map_break:` function #2.

Any other signature, as well as a function without signature are errors, and in such case the definition is aborted.

<code>_kernel_quark_new_conditional:Nn</code>	<code>_kernel_quark_new_conditional:Nn</code>
	<code>_<namespace>_quark_if_<name>:<arg spec> {\<conditions>}</code>

Defines a collection of quark conditionals that test if their argument is the quark `\q_<namespace>_<name>` and perform suitable actions. The $\langle conditions \rangle$ are a comma-separated list of one or more of p, T, F, and TF, and one conditional is defined for each $\langle condition \rangle$ in the list, as described for `\prg_new_conditional:Npnn`. The conditionals are defined using `\prg_new_conditional:Npnn`, so that their name is obtained by adding p, T, F, or TF to the base name `_<namespace>_quark_if_<name>:<arg spec>`.

The first argument of `_kernel_quark_new_conditional:Nn` must contain `_quark_if_<name>:`, as these markers are used to determine the $\langle name \rangle$ of the quark `\q_<namespace>_<name>` to be tested. This quark should be manually defined with `\quark_new:N`, as `_kernel_quark_new_conditional:Nn` does *not* define it.

The function `_kernel_quark_new_conditional:Nn` can define 2 different types of quark conditionals. Which one is defined depends on the $\langle arg spec \rangle$, which *must* be one of the following options, modeled after `\quark_if_nil:(N|n)(TF)`.

`n` defines `_<namespace>_quark_if_<name>:n(TF)` such that it checks if #1 contains only `\q_<namespace>_<name>`, and executes the proper conditional branch.

`N` defines `_<namespace>_quark_if_<name>:N(TF)` such that it checks if #1 is `\q_<namespace>_<name>`, and executes the proper conditional branch.

Any other signature, as well as a function without signature are errors, and in such case the definition is aborted.

`\c_kernel_randint_max_int` Maximal allowed argument to `_kernel_randint:n`. Equal to $2^{17} - 1$.

(End definition for `\c_kernel_randint_max_int`.)

<code>_kernel_randint:n</code>	<code>_kernel_randint:n {\<max>}</code>
----------------------------------	---

Used in an integer expression this gives a pseudo-random number between 1 and $\langle max \rangle$ included. One must have $\langle max \rangle \leq 2^{17} - 1$. The $\langle max \rangle$ must be suitable for `\int_value:w` (and any `\int_eval:w` must be terminated by `\scan_stop:` or equivalent).

<code>_kernel_randint:nn</code>	<code>_kernel_randint:nn {\<min>} {\<max>}</code>
-----------------------------------	---

Used in an integer expression this gives a pseudo-random number between $\langle min \rangle$ and $\langle max \rangle$ included. The $\langle min \rangle$ and $\langle max \rangle$ must be suitable for `\int_value:w` (and any `\int_eval:w` must be terminated by `\scan_stop:` or equivalent). For small ranges $R = \langle max \rangle - \langle min \rangle + 1 \leq 2^{17} - 1$, $\langle min \rangle - 1 + _kernel_randint:n\{R\}$ is faster.

<code>_kernel_register_show:N</code>	<code>_kernel_register_show:N <register></code>
<code>_kernel_register_show:c</code>	

Used to show the contents of a T_EX register at the terminal, formatted such that internal parts of the mechanism are not visible.

<code>_kernel_register_log:N</code>	<code>_kernel_register_log:N <register></code>
<code>_kernel_register_log:c</code>	

Used to write the contents of a T_EX register to the log file in a form similar to `_kernel_register_show:N`.

<code>_kernel_str_to_other:n</code> ★	<code>_kernel_str_to_other:n {⟨token list⟩}</code>
--	---

Converts the $\langle token list \rangle$ to a $\langle other string \rangle$, where spaces have category code “other”. This function can be f-expanded without fear of losing a leading space, since spaces do not have category code 10 in its result. It takes a time quadratic in the character count of the string.

<code>_kernel_str_to_other_fast:n</code> ☆	<code>_kernel_str_to_other_fast:n {⟨token list⟩}</code>
---	--

Same behaviour `_kernel_str_to_other:n` but only restricted-expandable. It takes a time linear in the character count of the string.

<code>_kernel_tl_to_str:w</code> ★	<code>_kernel_tl_to_str:w ⟨expandable tokens⟩ {⟨tokens⟩}</code>
-------------------------------------	--

Carries out expansion on the $\langle expandable tokens \rangle$ before conversion of the $\langle tokens \rangle$ to a string as describe for `\tl_to_str:n`. Typically, the $\langle expandable tokens \rangle$ will alter the nature of the $\langle tokens \rangle$, *i.e.* allow it to be generated in some way. This function requires only a single expansion.

<code>_kernel_tl_set:Nx</code> <code>_kernel_tl_gset:Nx</code>	<code>_kernel_tl_set:Nx ⟨tl var⟩ {⟨tokens⟩}</code>
---	---

Fully expands $\langle tokens \rangle$ and assigns the result to $\langle tl var \rangle$. $\langle tokens \rangle$ must be given in braces and there must be no token between $\langle tl var \rangle$ and $\langle tokens \rangle$.

40.2 Kernel backend functions

These functions are required to pass information to the backend. The nature of these means that they are defined only when the relevant backend is in use.

<code>_kernel_backend_literal:n</code> <code>_kernel_backend_literal:(e x)</code>	<code>_kernel_backend_literal:n {⟨content⟩}</code>
--	---

Adds the $\langle content \rangle$ literally to the current vertical list as a whatsit. The nature of the $\langle content \rangle$ will depend on the backend in use.

<code>_kernel_backend_literal_postscript:n</code> <code>_kernel_backend_literal_postscript:x</code>	<code>_kernel_backend_literal_postscript:n {⟨PostScript⟩}</code>
--	---

Adds the $\langle PostScript \rangle$ literally to the current vertical list as a whatsit. No positioning is applied.

<code>_kernel_backend_literal_pdf:n</code> <code>_kernel_backend_literal_pdf:x</code>	<code>_kernel_backend_literal_pdf:n {⟨PDF instructions⟩}</code>
--	--

Adds the $\langle PDF instructions \rangle$ literally to the current vertical list as a whatsit. No positioning is applied.

<code>_kernel_backend_literal_svg:n</code> <code>_kernel_backend_literal_svg:x</code>	<code>_kernel_backend_literal_svg:n {⟨SVG instructions⟩}</code>
--	--

Adds the $\langle SVG instructions \rangle$ literally to the current vertical list as a whatsit. No positioning is applied.

<code>_kernel_backend_postscript:n</code>	<code>_kernel_backend_postscript:n {<i>PostScript</i>}</code>
<code>_kernel_backend_postscript:x</code>	

Adds the *PostScript* to the current vertical list as a whatsit. The PostScript reference point is adjusted to match the current position. The PostScript is inserted inside a SDict begin/end pair.

<code>_kernel_backend_align_begin:</code>	<code>_kernel_backend_align_begin:</code>
<code>_kernel_backend_align_end:</code>	<code><i>PostScript literals</i></code>
	<code>_kernel_backend_align_end:</code>

Arranges to align the PostScript and DVI current positions and scales.

<code>_kernel_backend_scope_begin:</code>	<code>_kernel_backend_scope_begin:</code>
<code>_kernel_backend_scope_end:</code>	<code><i>content</i></code>
	<code>_kernel_backend_scope_end:</code>

Creates a scope for instructions at the backend level.

<code>_kernel_backend_matrix:n</code>	<code>_kernel_backend_matrix:n {<i>matrix</i>}</code>
<code>_kernel_backend_matrix:x</code>	Applies the <i>matrix</i> to the current transformation matrix.

<code>\g_kernel_backend_header_bool</code>
--

Specifies whether to write headers for the backend.

<code>\l_kernel_color_stack_int</code>	The color stack used in pdfTeX and LuaTeX for the main color.
--	---

Chapter 41

l3basics implementation

```
1415 <*package>
```

41.1 Renaming some T_EX primitives (again)

Having given all the T_EX primitives a consistent name, we need to give sensible names to the ones we actually want to use. These will be defined as needed in the appropriate modules, but we do a few now, just to get started.⁸

```
\if_true: Then some conditionals.
\if_false: 1416 \tex_let:D \if_true:      \tex_iftrue:D
\or:       1417 \tex_let:D \if_false:    \tex_iffalse:D
\else:     1418 \tex_let:D \or:          \tex_or:D
\fi:       1419 \tex_let:D \else:        \tex_else:D
\reverse_if:N 1420 \tex_let:D \fi:          \tex_fi:D
\if:w      1421 \tex_let:D \reverse_if:N    \tex_unless:D
\if_charcode:w 1422 \tex_let:D \if:w          \tex_if:D
\if_catcode:w 1423 \tex_let:D \if_charcode:w    \tex_if:D
\if_meaning:w 1424 \tex_let:D \if_catcode:w    \tex_ifcat:D
            1425 \tex_let:D \if_meaning:w    \tex_ifx:D
            1426 \tex_let:D \if_bool:N      \tex_ifodd:D
```

(End definition for \if_true: and others. These functions are documented on page 27.)

```
\if_mode_math: TEX lets us detect some if its modes.
\if_mode_horizontal: 1427 \tex_let:D \if_mode_math:    \tex_ifmmode:D
\if_mode_vertical:   1428 \tex_let:D \if_mode_horizontal: \tex_ifhmode:D
\if_mode_inner:      1429 \tex_let:D \if_mode_vertical:    \tex_ifvmode:D
                    1430 \tex_let:D \if_mode_inner:      \tex_ifinner:D
```

(End definition for \if_mode_math: and others. These functions are documented on page 28.)

```
\if_cs_exist:N Building csnames and testing if control sequences exist.
\if_cs_exist:w 1431 \tex_let:D \if_cs_exist:N    \tex_ifdefined:D
\cs:w          1432 \tex_let:D \if_cs_exist:w    \tex_ifcurname:D
\cs_end:       1433 \tex_let:D \cs:w      \tex_csname:D
            1434 \tex_let:D \cs_end:      \tex_endcurname:D
```

⁸This renaming gets expensive in terms of csname usage, an alternative scheme would be to just use the `\tex_...:D` name in the cases where no good alternative exists.

(End definition for `\if_cs_exist:N` and others. These functions are documented on page 28.)

`\exp_after:wN` The five `\exp_` functions are used in the `l3expan` module where they are described.

```

\exp_not:N      1435 \tex_let:D \exp_after:wN      \tex_expandafter:D
\exp_not:n      1436 \tex_let:D \exp_not:N      \tex_noexpand:D
                1437 \tex_let:D \exp_not:n      \tex_unexpanded:D
                1438 \tex_let:D \exp:w      \tex_romannumeral:D
                1439 \tex_chardef:D \exp_end: = 0 ~

```

(End definition for `\exp_after:wN`, `\exp_not:N`, and `\exp_not:n`. These functions are documented on page 38.)

`\token_to_meaning:N` Examining a control sequence or token.

```

\cs_meaning:N      1440 \tex_let:D \token_to_meaning:N \tex_meaning:D
                  1441 \tex_let:D \cs_meaning:N      \tex_meaning:D

```

(End definition for `\token_to_meaning:N` and `\cs_meaning:N`. These functions are documented on page 187.)

`\tl_to_str:n` Making strings.

```

\token_to_str:N      1442 \tex_let:D \tl_to_str:n      \tex_detokenize:D
\__kernel_tl_to_str:w 1443 \tex_let:D \token_to_str:N      \tex_string:D
                  1444 \tex_let:D \__kernel_tl_to_str:w \tex_detokenize:D

```

(End definition for `\tl_to_str:n`, `\token_to_str:N`, and `__kernel_tl_to_str:w`. These functions are documented on page 107.)

`\scan_stop:` The next three are basic functions for which there also exist versions that are safe inside alignments. These safe versions are defined in the `l3prg` module.

```

\group_begin:      1445 \tex_let:D \scan_stop:      \tex_relax:D
\group_end:        1446 \tex_let:D \group_begin:      \tex_begingroup:D
                  1447 \tex_let:D \group_end:      \tex_endgroup:D

```

(End definition for `\scan_stop:`, `\group_begin:`, and `\group_end:`. These functions are documented on page 13.)

```
1448 <@@=int>
```

`\if_int_compare:w` For integers.

```

\__int_to_roman:w  1449 \tex_let:D \if_int_compare:w \tex_ifnum:D
                  1450 \tex_let:D \__int_to_roman:w \tex_romannumeral:D

```

(End definition for `\if_int_compare:w` and `__int_to_roman:w`. This function is documented on page 167.)

`\group_insert_after:N` Adding material after the end of a group.

```
1451 \tex_let:D \group_insert_after:N \tex_aftergroup:D
```

(End definition for `\group_insert_after:N`. This function is documented on page 14.)

`\exp_args:Nc` Discussed in `l3expan`, but needed much earlier.

```

\exp_args:cc      1452 \tex_long:D \tex_def:D \exp_args:Nc #1#2
                  1453 { \exp_after:wN #1 \cs:w #2 \cs_end: }
                  1454 \tex_long:D \tex_def:D \exp_args:cc #1#2
                  1455 { \cs:w #1 \exp_after:wN \cs_end: \cs:w #2 \cs_end: }

```

(End definition for `\exp_args:Nc` and `\exp_args:cc`. These functions are documented on page 34.)

`\token_to_meaning:c` A small number of variants defined by hand. Some of the necessary functions (`\use_i:nn`, `\use_ii:nn`, and `\exp_args:Nnc`) are not defined at that point yet, but will be defined before those variants are used. The `\cs_meaning:c` command must check for an undefined control sequence to avoid defining it mistakenly.

```

1456 \tex_def:D \token_to_str:c { \exp_args:Nc \token_to_str:N }
1457 \tex_long:D \tex_def:D \cs_meaning:c #1
1458 {
1459   \if_cs_exist:w #1 \cs_end:
1460     \exp_after:wN \use_i:nn
1461   \else:
1462     \exp_after:wN \use_ii:nn
1463   \fi:
1464   { \exp_args:Nc \cs_meaning:N {#1} }
1465   { \tl_to_str:n {undefined} }
1466 }
1467 \tex_let:D \token_to_meaning:c = \cs_meaning:c

```

(End definition for `\token_to_meaning:N`. This function is documented on page 187.)

41.2 Defining some constants

`\c_zero_int` We need the constant `\c_zero_int` which is used by some functions in the `l3alloc` module. The rest are defined in the `l3int` module – at least for the ones that can be defined with `\tex_chardef:D` or `\tex_mathchardef:D`. For other constants the `l3int` module is required but it can’t be used until the allocation has been set up properly!

```

1468 \tex_chardef:D \c_zero_int = 0 ~

```

(End definition for `\c_zero_int`. This variable is documented on page 166.)

`\c_max_register_int` This is here as this particular integer is needed both in package mode and to bootstrap `l3alloc`, and is documented in `l3int`. LuaTeX and those which contain parts of the Omega extensions have more registers available than ϵ -TeX.

```

1469 \tex_ifdefined:D \tex_luatexversion:D
1470   \tex_chardef:D \c_max_register_int = 65 535 ~
1471 \tex_else:D
1472   \tex_ifdefined:D \tex_omathchardef:D
1473     \tex_omathchardef:D \c_max_register_int = 65535 ~
1474   \tex_else:D
1475     \tex_mathchardef:D \c_max_register_int = 32767 ~
1476   \tex_fi:D
1477 \tex_fi:D

```

(End definition for `\c_max_register_int`. This variable is documented on page 166.)

41.3 Defining functions

We start by providing functions for the typical definition functions. First the local ones.

`\cs_set_nopar:Npn` All assignment functions in L^AT_EX₃ should be naturally protected; after all, the T_EX primitives for assignments are and it can be a cause of problems if others aren’t.

```

\cs_set_nopar:Npx
\cs_set:Npn
\cs_set:Npx
1478 \tex_let:D \cs_set_nopar:Npn \tex_def:D
1479 \tex_let:D \cs_set_nopar:Npx \tex_edef:D

```

```

\cs_set_protected_nopar:Npn
\cs_set_protected_nopar:Npx
\cs_set_protected:Npn
\cs_set_protected:Npx

```

```

1480 \tex_protected:D \tex_long:D \tex_def:D \cs_set:Npn
1481   { \tex_long:D \tex_def:D }
1482 \tex_protected:D \tex_long:D \tex_def:D \cs_set:Npx
1483   { \tex_long:D \tex_edef:D }
1484 \tex_protected:D \tex_long:D \tex_def:D \cs_set_protected_nopar:Npn
1485   { \tex_protected:D \tex_def:D }
1486 \tex_protected:D \tex_long:D \tex_def:D \cs_set_protected_nopar:Npx
1487   { \tex_protected:D \tex_edef:D }
1488 \tex_protected:D \tex_long:D \tex_def:D \cs_set_protected:Npn
1489   { \tex_protected:D \tex_long:D \tex_def:D }
1490 \tex_protected:D \tex_long:D \tex_def:D \cs_set_protected:Npx
1491   { \tex_protected:D \tex_long:D \tex_edef:D }

```

(End definition for `\cs_set_nopar:Npn` and others. These functions are documented on page 16.)

`\cs_gset_nopar:Npn` Global versions of the above functions.

```

\cs_gset_nopar:Npx 1492 \tex_let:D \cs_gset_nopar:Npn          \tex_gdef:D
\cs_gset:Npn        1493 \tex_let:D \cs_gset_nopar:Npx          \tex_xdef:D
\cs_gset:Npx        1494 \cs_set_protected:Npn \cs_gset:Npn
\cs_gset_protected_nopar:Npn 1495   { \tex_long:D \tex_gdef:D }
\cs_gset_protected_nopar:Npx 1496 \cs_set_protected:Npn \cs_gset:Npx
\cs_gset_protected:Npn 1497   { \tex_long:D \tex_xdef:D }
\cs_gset_protected:Npx 1498 \cs_set_protected:Npn \cs_gset_protected_nopar:Npn
\cs_gset_protected:Npx 1499   { \tex_protected:D \tex_gdef:D }
1500 \cs_set_protected:Npn \cs_gset_protected_nopar:Npx
1501   { \tex_protected:D \tex_xdef:D }
1502 \cs_set_protected:Npn \cs_gset_protected:Npn
1503   { \tex_protected:D \tex_long:D \tex_gdef:D }
1504 \cs_set_protected:Npn \cs_gset_protected:Npx
1505   { \tex_protected:D \tex_long:D \tex_xdef:D }

```

(End definition for `\cs_gset_nopar:Npn` and others. These functions are documented on page 16.)

41.4 Selecting tokens

```

1506 <@@=exp>

```

`\l__exp_internal_tl` Scratch token list variable for `l3expan`, used by `\use:x`, used in defining conditionals. We don't use `tl` methods because `l3basics` is loaded earlier.

```

1507 \cs_set_nopar:Npn \l__exp_internal_tl { }

```

(End definition for `\l__exp_internal_tl`.)

`\use:c` This macro grabs its argument and returns a csname from it.

```

1508 \cs_set:Npn \use:c #1 { \cs:w #1 \cs_end: }

```

(End definition for `\use:c`. This function is documented on page 20.)

`\use:x` Fully expands its argument and passes it to the input stream. Uses the reserved `\l__exp_internal_tl` which we've set up above.

```

1509 \cs_set_protected:Npn \use:x #1
1510   {
1511     \cs_set_nopar:Npx \l__exp_internal_tl {#1}
1512     \l__exp_internal_tl
1513   }

```


(End definition for `\use:x`. This function is documented on page 25.)

```
1514 <@@=use>
```

`\use:e` In non-LuaTeXengines older than 2019, `\expanded` is emulated.

```
1515 \cs_set:Npn \use:e #1 { \tex_expanded:D {#1} }
1516 \tex_ifdefined:D \tex_expanded:D \tex_else:D
1517   \cs_set:Npn \use:e #1 { \exp_args:Ne \use:n {#1} }
1518 \tex_fi:D
```

(End definition for `\use:e`. This function is documented on page 25.)

```
1519 <@@=exp>
```

`\use:n` These macros grab their arguments and return them back to the input (with outer braces removed).

```
\use:nnn 1520 \cs_set:Npn \use:n   #1      {#1}
\use:nnnn 1521 \cs_set:Npn \use:nn  #1#2    {#1#2}
          1522 \cs_set:Npn \use:nnn  #1#2#3   {#1#2#3}
          1523 \cs_set:Npn \use:nnnn #1#2#3#4 {#1#2#3#4}
```

(End definition for `\use:n` and others. These functions are documented on page 23.)

`\use_i:nn` The equivalent to L^AT_EX 2_ε's `\@firstoftwo` and `\@secondoftwo`.

```
\use_ii:nn 1524 \cs_set:Npn \use_i:nn  #1#2 {#1}
           1525 \cs_set:Npn \use_ii:nn #1#2 {#2}
```

(End definition for `\use_i:nn` and `\use_ii:nn`. These functions are documented on page 24.)

`\use_i:nnn` We also need something for picking up arguments from a longer list.

```
\use_ii:nnn 1526 \cs_set:Npn \use_i:nnn   #1#2#3 {#1}
\use_iii:nnn 1527 \cs_set:Npn \use_ii:nnn  #1#2#3 {#2}
\use_i_ii:nnn 1528 \cs_set:Npn \use_iii:nnn  #1#2#3 {#3}
\use_i:nnnn 1529 \cs_set:Npn \use_i_ii:nnn  #1#2#3 {#1#2}
\use_ii:nnnn 1530 \cs_set:Npn \use_i:nnnn   #1#2#3#4 {#1}
\use_iii:nnnn 1531 \cs_set:Npn \use_ii:nnnn  #1#2#3#4 {#2}
\use_iv:nnnn 1532 \cs_set:Npn \use_iii:nnnn  #1#2#3#4 {#3}
           1533 \cs_set:Npn \use_iv:nnnn   #1#2#3#4 {#4}
```

(End definition for `\use_i:nnn` and others. These functions are documented on page 24.)

`\use_ii_i:nn`

```
1534 \cs_set:Npn \use_ii_i:nn #1#2 { #2 #1 }
```

(End definition for `\use_ii_i:nn`. This function is documented on page 24.)

`\use_none_delimit_by_q_nil:w` Functions that gobble everything until they see either `\q_nil`, `\q_stop`, or `\q_recursion_stop`, respectively.

```
\use_none_delimit_by_q_stop:w 1535 \cs_set:Npn \use_none_delimit_by_q_nil:w #1 \q_nil { }
\use_none_delimit_by_q_recursion_stop:w 1536 \cs_set:Npn \use_none_delimit_by_q_stop:w #1 \q_stop { }
1537 \cs_set:Npn \use_none_delimit_by_q_recursion_stop:w #1 \q_recursion_stop { }
```

(End definition for `\use_none_delimit_by_q_nil:w`, `\use_none_delimit_by_q_stop:w`, and `\use_none_delimit_by_q_recursion_stop:w`. These functions are documented on page 25.)

```

\use_i_delimit_by_q_nil:nw
\use_i_delimit_by_q_stop:nw
\use_i_delimit_by_q_recursion_stop:nw

```

Same as above but execute first argument after gobbling. Very useful when you need to skip the rest of a mapping sequence but want an easy way to control what should be expanded next.

```

1538 \cs_set:Npn \use_i_delimit_by_q_nil:nw #1#2 \q_nil {#1}
1539 \cs_set:Npn \use_i_delimit_by_q_stop:nw #1#2 \q_stop {#1}
1540 \cs_set:Npn \use_i_delimit_by_q_recursion_stop:nw
1541 #1#2 \q_recursion_stop {#1}

```

(End definition for `\use_i_delimit_by_q_nil:nw`, `\use_i_delimit_by_q_stop:nw`, and `\use_i_delimit_by_q_recursion_stop:nw`. These functions are documented on page 25.)

41.5 Gobbling tokens from input

```

\use_none:n
\use_none:nn
\use_none:nnn
\use_none:nnnn
\use_none:nnnnn
\use_none:nnnnnn
\use_none:nnnnnnn
\use_none:nnnnnnnn

```

To gobble tokens from the input we use a standard naming convention: the number of tokens gobbled is given by the number of `n`'s following the `:` in the name. Although we could define functions to remove ten arguments or more using separate calls of `\use_none:nnnnn`, this is very non-intuitive to the programmer who will assume that expanding such a function once takes care of gobbling all the tokens in one go.

```

1542 \cs_set:Npn \use_none:n #1 { }
1543 \cs_set:Npn \use_none:nn #1#2 { }
1544 \cs_set:Npn \use_none:nnn #1#2#3 { }
1545 \cs_set:Npn \use_none:nnnn #1#2#3#4 { }
1546 \cs_set:Npn \use_none:nnnnn #1#2#3#4#5 { }
1547 \cs_set:Npn \use_none:nnnnnn #1#2#3#4#5#6 { }
1548 \cs_set:Npn \use_none:nnnnnnn #1#2#3#4#5#6#7 { }
1549 \cs_set:Npn \use_none:nnnnnnnn #1#2#3#4#5#6#7#8 { }
1550 \cs_set:Npn \use_none:nnnnnnnnn #1#2#3#4#5#6#7#8#9 { }

```

(End definition for `\use_none:n` and others. These functions are documented on page 25.)

41.6 Debugging and patching later definitions

```

1551 <@@=debug>

```

```

\__kernel_if_debug:TF

```

A more meaningful test of whether debugging is enabled than messing up with guards. We can also more easily change the logic in one place then. This is needed primarily for deprecations.

```

1552 \cs_set_protected:Npn \__kernel_if_debug:TF #1#2 {#2}

```

(End definition for `__kernel_if_debug:TF`.)

```

\debug_on:n
\debug_off:n

```

Stubs.

```

1553 \cs_set_protected:Npn \debug_on:n #1
1554 {
1555   \msg_error:nnx { debug } { enable-debug }
1556   { \tl_to_str:n { \debug_on:n {#1} } }
1557 }
1558 \cs_set_protected:Npn \debug_off:n #1
1559 {
1560   \msg_error:nnx { debug } { enable-debug }
1561   { \tl_to_str:n { \debug_off:n {#1} } }
1562 }

```

(End definition for `\debug_on:n` and `\debug_off:n`. These functions are documented on page 29.)

```
\debug_suspend:
\debug_resume: 1563 \cs_set_protected:Npn \debug_suspend: { }
                1564 \cs_set_protected:Npn \debug_resume: { }
```

(End definition for `\debug_suspend:` and `\debug_resume:`. These functions are documented on page 29.)

`__kernel_deprecation_code:nn` Some commands were more recently deprecated and not yet removed; only make these into errors if the user requests it. This relies on two token lists, filled up in `l3deprecation`.

```
\g__debug_deprecation_on_tl
\g__debug_deprecation_off_tl 1565 \cs_set_nopar:Npn \g__debug_deprecation_on_tl { }
                             1566 \cs_set_nopar:Npn \g__debug_deprecation_off_tl { }
                             1567 \cs_set_protected:Npn \__kernel_deprecation_code:nn #1#2
                             1568 {
                             1569   \tl_gput_right:Nn \g__debug_deprecation_on_tl {#1}
                             1570   \tl_gput_right:Nn \g__debug_deprecation_off_tl {#2}
                             1571 }
```

(End definition for `__kernel_deprecation_code:nn`, `\g__debug_deprecation_on_tl`, and `\g__debug_deprecation_off_tl`.)

41.7 Conditional processing and definitions

```
1572 <@@=prg>
```

Underneath any predicate function (`_p`) or other conditional forms (TF, etc.) is a built-in logic saying that it after all of the testing and processing must return the *state* this leaves \TeX in. Therefore, a simple user interface could be something like

```
\if_meaning:w #1#2
  \prg_return_true:
\else:
  \if_meaning:w #1#3
    \prg_return_true:
  \else:
    \prg_return_false:
  \fi:
\fi:
```

Usually, a \TeX programmer would have to insert a number of `\exp_after:wN`s to ensure the state value is returned at exactly the point where the last conditional is finished. However, that obscures the code and forces the \TeX programmer to prove that he/she knows the $2^n - 1$ table. We therefore provide the simpler interface.

`\prg_return_true:` The idea here is that `\exp:w` expands fully any `\else:` and `\fi:` that are waiting to be discarded, before reaching the `\exp_end:` which leaves an empty expansion. The code can then leave either the first or second argument in the input stream. This means that all of the branching code has to contain at least two tokens: see how the logical tests are actually implemented to see this.

```
1573 \cs_set:Npn \prg_return_true:
1574 { \exp_after:wN \use_i:nn \exp:w }
1575 \cs_set:Npn \prg_return_false:
1576 { \exp_after:wN \use_ii:nn \exp:w }
```

An extended state space could be implemented by including a more elaborate function in place of `\use_i:nn/\use_ii:nn`. Provided two arguments are absorbed then the code would work.

(End definition for `\prg_return_true:` and `\prg_return_false:`. These functions are documented on page 64.)

`\prg_use_none_delimit_by_q_recursion_stop:w`

Private version of `\use_none_delimit_by_q_recursion_stop:w`.

```
1577 \cs_set:Npn \__prg_use_none_delimit_by_q_recursion_stop:w
1578   #1 \q__prg_recursion_stop { }
```

(End definition for `__prg_use_none_delimit_by_q_recursion_stop:w`.)

`\prg_set_conditional:Npnn`

`\prg_new_conditional:Npnn`

`\prg_set_protected_conditional:Npnn`

`\prg_new_protected_conditional:Npnn`

`__prg_generate_conditional_parm:NNNpnn`

The user functions for the types using parameter text from the programmer. The various functions only differ by which function is used for the assignment. For those `Npnn` type functions, we must grab the parameter text, reading everything up to a left brace before continuing. Then split the base function into name and signature, and feed $\{\langle name \rangle\}$ $\{\langle signature \rangle\}$ $\langle boolean \rangle$ $\{\langle set \text{ or } new \rangle\}$ $\{\langle maybe \text{ protected} \rangle\}$ $\{\langle parameters \rangle\}$ $\{TF, \dots\}$ $\{\langle code \rangle\}$ to the auxiliary function responsible for defining all conditionals. Note that `e` stands for expandable and `p` for protected.

```
1579 \cs_set_protected:Npn \prg_set_conditional:Npnn
1580   { \__prg_generate_conditional_parm:NNNpnn \cs_set:Npn e }
1581 \cs_set_protected:Npn \prg_new_conditional:Npnn
1582   { \__prg_generate_conditional_parm:NNNpnn \cs_new:Npn e }
1583 \cs_set_protected:Npn \prg_set_protected_conditional:Npnn
1584   { \__prg_generate_conditional_parm:NNNpnn \cs_set_protected:Npn p }
1585 \cs_set_protected:Npn \prg_new_protected_conditional:Npnn
1586   { \__prg_generate_conditional_parm:NNNpnn \cs_new_protected:Npn p }
1587 \cs_set_protected:Npn \__prg_generate_conditional_parm:NNNpnn #1#2#3#4#
1588   {
1589     \use:x
1590     {
1591       \__prg_generate_conditional:nnNNNnnn
1592       \cs_split_function:N #3
1593     }
1594     #1 #2 {#4}
1595   }
```

(End definition for `\prg_set_conditional:Npnn` and others. These functions are documented on page 62.)

`\prg_set_conditional:Nnn`

`\prg_new_conditional:Nnn`

`\prg_set_protected_conditional:Nnn`

`\prg_new_protected_conditional:Nnn`

`__prg_generate_conditional_count:NNNnn`

`__prg_generate_conditional_count:nnNNNnn`

The user functions for the types automatically inserting the correct parameter text based on the signature. The various functions only differ by which function is used for the assignment. Split the base function into name and signature. The second auxiliary generates the parameter text from the number of letters in the signature. Then feed $\{\langle name \rangle\}$ $\{\langle signature \rangle\}$ $\langle boolean \rangle$ $\{\langle set \text{ or } new \rangle\}$ $\{\langle maybe \text{ protected} \rangle\}$ $\{\langle parameters \rangle\}$ $\{TF, \dots\}$ $\{\langle code \rangle\}$ to the auxiliary function responsible for defining all conditionals. If the $\langle signature \rangle$ has more than 9 letters, the definition is aborted since \TeX macros have at most 9 arguments. The erroneous case where the function name contains no colon is captured later.

```
1596 \cs_set_protected:Npn \prg_set_conditional:Nnn
1597   { \__prg_generate_conditional_count:NNNnn \cs_set:Npn e }
1598 \cs_set_protected:Npn \prg_new_conditional:Nnn
1599   { \__prg_generate_conditional_count:NNNnn \cs_new:Npn e }
```

```

1600 \cs_set_protected:Npn \prg_set_protected_conditional:Nnn
1601   { \__prg_generate_conditional_count:NNNnn \cs_set_protected:Npn p }
1602 \cs_set_protected:Npn \prg_new_protected_conditional:Nnn
1603   { \__prg_generate_conditional_count:NNNnn \cs_new_protected:Npn p }
1604 \cs_set_protected:Npn \__prg_generate_conditional_count:NNNnn #1#2#3
1605   {
1606     \use:x
1607     {
1608       \__prg_generate_conditional_count:nnNNNnn
1609       \cs_split_function:N #3
1610     }
1611     #1 #2
1612   }
1613 \cs_set_protected:Npn \__prg_generate_conditional_count:nnNNNnn #1#2#3#4#5
1614   {
1615     \__kernel_cs_parm_from_arg_count:nnF
1616     { \__prg_generate_conditional:nnNNNnnn {#1} {#2} #3 #4 #5 }
1617     { \tl_count:n {#2} }
1618     {
1619       \msg_error:nnxx { kernel } { bad-number-of-arguments }
1620       { \token_to_str:c { #1 : #2 } }
1621       { \tl_count:n {#2} }
1622       \use_none:nn
1623     }
1624   }

```

(End definition for `\prg_set_conditional:Nnn` and others. These functions are documented on page 62.)

```

\__prg_generate_conditional:nnNNNnnn
\__prg_generate_conditional:NNnnnnNw
\__prg_generate_conditional_test:w
\__prg_generate_conditional_fast:nw

```

The workhorse here is going through a list of desired forms, *i.e.*, `p`, `TF`, `T` and `F`. The first three arguments come from splitting up the base form of the conditional, which gives the name, signature and a boolean to signal whether or not there was a colon in the name. In the absence of a colon, we throw an error and don't define any conditional. The fourth and fifth arguments build up the defining function. The sixth is the parameters to use (possibly empty), the seventh is the list of forms to define, the eighth is the replacement text which we will augment when defining the forms. The use of `\tl_to_str:n` makes the later loop more robust.

A large number of our low-level conditionals look like `\prg_return_true:` `\else:` `\prg_return_false:` `\fi:` so we optimize this special case by calling `__prg_generate_conditional_fast:nw` `{\code}`. This passes `\use_i:nn` instead of `\use_i_ii:nnn` to functions such as `__prg_generate_p_form:wNNnnnnN`.

```

1625 \cs_set_protected:Npn \__prg_generate_conditional:nnNNNnnn #1#2#3#4#5#6#7#8
1626   {
1627     \if_meaning:w \c_false_bool #3
1628     \msg_error:nnx { kernel } { missing-colon }
1629     { \token_to_str:c {#1} }
1630     \exp_after:wN \use_none:nn
1631     \fi:
1632     \use:x
1633     {
1634       \exp_not:N \__prg_generate_conditional:NNnnnnNw
1635       \exp_not:n { #4 #5 {#1} {#2} {#6} }
1636       \__prg_generate_conditional_test:w

```

```

1637         #8 \s__prg_mark
1638         \__prg_generate_conditional_fast:nw
1639         \prg_return_true: \else: \prg_return_false: \fi: \s__prg_mark
1640         \use_none:n
1641         \exp_not:n { {#8} \use_i_ii:nnn }
1642         \tl_to_str:n {#7}
1643         \exp_not:n { , \q__prg_recursion_tail , \q__prg_recursion_stop }
1644     }
1645 }
1646 \cs_set:Npn \__prg_generate_conditional_test:w
1647     #1 \prg_return_true: \else: \prg_return_false: \fi: \s__prg_mark #2
1648     { #2 {#1} }
1649 \cs_set:Npn \__prg_generate_conditional_fast:nw #1#2 \exp_not:n #3
1650     { \exp_not:n { {#1} \use_i:nn } }

```

Looping through the list of desired forms. First are six arguments and seventh is the form. Use the form to call the correct type. If the form does not exist, the `\use:c` construction results in `\relax`, and the error message is displayed (unless the form is empty, to allow for `{T, , F}`), then `\use_none:nnnnnnnn` cleans up. Otherwise, the error message is removed by the variant form.

```

1651 \cs_set_protected:Npn \__prg_generate_conditional:NNnnnnNw #1#2#3#4#5#6#7#8 ,
1652 {
1653     \if_meaning:w \q__prg_recursion_tail #8
1654     \exp_after:wN \__prg_use_none_delimit_by_q_recursion_stop:w
1655     \fi:
1656     \use:c { __prg_generate_ #8 _form:wNNnnnnN }
1657     \tl_if_empty:nF {#8}
1658     {
1659         \msg_error:nnxx
1660         { kernel } { conditional-form-unknown }
1661         {#8} { \token_to_str:c { #3 : #4 } }
1662     }
1663     \use_none:nnnnnnnn
1664     \s__prg_stop
1665     #1 #2 {#3} {#4} {#5} {#6} #7
1666     \__prg_generate_conditional:NNnnnnNw #1 #2 {#3} {#4} {#5} {#6} #7
1667 }

```

(End definition for `__prg_generate_conditional:nnNNnnnn` and others.)

```

\__prg_generate_p_form:wNNnnnnN
\__prg_generate_TF_form:wNNnnnnN
\__prg_generate_T_form:wNNnnnnN
\__prg_generate_F_form:wNNnnnnN
\__prg_p_true:w
\__prg_T_true:w
\__prg_F_true:w
\__prg_TF_true:w

```

How to generate the various forms. Those functions take the following arguments: 1: junk, 2: `\cs_set:Npn` or similar, 3: `p` (for protected conditionals) or `e`, 4: function name, 5: signature, 6: parameter text, 7: replacement (possibly trimmed by `__prg_generate_conditional_fast:nw`), 8: `\use_i_ii:nnn` or `\use_i:nn` (for “fast” conditionals). Remember that the logic-returning functions expect two arguments to be present after `\exp_end::` notice the construction of the different variants relies on this, and that the TF and F variants will be slightly faster than the T version. The `p` form is only valid for expandable tests, we check for that by making sure that the second argument is empty. For “fast” conditionals, #7 has an extra `\if....` To optimize a bit further we don’t use `\exp_after:wN \use_ii:nnn` and similar but instead use `__prg_TF_true:w` and similar to swap out the macro after `\fi:`. It would be a tiny bit faster if we directly grabbed the T and F arguments there, but if those are actually missing, the recovery from the runaway argument would not insert `\fi:` back, messing up nesting of conditionals.

```

1668 \cs_set_protected:Npn \__prg_generate_p_form:wNNnnnnN
1669   #1 \s__prg_stop #2#3#4#5#6#7#8
1670   {
1671     \if_meaning:w e #3
1672     \exp_after:wN \use_i:nn
1673   \else:
1674     \exp_after:wN \use_ii:nn
1675   \fi:
1676   {
1677     #8
1678     { \exp_args:Nc #2 { #4 _p: #5 } #6 }
1679     { { #7 \exp_end: \c_true_bool \c_false_bool } }
1680     { #7 \__prg_p_true:w \fi: \c_false_bool }
1681   }
1682   {
1683     \msg_error:nnx { kernel } { protected-predicate }
1684     { \token_to_str:c { #4 _p: #5 } }
1685   }
1686 }
1687 \cs_set_protected:Npn \__prg_generate_T_form:wNNnnnnN
1688   #1 \s__prg_stop #2#3#4#5#6#7#8
1689   {
1690     #8
1691     { \exp_args:Nc #2 { #4 : #5 T } #6 }
1692     { { #7 \exp_end: \use:n \use_none:n } }
1693     { #7 \__prg_T_true:w \fi: \use_none:n }
1694   }
1695 \cs_set_protected:Npn \__prg_generate_F_form:wNNnnnnN
1696   #1 \s__prg_stop #2#3#4#5#6#7#8
1697   {
1698     #8
1699     { \exp_args:Nc #2 { #4 : #5 F } #6 }
1700     { { #7 \exp_end: { } } }
1701     { #7 \__prg_F_true:w \fi: \use:n }
1702   }
1703 \cs_set_protected:Npn \__prg_generate_TF_form:wNNnnnnN
1704   #1 \s__prg_stop #2#3#4#5#6#7#8
1705   {
1706     #8
1707     { \exp_args:Nc #2 { #4 : #5 TF } #6 }
1708     { { #7 \exp_end: { } } }
1709     { #7 \__prg_TF_true:w \fi: \use_ii:nn }
1710   }
1711 \cs_set:Npn \__prg_p_true:w \fi: \c_false_bool { \fi: \c_true_bool }
1712 \cs_set:Npn \__prg_T_true:w \fi: \use_none:n { \fi: \use:n }
1713 \cs_set:Npn \__prg_F_true:w \fi: \use:n { \fi: \use_none:n }
1714 \cs_set:Npn \__prg_TF_true:w \fi: \use_ii:nn { \fi: \use_i:nn }

```

(End definition for __prg_generate_p_form:wNNnnnnN and others.)

\prg_set_eq_conditional:NNn The setting-equal functions. Split both functions and feed $\{\langle name_1 \rangle\}$ $\{\langle signature_1 \rangle\}$
\prg_new_eq_conditional:NNn $\langle boolean_1 \rangle$ $\{\langle name_2 \rangle\}$ $\{\langle signature_2 \rangle\}$ $\langle boolean_2 \rangle$ $\langle copying\ function \rangle$ $\langle conditions \rangle$, \q__-
__prg_set_eq_conditional:NNn prg_recursion_tail , \q__prg_recursion_stop to a first auxiliary.

```

1715 \cs_set_protected:Npn \prg_set_eq_conditional:NNn

```

```

1716 { \_prg_set_eq_conditional:NNNn \cs_set_eq:cc }
1717 \cs_set_protected:Npn \prg_new_eq_conditional:NNn
1718 { \_prg_set_eq_conditional:NNNn \cs_new_eq:cc }
1719 \cs_set_protected:Npn \_prg_set_eq_conditional:NNNn #1#2#3#4
1720 {
1721   \use:x
1722   {
1723     \exp_not:N \_prg_set_eq_conditional:nnNnnNNw
1724     \cs_split_function:N #2
1725     \cs_split_function:N #3
1726     \exp_not:N #1
1727     \tl_to_str:n {#4}
1728     \exp_not:n { , \q__prg_recursion_tail , \q__prg_recursion_stop }
1729   }
1730 }

```

(End definition for `\prg_set_eq_conditional:NNn`, `\prg_new_eq_conditional:NNn`, and `_prg_set_eq_conditional:NNNn`. These functions are documented on page 64.)

```

\_prg_set_eq_conditional:nnNnnNNw
\_prg_set_eq_conditional_loop:nnnnNw
\_prg_set_eq_conditional_p_form:nnn
\_prg_set_eq_conditional_TF_form:nnn
\_prg_set_eq_conditional_T_form:nnn
\_prg_set_eq_conditional_F_form:nnn

```

Split the function to be defined, and setup a manual clist loop over argument #6 of the first auxiliary. The second auxiliary receives twice three arguments coming from splitting the function to be defined and the function to copy. Make sure that both functions contained a colon, otherwise we don't know how to build conditionals, hence abort. Call the looping macro, with arguments $\{\langle name_1 \rangle\} \{\langle signature_1 \rangle\} \{\langle name_2 \rangle\} \{\langle signature_2 \rangle\}$ $\langle copying\ function \rangle$ and followed by the comma list. At each step in the loop, make sure that the conditional form we copy is defined, and copy it, otherwise abort.

```

1731 \cs_set_protected:Npn \_prg_set_eq_conditional:nnNnnNNw #1#2#3#4#5#6
1732 {
1733   \if_meaning:w \c_false_bool #3
1734   \msg_error:nnx { kernel } { missing-colon }
1735   { \token_to_str:c {#1} }
1736   \exp_after:wN \_prg_use_none_delimit_by_q_recursion_stop:w
1737   \fi:
1738   \if_meaning:w \c_false_bool #6
1739   \msg_error:nnx { kernel } { missing-colon }
1740   { \token_to_str:c {#4} }
1741   \exp_after:wN \_prg_use_none_delimit_by_q_recursion_stop:w
1742   \fi:
1743   \_prg_set_eq_conditional_loop:nnnnNw {#1} {#2} {#4} {#5}
1744 }
1745 \cs_set_protected:Npn \_prg_set_eq_conditional_loop:nnnnNw #1#2#3#4#5#6 ,
1746 {
1747   \if_meaning:w \q__prg_recursion_tail #6
1748   \exp_after:wN \_prg_use_none_delimit_by_q_recursion_stop:w
1749   \fi:
1750   \use:c { \_prg_set_eq_conditional_ #6 _form:wNnnnn }
1751   \tl_if_empty:nF {#6}
1752   {
1753     \msg_error:nnxx
1754     { kernel } { conditional-form-unknown }
1755     {#6} { \token_to_str:c { #1 : #2 } }
1756   }
1757   \use_none:nnnnnn
1758   \s__prg_stop

```



```

1759      #5 {#1} {#2} {#3} {#4}
1760      \__prg_set_eq_conditional_loop:nnnnNw {#1} {#2} {#3} {#4} #5
1761    }
1762    \cs_set:Npn \__prg_set_eq_conditional_p_form:wNnnnn #1 \s__prg_stop #2#3#4#5#6
1763      { #2 { #3 _p : #4      }      { #5 _p : #6      } }
1764    \cs_set:Npn \__prg_set_eq_conditional_TF_form:wNnnnn #1 \s__prg_stop #2#3#4#5#6
1765      { #2 { #3      : #4 TF }      { #5      : #6 TF } }
1766    \cs_set:Npn \__prg_set_eq_conditional_T_form:wNnnnn #1 \s__prg_stop #2#3#4#5#6
1767      { #2 { #3      : #4 T  }      { #5      : #6 T  } }
1768    \cs_set:Npn \__prg_set_eq_conditional_F_form:wNnnnn #1 \s__prg_stop #2#3#4#5#6
1769      { #2 { #3      : #4 F  }      { #5      : #6 F  } }

```

(End definition for __prg_set_eq_conditional:nnnnNw and others.)

All that is left is to define the canonical boolean true and false. I think Michael originated the idea of expandable boolean tests. At first these were supposed to expand into either TT or TF to be tested using \if:w but this was later changed to 00 and 01, so they could be used in logical operations. Later again they were changed to being numerical constants with values of 1 for true and 0 for false. We need this from the get-go.

`\c_true_bool` Here are the canonical boolean values.
`\c_false_bool`

```

1770 \tex_chardef:D \c_true_bool  = 1 ~
1771 \tex_chardef:D \c_false_bool = 0 ~

```

(End definition for \c_true_bool and \c_false_bool. These variables are documented on page 26.)

41.8 Dissecting a control sequence

```

1772 <@@=cs>

```

```

\__cs_count_signature:N \__cs_count_signature:N <function>

```

Splits the *<function>* into the *<name>* (i.e. the part before the colon) and the *<signature>* (i.e. after the colon). The *<number>* of tokens in the *<signature>* is then left in the input stream. If there was no *<signature>* then the result is the marker value -1 .

```

\__cs_get_function_name:N * \__cs_get_function_name:N <function>

```

Splits the *<function>* into the *<name>* (i.e. the part before the colon) and the *<signature>* (i.e. after the colon). The *<name>* is then left in the input stream without the escape character present made up of tokens with category code 12 (other).

```

\__cs_get_function_signature:N * \__cs_get_function_signature:N <function>

```

Splits the *<function>* into the *<name>* (i.e. the part before the colon) and the *<signature>* (i.e. after the colon). The *<signature>* is then left in the input stream made up of tokens with category code 12 (other).

`__cs_tmp:w` Function used for various short-term usages, for instance defining functions whose definition involves tokens which are hard to insert normally (spaces, characters with category other).

`\cs_to_str:N` This converts a control sequence into the character string of its name, removing the leading escape character. This turns out to be a non-trivial matter as there are different cases:

- The usual case of a printable escape character;
- the case of a non-printable escape character, e.g., when the value of the `\escapechar` is negative;
- when the escape character is a space.

One approach to solve this is to test how many tokens result from `\token_to_str:N \a`. If there are two tokens, then the escape character is printable, while if it is non-printable then only one is present.

However, there is an additional complication: the control sequence itself may start with a space. Clearly that should *not* be lost in the process of converting to a string. So the approach adopted is a little more intricate still. When the escape character is printable, `\token_to_str:N _` yields the escape character itself and a space. The character codes are different, thus the `\if:w` test is false, and TeX reads `_cs_to_str:N` after turning the following control sequence into a string; this auxiliary removes the escape character, and stops the expansion of the initial `\tex_romannumeral:D`. The second case is that the escape character is not printable. Then the `\if:w` test is unfinished after reading a space from `\token_to_str:N _`, and the auxiliary `_cs_to_str:w` is expanded, feeding – as a second character for the test; the test is false, and TeX skips to `\fi:`, then performs `\token_to_str:N`, and stops the `\tex_romannumeral:D` with `\c_zero_int`. The last case is that the escape character is itself a space. In this case, the `\if:w` test is true, and the auxiliary `_cs_to_str:w` comes into play, inserting `-\int_value:w`, which expands `\c_zero_int` to the character 0. The initial `\tex_romannumeral:D` then sees 0, which is not a terminated number, followed by the escape character, a space, which is removed, terminating the expansion of `\tex_romannumeral:D`. In all three cases, `\cs_to_str:N` takes two expansion steps to be fully expanded.

```
1773 \cs_set:Npn \cs_to_str:N
1774 {
```

We implement the expansion scheme using `\tex_romannumeral:D` terminating it with `\c_zero_int` rather than using `\exp:w` and `\exp_end:` as we normally do. The reason is that the code heavily depends on terminating the expansion with `\c_zero_int` so we make this dependency explicit.

```
1775 \tex_romannumeral:D
1776 \if:w \token_to_str:N \\_cs_to_str:w \fi:
1777 \exp_after:wN \_cs_to_str:N \token_to_str:N
1778 }
1779 \cs_set:Npn \_cs_to_str:N #1 { \c_zero_int }
1780 \cs_set:Npn \_cs_to_str:w #1 \_cs_to_str:N
1781 { - \int_value:w \fi: \exp_after:wN \c_zero_int }
```

If speed is a concern we could use `\csstring` in LuaTeX. For the empty csname that primitive gives an empty result while the current `\cs_to_str:N` gives incorrect results in all engines (this is impossible to fix without huge performance hit).

(End definition for `\cs_to_str:N`, `_cs_to_str:N`, and `_cs_to_str:w`. This function is documented on page 21.)

`\cs_split_function:N` This function takes a function name and splits it into name with the escape char removed and argument specification. In addition to this, a third argument, a boolean $\langle true \rangle$ or $\langle false \rangle$ is returned with $\langle true \rangle$ for when there is a colon in the function and $\langle false \rangle$ if there is not.

We cannot use `:` directly as it has the wrong category code so an `x`-type expansion is used to force the conversion.

First ensure that we actually get a properly evaluated string by expanding `\cs_to_str:N` twice. If the function contained a colon, the auxiliary takes as `#1` the function name, delimited by the first colon, then the signature `#2`, delimited by `\s__cs_mark`, then `\c_true_bool` as `#3`, and `#4` cleans up until `\s__cs_stop`. Otherwise, the `#1` contains the function name and `\s__cs_mark \c_true_bool`, `#2` is empty, `#3` is `\c_false_bool`, and `#4` cleans up. The second auxiliary trims the trailing `\s__cs_mark` from the function name if present (that is, if the original function had no colon).

```

1782 \cs_set_protected:Npn \__cs_tmp:w #1
1783 {
1784   \cs_set:Npn \cs_split_function:N ##1
1785   {
1786     \exp_after:wN \exp_after:wN \exp_after:wN
1787     \__cs_split_function_auxi:w
1788     \cs_to_str:N ##1 \s__cs_mark \c_true_bool
1789     #1 \s__cs_mark \c_false_bool \s__cs_stop
1790   }
1791   \cs_set:Npn \__cs_split_function_auxi:w
1792   ##1 #1 ##2 \s__cs_mark ##3##4 \s__cs_stop
1793   { \__cs_split_function_auxii:w ##1 \s__cs_mark \s__cs_stop {##2} ##3 }
1794   \cs_set:Npn \__cs_split_function_auxii:w ##1 \s__cs_mark ##2 \s__cs_stop
1795   { {##1} }
1796 }
1797 \exp_after:wN \__cs_tmp:w \token_to_str:N :
```

(End definition for `\cs_split_function:N`, `__cs_split_function_auxi:w`, and `__cs_split_function_auxii:w`. This function is documented on page 22.)

41.9 Exist or free

A control sequence is said to *exist* (to be used) if has an entry in the hash table and its meaning is different from the primitive `\relax` token. A control sequence is said to be *free* (to be defined) if it does not already exist.

`\cs_if_exist_p:N` Two versions for checking existence. For the `N` form we firstly check for `\scan_stop:` and then if it is in the hash table. There is no problem when inputting something like `\else:` or `\fi:` as \TeX will only ever skip input in case the token tested against is `\scan_stop:`.

```

\cs_if_exist:N $\underline{TF}$ 
\cs_if_exist:c $\underline{TF}$ 
1798 \prg_set_conditional:Npnn \cs_if_exist:N #1 { p , T , F , TF }
1799 {
1800   \if_meaning:w #1 \scan_stop:
1801   \prg_return_false:
1802   \else:
1803     \if_cs_exist:N #1
1804     \prg_return_true:
1805     \else:
1806       \prg_return_false:
1807     \fi:
```

```

1808     \fi:
1809 }

```

For the `c` form we firstly check if it is in the hash table and then for `\scan_stop`: so that we do not add it to the hash table unless it was already there. Here we have to be careful as the text to be skipped if the first test is false may contain tokens that disturb the scanner. Therefore, we ensure that the second test is performed after the first one has concluded completely.

```

1810 \prg_set_conditional:Npnn \cs_if_exist:c #1 { p , T , F , TF }
1811 {
1812     \if_cs_exist:w #1 \cs_end:
1813     \exp_after:wN \use_i:nn
1814     \else:
1815     \exp_after:wN \use_ii:nn
1816     \fi:
1817     {
1818     \exp_after:wN \if_meaning:w \cs:w #1 \cs_end: \scan_stop:
1819     \prg_return_false:
1820     \else:
1821     \prg_return_true:
1822     \fi:
1823     }
1824     \prg_return_false:
1825 }

```

(End definition for `\cs_if_exist:NTF`. This function is documented on page 27.)

`\cs_if_free_p:N`
`\cs_if_free_p:c`
`\cs_if_free:NTF`
`\cs_if_free:cTF`

The logical reversal of the above.

```

1826 \prg_set_conditional:Npnn \cs_if_free:N #1 { p , T , F , TF }
1827 {
1828     \if_meaning:w #1 \scan_stop:
1829     \prg_return_true:
1830     \else:
1831     \if_cs_exist:N #1
1832     \prg_return_false:
1833     \else:
1834     \prg_return_true:
1835     \fi:
1836     \fi:
1837 }
1838 \prg_set_conditional:Npnn \cs_if_free:c #1 { p , T , F , TF }
1839 {
1840     \if_cs_exist:w #1 \cs_end:
1841     \exp_after:wN \use_i:nn
1842     \else:
1843     \exp_after:wN \use_ii:nn
1844     \fi:
1845     {
1846     \exp_after:wN \if_meaning:w \cs:w #1 \cs_end: \scan_stop:
1847     \prg_return_true:
1848     \else:
1849     \prg_return_false:
1850     \fi:
1851     }

```

```

1852     { \prg_return_true: }
1853 }

```

(End definition for `\cs_if_free:NTF`. This function is documented on page 27.)

`\cs_if_exist_use:N` The `\cs_if_exist_use:...` functions cannot be implemented as conditionals because the true branch must leave both the control sequence itself and the true code in the input stream. For the `c` variants, we are careful not to put the control sequence in the hash table if it does not exist. In LuaTeX we could use the `\lastnamedcs` primitive.

```

1854 \cs_set:Npn \cs_if_exist_use:NTF #1#2
1855   { \cs_if_exist:NTF #1 { #1 #2 } }
1856 \cs_set:Npn \cs_if_exist_use:NF #1
1857   { \cs_if_exist:NTF #1 { #1 } }
1858 \cs_set:Npn \cs_if_exist_use:NT #1 #2
1859   { \cs_if_exist:NTF #1 { #1 #2 } { } }
1860 \cs_set:Npn \cs_if_exist_use:N #1
1861   { \cs_if_exist:NTF #1 { #1 } { } }
1862 \cs_set:Npn \cs_if_exist_use:cTF #1#2
1863   { \cs_if_exist:cTF {#1} { \use:c {#1} #2 } }
1864 \cs_set:Npn \cs_if_exist_use:cF #1
1865   { \cs_if_exist:cTF {#1} { \use:c {#1} } }
1866 \cs_set:Npn \cs_if_exist_use:cT #1#2
1867   { \cs_if_exist:cTF {#1} { \use:c {#1} #2 } { } }
1868 \cs_set:Npn \cs_if_exist_use:c #1
1869   { \cs_if_exist:cTF {#1} { \use:c {#1} } { } }

```

(End definition for `\cs_if_exist_use:NTF`. This function is documented on page 21.)

41.10 Preliminaries for new functions

We provide two kinds of functions that can be used to define control sequences. On the one hand we have functions that check if their argument doesn't already exist, they are called `\..._new`. The second type of defining functions doesn't check if the argument is already defined.

Before we can define them, we need some auxiliary macros that allow us to generate error messages. The next few definitions here are only temporary, they will be redefined later on.

`\msg_error:nxxx` If an internal error occurs before L^AT_EX3 has loaded l3msg then the code should issue a usable if terse error message and halt. This can only happen if a coding error is made by the team, so this is a reasonable response. Setting the `\newlinechar` is needed, to turn `^^J` into a proper line break in plain T_EX.

```

1870 \cs_set_protected:Npn \msg_error:nxxx #1#2#3#4
1871   {
1872     \tex_newlinechar:D = '\^^J \scan_stop:
1873     \tex_errmessage:D
1874     {
1875       !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!~! ^^J
1876       Argh,~internal~LaTeX3~error! ^^J ^^J
1877       Module ~ #1 , ~ message~name~"#2": ^^J
1878       Arguments~'#3'~and~'#4' ^^J ^^J
1879       This~is~one~for~The~LaTeX3~Project:~bailing~out

```

```

1880     }
1881     \tex_end:D
1882   }
1883   \cs_set_protected:Npn \msg_error:nnx #1#2#3
1884     { \msg_error:nxxx {#1} {#2} {#3} { } }
1885   \cs_set_protected:Npn \msg_error:nn #1#2
1886     { \msg_error:nxxx {#1} {#2} { } { } }

```

(End definition for `\msg_error:nxxx`, `\msg_error:nnx`, and `\msg_error:nn`. These functions are documented on page ??.)

`\msg_line_context:` Another one from `l3msg` which will be altered later.

```

1887 \cs_set:Npn \msg_line_context:
1888   { on~line~ \tex_the:D \tex_inputlineno:D }

```

(End definition for `\msg_line_context:`. This function is documented on page 78.)

`\iow_log:x` We define a routine to write only to the log file. And a similar one for writing to both
`\iow_term:x` the log file and the terminal. These will be redefined later by `l3io`.

```

1889 \cs_set_protected:Npn \iow_log:x
1890   { \tex_immediate:D \tex_write:D -1 }
1891 \cs_set_protected:Npn \iow_term:x
1892   { \tex_immediate:D \tex_write:D 16 }

```

(End definition for `\iow_log:n`. This function is documented on page 90.)

`__kernel_chk_if_free_cs:N` This command is called by `\cs_new_nopar:Npn` and `\cs_new_eq:NN` etc. to make sure
`__kernel_chk_if_free_cs:c` that the argument sequence is not already in use. If it is, an error is signalled. It checks
if `\csname` is undefined or `\scan_stop:`. Otherwise an error message is issued. We have
to make sure we don't put the argument into the conditional processing since it may be
an `\if...` type function!

```

1893 \cs_set_protected:Npn \__kernel_chk_if_free_cs:N #1
1894   {
1895     \cs_if_free:NF #1
1896     {
1897       \msg_error:nxxx { kernel } { command-already-defined }
1898       { \token_to_str:N #1 } { \token_to_meaning:N #1 }
1899     }
1900   }
1901 \cs_set_protected:Npn \__kernel_chk_if_free_cs:c
1902   { \exp_args:Nc \__kernel_chk_if_free_cs:N }

```

(End definition for `__kernel_chk_if_free_cs:N`.)

41.11 Defining new functions

```

1903 <@@=cs>

```

`\cs_new_nopar:Npn` Function which check that the control sequence is free before defining it.
`\cs_new_nopar:Npx`
`\cs_new:Npn`
`\cs_new:Npx`
`\cs_new_protected_nopar:Npn`
`\cs_new_protected_nopar:Npx`
`\cs_new_protected:Npn`
`\cs_new_protected:Npx`
`__cs_tmp:w`

```

1904 \cs_set:Npn \__cs_tmp:w #1#2
1905   {
1906     \cs_set_protected:Npn #1 ##1
1907     {
1908       \__kernel_chk_if_free_cs:N ##1

```

```

1909         #2 ##1
1910     }
1911 }
1912 \__cs_tmp:w \cs_new_nopar:Npn          \cs_gset_nopar:Npn
1913 \__cs_tmp:w \cs_new_nopar:Npx          \cs_gset_nopar:Npx
1914 \__cs_tmp:w \cs_new:Npn                \cs_gset:Npn
1915 \__cs_tmp:w \cs_new:Npx                \cs_gset:Npx
1916 \__cs_tmp:w \cs_new_protected_nopar:Npn \cs_gset_protected_nopar:Npn
1917 \__cs_tmp:w \cs_new_protected_nopar:Npx \cs_gset_protected_nopar:Npx
1918 \__cs_tmp:w \cs_new_protected:Npn      \cs_gset_protected:Npn
1919 \__cs_tmp:w \cs_new_protected:Npx      \cs_gset_protected:Npx

```

(End definition for `\cs_new_nopar:Npn` and others. These functions are documented on page 15.)

`\cs_set_nopar:cpn` Like `\cs_set_nopar:Npn` and `\cs_new_nopar:Npn`, except that the first argument consists of the sequence of characters that should be used to form the name of the desired control sequence (the `c` stands for `csname` argument, see the expansion module). Global versions are also provided.

`\cs_set_nopar:cpn` $\langle string \rangle \langle rep-text \rangle$ turns $\langle string \rangle$ into a `csname` and then assigns $\langle rep-text \rangle$ to it by using `\cs_set_nopar:Npn`. This means that there might be a parameter string between the two arguments.

```

1920 \cs_set:Npn \__cs_tmp:w #1#2
1921 { \cs_new_protected_nopar:Npn #1 { \exp_args:Nc #2 } }
1922 \__cs_tmp:w \cs_set_nopar:cpn \cs_set_nopar:Npn
1923 \__cs_tmp:w \cs_set_nopar:cpx \cs_set_nopar:Npx
1924 \__cs_tmp:w \cs_gset_nopar:cpn \cs_gset_nopar:Npn
1925 \__cs_tmp:w \cs_gset_nopar:cpx \cs_gset_nopar:Npx
1926 \__cs_tmp:w \cs_new_nopar:cpn \cs_new_nopar:Npn
1927 \__cs_tmp:w \cs_new_nopar:cpx \cs_new_nopar:Npx

```

(End definition for `\cs_set_nopar:Npn`. This function is documented on page 16.)

`\cs_set:cpn` Variants of the `\cs_set:Npn` versions which make a `csname` out of the first arguments.

`\cs_set:cpx` We may also do this globally.

`\cs_gset:cpn` `\cs_gset:cpn` `\cs_gset:cpn` `\cs_gset:cpn`

`\cs_gset:cpx` `\cs_gset:cpx` `\cs_gset:cpx` `\cs_gset:cpx`

`\cs_new:cpn` `\cs_new:cpn` `\cs_new:cpn` `\cs_new:cpn`

`\cs_new:cpx` `\cs_new:cpx` `\cs_new:cpx` `\cs_new:cpx`

(End definition for `\cs_set:Npn`. This function is documented on page 16.)

`\cs_set_protected_nopar:cpn` Variants of the `\cs_set_protected_nopar:Npn` versions which make a `csname` out of the first arguments. We may also do this globally.

`\cs_set_protected_nopar:cpx`

`\cs_gset_protected_nopar:cpn` `\cs_gset_protected_nopar:cpn` `\cs_gset_protected_nopar:cpn` `\cs_gset_protected_nopar:cpn`

`\cs_gset_protected_nopar:cpx` `\cs_gset_protected_nopar:cpx` `\cs_gset_protected_nopar:cpx` `\cs_gset_protected_nopar:cpx`

`\cs_new_protected_nopar:cpn` `\cs_new_protected_nopar:cpn` `\cs_new_protected_nopar:cpn` `\cs_new_protected_nopar:cpn`

`\cs_new_protected_nopar:cpx` `\cs_new_protected_nopar:cpx` `\cs_new_protected_nopar:cpx` `\cs_new_protected_nopar:cpx`

(End definition for `\cs_set_protected_nopar:Npn`. This function is documented on page 16.)

<code>\cs_set_protected:cpn</code>	1940	<code>__cs_tmp:w \cs_set_protected:cpn \cs_set_protected:Npn</code>
<code>\cs_set_protected:cpx</code>	1941	<code>__cs_tmp:w \cs_set_protected:cpx \cs_set_protected:Npx</code>
<code>\cs_gset_protected:cpn</code>	1942	<code>__cs_tmp:w \cs_gset_protected:cpn \cs_gset_protected:Npn</code>
<code>\cs_gset_protected:cpx</code>	1943	<code>__cs_tmp:w \cs_gset_protected:cpx \cs_gset_protected:Npx</code>
<code>\cs_new_protected:cpn</code>	1944	<code>__cs_tmp:w \cs_new_protected:cpn \cs_new_protected:Npn</code>
<code>\cs_new_protected:cpx</code>	1945	<code>__cs_tmp:w \cs_new_protected:cpx \cs_new_protected:Npx</code>

(End definition for `\cs_set_protected:Npn`. This function is documented on page 16.)

41.12 Copying definitions

<code>\cs_set_eq:NN</code>	These macros allow us to copy the definition of a control sequence to another control
<code>\cs_set_eq:cN</code>	sequence.
<code>\cs_set_eq:Nc</code>	The = sign allows us to define funny char tokens like = itself or <code>_</code> with this function.
<code>\cs_set_eq:cc</code>	For the definition of <code>\c_space_char{~}</code> to work we need the <code>~</code> after the =.
<code>\cs_gset_eq:NN</code>	<code>\cs_set_eq:NN</code> is long to avoid problems with a literal argument of <code>\par</code> . While
<code>\cs_gset_eq:cN</code>	<code>\cs_new_eq:NN</code> will probably never be correct with a first argument of <code>\par</code> , define it
<code>\cs_gset_eq:Nc</code>	long in order to throw an “already defined” error rather than “runaway argument”.
<code>\cs_gset_eq:cc</code>	1946 <code>\cs_new_protected:Npn \cs_set_eq:NN #1 { \tex_let:D #1 =~ }</code>
<code>\cs_new_eq:NN</code>	1947 <code>\cs_new_protected:Npn \cs_set_eq:cN { \exp_args:Nc \cs_set_eq:NN }</code>
<code>\cs_new_eq:cN</code>	1948 <code>\cs_new_protected:Npn \cs_set_eq:Nc { \exp_args:NNc \cs_set_eq:NN }</code>
<code>\cs_new_eq:Nc</code>	1949 <code>\cs_new_protected:Npn \cs_set_eq:cc { \exp_args:Ncc \cs_set_eq:NN }</code>
<code>\cs_new_eq:cc</code>	1950 <code>\cs_new_protected:Npn \cs_gset_eq:NN { \tex_global:D \cs_set_eq:NN }</code>
	1951 <code>\cs_new_protected:Npn \cs_gset_eq:Nc { \exp_args:NNc \cs_gset_eq:NN }</code>
	1952 <code>\cs_new_protected:Npn \cs_gset_eq:cN { \exp_args:Nc \cs_gset_eq:NN }</code>
	1953 <code>\cs_new_protected:Npn \cs_gset_eq:cc { \exp_args:Ncc \cs_gset_eq:NN }</code>
	1954 <code>\cs_new_protected:Npn \cs_new_eq:NN #1</code>
	1955 <code>{</code>
	1956 <code> __kernel_chk_if_free_cs:N #1</code>
	1957 <code> \tex_global:D \cs_set_eq:NN #1</code>
	1958 <code>}</code>
	1959 <code>\cs_new_protected:Npn \cs_new_eq:cN { \exp_args:Nc \cs_new_eq:NN }</code>
	1960 <code>\cs_new_protected:Npn \cs_new_eq:Nc { \exp_args:NNc \cs_new_eq:NN }</code>
	1961 <code>\cs_new_protected:Npn \cs_new_eq:cc { \exp_args:Ncc \cs_new_eq:NN }</code>

(End definition for `\cs_set_eq:NN`, `\cs_gset_eq:NN`, and `\cs_new_eq:NN`. These functions are documented on page 19.)

41.13 Undefining functions

<code>\cs_undefine:N</code>	The following function is used to free the main memory from the definition of some
<code>\cs_undefine:c</code>	function that isn’t in use any longer. The <code>c</code> variant is careful not to add the control
	sequence to the hash table if it isn’t there yet, and it also avoids nesting \TeX conditionals
	in case <code>#1</code> is unbalanced in this matter.

```

1962 \cs_new_protected:Npn \cs_undefine:N #1
1963 { \cs_gset_eq:NN #1 \tex_undefined:D }
1964 \cs_new_protected:Npn \cs_undefine:c #1
1965 {
1966   \if_cs_exist:w #1 \cs_end:

```



```

1967     \exp_after:wN \use:n
1968 \else:
1969     \exp_after:wN \use_none:n
1970 \fi:
1971 { \cs_gset_eq:cN {#1} \tex_undefined:D }
1972 }

```

(End definition for `\cs_undefine:N`. This function is documented on page 20.)

41.14 Generating parameter text from argument count

```

1973 <@@=cs>

```

```

\__kernel_cs_parm_from_arg_count:nnF
\__cs_parm_from_arg_count_test:nnF

```

L^AT_EX3 provides shorthands to define control sequences and conditionals with a simple parameter text, derived directly from the signature, or more generally from knowing the number of arguments, between 0 and 9. This function expands to its first argument, untouched, followed by a brace group containing the parameter text `{#1...#n}`, where n is the result of evaluating the second argument (as described in `\int_eval:n`). If the second argument gives a result outside the range $[0, 9]$, the third argument is returned instead, normally an error message. Some of the functions use here are not defined yet, but will be defined before this function is called.

```

1974 \cs_set_protected:Npn \__kernel_cs_parm_from_arg_count:nnF #1#2
1975 {
1976   \exp_args:Nx \__cs_parm_from_arg_count_test:nnF
1977   {
1978     \exp_after:wN \exp_not:n
1979     \if_case:w \int_eval:n {#2}
1980     { }
1981     \or: { ##1 }
1982     \or: { ##1##2 }
1983     \or: { ##1##2##3 }
1984     \or: { ##1##2##3##4 }
1985     \or: { ##1##2##3##4##5 }
1986     \or: { ##1##2##3##4##5##6 }
1987     \or: { ##1##2##3##4##5##6##7 }
1988     \or: { ##1##2##3##4##5##6##7##8 }
1989     \or: { ##1##2##3##4##5##6##7##8##9 }
1990     \else: { \c_false_bool }
1991     \fi:
1992   }
1993   {#1}
1994 }
1995 \cs_set_protected:Npn \__cs_parm_from_arg_count_test:nnF #1#2
1996 {
1997   \if_meaning:w \c_false_bool #1
1998   \exp_after:wN \use_ii:nn
1999 \else:
2000   \exp_after:wN \use_i:nn
2001 \fi:
2002 { #2 {#1} }
2003 }

```

(End definition for `__kernel_cs_parm_from_arg_count:nnF` and `__cs_parm_from_arg_count_test:nnF`.)

41.15 Defining functions from a given number of arguments

2004 `<@@=cs>`

`__cs_count_signature:N` Counting the number of tokens in the signature, *i.e.*, the number of arguments the function should take. Since this is not used in any time-critical function, we simply use `__cs_count_signature:c` `__cs_count_signature:n` `__cs_count_signature:nnN` `\tl_count:n` if there is a signature, otherwise `-1` arguments to signal an error. We need a variant form right away.

```
2005 \cs_new:Npn \__cs_count_signature:N #1
2006   { \exp_args:Nf \__cs_count_signature:n { \cs_split_function:N #1 } }
2007 \cs_new:Npn \__cs_count_signature:n #1
2008   { \int_eval:n { \__cs_count_signature:nnN #1 } }
2009 \cs_new:Npn \__cs_count_signature:nnN #1#2#3
2010   {
2011     \if_meaning:w \c_true_bool #3
2012       \tl_count:n {#2}
2013     \else:
2014       -1
2015     \fi:
2016   }
2017 \cs_new:Npn \__cs_count_signature:c
2018   { \exp_args:Nc \__cs_count_signature:N }
```

(End definition for `__cs_count_signature:N`, `__cs_count_signature:n`, and `__cs_count_signature:nnN`.)

`\cs_generate_from_arg_count:NNnn`
`\cs_generate_from_arg_count:cNnn`
`\cs_generate_from_arg_count:Ncnn`

We provide a constructor function for defining functions with a given number of arguments. For this we need to choose the correct parameter text and then use that when defining. Since TeX supports from zero to nine arguments, we use a simple switch to choose the correct parameter text, ensuring the result is returned after finishing the conditional. If it is not between zero and nine, we throw an error.

1: function to define, 2: with what to define it, 3: the number of args it requires and 4: the replacement text

```
2019 \cs_new_protected:Npn \cs_generate_from_arg_count:NNnn #1#2#3#4
2020   {
2021     \__kernel_cs_parm_from_arg_count:nnF { \use:nnn #2 #1 } {#3}
2022     {
2023       \msg_error:nnxx { kernel } { bad-number-of-arguments }
2024       { \token_to_str:N #1 } { \int_eval:n {#3} }
2025       \use_none:n
2026     }
2027     {#4}
2028   }
```

A variant form we need right away, plus one which is used elsewhere but which is most logically created here.

```
2029 \cs_new_protected:Npn \cs_generate_from_arg_count:cNnn
2030   { \exp_args:Nc \cs_generate_from_arg_count:NNnn }
2031 \cs_new_protected:Npn \cs_generate_from_arg_count:Ncnn
2032   { \exp_args:NNc \cs_generate_from_arg_count:NNnn }
```

(End definition for `\cs_generate_from_arg_count:NNnn`. This function is documented on page 19.)

41.16 Using the signature to define functions

2033 <@@=cs>

We can now combine some of the tools we have to provide a simple interface for defining functions, where the number of arguments is read from the signature. For instance, `\cs_set:Nn \foo_bar:nn {#1,#2}`.

We want to define `\cs_set:Nn` as

```
\cs_set:Nn
\cs_set:Nx
\cs_set_nopar:Nn
\cs_set_nopar:Nx
\cs_set_protected:Nn
\cs_set_protected:Nx
\cs_set_protected_nopar:Nn
\cs_set_protected_nopar:Nx
\cs_gset:Nn
\cs_gset:Nx
\cs_gset_nopar:Nn
\cs_gset_nopar:Nx
\cs_gset_protected:Nn
\cs_gset_protected:Nx
\cs_gset_protected_nopar:Nn
\cs_gset_protected_nopar:Nx
\cs_new:Nn
\cs_new:Nx
\cs_new_nopar:Nn
\cs_new_nopar:Nx
\cs_new_protected:Nn
\cs_new_protected:Nx
\cs_new_protected_nopar:Nn
\cs_new_protected_nopar:Nx
```

In short, to define `\cs_set:Nn` we need just use `\cs_set:Npn`, everything else is the same for each variant. Therefore, we can make it simpler by temporarily defining a function to do this for us.

```
2034 \cs_set:Npn \__cs_tmp:w #1#2#3
2035 {
2036   \cs_new_protected:cpx { cs_ #1 : #2 }
2037   {
2038     \exp_not:N \__cs_generate_from_signature:NNn
2039     \exp_after:wN \exp_not:N \cs:w cs_ #1 : #3 \cs_end:
2040   }
2041 }
2042 \cs_new_protected:Npn \__cs_generate_from_signature:NNn #1#2
2043 {
2044   \use:x
2045   {
2046     \__cs_generate_from_signature:nnNNNn
2047     \cs_split_function:N #2
2048   }
2049   #1 #2
2050 }
2051 \cs_new_protected:Npn \__cs_generate_from_signature:nnNNNn #1#2#3#4#5#6
2052 {
2053   \bool_if:NTF #3
2054   {
2055     \cs_set_nopar:Npx \__cs_tmp:w
2056     { \tl_map_function:nN {#2} \__cs_generate_from_signature:n }
2057     \tl_if_empty:oF \__cs_tmp:w
2058     {
2059       \msg_error:nnxxx { kernel } { non-base-function }
2060       { \token_to_str:N #5 } {#2} { \__cs_tmp:w }
2061     }
2062     \cs_generate_from_arg_count:NNnn
2063     #5 #4 { \tl_count:n {#2} } {#6}
2064   }
2065   {
2066     \msg_error:nnx { kernel } { missing-colon }
2067     { \token_to_str:N #5 }
2068   }
2069 }
```

```

2070 \cs_new:Npn \__cs_generate_from_signature:n #1
2071 {
2072   \if:w n #1 \else: \if:w N #1 \else:
2073     \if:w T #1 \else: \if:w F #1 \else: #1 \fi: \fi: \fi: \fi:
2074 }

```

Then we define the 24 variants beginning with N.

```

2075 \__cs_tmp:w { set } { Nn } { Npn }
2076 \__cs_tmp:w { set } { Nx } { Npx }
2077 \__cs_tmp:w { set_nopar } { Nn } { Npn }
2078 \__cs_tmp:w { set_nopar } { Nx } { Npx }
2079 \__cs_tmp:w { set_protected } { Nn } { Npn }
2080 \__cs_tmp:w { set_protected } { Nx } { Npx }
2081 \__cs_tmp:w { set_protected_nopar } { Nn } { Npn }
2082 \__cs_tmp:w { set_protected_nopar } { Nx } { Npx }
2083 \__cs_tmp:w { gset } { Nn } { Npn }
2084 \__cs_tmp:w { gset } { Nx } { Npx }
2085 \__cs_tmp:w { gset_nopar } { Nn } { Npn }
2086 \__cs_tmp:w { gset_nopar } { Nx } { Npx }
2087 \__cs_tmp:w { gset_protected } { Nn } { Npn }
2088 \__cs_tmp:w { gset_protected } { Nx } { Npx }
2089 \__cs_tmp:w { gset_protected_nopar } { Nn } { Npn }
2090 \__cs_tmp:w { gset_protected_nopar } { Nx } { Npx }
2091 \__cs_tmp:w { new } { Nn } { Npn }
2092 \__cs_tmp:w { new } { Nx } { Npx }
2093 \__cs_tmp:w { new_nopar } { Nn } { Npn }
2094 \__cs_tmp:w { new_nopar } { Nx } { Npx }
2095 \__cs_tmp:w { new_protected } { Nn } { Npn }
2096 \__cs_tmp:w { new_protected } { Nx } { Npx }
2097 \__cs_tmp:w { new_protected_nopar } { Nn } { Npn }
2098 \__cs_tmp:w { new_protected_nopar } { Nx } { Npx }

```

(End definition for \cs_set:Nn and others. These functions are documented on page 17.)

\cs_set:cn The 24 c variants simply use \exp_args:Nc.

```

\cs_set:cx 2099 \cs_set:Npn \__cs_tmp:w #1#2
\cs_set_nopar:cn 2100 {
\cs_set_nopar:cx 2101   \cs_new_protected:cpx { cs_ #1 : c #2 }
\cs_set_protected:cn 2102   {
\cs_set_protected:cx 2103     \exp_not:N \exp_args:Nc
\cs_set_protected_nopar:cn 2104     \exp_after:wN \exp_not:N \cs:w cs_ #1 : N #2 \cs_end:
\cs_set_protected_nopar:cx 2105   }
2106 }
\cs_gset:cn 2107 \__cs_tmp:w { set } { n }
\cs_gset:cx 2108 \__cs_tmp:w { set } { x }
\cs_gset_nopar:cn 2109 \__cs_tmp:w { set_nopar } { n }
\cs_gset_nopar:cx 2110 \__cs_tmp:w { set_nopar } { x }
\cs_gset_protected:cn 2111 \__cs_tmp:w { set_protected } { n }
\cs_gset_protected:cx 2112 \__cs_tmp:w { set_protected } { x }
\cs_gset_protected_nopar:cn 2113 \__cs_tmp:w { set_protected_nopar } { n }
\cs_gset_protected_nopar:cx 2114 \__cs_tmp:w { set_protected_nopar } { x }
\cs_new:cn 2115 \__cs_tmp:w { gset } { n }
\cs_new:cx 2116 \__cs_tmp:w { gset } { x }
\cs_new_nopar:cn 2117 \__cs_tmp:w { gset_nopar } { n }
\cs_new_nopar:cx 2118 \__cs_tmp:w { gset_nopar } { x }
\cs_new_protected:cn
\cs_new_protected:cx
\cs_new_protected_nopar:cn
\cs_new_protected_nopar:cx

```

```

2119 \__cs_tmp:w { gset_protected } { n }
2120 \__cs_tmp:w { gset_protected } { x }
2121 \__cs_tmp:w { gset_protected_nopar } { n }
2122 \__cs_tmp:w { gset_protected_nopar } { x }
2123 \__cs_tmp:w { new } { n }
2124 \__cs_tmp:w { new } { x }
2125 \__cs_tmp:w { new_nopar } { n }
2126 \__cs_tmp:w { new_nopar } { x }
2127 \__cs_tmp:w { new_protected } { n }
2128 \__cs_tmp:w { new_protected } { x }
2129 \__cs_tmp:w { new_protected_nopar } { n }
2130 \__cs_tmp:w { new_protected_nopar } { x }

```

(End definition for \cs_set:Nn. This function is documented on page 17.)

41.17 Checking control sequence equality

\cs_if_eq_p:NN Check if two control sequences are identical.

```

\cs_if_eq_p:cN 2131 \prg_new_conditional:Npnn \cs_if_eq:NN #1#2 { p , T , F , TF }
\cs_if_eq_p:Nc 2132 {
\cs_if_eq_p:cc 2133   \if_meaning:w #1#2
\cs_if_eq:NNTF 2134   \prg_return_true: \else: \prg_return_false: \fi:
\cs_if_eq:cNTF 2135 }
\cs_if_eq:NcTF 2136 \cs_new:Npn \cs_if_eq_p:cN { \exp_args:Nc \cs_if_eq_p:NN }
\cs_if_eq:NcTF 2137 \cs_new:Npn \cs_if_eq:cNTF { \exp_args:Nc \cs_if_eq:NNTF }
\cs_if_eq:ccTF 2138 \cs_new:Npn \cs_if_eq:cNT { \exp_args:Nc \cs_if_eq:NNT }
2139 \cs_new:Npn \cs_if_eq:cNF { \exp_args:Nc \cs_if_eq:NNF }
2140 \cs_new:Npn \cs_if_eq_p:Nc { \exp_args:NNc \cs_if_eq_p:NN }
2141 \cs_new:Npn \cs_if_eq:NcTF { \exp_args:NNc \cs_if_eq:NNTF }
2142 \cs_new:Npn \cs_if_eq:NcT { \exp_args:NNc \cs_if_eq:NNT }
2143 \cs_new:Npn \cs_if_eq:NcF { \exp_args:NNc \cs_if_eq:NNF }
2144 \cs_new:Npn \cs_if_eq_p:cc { \exp_args:Ncc \cs_if_eq_p:NN }
2145 \cs_new:Npn \cs_if_eq:ccTF { \exp_args:Ncc \cs_if_eq:NNTF }
2146 \cs_new:Npn \cs_if_eq:ccT { \exp_args:Ncc \cs_if_eq:NNT }
2147 \cs_new:Npn \cs_if_eq:ccF { \exp_args:Ncc \cs_if_eq:NNF }

```

(End definition for \cs_if_eq:NNTF. This function is documented on page 27.)

41.18 Diagnostic functions

```

2148 <@@=kernel>
\__kernel_chk_defined:NT Error if the variable #1 is not defined.
2149 \cs_new_protected:Npn \__kernel_chk_defined:NT #1#2
2150 {
2151   \cs_if_exist:NTF #1
2152   {#2}
2153   {
2154     \msg_error:nnx { kernel } { variable-not-defined }
2155     { \token_to_str:N #1 }
2156   }
2157 }

```

(End definition for `_kernel_chk_defined:NT`.)

`_kernel_register_show:N`
`_kernel_register_show:c`
`_kernel_register_log:N`
`_kernel_register_log:c`
`_kernel_register_show_aux:NN`
`_kernel_register_show_aux:nNN`

Simply using the `\show` primitive does not allow for line-wrapping, so instead use `\tl_show:n` and `\tl_log:n` (defined in `l3tl` and that performs line-wrapping). This displays `>~⟨variable⟩=⟨value⟩`. We expand the value before-hand as otherwise some integers (such as `\currentgrouplevel` or `\currentgrouptype`) altered by the line-wrapping code would show wrong values.

```

2158 \cs_new_protected:Npn \_kernel_register_show:N
2159   { \_kernel_register_show_aux:NN \tl_show:n }
2160 \cs_new_protected:Npn \_kernel_register_show:c
2161   { \exp_args:Nc \_kernel_register_show:N }
2162 \cs_new_protected:Npn \_kernel_register_log:N
2163   { \_kernel_register_show_aux:NN \tl_log:n }
2164 \cs_new_protected:Npn \_kernel_register_log:c
2165   { \exp_args:Nc \_kernel_register_log:N }
2166 \cs_new_protected:Npn \_kernel_register_show_aux:NN #1#2
2167   {
2168     \_kernel_chk_defined:NT #2
2169     {
2170       \exp_args:No \_kernel_register_show_aux:nNN
2171       { \tex_the:D #2 } #2 #1
2172     }
2173   }
2174 \cs_new_protected:Npn \_kernel_register_show_aux:nNN #1#2#3
2175   { \exp_args:No #3 { \token_to_str:N #2 = #1 } }

```

(End definition for `_kernel_register_show:N` and others.)

`\cs_show:N`
`\cs_show:c`
`\cs_log:N`
`\cs_log:c`
`_kernel_show:NN`

Some control sequences have a very long name or meaning. Thus, simply using TeX's primitive `\show` could lead to overlong lines. The output of this primitive is mimicked to some extent, then the re-built string is given to `\tl_show:n` or `\tl_log:n` for line-wrapping. We must expand the meaning before passing it to the wrapping code as otherwise we would wrongly see the definitions that are in place there. To get correct escape characters, set the `\escapechar` in a group; this also localizes the assignment performed by x-expansion. The `\cs_show:c` and `\cs_log:c` commands convert their argument to a control sequence within a group to avoid showing `\relax` for undefined control sequences.

```

2176 \cs_new_protected:Npn \cs_show:N { \_kernel_show:NN \tl_show:n }
2177 \cs_new_protected:Npn \cs_show:c
2178   { \group_begin: \exp_args:NNc \group_end: \cs_show:N }
2179 \cs_new_protected:Npn \cs_log:N { \_kernel_show:NN \tl_log:n }
2180 \cs_new_protected:Npn \cs_log:c
2181   { \group_begin: \exp_args:NNc \group_end: \cs_log:N }
2182 \cs_new_protected:Npn \_kernel_show:NN #1#2
2183   {
2184     \group_begin:
2185     \int_set:Nn \tex_escapechar:D { '\ }
2186     \exp_args:NNx
2187     \group_end:
2188     #1 { \token_to_str:N #2 = \cs_meaning:N #2 }
2189   }

```

(End definition for `\cs_show:N`, `\cs_log:N`, and `_kernel_show:NN`. These functions are documented on page 20.)

`\group_show_list:` Wrapper around `\showgroups`. Getting TeX to write to the log without interruption the run is done by altering the interaction mode.

`\group_log_list:`

`__kernel_group_show:NN`

```

2190 \cs_new_protected:Npn \group_show_list:
2191 { \__kernel_group_show:NN \use_none:n 1 }
2192 \cs_new_protected:Npn \group_log_list:
2193 { \__kernel_group_show:NN \int_zero:N 0 }
2194 \cs_new_protected:Npn \__kernel_group_show:NN #1#2
2195 {
2196   \use:x
2197   {
2198     #1 \tex_interactionmode:D
2199     \int_set:Nn \tex_tracingonline:D {#2}
2200     \int_set:Nn \tex_errorcontextlines:D { -1 }
2201     \exp_not:N \exp_after:wN \scan_stop:
2202     \tex_showgroups:D
2203     \int_set:Nn \tex_interactionmode:D
2204     { \int_use:N \tex_interactionmode:D }
2205     \int_set:Nn \tex_tracingonline:D
2206     { \int_use:N \tex_tracingonline:D }
2207     \int_set:Nn \tex_errorcontextlines:D
2208     { \int_use:N \tex_errorcontextlines:D }
2209   }
2210 }

```

(End definition for `\group_show_list:`, `\group_log_list:`, and `__kernel_group_show:NN`. These functions are documented on page 14.)

41.19 Decomposing a macro definition

`\cs_prefix_spec:N` We sometimes want to test if a control sequence can be expanded to reveal a hidden value.

`\cs_argument_spec:N` However, we cannot just expand the macro blindly as it may have arguments and none might be present. Therefore we define these functions to pick either the prefix(es), the

`\cs_replacement_spec:N` argument specification, or the replacement text from a macro. All of this information is returned as characters with catcode 12. If the token in question isn't a macro, the token

`__kernel_prefix_arg_replacement:wN` `\scan_stop:` is returned instead.

```

2211 \use:x
2212 {
2213   \exp_not:n { \cs_new:Npn \__kernel_prefix_arg_replacement:wN #1 }
2214   \tl_to_str:n { macro : } \exp_not:n { #2 -> #3 \s__kernel_stop #4 }
2215 }
2216 { #4 {#1} {#2} {#3} }
2217 \cs_new:Npn \cs_prefix_spec:N #1
2218 {
2219   \token_if_macro:NTF #1
2220   {
2221     \exp_after:wN \__kernel_prefix_arg_replacement:wN
2222     \token_to_meaning:N #1 \s__kernel_stop \use_i:nnn
2223   }
2224   { \scan_stop: }
2225 }
2226 \cs_new:Npn \cs_argument_spec:N #1
2227 {
2228   \token_if_macro:NTF #1

```

```

2229     {
2230         \exp_after:wN \_kernel_prefix_arg_replacement:wN
2231         \token_to_meaning:N #1 \s__kernel_stop \use_ii:nnn
2232     }
2233     { \scan_stop: }
2234 }
2235 \cs_new:Npn \cs_replacement_spec:N #1
2236 {
2237     \token_if_macro:NTF #1
2238     {
2239         \exp_after:wN \_kernel_prefix_arg_replacement:wN
2240         \token_to_meaning:N #1 \s__kernel_stop \use_iii:nnn
2241     }
2242     { \scan_stop: }
2243 }

```

(End definition for `\cs_prefix_spec:N` and others. These functions are documented on page 22.)

41.20 Doing nothing functions

`\prg_do_nothing:` This does not fit anywhere else!

```

2244 \cs_new:Npn \prg_do_nothing: { }

```

(End definition for `\prg_do_nothing:`. This function is documented on page 13.)

41.21 Breaking out of mapping functions

```

2245 <@@=prg>

```

`\prg_break_point:Nn` In inline mappings, the nesting level must be reset at the end of the mapping, even when the user decides to break out. This is done by putting the code that must be performed as an argument of `_prg_break_point:Nn`. The breaking functions are then defined to jump to that point and perform the argument of `_prg_break_point:Nn`, before the user's code (if any). There is a check that we close the correct loop, otherwise we continue breaking.

`\prg_map_break:Nn`

```

2246 \cs_new_eq:NN \prg_break_point:Nn \use_ii:nn
2247 \cs_new:Npn \prg_map_break:Nn #1#2#3 \prg_break_point:Nn #4#5
2248 {
2249     #5
2250     \if_meaning:w #1 #4
2251     \exp_after:wN \use_iii:nnn
2252     \fi:
2253     \prg_map_break:Nn #1 {#2}
2254 }

```

(End definition for `\prg_break_point:Nn` and `\prg_map_break:Nn`. These functions are documented on page 70.)

`\prg_break_point:` Very simple analogues of `\prg_break_point:Nn` and `\prg_map_break:Nn`, for use in fast short-term recursions which are not mappings, do not need to support nesting, and in which nothing has to be done at the end of the loop.

`\prg_break:`

`\prg_break:n`

```

2255 \cs_new_eq:NN \prg_break_point: \prg_do_nothing:
2256 \cs_new:Npn \prg_break: #1 \prg_break_point: { }
2257 \cs_new:Npn \prg_break:n #1#2 \prg_break_point: {#1}

```


(End definition for `\prg_break_point:`, `\prg_break:`, and `\prg_break:n`. These functions are documented on page 71.)

41.22 Starting a paragraph

`\mode_leave_vertical:` The approach here is different to that used by L^AT_EX 2_ε or plain T_EX, which unbox a void box to force horizontal mode. That inserts the `\everypar` tokens *before* the re-inserted unboxing tokens. The approach here uses either the `\quitvmode` primitive or the equivalent protected macro. In vertical mode, the `\indent` primitive is inserted: this will switch to horizontal mode and insert `\everypar` tokens and nothing else. Unlike the L^AT_EX 2_ε version, the availability of ε-T_EX means using a mode test can be done at for example the start of an `\halign`.

```

2258 \cs_new_protected:Npn \mode_leave_vertical:
2259 {
2260   \if_mode_vertical:
2261     \exp_after:wN \tex_indent:D
2262   \fi:
2263 }
```

(End definition for `\mode_leave_vertical:`. This function is documented on page 28.)

```

2264 </package>
```

Chapter 42

l3expan implementation

```
2265 \*package>
2266 \@@=exp>

\l__exp_internal_tl The \exp_ module has its private variable to temporarily store the result of x-type argu-
ment expansion. This is done to avoid interference with other functions using temporary
variables.

(End definition for \l__exp_internal_tl.)

\exp_after:wN These are defined in l3basics, as they are needed “early”. This is just a reminder of that
\exp_not:N fact!
\exp_not:n (End definition for \exp_after:wN, \exp_not:N, and \exp_not:n. These functions are documented on
page 38.)
```

42.1 General expansion

In this section a general mechanism for defining functions that handle arguments is defined. These general expansion functions are expandable unless `x` is used. (Any version of `x` is going to have to use one of the L^AT_EX3 names for `\cs_set:Npx` at some point, and so is never going to be expandable.)

The definition of expansion functions with this technique happens in section 42.8. In section 42.2 some common cases are coded by a more direct method for efficiency, typically using calls to `\exp_after:wN`.

```
\l__exp_internal_tl This scratch token list variable is defined in l3basics.

(End definition for \l__exp_internal_tl.)
```

This code uses internal functions with names that start with `\::` to perform the expansions. All macros are `long` since the tokens undergoing expansion may be arbitrary user input.

An argument manipulator `\::<Z>` always has signature `#1\:::#2#3` where `#1` holds the remaining argument manipulations to be performed, `\:::` serves as an end marker for the list of manipulations, `#2` is the carried over result of the previous expansion steps and `#3` is the argument about to be processed. One exception to this rule is `\::p`, which has to grab an argument delimited by a left brace.

`__exp_arg_next:nnn` #1 is the result of an expansion step, #2 is the remaining argument manipulations and
`__exp_arg_next:Nnn` #3 is the current result of the expansion chain. This auxiliary function moves #1 back
after #3 in the input stream and checks if any expansion is left to be done by calling
#2. In by far the most cases we need to add a set of braces to the result of an argument
manipulation so it is more effective to do it directly here. Actually, so far only the `c` of
the final argument manipulation variants does not require a set of braces.

```
2267 \cs_new:Npn \__exp_arg_next:nnn #1#2#3 { #2 \::: { #3 {#1} } }
2268 \cs_new:Npn \__exp_arg_next:Nnn #1#2#3 { #2 \::: { #3 #1 } }
```

(End definition for `__exp_arg_next:nnn` and `__exp_arg_next:Nnn`.)

`\:::` The end marker is just another name for the identity function.

```
2269 \cs_new:Npn \::: #1 {#1}
```

(End definition for `\:::`. This function is documented on page 42.)

`\::n` This function is used to skip an argument that doesn't need to be expanded.

```
2270 \cs_new:Npn \::n #1 \::: #2#3 { #1 \::: { #2 {#3} } }
```

(End definition for `\::n`. This function is documented on page 42.)

`\::N` This function is used to skip an argument that consists of a single token and doesn't need
to be expanded.

```
2271 \cs_new:Npn \::N #1 \::: #2#3 { #1 \::: {#2#3} }
```

(End definition for `\::N`. This function is documented on page 42.)

`\::p` This function is used to skip an argument that is delimited by a left brace and doesn't
need to be expanded. It is not wrapped in braces in the result.

```
2272 \cs_new:Npn \::p #1 \::: #2#3# { #1 \::: {#2#3} }
```

(End definition for `\::p`. This function is documented on page 42.)

`\::c` This function is used to skip an argument that is turned into a control sequence without
expansion.

```
2273 \cs_new:Npn \::c #1 \::: #2#3
2274 { \exp_after:wN \__exp_arg_next:Nnn \cs:w #3 \cs_end: {#1} {#2} }
```

(End definition for `\::c`. This function is documented on page 42.)

`\::o` This function is used to expand an argument once.

```
2275 \cs_new:Npn \::o #1 \::: #2#3
2276 { \exp_after:wN \__exp_arg_next:nnn \exp_after:wN {#3} {#1} {#2} }
```

(End definition for `\::o`. This function is documented on page 42.)

`\::e` With the `\expanded` primitive available, just expand. Otherwise defer to `\exp_args:Ne`
implemented later.

```
2277 \cs_if_exist:NTF \tex_expanded:D
2278 {
2279   \cs_new:Npn \::e #1 \::: #2#3
2280   { \tex_expanded:D { \exp_not:n { #1 \::: } { \exp_not:n {#2} {#3} } } }
2281 }
2282 {
2283   \cs_new:Npn \::e #1 \::: #2#3
2284   { \exp_args:Ne \__exp_arg_next:nnn {#3} {#1} {#2} }
2285 }
```

(End definition for `\::e`. This function is documented on page 42.)

`\::f` This function is used to expand a token list until the first unexpandable token is found. This is achieved through `\exp:w \exp_end_continue_f:w` that expands everything in its way following it. This scanning procedure is terminated once the expansion hits something non-expandable (if that is a space it is removed). We introduce `\exp_stop_f:` to mark such an end-of-expansion marker. For example, `f`-expanding `\cs_set_eq:Nc \aaa { b \l_tmpa_tl b }` where `\l_tmpa_tl` contains the characters `lur` gives `\tex_let:D \aaa = \blurb` which then turns out to start with the non-expandable token `\tex_let:D`. Since the expansion of `\exp:w \exp_end_continue_f:w` is empty, we wind up with a fully expanded list, only TeX has not tried to execute any of the non-expandable tokens. This is what differentiates this function from the `x` argument type.

```

2286 \cs_new:Npn \::f #1 \::: #2#3
2287 {
2288   \exp_after:wN \__exp_arg_next:nnn
2289   \exp_after:wN { \exp:w \exp_end_continue_f:w #3 }
2290   {#1} {#2}
2291 }
2292 \use:nn { \cs_new_eq:NN \exp_stop_f: } { ~ }
```

(End definition for `\::f` and `\exp_stop_f:`. These functions are documented on page 42.)

`\::x` This function is used to expand an argument fully. We build in the expansion of `__exp_arg_next:nnn`.

```

2293 \cs_new_protected:Npn \::x #1 \::: #2#3
2294 {
2295   \cs_set_nopar:Npx \l__exp_internal_tl
2296   { \exp_not:n { #1 \::: } { \exp_not:n {#2} {#3} } }
2297   \l__exp_internal_tl
2298 }
```

(End definition for `\::x`. This function is documented on page 42.)

`\::v` These functions return the value of a register, i.e., one of `tl`, `clist`, `int`, `skip`, `dim`, `muskip`, or built-in TeX register. The `V` version expects a single token whereas `v` like `c` creates a cname from its argument given in braces and then evaluates it as if it was a `V`. The `\exp:w` sets off an expansion similar to an `f`-type expansion, which we terminate using `\exp_end:`. The argument is returned in braces.

```

2299 \cs_new:Npn \::V #1 \::: #2#3
2300 {
2301   \exp_after:wN \__exp_arg_next:nnn
2302   \exp_after:wN { \exp:w \__exp_eval_register:N #3 }
2303   {#1} {#2}
2304 }
2305 \cs_new:Npn \::v #1 \::: #2#3
2306 {
2307   \exp_after:wN \__exp_arg_next:nnn
2308   \exp_after:wN { \exp:w \__exp_eval_register:c {#3} }
2309   {#1} {#2}
2310 }
```

(End definition for `\::v` and `\::V`. These functions are documented on page 42.)

`__exp_eval_register:N` This function evaluates a register. Now a register might exist as one of two things: A parameter-less macro or a built-in \TeX register such as `\count`. For the \TeX registers we have to utilize a `\the` whereas for the macros we merely have to expand them once. The trick is to find out when to use `\the` and when not to. What we want here is to find out whether the token expands to something else when hit with `\exp_after:wN`. The technique is to compare the meaning of the token in question when it has been prefixed with `\exp_not:N` and the token itself. If it is a macro, the prefixed `\exp_not:N` temporarily turns it into the primitive `\scan_stop:.`

```

2311 \cs_new:Npn \__exp_eval_register:N #1
2312 {
2313   \exp_after:wN \if_meaning:w \exp_not:N #1 #1

```

If the token was not a macro it may be a malformed variable from a `c` expansion in which case it is equal to the primitive `\scan_stop:.` In that case we throw an error. We could let \TeX do it for us but that would result in the rather obscure

! You can't use '`\relax`' after `\the`.

which while quite true doesn't give many hints as to what actually went wrong. We provide something more sensible.

```

2314   \if_meaning:w \scan_stop: #1
2315   \__exp_eval_error_msg:w
2316   \fi:

```

The next bit requires some explanation. The function must be initiated by `\exp:w` and we want to terminate this expansion chain by inserting the `\exp_end:` token. However, we have to expand the register `#1` before we do that. If it is a \TeX register, we need to execute the sequence `\exp_after:wN \exp_end: \tex_the:D #1` and if it is a macro we need to execute `\exp_after:wN \exp_end: #1`. We therefore issue the longer of the two sequences and if the register is a macro, we remove the `\tex_the:D`.

```

2317   \else:
2318     \exp_after:wN \use_i_ii:nnn
2319   \fi:
2320   \exp_after:wN \exp_end: \tex_the:D #1
2321 }
2322 \cs_new:Npn \__exp_eval_register:c #1
2323 { \exp_after:wN \__exp_eval_register:N \cs:w #1 \cs_end: }

```

Clean up nicely, then call the undefined control sequence. The result is an error message looking like this:

```

! Undefined control sequence.
<argument> \LaTeX3 error:
                               Erroneous variable used!
1.55 \tl_set:Nv \l_tmpa_tl {undefined_tl}

```

```

2324 \cs_new:Npn \__exp_eval_error_msg:w #1 \tex_the:D #2
2325 {
2326   \fi:
2327   \fi:
2328   \msg_expandable_error:nnn { kernel } { bad-variable } {#2}
2329   \exp_end:
2330 }

```

(End definition for `__exp_eval_register:N` and `__exp_eval_error_msg:w`.)

42.2 Hand-tuned definitions

One of the most important features of these functions is that they are fully expandable.

`\exp_args:Nc` In l3basics.

`\exp_args:cc`

(End definition for `\exp_args:Nc` and `\exp_args:cc`. These functions are documented on page 34.)

`\exp_args:NNc`

`\exp_args:Ncc`

`\exp_args:Nccc`

Here are the functions that turn their argument into csnames but are expandable.

```

2331 \cs_new:Npn \exp_args:NNc #1#2#3
2332 { \exp_after:wN #1 \exp_after:wN #2 \cs:w # 3\cs_end: }
2333 \cs_new:Npn \exp_args:Ncc #1#2#3
2334 { \exp_after:wN #1 \cs:w #2 \exp_after:wN \cs_end: \cs:w #3 \cs_end: }
2335 \cs_new:Npn \exp_args:Nccc #1#2#3#4
2336 {
2337   \exp_after:wN #1
2338   \cs:w #2 \exp_after:wN \cs_end:
2339   \cs:w #3 \exp_after:wN \cs_end:
2340   \cs:w #4 \cs_end:
2341 }
```

(End definition for `\exp_args:NNc`, `\exp_args:Ncc`, and `\exp_args:Nccc`. These functions are documented on page 36.)

`\exp_args:No`

`\exp_args:NNo`

`\exp_args:NNNo`

Those lovely runs of expansion!

```

2342 \cs_new:Npn \exp_args:No #1#2 { \exp_after:wN #1 \exp_after:wN {#2} }
2343 \cs_new:Npn \exp_args:NNo #1#2#3
2344 { \exp_after:wN #1 \exp_after:wN #2 \exp_after:wN {#3} }
2345 \cs_new:Npn \exp_args:NNNo #1#2#3#4
2346 { \exp_after:wN #1 \exp_after:wN#2 \exp_after:wN #3 \exp_after:wN {#4} }
```

(End definition for `\exp_args:No`, `\exp_args:NNo`, and `\exp_args:NNNo`. These functions are documented on page 35.)

`\exp_args:Ne`

When the `\expanded` primitive is available, use it. Otherwise use `__exp_e:nn`, defined later, to fully expand tokens.

```

2347 \cs_if_exist:NTF \tex_expanded:D
2348 {
2349   \cs_new:Npn \exp_args:Ne #1#2
2350   { \exp_after:wN #1 \tex_expanded:D { {#2} } }
2351 }
2352 {
2353   \cs_new:Npn \exp_args:Ne #1#2
2354   {
2355     \exp_after:wN #1 \exp_after:wN
2356     { \exp:w \__exp_e:nn {#2} { } }
2357   }
2358 }
```

(End definition for `\exp_args:Ne`. This function is documented on page 35.)

\exp_args:Nf
\exp_args:NV
\exp_args:Nv

```

2359 \cs_new:Npn \exp_args:Nf #1#2
2360 { \exp_after:wN #1 \exp_after:wN { \exp:w \exp_end_continue_f:w #2 } }
2361 \cs_new:Npn \exp_args:Nv #1#2
2362 {
2363   \exp_after:wN #1 \exp_after:wN
2364   { \exp:w \__exp_eval_register:c {#2} }
2365 }
2366 \cs_new:Npn \exp_args:NV #1#2
2367 {
2368   \exp_after:wN #1 \exp_after:wN
2369   { \exp:w \__exp_eval_register:N #2 }
2370 }

```

(End definition for `\exp_args:Nf`, `\exp_args:NV`, and `\exp_args:Nv`. These functions are documented on page 35.)

\exp_args:NNV
\exp_args:NNv
\exp_args:NNe
\exp_args:NNf
\exp_args:Nco
\exp_args:NcV
\exp_args:Ncv
\exp_args:Ncf
\exp_args:NVV

Some more hand-tuned function with three arguments. If we forced that an `o` argument always has braces, we could implement `\exp_args:Nco` with less tokens and only two arguments.

```

2371 \cs_new:Npn \exp_args:NNV #1#2#3
2372 {
2373   \exp_after:wN #1
2374   \exp_after:wN #2
2375   \exp_after:wN { \exp:w \__exp_eval_register:N #3 }
2376 }
2377 \cs_new:Npn \exp_args:NNv #1#2#3
2378 {
2379   \exp_after:wN #1
2380   \exp_after:wN #2
2381   \exp_after:wN { \exp:w \__exp_eval_register:c {#3} }
2382 }
2383 \cs_if_exist:NTF \tex_expanded:D
2384 {
2385   \cs_new:Npn \exp_args:NNe #1#2#3
2386   {
2387     \exp_after:wN #1
2388     \exp_after:wN #2
2389     \tex_expanded:D { {#3} }
2390   }
2391 }
2392 { \cs_new:Npn \exp_args:NNe { \::N \::e \::: } }
2393 \cs_new:Npn \exp_args:NNf #1#2#3
2394 {
2395   \exp_after:wN #1
2396   \exp_after:wN #2
2397   \exp_after:wN { \exp:w \exp_end_continue_f:w #3 }
2398 }
2399 \cs_new:Npn \exp_args:Nco #1#2#3
2400 {
2401   \exp_after:wN #1
2402   \cs:w #2 \exp_after:wN \cs_end:
2403   \exp_after:wN {#3}
2404 }

```

```

2405 \cs_new:Npn \exp_args:NcV #1#2#3
2406 {
2407     \exp_after:wN #1
2408     \cs:w #2 \exp_after:wN \cs_end:
2409     \exp_after:wN { \exp:w \__exp_eval_register:N #3 }
2410 }
2411 \cs_new:Npn \exp_args:Ncv #1#2#3
2412 {
2413     \exp_after:wN #1
2414     \cs:w #2 \exp_after:wN \cs_end:
2415     \exp_after:wN { \exp:w \__exp_eval_register:c {#3} }
2416 }
2417 \cs_new:Npn \exp_args:Ncf #1#2#3
2418 {
2419     \exp_after:wN #1
2420     \cs:w #2 \exp_after:wN \cs_end:
2421     \exp_after:wN { \exp:w \exp_end_continue_f:w #3 }
2422 }
2423 \cs_new:Npn \exp_args:NVV #1#2#3
2424 {
2425     \exp_after:wN #1
2426     \exp_after:wN { \exp:w \exp_after:wN
2427         \__exp_eval_register:N \exp_after:wN #2 \exp_after:wN }
2428     \exp_after:wN { \exp:w \__exp_eval_register:N #3 }
2429 }

```

(End definition for `\exp_args:NNV` and others. These functions are documented on page 36.)

`\exp_args:NNNV`
`\exp_args:NNNv`
`\exp_args:NcNc`
`\exp_args:NcNo`
`\exp_args:Ncco`

A few more that we can hand-tune.

```

2430 \cs_new:Npn \exp_args:NNNV #1#2#3#4
2431 {
2432     \exp_after:wN #1
2433     \exp_after:wN #2
2434     \exp_after:wN #3
2435     \exp_after:wN { \exp:w \__exp_eval_register:N #4 }
2436 }
2437 \cs_new:Npn \exp_args:NNNv #1#2#3#4
2438 {
2439     \exp_after:wN #1
2440     \exp_after:wN #2
2441     \exp_after:wN #3
2442     \exp_after:wN { \exp:w \__exp_eval_register:c {#4} }
2443 }
2444 \cs_new:Npn \exp_args:NcNc #1#2#3#4
2445 {
2446     \exp_after:wN #1
2447     \cs:w #2 \exp_after:wN \cs_end:
2448     \exp_after:wN #3
2449     \cs:w #4 \cs_end:
2450 }
2451 \cs_new:Npn \exp_args:NcNo #1#2#3#4
2452 {
2453     \exp_after:wN #1
2454     \cs:w #2 \exp_after:wN \cs_end:

```



```

2455     \exp_after:wN #3
2456     \exp_after:wN {#4}
2457   }
2458 \cs_new:Npn \exp_args:Ncco #1#2#3#4
2459 {
2460   \exp_after:wN #1
2461   \cs:w #2 \exp_after:wN \cs_end:
2462   \cs:w #3 \exp_after:wN \cs_end:
2463   \exp_after:wN {#4}
2464 }

```

(End definition for `\exp_args:NNNV` and others. These functions are documented on page 37.)

`\exp_args:Nx`

```

2465 \cs_new_protected:Npn \exp_args:Nx #1#2
2466 { \use:x { \exp_not:N #1 {#2} } }

```

(End definition for `\exp_args:Nx`. This function is documented on page 36.)

42.3 Last-unbraced versions

`__exp_arg_last_unbraced:nn` There are a few places where the last argument needs to be available unbraced. First some helper macros.

```

\::o_unbraced
\::V_unbraced
\::v_unbraced
\::e_unbraced
\::f_unbraced
\::x_unbraced
2467 \cs_new:Npn \__exp_arg_last_unbraced:nn #1#2 { #2#1 }
2468 \cs_new:Npn \::o_unbraced \::: #1#2
2469 { \exp_after:wN \__exp_arg_last_unbraced:nn \exp_after:wN {#2} {#1} }
2470 \cs_new:Npn \::V_unbraced \::: #1#2
2471 {
2472   \exp_after:wN \__exp_arg_last_unbraced:nn
2473   \exp_after:wN { \exp:w \__exp_eval_register:N #2 } {#1}
2474 }
2475 \cs_new:Npn \::v_unbraced \::: #1#2
2476 {
2477   \exp_after:wN \__exp_arg_last_unbraced:nn
2478   \exp_after:wN { \exp:w \__exp_eval_register:c {#2} } {#1}
2479 }
2480 \cs_if_exist:NTF \tex_expanded:D
2481 {
2482   \cs_new:Npn \::e_unbraced \::: #1#2
2483   { \tex_expanded:D { \exp_not:n {#1} #2 } }
2484 }
2485 {
2486   \cs_new:Npn \::e_unbraced \::: #1#2
2487   { \exp:w \__exp_e:nn {#2} {#1} }
2488 }
2489 \cs_new:Npn \::f_unbraced \::: #1#2
2490 {
2491   \exp_after:wN \__exp_arg_last_unbraced:nn
2492   \exp_after:wN { \exp:w \exp_end_continue_f:w #2 } {#1}
2493 }
2494 \cs_new_protected:Npn \::x_unbraced \::: #1#2
2495 {
2496   \cs_set_nopar:Npx \l__exp_internal_tl { \exp_not:n {#1} #2 }

```

```

2497 \l__exp_internal_tl
2498 }

```

(End definition for `__exp_arg_last_unbraced:nn` and others. These functions are documented on page 42.)

```

\exp_last_unbraced:No
\exp_last_unbraced:NV
\exp_last_unbraced:Nv
\exp_last_unbraced:Ne
\exp_last_unbraced:Nf
\exp_last_unbraced:NNo
\exp_last_unbraced:NNV
\exp_last_unbraced:NNf
\exp_last_unbraced:Nco
\exp_last_unbraced:NcV
\exp_last_unbraced:NNNo
\exp_last_unbraced:NNNV
\exp_last_unbraced:NNNf
\exp_last_unbraced:Nno
\exp_last_unbraced:Noo
\exp_last_unbraced:Nfo
\exp_last_unbraced:NnNo
\exp_last_unbraced:NNNNo
\exp_last_unbraced:NNNNf
\exp_last_unbraced:Nx

```

Now the business end: most of these are hand-tuned for speed, but the general system is in place.

```

2499 \cs_new:Npn \exp_last_unbraced:No #1#2 { \exp_after:wN #1 #2 }
2500 \cs_new:Npn \exp_last_unbraced:NV #1#2
2501 { \exp_after:wN #1 \exp:w \__exp_eval_register:N #2 }
2502 \cs_new:Npn \exp_last_unbraced:Nv #1#2
2503 { \exp_after:wN #1 \exp:w \__exp_eval_register:c {#2} }
2504 \cs_if_exist:NTF \tex_expanded:D
2505 {
2506   \cs_new:Npn \exp_last_unbraced:Ne #1#2
2507   { \exp_after:wN #1 \tex_expanded:D {#2} }
2508 }
2509 { \cs_new:Npn \exp_last_unbraced:Ne { \::e_unbraced \:: } }
2510 \cs_new:Npn \exp_last_unbraced:Nf #1#2
2511 { \exp_after:wN #1 \exp:w \exp_end_continue_f:w #2 }
2512 \cs_new:Npn \exp_last_unbraced:NNo #1#2#3
2513 { \exp_after:wN #1 \exp_after:wN #2 #3 }
2514 \cs_new:Npn \exp_last_unbraced:NNV #1#2#3
2515 {
2516   \exp_after:wN #1
2517   \exp_after:wN #2
2518   \exp:w \__exp_eval_register:N #3
2519 }
2520 \cs_new:Npn \exp_last_unbraced:NNf #1#2#3
2521 {
2522   \exp_after:wN #1
2523   \exp_after:wN #2
2524   \exp:w \exp_end_continue_f:w #3
2525 }
2526 \cs_new:Npn \exp_last_unbraced:Nco #1#2#3
2527 { \exp_after:wN #1 \cs:w #2 \exp_after:wN \cs_end: #3 }
2528 \cs_new:Npn \exp_last_unbraced:NcV #1#2#3
2529 {
2530   \exp_after:wN #1
2531   \cs:w #2 \exp_after:wN \cs_end:
2532   \exp:w \__exp_eval_register:N #3
2533 }
2534 \cs_new:Npn \exp_last_unbraced:NNNo #1#2#3#4
2535 { \exp_after:wN #1 \exp_after:wN #2 \exp_after:wN #3 #4 }
2536 \cs_new:Npn \exp_last_unbraced:NNNV #1#2#3#4
2537 {
2538   \exp_after:wN #1
2539   \exp_after:wN #2
2540   \exp_after:wN #3
2541   \exp:w \__exp_eval_register:N #4
2542 }
2543 \cs_new:Npn \exp_last_unbraced:NNNf #1#2#3#4
2544 {

```

```

2545 \exp_after:wN #1
2546 \exp_after:wN #2
2547 \exp_after:wN #3
2548 \exp:w \exp_end_continue_f:w #4
2549 }
2550 \cs_new:Npn \exp_last_unbraced:Nno { \::n \::o_unbraced \:: }
2551 \cs_new:Npn \exp_last_unbraced:Noo { \::o \::o_unbraced \:: }
2552 \cs_new:Npn \exp_last_unbraced:Nfo { \::f \::o_unbraced \:: }
2553 \cs_new:Npn \exp_last_unbraced:NnNo { \::n \::N \::o_unbraced \:: }
2554 \cs_new:Npn \exp_last_unbraced:NNNNo #1#2#3#4#5
2555 { \exp_after:wN #1 \exp_after:wN #2 \exp_after:wN #3 \exp_after:wN #4 #5 }
2556 \cs_new:Npn \exp_last_unbraced:NNNNf #1#2#3#4#5
2557 {
2558 \exp_after:wN #1
2559 \exp_after:wN #2
2560 \exp_after:wN #3
2561 \exp_after:wN #4
2562 \exp:w \exp_end_continue_f:w #5
2563 }
2564 \cs_new_protected:Npn \exp_last_unbraced:Nx { \::x_unbraced \:: }

```

(End definition for `\exp_last_unbraced:No` and others. These functions are documented on page 38.)

`\exp_last_two_unbraced:Noo`
`_exp_last_two_unbraced:noN`

If #2 is a single token then this can be implemented as

```

\cs_new:Npn \exp_last_two_unbraced:Noo #1 #2 #3
{ \exp_after:wN \exp_after:wN \exp_after:wN #1 \exp_after:wN #2 #3 }

```

However, for robustness this is not suitable. Instead, a bit of a shuffle is used to ensure that #2 can be multiple tokens.

```

2565 \cs_new:Npn \exp_last_two_unbraced:Noo #1#2#3
2566 { \exp_after:wN \_exp_last_two_unbraced:noN \exp_after:wN {#3} {#2} #1 }
2567 \cs_new:Npn \_exp_last_two_unbraced:noN #1#2#3
2568 { \exp_after:wN #3 #2 #1 }

```

(End definition for `\exp_last_two_unbraced:Noo` and `_exp_last_two_unbraced:noN`. This function is documented on page 38.)

42.4 Preventing expansion

`_kernel_exp_not:w`

At the kernel level, we need the primitive behaviour to allow expansion *before* the brace group.

```

2569 \cs_new_eq:NN \_kernel_exp_not:w \tex_unexpanded:D

```

(End definition for `_kernel_exp_not:w`.)

`\exp_not:c`
`\exp_not:o`
`\exp_not:e`
`\exp_not:f`
`\exp_not:V`
`\exp_not:v`

All these except `\exp_not:c` call the kernel-internal `_kernel_exp_not:w` namely `\tex_unexpanded:D`.

```

2570 \cs_new:Npn \exp_not:c #1 { \exp_after:wN \exp_not:N \cs:w #1 \cs_end: }
2571 \cs_new:Npn \exp_not:o #1 { \_kernel_exp_not:w \exp_after:wN {#1} }
2572 \cs_if_exist:NTF \tex_expanded:D
2573 {
2574 \cs_new:Npn \exp_not:e #1
2575 { \_kernel_exp_not:w \tex_expanded:D { {#1} } }

```

```

2576 }
2577 {
2578   \cs_new:Npn \exp_not:e
2579     { \__kernel_exp_not:w \exp_args:Ne \prg_do_nothing: }
2580 }
2581 \cs_new:Npn \exp_not:f #1
2582   { \__kernel_exp_not:w \exp_after:wN { \exp:w \exp_end_continue_f:w #1 } }
2583 \cs_new:Npn \exp_not:V #1
2584   {
2585     \__kernel_exp_not:w \exp_after:wN
2586     { \exp:w \__exp_eval_register:N #1 }
2587   }
2588 \cs_new:Npn \exp_not:v #1
2589   {
2590     \__kernel_exp_not:w \exp_after:wN
2591     { \exp:w \__exp_eval_register:c {#1} }
2592   }

```

(End definition for `\exp_not:c` and others. These functions are documented on page 39.)

42.5 Controlled expansion

`\exp:w`
`\exp_end:`
`\exp_end_continue_f:w`
`\exp_end_continue_f:nw`

To trigger a sequence of “arbitrarily” many expansions we need a method to invoke T_EX’s expansion mechanism in such a way that (a) we are able to stop it in a controlled manner and (b) the result of what triggered the expansion in the first place is null, i.e., that we do not get any unwanted side effects. There aren’t that many possibilities in T_EX; in fact the one explained below might well be the only one (as normally the result of expansion is not null).

The trick here is to make use of the fact that `\tex_romannumeral:D` expands the tokens following it when looking for a number and that its expansion is null if that number turns out to be zero or negative. So we use that to start the expansion sequence: `\exp:w` is set equal to `\tex_romannumeral:D` in `l3basics`. To stop the expansion sequence in a controlled way all we need to provide is a constant integer zero as part of expanded tokens. As this is an integer constant it immediately stops `\tex_romannumeral:D`’s search for a number. Again, the definition of `\exp_end:` as the integer constant zero is in `l3basics`. (Note that according to our specification all tokens we expand initiated by `\exp:w` are supposed to be expandable (as well as their replacement text in the expansion) so we will not encounter a “number” that actually result in a roman numeral being generated. Or if we do then the programmer made a mistake.)

If on the other hand we want to stop the initial expansion sequence but continue with an f-type expansion we provide the alphabetic constant `’^^@` that also represents 0 but this time T_EX’s syntax for a *⟨number⟩* continues searching for an optional space (and it continues expansion doing that) — see T_EXbook page 269 for details.

```

2593 \group_begin:
2594   \tex_catcode:D ‘^^@ = 13
2595   \cs_new_protected:Npn \exp_end_continue_f:w { ‘^^@ }

```

If the above definition ever appears outside its proper context the active character `^^@` will be executed so we turn this into an error. The test for existence covers the (unlikely) case that some other code has already defined `^^@`: this is true for example for `xmltex.tex`.

```

2596   \if_cs_exist:N ^^@
2597   \else:

```

```

2598 \cs_new:Npn ^^@
2599 { \msg_expandable_error:nn { kernel } { bad-exp-end-f } }
2600 \fi:

```

The same but grabbing an argument to remove spaces and braces.

```

2601 \cs_new:Npn \exp_end_continue_f:nw #1 { ^^@ #1 }
2602 \group_end:

```

(End definition for `\exp:w` and others. These functions are documented on page 41.)

42.6 Emulating e-type expansion

When the `\expanded` primitive is available it is used to implement e-type expansion; otherwise we emulate it.

```

2603 \cs_if_exist:NF \tex_expanded:D
2604 {

```

`__exp_e:nn` Repeatedly expand tokens, keeping track of fully-expanded tokens in the second argument to `__exp_e:nn`; this function eventually calls `__exp_e_end:nn` to leave `\exp_end:` in the input stream, followed by the result of the expansion. There are many special cases: spaces, brace groups, `\noexpand`, `\unexpanded`, `\the`, `\primitive`. While we use brace tricks `\if_false: { \fi:`, the expansion of this function is always triggered by `\exp:w` so brace balance is eventually restored after that is hit with a single step of expansion. Otherwise we could not nest e-type expansions within each other.

```

2605 \cs_new:Npn \__exp_e:nn #1
2606 {
2607   \if_false: { \fi:
2608     \tl_if_head_is_N_type:nTF {#1}
2609     { \__exp_e:N }
2610     {
2611       \tl_if_head_is_group:nTF {#1}
2612       { \__exp_e_group:n }
2613       {
2614         \tl_if_empty:nTF {#1}
2615         { \exp_after:wN \__exp_e_end:nn }
2616         { \exp_after:wN \__exp_e_space:nn }
2617         \exp_after:wN { \if_false: } \fi:
2618       }
2619     }
2620     #1
2621   }
2622 }
2623 \cs_new:Npn \__exp_e_end:nn #1#2 { \exp_end: #2 }

```

(End definition for `__exp_e:nn` and `__exp_e_end:nn`.)

`__exp_e_space:nn` For an explicit space character, remove it by f-expansion and put it in the (future) output.

```

2624 \cs_new:Npn \__exp_e_space:nn #1#2
2625 { \exp_args:Nf \__exp_e:nn {#1} { #2 ~ } }

```

(End definition for `__exp_e_space:nn`.)

`__exp_e_group:n` For a group, expand its contents, wrap it in two pairs of braces, and call `__exp_e_put:nn`. This function places the first item (the double-brace wrapped result) into the output. Importantly, `\tl_head:n` works even if the input contains quarks.

```

2626 \cs_new:Npn \__exp_e_group:n #1
2627 {
2628   \exp_after:wN \__exp_e_put:nn
2629   \exp_after:wN { \exp_after:wN { \exp_after:wN {
2630     \exp:w \if_false: } \fi: \__exp_e:nn {#1} { } } }
2631 }
2632 \cs_new:Npn \__exp_e_put:nn #1
2633 {
2634   \exp_args:NNo \exp_args:No \__exp_e_put:nnn
2635   { \tl_head:n {#1} } {#1}
2636 }
2637 \cs_new:Npn \__exp_e_put:nnn #1#2#3
2638 { \exp_args:No \__exp_e:nn { \use_none:n #2 } { #3 #1 } }

```

(End definition for `__exp_e_group:n`, `__exp_e_put:nn`, and `__exp_e_put:nnn`.)

`__exp_e:N` For an N-type token, call `__exp_e:Nnn` with arguments the *first token*, the remaining tokens to expand and what's already been expanded. If the *first token* is non-expandable, including `\protected` (`\long` or not) macros, it is put in the result by `__exp_e_protected:Nnn`. The four special primitives `\unexpanded`, `\noexpand`, `\the`, `\primitive` are detected; otherwise the token is expanded by `__exp_e_expandable:Nnn`.

```

2639 \cs_new:Npn \__exp_e:N #1
2640 {
2641   \exp_after:wN \__exp_e:Nnn
2642   \exp_after:wN #1
2643   \exp_after:wN { \if_false: } \fi:
2644 }
2645 \cs_new:Npn \__exp_e:Nnn #1
2646 {
2647   \if_case:w
2648     \exp_after:wN \if_meaning:w \exp_not:N #1 #1 1 ~ \fi:
2649     \token_if_protected_macro:NT #1 { 1 ~ }
2650     \token_if_protected_long_macro:NT #1 { 1 ~ }
2651     \if_meaning:w \exp_not:n #1 2 ~ \fi:
2652     \if_meaning:w \exp_not:N #1 3 ~ \fi:
2653     \if_meaning:w \tex_the:D #1 4 ~ \fi:
2654     \if_meaning:w \tex_primitive:D #1 5 ~ \fi:
2655     0 ~
2656     \exp_after:wN \__exp_e_expandable:Nnn
2657   \or: \exp_after:wN \__exp_e_protected:Nnn
2658   \or: \exp_after:wN \__exp_e_unexpanded:Nnn
2659   \or: \exp_after:wN \__exp_e_noexpand:Nnn
2660   \or: \exp_after:wN \__exp_e_the:Nnn
2661   \or: \exp_after:wN \__exp_e_primitive:Nnn
2662   \fi:
2663   #1
2664 }
2665 \cs_new:Npn \__exp_e_protected:Nnn #1#2#3
2666 { \__exp_e:nn {#2} { #3 #1 } }
2667 \cs_new:Npn \__exp_e_expandable:Nnn #1#2
2668 { \exp_args:No \__exp_e:nn { #1 #2 } }

```

(End definition for `_exp_e:N` and others.)

```

\__exp_e_primitive:Nnn
\__exp_e_primitive_aux:NNw
\__exp_e_primitive_aux:NNnn
  \__exp_e_primitive_other:NNnn
  \__exp_e_primitive_other_aux:nNNnn

```

We don't try hard to make sensible error recovery since the error recovery of `\tex_primitive:D` when followed by something else than a primitive depends on the engine. The only valid case is when what follows is N-type. Then distinguish special primitives `\unexpanded`, `\noexpand`, `\the`, `\primitive` from other primitives. In the “other” case, the only reasonable way to check if the primitive that follows `\tex_primitive:D` is expandable is to expand and compare the before-expansion and after-expansion results. If they coincide then probably the primitive is non-expandable and should be put in the output together with `\tex_primitive:D` (one can cook up contrived counter-examples where the true `\expanded` would have an infinite loop), and otherwise one should continue expanding.

```

2669   \cs_new:Npn \__exp_e_primitive:Nnn #1#2
2670   {
2671     \if_false: { \fi:
2672       \tl_if_head_is_N_type:nTF {#2}
2673       { \__exp_e_primitive_aux:NNw #1 }
2674       {
2675         \msg_expandable_error:nnn { kernel } { e-type }
2676         { Missing~primitive-name }
2677         \__exp_e_primitive_aux:NNw #1 \c_empty_tl
2678       }
2679       #2
2680     }
2681   }
2682   \cs_new:Npn \__exp_e_primitive_aux:NNw #1#2
2683   {
2684     \exp_after:wN \__exp_e_primitive_aux:NNnn
2685     \exp_after:wN #1
2686     \exp_after:wN #2
2687     \exp_after:wN { \if_false: } \fi:
2688   }
2689   \cs_new:Npn \__exp_e_primitive_aux:NNnn #1#2
2690   {
2691     \exp_args:Nf \str_case_e:nnTF { \cs_to_str:N #2 }
2692     {
2693       { unexpanded } { \__exp_e_unexpanded:Nnn \exp_not:n }
2694       { noexpand } { \__exp_e_noexpand:NNnn \exp_not:N }
2695       { the } { \__exp_e_the:NNnn \tex_the:D }
2696       {
2697         \sys_if_engine_xetex:T { pdf }
2698         \sys_if_engine luatex:T { pdf }
2699         primitive
2700       } { \__exp_e_primitive:Nnn #1 }
2701     }
2702     { \__exp_e_primitive_other:NNnn #1 #2 }
2703   }
2704   \cs_new:Npn \__exp_e_primitive_other:NNnn #1#2#3
2705   {
2706     \exp_args:No \__exp_e_primitive_other_aux:nNNnn
2707     { #1 #2 #3 }
2708     #1 #2 {#3}
2709   }

```

```

2710 \cs_new:Npn \__exp_e_primitive_other_aux:nNnn #1#2#3#4#5
2711 {
2712   \str_if_eq:nnTF {#1} { #2 #3 #4 }
2713   { \__exp_e:nn {#4} { #5 #2 #3 } }
2714   { \__exp_e:nn {#1} {#5} }
2715 }

```

(End definition for `__exp_e_primitive:Nnn` and others.)

`__exp_e_noexpand:Nnn` The `\noexpand` primitive has no effect when followed by a token that is not N-type; otherwise `__exp_e_put:nn` can grab the next token and put it in the result unchanged.

```

2716 \cs_new:Npn \__exp_e_noexpand:Nnn #1#2
2717 {
2718   \tl_if_head_is_N_type:nTF {#2}
2719   { \__exp_e_put:nn } { \__exp_e:nn } {#2}
2720 }

```

(End definition for `__exp_e_noexpand:Nnn`.)

`__exp_e_unexpanded:Nnn` The `\unexpanded` primitive expands and ignores any space, `\scan_stop:`, or token affected by `\exp_not:N`, then expects a brace group. Since we only support brace-balanced token lists it is impossible to support the case where the argument of `\unexpanded` starts with an implicit brace. Even though we want to expand and ignore spaces we cannot blindly f-expand because tokens affected by `\exp_not:N` should be discarded without being expanded further.

As usual distinguish four cases: brace group (the normal case, where we just put the item in the result), space (just f-expand to remove the space), empty (an error), or N-type *token*. In the last case call `__exp_e_unexpanded:nN` triggered by an f-expansion. Having a non-expandable *token* after `\unexpanded` is an error (we recover by passing `{}` to `\unexpanded`; this is different from \TeX because the error recovery of `\unexpanded` changes the balance of braces), unless that *token* is `\scan_stop:` or a space (recall that we don't implement the case of an implicit begin-group token). An expandable *token* is instead expanded, unless it is `\noexpand`. The latter primitive can be followed by an expandable N-type token (removed), by a non-expandable one (kept and later causing an error), by a space (removed by f-expansion), or by a brace group or nothing (later causing an error).

```

2721 \cs_new:Npn \__exp_e_unexpanded:Nnn #1 { \__exp_e_unexpanded:nn }
2722 \cs_new:Npn \__exp_e_unexpanded:nn #1
2723 {
2724   \tl_if_head_is_N_type:nTF {#1}
2725   {
2726     \exp_args:Nf \__exp_e_unexpanded:nn
2727     { \__exp_e_unexpanded:nN {#1} #1 }
2728   }
2729   {
2730     \tl_if_head_is_group:nTF {#1}
2731     { \__exp_e_put:nn }
2732     {
2733       \tl_if_empty:nTF {#1}
2734       {
2735         \msg_expandable_error:nnn
2736         { kernel } { e-type }
2737         { \unexpanded missing~brace }

```



```

2738         \__exp_e_end:nn
2739     }
2740     { \exp_args:Nf \__exp_e_unexpanded:nn }
2741 }
2742 {#1}
2743 }
2744 }
2745 \cs_new:Npn \__exp_e_unexpanded:nN #1#2
2746 {
2747     \exp_after:wN \if_meaning:w \exp_not:N #2 #2
2748     \exp_after:wN \use_i:nn
2749     \else:
2750     \exp_after:wN \use_ii:nn
2751     \fi:
2752     {
2753         \token_if_eq_catcode:NNTF #2 \c_space_token
2754         { \exp_stop_f: }
2755         {
2756             \token_if_eq_meaning:NNTF #2 \scan_stop:
2757             { \exp_stop_f: }
2758             {
2759                 \msg_expandable_error:nnn
2760                 { kernel } { e-type }
2761                 { \unexpanded missing~brace }
2762                 { }
2763             }
2764         }
2765     }
2766     {
2767         \token_if_eq_meaning:NNTF #2 \exp_not:N
2768         {
2769             \exp_args:No \tl_if_head_is_N_type:nT { \use_none:n #1 }
2770             { \__exp_e_unexpanded:N }
2771         }
2772         { \exp_after:wN \exp_stop_f: #2 }
2773     }
2774 }
2775 \cs_new:Npn \__exp_e_unexpanded:N #1
2776 {
2777     \exp_after:wN \if_meaning:w \exp_not:N #1 #1 \else:
2778     \exp_after:wN \use_i:nn
2779     \fi:
2780     \exp_stop_f: #1
2781 }

```

(End definition for __exp_e_unexpanded:Nnn and others.)

__exp_e_the:Nnn Finally implement \the. Followed by anything other than an N-type $\langle token \rangle$ this causes an error (we just let \TeX make one), otherwise we test the $\langle token \rangle$. If the $\langle token \rangle$ is expandable, expand it. Otherwise it could be any kind of register, or things like \numexpr , so there is no way to deal with all cases. Thankfully, only \toks data needs to be protected from expansion since everything else gives a string of characters. If the $\langle token \rangle$ is \toks we find a number and unpack using the \the_toks functions. If it is a token register we unpack it in a brace group and call __exp_e_put:nn to move it to

the result. Otherwise we unpack and continue expanding (useless but safe) since it is basically impossible to have a handle on where the result of `\the` ends.

```

2782 \cs_new:Npn \__exp_e_the:Nnn #1#2
2783 {
2784   \tl_if_head_is_N_type:nTF {#2}
2785   { \if_false: { \fi: \__exp_e_the:N #2 } }
2786   { \exp_args:No \__exp_e:nn { \tex_the:D #2 } }
2787 }
2788 \cs_new:Npn \__exp_e_the:N #1
2789 {
2790   \exp_after:wN \if_meaning:w \exp_not:N #1 #1
2791   \exp_after:wN \use_i:nn
2792   \else:
2793     \exp_after:wN \use_ii:nn
2794   \fi:
2795   {
2796     \if_meaning:w \tex_toks:D #1
2797     \exp_after:wN \__exp_e_the_toks:wnn \int_value:w
2798     \exp_after:wN \__exp_e_the_toks:n
2799     \exp_after:wN { \int_value:w \if_false: } \fi:
2800     \else:
2801       \__exp_e_if_toks_register:NTF #1
2802       { \exp_after:wN \__exp_e_the_toks_reg:N }
2803       {
2804         \exp_after:wN \__exp_e:nn \exp_after:wN {
2805           \tex_the:D \if_false: } \fi:
2806       }
2807       \exp_after:wN #1
2808     \fi:
2809   }
2810   {
2811     \exp_after:wN \__exp_e_the:Nnn \exp_after:wN ?
2812     \exp_after:wN { \exp:w \if_false: } \fi:
2813     \exp_after:wN \exp_end: #1
2814   }
2815 }
2816 \cs_new:Npn \__exp_e_the_toks_reg:N #1
2817 {
2818   \exp_after:wN \__exp_e_put:nn \exp_after:wN {
2819     \exp_after:wN {
2820       \tex_the:D \if_false: } \fi: #1 }
2821 }

```

(End definition for `__exp_e_the:Nnn`, `__exp_e_the:N`, and `__exp_e_the_toks_reg:N`.)

<code>__exp_e_the_toks:wnn</code> <code>__exp_e_the_toks:n</code> <code>__exp_e_the_toks:N</code>	<p>The calling function has applied <code>\int_value:w</code> so we collect digits with <code>__exp_e_the_toks:n</code> (which gets the token list as an argument) and <code>__exp_e_the_toks:N</code> (which gets the first token in case it is N-type). The digits are themselves collected into an <code>\int_value:w</code> argument to <code>__exp_e_the_toks:wnn</code>. Then that function unpacks the <code>\toks<number></code> into the result. We include <code>?</code> because <code>__exp_e_put:nnn</code> removes one item from its second argument. Note that our approach is rather crude: in cases like <code>\the\toks12~34</code> the first <code>\int_value:w</code> removes the space and we will incorrectly unpack the <code>\the\toks1234</code>.</p>
--	--

```

2822 \cs_new:Npn \__exp_e_the_toks:wnn #1; #2
2823 {
2824   \exp_args:No \__exp_e_put:nnn
2825   { \tex_the:D \tex_toks:D #1 } { ? #2 }
2826 }
2827 \cs_new:Npn \__exp_e_the_toks:n #1
2828 {
2829   \tl_if_head_is_N_type:NTF {#1}
2830   { \exp_after:wN \__exp_e_the_toks:N \if_false: { \fi: #1 } }
2831   { ; {#1} }
2832 }
2833 \cs_new:Npn \__exp_e_the_toks:N #1
2834 {
2835   \if_int_compare:w 10 < 9 \token_to_str:N #1 \exp_stop_f:
2836   \exp_after:wN \use_i:nn
2837   \else:
2838     \exp_after:wN \use_ii:nn
2839   \fi:
2840   {
2841     #1
2842     \exp_after:wN \__exp_e_the_toks:n
2843     \exp_after:wN { \if_false: } \fi:
2844   }
2845   {
2846     \exp_after:wN ;
2847     \exp_after:wN { \if_false: } \fi: #1
2848   }
2849 }

```

(End definition for `__exp_e_the_toks:wnn`, `__exp_e_the_toks:n`, and `__exp_e_the_toks:N`.)

`__exp_e_if_toks_register:N` We need to detect both `\toks` registers like `\toks@` in L^AT_EX 2_ε and parameters such as `\everypar`, as the result of unpacking the register should not expand further. Registers are found by `\token_if_toks_register:N` by inspecting the meaning. The list of parameters is finite so we just use a `\cs_if_exist:cTF` test to look up in a table. We abuse `\cs_to_str:N`'s ability to remove a leading escape character whatever it is.

```

\__exp_e_if_toks_register:NTF
\__exp_e_the_XeTeXinterchartoks:
\__exp_e_the_errhelp:
\__exp_e_the_everycr:
\__exp_e_the_everydisplay:
\__exp_e_the_everyeof:
\__exp_e_the_everyhbox:
\__exp_e_the_everyjob:
\__exp_e_the_everymath:
\__exp_e_the_everypar:
\__exp_e_the_everyvbox:
\__exp_e_the_output:
\__exp_e_the_pdffpageattr:
\__exp_e_the_pdfpageresources:
\__exp_e_the_pdfpagesattr:
\__exp_e_the_pdfpkmode:
2850 \prg_new_conditional:Nppn \__exp_e_if_toks_register:N #1 { TF }
2851 {
2852   \token_if_toks_register:NTF #1 { \prg_return_true: }
2853   {
2854     \cs_if_exist:cTF
2855     {
2856       __exp_e_the_
2857       \exp_after:wN \cs_to_str:N
2858       \token_to_meaning:N #1
2859       :
2860     } { \prg_return_true: } { \prg_return_false: }
2861   }
2862 }
2863 \cs_new_eq:NN \__exp_e_the_XeTeXinterchartoks: ?
2864 \cs_new_eq:NN \__exp_e_the_errhelp: ?
2865 \cs_new_eq:NN \__exp_e_the_everycr: ?
2866 \cs_new_eq:NN \__exp_e_the_everydisplay: ?
2867 \cs_new_eq:NN \__exp_e_the_everyeof: ?

```

```

2868 \cs_new_eq:NN \__exp_e_the_everyhbox: ?
2869 \cs_new_eq:NN \__exp_e_the_everyjob: ?
2870 \cs_new_eq:NN \__exp_e_the_everymath: ?
2871 \cs_new_eq:NN \__exp_e_the_everypar: ?
2872 \cs_new_eq:NN \__exp_e_the_everyvbox: ?
2873 \cs_new_eq:NN \__exp_e_the_output: ?
2874 \cs_new_eq:NN \__exp_e_the_pdfpageattr: ?
2875 \cs_new_eq:NN \__exp_e_the_pdfpageresources: ?
2876 \cs_new_eq:NN \__exp_e_the_pdfpagesattr: ?
2877 \cs_new_eq:NN \__exp_e_the_pdfpkmode: ?

```

(End definition for `__exp_e_if_toks_register:NTF` and others.)

We are done emulating e-type argument expansion when `\expanded` is unavailable.

```

2878 }

```

42.7 Defining function variants

```

2879 <@@=cs>

```

`\s__cs_mark` Internal scan marks. No l3quark yet, so do things by hand.

```

\s__cs_stop 2880 \cs_new_eq:NN \s__cs_mark \scan_stop:
2881 \cs_new_eq:NN \s__cs_stop \scan_stop:

```

(End definition for `\s__cs_mark` and `\s__cs_stop`.)

`\q__cs_recursion_stop` Internal recursion quarks. No l3quark yet, so do things by hand.

```

2882 \cs_new:Npn \q__cs_recursion_stop { \q__cs_recursion_stop }

```

(End definition for `\q__cs_recursion_stop`.)

`__cs_use_none_delimit_by_s_stop:w` Internal scan marks.

```

\__cs_use_i_delimit_by_s_stop:nw 2883 \cs_new:Npn \__cs_use_none_delimit_by_s_stop:w #1 \s__cs_stop { }
\__cs_use_none_delimit_by_q_recursion_stop:w 2884 \cs_new:Npn \__cs_use_i_delimit_by_s_stop:nw #1 #2 \s__cs_stop {#1}
2885 \cs_new:Npn \__cs_use_none_delimit_by_q_recursion_stop:w
2886 #1 \q__cs_recursion_stop { }

```

(End definition for `__cs_use_none_delimit_by_s_stop:w`, `__cs_use_i_delimit_by_s_stop:nw`, and `__cs_use_none_delimit_by_q_recursion_stop:w`.)

`\cs_generate_variant:Nn` #1 : Base form of a function; e.g., `\tl_set:Nn`

`\cs_generate_variant:cn` #2 : One or more variant argument specifiers; e.g., `{Nx,c,cx}`

After making sure that the base form exists, test whether it is protected or not and define `__cs_tmp:w` as either `\cs_new:Npx` or `\cs_new_protected:Npx`, which is then used to define all the variants (except those involving x-expansion, always protected). Split up the original base function only once, to grab its name and signature. Then we wish to iterate through the comma list of variant argument specifiers, which we first convert to a string: the reason is explained later.

```

2887 \cs_new_protected:Npn \cs_generate_variant:Nn #1#2
2888 {
2889   \__cs_generate_variant:N #1
2890   \use:x
2891   {
2892     \__cs_generate_variant:nnNN
2893     \cs_split_function:N #1

```

```

2894         \exp_not:N #1
2895         \tl_to_str:n {#2} ,
2896         \exp_not:N \scan_stop: ,
2897         \exp_not:N \q__cs_recursion_stop
2898     }
2899 }
2900 \cs_new_protected:Npn \cs_generate_variant:cn
2901 { \exp_args:Nc \cs_generate_variant:Nn }

```

(End definition for `\cs_generate_variant:Nn`. This function is documented on page 32.)

```

\__cs_generate_variant:N
\__cs_generate_variant:ww
\__cs_generate_variant:wwNw

```

The goal here is to pick up protected parent functions. There are four cases: the parent function can be a primitive or a macro, and can be expandable or not. For non-expandable primitives, all variants should be protected; skipping the `\else:` branch is safe because non-expandable primitives cannot be TeX conditionals.

The other case where variants should be protected is when the parent function is a protected macro: then `protected` appears in the meaning before the first occurrence of `macro`. The `ww` auxiliary removes everything in the meaning string after the first `ma`. We use `ma` rather than the full `macro` because the meaning of the `\firstmark` primitive (and four others) can contain an arbitrary string after a leading `firstmark:`. Then, look for `pr` in the part we extracted: no need to look for anything longer: the only strings we can have are an empty string, `\long_`, `\protected_`, `\protected\long_`, `\first`, `\top`, `\bot`, `\splittop`, or `\splitbot`, with `\` replaced by the appropriate escape character. If `pr` appears in the part before `ma`, the first `\s__cs_mark` is taken as an argument of the `wwNw` auxiliary, and `#3` is `\cs_new_protected:Npx`, otherwise it is `\cs_new:Npx`.

```

2902 \cs_new_protected:Npx \__cs_generate_variant:N #1
2903 {
2904     \exp_not:N \exp_after:wN \exp_not:N \if_meaning:w
2905     \exp_not:N \exp_not:N #1 #1
2906     \cs_set_eq:NN \exp_not:N \__cs_tmp:w \cs_new_protected:Npx
2907     \exp_not:N \else:
2908     \exp_not:N \exp_after:wN \exp_not:N \__cs_generate_variant:ww
2909     \exp_not:N \token_to_meaning:N #1 \tl_to_str:n { ma }
2910     \s__cs_mark
2911     \s__cs_mark \cs_new_protected:Npx
2912     \tl_to_str:n { pr }
2913     \s__cs_mark \cs_new:Npx
2914     \s__cs_stop
2915     \exp_not:N \fi:
2916 }
2917 \exp_last_unbraced:NNNNo
2918 \cs_new_protected:Npn \__cs_generate_variant:ww
2919 #1 { \tl_to_str:n { ma } } #2 \s__cs_mark
2920 { \__cs_generate_variant:wwNw #1 }
2921 \exp_last_unbraced:NNNNo
2922 \cs_new_protected:Npn \__cs_generate_variant:wwNw
2923 #1 { \tl_to_str:n { pr } } #2 \s__cs_mark #3 #4 \s__cs_stop
2924 { \cs_set_eq:NN \__cs_tmp:w #3 }

```

(End definition for `__cs_generate_variant:N`, `__cs_generate_variant:ww`, and `__cs_generate_variant:wwNw`.)

```

\__cs_generate_variant:nnNN #1 : Base name.
                             #2 : Base signature.

```

#3 : Boolean.

#4 : Base function.

If the boolean is `\c_false_bool`, the base function has no colon and we abort with an error; otherwise, set off a loop through the desired variant forms. The original function is retained as #4 for efficiency.

```

2925 \cs_new_protected:Npn \__cs_generate_variant:nnNN #1#2#3#4
2926 {
2927   \if_meaning:w \c_false_bool #3
2928     \msg_error:nnx { kernel } { missing-colon }
2929     { \token_to_str:c {#1} }
2930     \exp_after:wN \__cs_use_none_delimit_by_q_recursion_stop:w
2931   \fi:
2932   \__cs_generate_variant:Nnnw #4 {#1}{#2}
2933 }

```

(End definition for `__cs_generate_variant:nnNN`.)

`__cs_generate_variant:Nnnw`

#1 : Base function.

#2 : Base name.

#3 : Base signature.

#4 : Beginning of variant signature.

First check whether to terminate the loop over variant forms. Then, for each variant form, construct a new function name using the original base name, the variant signature consisting of l letters and the last $k - l$ letters of the base signature (of length k). For example, for a base function `\prop_put:Nnn` which needs a `cV` variant form, we want the new signature to be `cVn`.

There are further subtleties:

- In `\cs_generate_variant:Nn \foo:nnTF {xxTF}`, we must define `\foo:xxTF` using `\exp_args:Nxx`, rather than a hypothetical `\exp_args:NxxTF`. Thus, we wish to trim a common trailing part from the base signature and the variant signature.
- In `\cs_generate_variant:Nn \foo:on {ox}`, the function `\foo:ox` must be defined using `\exp_args:Nnx`, not `\exp_args:Nox`, to avoid double `o` expansion.
- Lastly, `\cs_generate_variant:Nn \foo:on {xn}` must trigger an error, because we do not have a means to replace `o`-expansion by `x`-expansion. More generally, we can only convert `N` to `c`, or convert `n` to `V`, `v`, `o`, `f`, `x`.

All this boils down to a few rules. Only `n` and `N`-type arguments can be replaced by `\cs_generate_variant:Nn`. Other argument types are allowed to be passed unchanged from the base form to the variant: in the process they are changed to `n` except for `N` and `p`-type arguments. A common trailing part is ignored.

We compare the base and variant signatures one character at a time within `x`-expansion. The result is given to `__cs_generate_variant:wwNN` (defined later) in the form `<processed variant signature> \s__cs_mark <errors> \s__cs_stop <base function> <new function>`. If all went well, `<errors>` is empty; otherwise, it is a kernel error message and some clean-up code.

Note the space after #3 and after the following brace group. Those are ignored by `TEX` when fetching the last argument for `__cs_generate_variant_loop:nNwN`, but can be used as a delimiter for `__cs_generate_variant_loop_end:nwwwNNnn`.

```

2934 \cs_new_protected:Npn \__cs_generate_variant:Nnnw #1#2#3#4 ,

```

```

2935 {
2936   \if_meaning:w \scan_stop: #4
2937   \exp_after:wN \__cs_use_none_delimit_by_q_recursion_stop:w
2938   \fi:
2939   \use:x
2940   {
2941     \exp_not:N \__cs_generate_variant:wwNN
2942     \__cs_generate_variant_loop:nNwN { }
2943     #4
2944     \__cs_generate_variant_loop_end:nwwwNNnn
2945     \s__cs_mark
2946     #3 ~
2947     { ~ { } } \fi: \__cs_generate_variant_loop_long:wNNnn } ~
2948     { }
2949     \s__cs_stop
2950     \exp_not:N #1 {#2} {#4}
2951   }
2952   \__cs_generate_variant:Nnnw #1 {#2} {#3}
2953 }

```

(End definition for __cs_generate_variant:Nnnw.)

<pre> __cs_generate_variant_loop:nNwN __cs_generate_variant_loop_base:N __cs_generate_variant_loop_same:w __cs_generate_variant_loop_end:nwwwNNnn __cs_generate_variant_loop_long:wNNnn __cs_generate_variant_loop_invalid:NNwNNnn __cs_generate_variant_loop_special:NNwNNnn </pre>	<pre> #1 : Last few consecutive letters common between the base and variant (more precisely, __cs_generate_variant_same:N <letter> for each letter). #2 : Next variant letter. #3 : Remainder of variant form. #4 : Next base letter. </pre>
---	---

The first argument is populated by __cs_generate_variant_loop_same:w when a variant letter and a base letter match. It is flushed into the input stream whenever the two letters are different: if the loop ends before, the argument is dropped, which means that trailing common letters are ignored.

The case where the two letters are different is only allowed if the base is N and the variant is c, or when the base is n and the variant is o, V, v, f or x. Otherwise, call __cs_generate_variant_loop_invalid:NNwNNnn to remove the end of the loop, get arguments at the end of the loop, and place an appropriate error message as a second argument of __cs_generate_variant:wwNN. If the letters are distinct and the base letter is indeed n or N, leave in the input stream whatever argument #1 was collected, and the next variant letter #2, then loop by calling __cs_generate_variant_loop:nNwN.

The loop can stop in three ways.

- If the end of the variant form is encountered first, #2 is __cs_generate_variant_loop_end:nwwwNNnn (expanded by the conditional \if:w), which inserts some tokens to end the conditional; grabs the *<base name>* as #7, the *<variant signature>* #8, the *<next base letter>* #1 and the part #3 of the base signature that wasn't read yet; and combines those into the *<new function>* to be defined.
- If the end of the base form is encountered first, #4 is ~{} \fi: which ends the conditional (with an empty expansion), followed by __cs_generate_variant_loop_long:wNNnn, which places an error as the second argument of __cs_generate_variant:wwNN.
- The loop can be interrupted early if the requested expansion is unavailable, namely when the variant and base letters differ and the base is not the right one (n or N to

support the variant). In that case too an error is placed as the second argument of
`__cs_generate_variant:wwNN`.

Note that if the variant form has the same length as the base form, #2 is as described in the first point, and #4 as described in the second point above. The `__cs_generate_variant_loop_end:nwwwNNnn` breaking function takes the empty brace group in #4 as its first argument: this empty brace group produces the correct signature for the full variant.

```

2954 \cs_new:Npn \__cs_generate_variant_loop:nNwN #1#2#3 \s__cs_mark #4
2955 {
2956   \if:w #2 #4
2957     \exp_after:wN \__cs_generate_variant_loop_same:w
2958   \else:
2959     \if:w #4 \__cs_generate_variant_loop_base:N #2 \else:
2960       \if:w 0
2961         \if:w N #4 \else: \if:w n #4 \else: 1 \fi: \fi:
2962         \if:w \scan_stop: \__cs_generate_variant_loop_base:N #2 1 \fi:
2963         0
2964         \__cs_generate_variant_loop_special:NNwNNnn #4#2
2965       \else:
2966         \__cs_generate_variant_loop_invalid:NNwNNnn #4#2
2967       \fi:
2968     \fi:
2969   \fi:
2970   #1
2971   \prg_do_nothing:
2972   #2
2973   \__cs_generate_variant_loop:nNwN { } #3 \s__cs_mark
2974 }
2975 \cs_new:Npn \__cs_generate_variant_loop_base:N #1
2976 {
2977   \if:w c #1 N \else:
2978     \if:w o #1 n \else:
2979       \if:w V #1 n \else:
2980         \if:w v #1 n \else:
2981         \if:w f #1 n \else:
2982         \if:w e #1 n \else:
2983         \if:w x #1 n \else:
2984         \if:w n #1 n \else:
2985         \if:w N #1 N \else:
2986         \scan_stop:
2987       \fi:
2988     \fi:
2989   \fi:
2990   \fi:
2991   \fi:
2992   \fi:
2993   \fi:
2994   \fi:
2995   \fi:
2996 }
2997 \cs_new:Npn \__cs_generate_variant_loop_same:w
2998 #1 \prg_do_nothing: #2#3#4
2999 { #3 { #1 \__cs_generate_variant_same:N #2 } }

```



```

3000 \cs_new:Npn \__cs_generate_variant_loop_end:nwwwNNnn
3001   #1#2 \s__cs_mark #3 ~ #4 \s__cs_stop #5#6#7#8
3002   {
3003     \scan_stop: \scan_stop: \fi:
3004     \s__cs_mark \s__cs_stop
3005     \exp_not:N #6
3006     \exp_not:c { #7 : #8 #1 #3 }
3007   }
3008 \cs_new:Npn \__cs_generate_variant_loop_long:wNNnn #1 \s__cs_stop #2#3#4#5
3009   {
3010     \exp_not:n
3011     {
3012       \s__cs_mark
3013       \msg_error:nnxx { kernel } { variant-too-long }
3014       {#5} { \token_to_str:N #3 }
3015       \use_none:nnn
3016       \s__cs_stop
3017       #3
3018       #3
3019     }
3020   }
3021 \cs_new:Npn \__cs_generate_variant_loop_invalid:NNwNNnn
3022   #1#2 \fi: \fi: \fi: #3 \s__cs_stop #4#5#6#7
3023   {
3024     \fi: \fi: \fi:
3025     \exp_not:n
3026     {
3027       \s__cs_mark
3028       \msg_error:nnxxxx { kernel } { invalid-variant }
3029       {#7} { \token_to_str:N #5 } {#1} {#2}
3030       \use_none:nnn
3031       \s__cs_stop
3032       #5
3033       #5
3034     }
3035   }
3036 \cs_new:Npn \__cs_generate_variant_loop_special:NNwNNnn
3037   #1#2#3 \s__cs_stop #4#5#6#7
3038   {
3039     #3 \s__cs_stop #4 #5 {#6} {#7}
3040     \exp_not:n
3041     {
3042       \msg_error:nnxxxx
3043       { kernel } { deprecated-variant }
3044       {#7} { \token_to_str:N #5 } {#1} {#2}
3045     }
3046   }

```

(End definition for __cs_generate_variant_loop:nNwN and others.)

__cs_generate_variant_same:N When the base and variant letters are identical, don't do any expansion. For most argument types, we can use the n-type no-expansion, but the N and p types require a slightly different behaviour with respect to braces. For V-type this function could output N to avoid adding useless braces but that is not a problem.

```

3047 \cs_new:Npn \__cs_generate_variant_same:N #1
3048 {
3049   \if:w N #1 #1 \else:
3050     \if:w p #1 #1 \else:
3051       \token_to_str:N n
3052       \if:w n #1 \else:
3053         \__cs_generate_variant_loop_special:NNwNNnn #1#1
3054       \fi:
3055     \fi:
3056   \fi:
3057 }

```

(End definition for __cs_generate_variant_same:N.)

`__cs_generate_variant:wwNN` If the variant form has already been defined, log its existence (provided `log-functions` is active). Otherwise, make sure that the `\exp_args:N #3` form is defined, and if it contains `x`, change `__cs_tmp:w` locally to `\cs_new_protected:Npx`. Then define the variant by combining the `\exp_args:N #3` variant and the base function.

```

3058 \cs_new_protected:Npn \__cs_generate_variant:wwNN
3059   #1 \s__cs_mark #2 \s__cs_stop #3#4
3060 {
3061   #2
3062   \cs_if_free:NT #4
3063   {
3064     \group_begin:
3065       \__cs_generate_internal_variant:n {#1}
3066       \__cs_tmp:w #4 { \exp_not:c { \exp_args:N #1 } \exp_not:N #3 }
3067     \group_end:
3068   }
3069 }

```

(End definition for __cs_generate_variant:wwNN.)

`__cs_generate_internal_variant:n`
`__cs_generate_internal_variant_loop:n`

First test for the presence of `x` (this is where working with strings makes our lives easier), as the result should be protected, and the next variant to be defined using that internal variant should be protected (done by setting `__cs_tmp:w`). Then call `__cs_generate_internal_variant:NNn` with arguments `\cs_new_protected:cpn \use:x` (for protected) or `\cs_new:cpn \tex_expanded:D` (expandable) and the signature. If `p` appears in the signature, or if the function to be defined is expandable and the primitive `\expanded` is not available, or if there are more than 8 arguments, call some fall-back code that just puts the appropriate `\::` commands. Otherwise, call `__cs_generate_internal_one_go:NNn` to construct the `\exp_args:N...` function as a macro taking up to 9 arguments and expanding them using `\use:x` or `\tex_expanded:D`.

```

3070 \cs_new_protected:Npx \__cs_generate_internal_variant:n #1
3071 {
3072   \exp_not:N \__cs_generate_internal_variant:wwnNwn
3073   #1 \s__cs_mark
3074   { \cs_set_eq:NN \exp_not:N \__cs_tmp:w \cs_new_protected:Npx }
3075   \cs_new_protected:cpn
3076   \use:x
3077   \token_to_str:N x \s__cs_mark
3078   { }
3079   \cs_new:cpn

```

```

3080         \exp_not:N \tex_expanded:D
3081     \s__cs_stop
3082     {#1}
3083 }
3084 \exp_last_unbraced:NNNNo
3085 \cs_new_protected:Npn \__cs_generate_internal_variant:wwnNwn #1
3086 { \token_to_str:N x } #2 \s__cs_mark #3#4#5#6 \s__cs_stop #7
3087 {
3088     #3
3089     \cs_if_free:cT { exp_args:N #7 }
3090     { \__cs_generate_internal_variant:NNn #4 #5 {#7} }
3091 }
3092 \cs_set_protected:Npn \__cs_tmp:w #1
3093 {
3094     \cs_new_protected:Npn \__cs_generate_internal_variant:NNn ##1##2##3
3095     {
3096         \if_catcode:w X \use_none:nnnnnnnn ##3
3097             \prg_do_nothing: \prg_do_nothing: \prg_do_nothing:
3098             \prg_do_nothing: \prg_do_nothing: \prg_do_nothing:
3099             \prg_do_nothing: \prg_do_nothing: X
3100             \exp_after:wN \__cs_generate_internal_test:Nw \exp_after:wN ##2
3101         \else:
3102             \exp_after:wN \__cs_generate_internal_test_aux:w \exp_after:wN #1
3103         \fi:
3104         ##3
3105         \s__cs_mark
3106         {
3107             \use:x
3108             {
3109                 ##1 { exp_args:N ##3 }
3110                 { \__cs_generate_internal_variant_loop:n ##3 { : \use_i:nn } }
3111             }
3112         }
3113         #1
3114         \s__cs_mark
3115         { \exp_not:n { \__cs_generate_internal_one_go:NNn ##1 ##2 {##3} } }
3116         \s__cs_stop
3117     }
3118 \cs_new_protected:Npn \__cs_generate_internal_test_aux:w
3119     ##1 #1 ##2 \s__cs_mark ##3 ##4 \s__cs_stop {##3}
3120 \cs_if_exist:NTF \tex_expanded:D
3121 {
3122     \cs_new_eq:NN \__cs_generate_internal_test:Nw
3123     \__cs_generate_internal_test_aux:w
3124 }
3125 {
3126     \cs_new_protected:Npn \__cs_generate_internal_test:Nw ##1
3127     {
3128         \if_meaning:w \tex_expanded:D ##1
3129             \exp_after:wN \__cs_generate_internal_test_aux:w
3130             \exp_after:wN #1
3131         \else:
3132             \exp_after:wN \__cs_generate_internal_test_aux:w
3133         \fi:

```

```

3134     }
3135 }
3136 }
3137 \exp_args:No \__cs_tmp:w { \token_to_str:N p }
3138 \cs_new_protected:Npn \__cs_generate_internal_one_go:NNn #1#2#3
3139 {
3140     \__cs_generate_internal_loop:nwnnw
3141     { \exp_not:N ##1 } 1 . { } { }
3142     #3 { ? \__cs_generate_internal_end:w } X ;
3143     23456789 { ? \__cs_generate_internal_long:w } ;
3144     #1 #2 {#3}
3145 }
3146 \cs_new_protected:Npn \__cs_generate_internal_loop:nwnnw #1#2 . #3#4#5#6 ; #7
3147 {
3148     \use_none:n #5
3149     \use_none:n #7
3150     \cs_if_exist_use:cF { __cs_generate_internal_#5:NN }
3151     { \__cs_generate_internal_other:NN }
3152     #5 #7
3153     #7 .
3154     { #3 #1 } { #4 ## #2 }
3155     #6 ;
3156 }
3157 \cs_new_protected:Npn \__cs_generate_internal_N:NN #1#2
3158 { \__cs_generate_internal_loop:nwnnw { \exp_not:N ###2 } }
3159 \cs_new_protected:Npn \__cs_generate_internal_c:NN #1#2
3160 { \exp_args:No \__cs_generate_internal_loop:nwnnw { \exp_not:c {###2} } }
3161 \cs_new_protected:Npn \__cs_generate_internal_n:NN #1#2
3162 { \__cs_generate_internal_loop:nwnnw { { \exp_not:n {###2} } } }
3163 \cs_new_protected:Npn \__cs_generate_internal_x:NN #1#2
3164 { \__cs_generate_internal_loop:nwnnw { {###2} } }
3165 \cs_new_protected:Npn \__cs_generate_internal_other:NN #1#2
3166 {
3167     \exp_args:No \__cs_generate_internal_loop:nwnnw
3168     {
3169         \exp_after:wN
3170         {
3171             \exp:w \exp_args:NNc \exp_after:wN \exp_end:
3172             { exp_not:#1 } {###2}
3173         }
3174     }
3175 }
3176 \cs_new_protected:Npn \__cs_generate_internal_end:w #1 . #2#3#4 ; #5 ; #6#7#8
3177 { #6 { exp_args:N #8 } #3 { #7 {#2} } }
3178 \cs_new_protected:Npn \__cs_generate_internal_long:w #1 N #2#3 . #4#5#6#
3179 {
3180     \exp_args:Nx \__cs_generate_internal_long:nnnNNn
3181     { \__cs_generate_internal_variant_loop:n #2 #6 { : \use_i:nn } }
3182     {#4} {#5}
3183 }
3184 \cs_new:Npn \__cs_generate_internal_long:nnnNNn #1#2#3#4 ; ; #5#6#7
3185 { #5 { exp_args:N #7 } #3 { #6 { \exp_not:n {#1} {#2} } } }

```

This command grabs char by char outputting \: #1 (not expanded further). We avoid tests by putting a trailing : \use_i:nn, which leaves \cs_end: and removes the looping

macro. The colon is in fact also turned into \::: so that the required structure for \exp_args:N... commands is correctly terminated.

```

3186 \cs_new:Npn \__cs_generate_internal_variant_loop:n #1
3187 {
3188     \exp_after:wN \exp_not:N \cs:w :: #1 \cs_end:
3189     \__cs_generate_internal_variant_loop:n
3190 }

```

(End definition for __cs_generate_internal_variant:n and __cs_generate_internal_variant_loop:n.)

\prg_generate_conditional_variant:Nnn

```

\__cs_generate_variant:nnNnn
\__cs_generate_variant:w
\__cs_generate_variant:n
\__cs_generate_variant_p_form:nnn
\__cs_generate_variant_T_form:nnn
\__cs_generate_variant_F_form:nnn
\__cs_generate_variant_TF_form:nnn

```

```

3191 \cs_new_protected:Npn \prg_generate_conditional_variant:Nnn #1
3192 {
3193     \use:x
3194     {
3195         \__cs_generate_variant:nnNnn
3196         \cs_split_function:N #1
3197     }
3198 }
3199 \cs_new_protected:Npn \__cs_generate_variant:nnNnn #1#2#3#4#5
3200 {
3201     \if_meaning:w \c_false_bool #3
3202     \msg_error:nnx { kernel } { missing-colon }
3203     { \token_to_str:c {#1} }
3204     \__cs_use_i_delimit_by_s_stop:nw
3205     \fi:
3206     \exp_after:wN \__cs_generate_variant:w
3207     \tl_to_str:n {#5} , \scan_stop: , \q__cs_recursion_stop
3208     \__cs_use_none_delimit_by_s_stop:w \s__cs_mark {#1} {#2} {#4} \s__cs_stop
3209 }
3210 \cs_new_protected:Npn \__cs_generate_variant:w
3211 #1 , #2 \s__cs_mark #3#4#5
3212 {
3213     \if_meaning:w \scan_stop: #1 \scan_stop:
3214     \if_meaning:w \q__cs_nil #1 \q__cs_nil
3215     \use_i:nnn
3216     \fi:
3217     \exp_after:wN \__cs_use_none_delimit_by_q_recursion_stop:w
3218     \else:
3219     \cs_if_exist_use:cTF { __cs_generate_variant_#1_form:nnn }
3220     { {#3} {#4} {#5} }
3221     {
3222         \msg_error:nnxx
3223         { kernel } { conditional-form-unknown }
3224         {#1} { \token_to_str:c { #3 : #4 } }
3225     }
3226     \fi:
3227     \__cs_generate_variant:w #2 \s__cs_mark {#3} {#4} {#5}
3228 }
3229 \cs_new_protected:Npn \__cs_generate_variant_p_form:nnn #1#2
3230 { \cs_generate_variant:cn { #1_p : #2 } }
3231 \cs_new_protected:Npn \__cs_generate_variant_T_form:nnn #1#2
3232 { \cs_generate_variant:cn { #1 : #2 T } }

```

```

3233 \cs_new_protected:Npn \__cs_generate_variant_F_form:nnn #1#2
3234 { \cs_generate_variant:cn { #1 : #2 F } }
3235 \cs_new_protected:Npn \__cs_generate_variant_TF_form:nnn #1#2
3236 { \cs_generate_variant:cn { #1 : #2 TF } }

```

(End definition for \prg_generate_conditional_variant:Nnn and others. This function is documented on page 64.)

\exp_args_generate:n This function is not used in the kernel hence we can use functions that are defined in later modules. It also does not need to be fast so use inline mappings. For each requested variant we check that there are no characters besides NnpcofVvx, in particular that there are no spaces. Then we just call the internal function.

```

3237 \cs_new_protected:Npn \exp_args_generate:n #1
3238 {
3239   \exp_args:No \clist_map_inline:nn { \tl_to_str:n {#1} }
3240   {
3241     \str_map_inline:nn {##1}
3242     {
3243       \str_if_in:nnF { NnpcofVvx } {####1}
3244       {
3245         \msg_error:nnnn { kernel } { invalid-exp-args }
3246         {####1} {##1}
3247         \str_map_break:n { \use_none:nn }
3248       }
3249     }
3250     \__cs_generate_internal_variant:n {##1}
3251   }
3252 }

```

(End definition for \exp_args_generate:n. This function is documented on page 301.)

42.8 Definitions with the automated technique

Some of these could be done more efficiently, but the complexity of coding then becomes an issue. Notice that the auto-generated functions actually take no arguments themselves.

\exp_args:Nnc Here are the actual function definitions, using the helper functions above. The group is used because __cs_generate_internal_variant:n redefines __cs_tmp:w locally.

```

\exp_args:Nnc
\exp_args:Nno
\exp_args:NnV
\exp_args:Nnv
\exp_args:Nne
\exp_args:Nnf
\exp_args:Noc
\exp_args:Noo
\exp_args:Nof
\exp_args:NVo
\exp_args:Nfo
\exp_args:Nff
\exp_args:Nee
\exp_args:NNx
\exp_args:Ncx
\exp_args:Nnx
\exp_args:Nox
\exp_args:Nxo
\exp_args:Nxx

```

```

3253 \cs_set_protected:Npn \__cs_tmp:w #1
3254 {
3255   \group_begin:
3256     \exp_args:No \__cs_generate_internal_variant:n
3257     { \tl_to_str:n {#1} }
3258   \group_end:
3259 }
3260 \__cs_tmp:w { nc }
3261 \__cs_tmp:w { no }
3262 \__cs_tmp:w { nV }
3263 \__cs_tmp:w { nv }
3264 \__cs_tmp:w { ne }
3265 \__cs_tmp:w { nf }
3266 \__cs_tmp:w { oc }
3267 \__cs_tmp:w { oo }

```

```

3268 \__cs_tmp:w { of }
3269 \__cs_tmp:w { Vo }
3270 \__cs_tmp:w { fo }
3271 \__cs_tmp:w { ff }
3272 \__cs_tmp:w { ee }
3273 \__cs_tmp:w { Nx }
3274 \__cs_tmp:w { cx }
3275 \__cs_tmp:w { nx }
3276 \__cs_tmp:w { ox }
3277 \__cs_tmp:w { xo }
3278 \__cs_tmp:w { xx }

```

(End definition for `\exp_args:Nnc` and others. These functions are documented on page 36.)

```

\exp_args:NNcf
\exp_args:NNno
\exp_args:NNnV
\exp_args:NNoo
\exp_args:NNVV
\exp_args:Ncno
\exp_args:NcnV
\exp_args:Ncoo
\exp_args:NcVV
\exp_args:Nnnc
\exp_args:Nnno
\exp_args:Nnnf
\exp_args:Nnff
\exp_args:Nooo
\exp_args:Noof
\exp_args:Nffo
\exp_args:Neee
\exp_args:NNNx
\exp_args:NNnx
\exp_args:NNox
\exp_args:Nccx
\exp_args:Ncnx
\exp_args:Nnnx
\exp_args:Nnox
\exp_args:Noox

```

```

3279 \__cs_tmp:w { Ncf }
3280 \__cs_tmp:w { Nno }
3281 \__cs_tmp:w { NnV }
3282 \__cs_tmp:w { Noo }
3283 \__cs_tmp:w { NVV }
3284 \__cs_tmp:w { cno }
3285 \__cs_tmp:w { cnV }
3286 \__cs_tmp:w { coo }
3287 \__cs_tmp:w { cVV }
3288 \__cs_tmp:w { nnc }
3289 \__cs_tmp:w { nno }
3290 \__cs_tmp:w { nnf }
3291 \__cs_tmp:w { nff }
3292 \__cs_tmp:w { ooo }
3293 \__cs_tmp:w { oof }
3294 \__cs_tmp:w { ffo }
3295 \__cs_tmp:w { eee }
3296 \__cs_tmp:w { NNx }
3297 \__cs_tmp:w { Nnx }
3298 \__cs_tmp:w { Nox }
3299 \__cs_tmp:w { nnx }
3300 \__cs_tmp:w { nox }
3301 \__cs_tmp:w { ccx }
3302 \__cs_tmp:w { cnx }
3303 \__cs_tmp:w { oox }

```

(End definition for `\exp_args:NNcf` and others. These functions are documented on page 37.)

```

3304 </package>

```

Chapter 43

l3sort implementation

```
3305 <*package>
3306 <@@=sort>
```

43.1 Variables

`\g__sort_internal_seq` Sorting happens in a group; the result is stored in those global variables before being copied outside the group to the proper places. For `seq` and `tl` this is more efficient than using `\use:x` (or some `\exp_args:NNNx`) to smuggle the definition outside the group since \TeX does not need to re-read tokens. For `clist` we don't gain anything since the result is converted from `seq` to `clist` anyways.

`\g__sort_internal_tl`

```
3307 \seq_new:N \g__sort_internal_seq
3308 \tl_new:N \g__sort_internal_tl
```

(End definition for `\g__sort_internal_seq` and `\g__sort_internal_tl`.)

`\l__sort_length_int` The sequence has `\l__sort_length_int` items and is stored from `\l__sort_min_int` to `\l__sort_top_int - 1`. While reading the sequence in memory, we check that `\l__sort_top_int` remains at most `\l__sort_max_int`, precomputed by `__sort_compute_range:.` That bound is such that the merge sort only uses `\toks` registers less than `\l__sort_true_max_int`, namely those that have not been allocated for use in other code: the user's comparison code could alter these.

`\l__sort_min_int`

`\l__sort_top_int`

`\l__sort_max_int`

`\l__sort_true_max_int`

```
3309 \int_new:N \l__sort_length_int
3310 \int_new:N \l__sort_min_int
3311 \int_new:N \l__sort_top_int
3312 \int_new:N \l__sort_max_int
3313 \int_new:N \l__sort_true_max_int
```

(End definition for `\l__sort_length_int` and others.)

`\l__sort_block_int` Merge sort is done in several passes. In each pass, blocks of size `\l__sort_block_int` are merged in pairs. The block size starts at 1, and, for a length in the range $[2^k + 1, 2^{k+1}]$, reaches 2^k in the last pass.

```
3314 \int_new:N \l__sort_block_int
```

(End definition for `\l__sort_block_int`.)

`\l__sort_begin_int` When merging two blocks, `\l__sort_begin_int` marks the lowest index in the two blocks, and `\l__sort_end_int` marks the highest index, plus 1.

```
3315 \int_new:N \l__sort_begin_int
3316 \int_new:N \l__sort_end_int
```

(End definition for `\l__sort_begin_int` and `\l__sort_end_int`.)

`\l__sort_A_int` When merging two blocks (whose end-points are `beg` and `end`), A starts from the high end of the low block, and decreases until reaching `beg`. The index B starts from the top of the range and marks the register in which a sorted item should be put. Finally, C points to the copy of the high block in the interval of registers starting at `\l__sort_length_int`, upwards. C starts from the upper limit of that range.

```
3317 \int_new:N \l__sort_A_int
3318 \int_new:N \l__sort_B_int
3319 \int_new:N \l__sort_C_int
```

(End definition for `\l__sort_A_int`, `\l__sort_B_int`, and `\l__sort_C_int`.)

`\s__sort_mark` Internal scan marks.

```
\s__sort_stop 3320 \scan_new:N \s__sort_mark
3321 \scan_new:N \s__sort_stop
```

(End definition for `\s__sort_mark` and `\s__sort_stop`.)

43.2 Finding available `\toks` registers

`__sort_shrink_range:` After `__sort_compute_range:` (defined below) determines that `\toks` registers between `\l__sort_min_int` (included) and `\l__sort_true_max_int` (excluded) have not yet been assigned, `__sort_shrink_range:` computes `\l__sort_max_int` to reflect the need for a buffer when merging blocks in the merge sort. Given $2^n \leq A \leq 2^n + 2^{n-1}$ registers we can sort $\lfloor A/2 \rfloor + 2^{n-2}$ items while if we have $2^n + 2^{n-1} \leq A \leq 2^{n+1}$ registers we can sort $A - 2^{n-1}$ items. We first find out a power 2^n such that $2^n \leq A \leq 2^{n+1}$ by repeatedly halving `\l__sort_block_int`, starting at 2^{15} or 2^{14} namely half the total number of registers, then we use the formulas and set `\l__sort_max_int`.

```
3322 \cs_new_protected:Npn \__sort_shrink_range:
3323 {
3324   \int_set:Nn \l__sort_A_int
3325     { \l__sort_true_max_int - \l__sort_min_int + 1 }
3326   \int_set:Nn \l__sort_block_int { \c_max_register_int / 2 }
3327   \__sort_shrink_range_loop:
3328   \int_set:Nn \l__sort_max_int
3329     {
3330     \int_compare:nNnTF
3331       { \l__sort_block_int * 3 / 2 } > \l__sort_A_int
3332       {
3333         \l__sort_min_int
3334         + ( \l__sort_A_int - 1 ) / 2
3335         + \l__sort_block_int / 4
3336         - 1
3337       }
3338       { \l__sort_true_max_int - \l__sort_block_int / 2 }
3339     }
```

```

3340 }
3341 \cs_new_protected:Npn \__sort_shrink_range_loop:
3342 {
3343   \if_int_compare:w \l__sort_A_int < \l__sort_block_int
3344     \tex_divide:D \l__sort_block_int 2 \exp_stop_f:
3345     \exp_after:wN \__sort_shrink_range_loop:
3346   \fi:
3347 }

```

(End definition for __sort_shrink_range: and __sort_shrink_range_loop:.)

__sort_compute_range: First find out what \toks have not yet been assigned. There are many cases. In L^AT_EX 2_ε with no package, available \toks range from \count15 + 1 to \c_max_register_int included (this was not altered despite the 2015 changes). When \loctoks is defined, namely in plain (e)T_EX, or when the package etex is loaded in L^AT_EX 2_ε, redefine __sort_compute_range: to use the range \count265 to \count275 − 1. The elocalloc package also defines \loctoks but uses yet another number for the upper bound, namely \e@alloc@top (minus one). We must check for \loctoks every time a sorting function is called, as etex or elocalloc could be loaded.

In ConT_EXt MkIV the range is from \c_syst_last_allocated_toks+1 to \c_max_register_int, and in MkII it is from \lastallocatedtoks+1 to \c_max_register_int. In all these cases, call __sort_shrink_range:.

```

3348 \cs_new_protected:Npn \__sort_compute_range:
3349 {
3350   \int_set:Nn \l__sort_min_int { \tex_count:D 15 + 1 }
3351   \int_set:Nn \l__sort_true_max_int { \c_max_register_int + 1 }
3352   \__sort_shrink_range:
3353   \if_meaning:w \loctoks \tex_undefined:D \else:
3354     \if_meaning:w \loctoks \scan_stop: \else:
3355       \__sort_redefine_compute_range:
3356       \__sort_compute_range:
3357     \fi:
3358   \fi:
3359 }
3360 \cs_new_protected:Npn \__sort_redefine_compute_range:
3361 {
3362   \cs_if_exist:cTF { ver@elocalloc.sty }
3363   {
3364     \cs_gset_protected:Npn \__sort_compute_range:
3365     {
3366       \int_set:Nn \l__sort_min_int { \tex_count:D 265 }
3367       \int_set_eq:NN \l__sort_true_max_int \e@alloc@top
3368       \__sort_shrink_range:
3369     }
3370   }
3371   {
3372     \cs_gset_protected:Npn \__sort_compute_range:
3373     {
3374       \int_set:Nn \l__sort_min_int { \tex_count:D 265 }
3375       \int_set:Nn \l__sort_true_max_int { \tex_count:D 275 }
3376       \__sort_shrink_range:
3377     }
3378   }

```

```

3379 }
3380 \cs_if_exist:NT \loctoks { \__sort_redefine_compute_range: }
3381 \tl_map_inline:nn { \lastallocatedtoks \c_syst_last_allocated_toks }
3382 {
3383   \cs_if_exist:NT #1
3384   {
3385     \cs_gset_protected:Npn \__sort_compute_range:
3386     {
3387       \int_set:Nn \l__sort_min_int { #1 + 1 }
3388       \int_set:Nn \l__sort_true_max_int { \c_max_register_int + 1 }
3389       \__sort_shrink_range:
3390     }
3391   }
3392 }

```

(End definition for `__sort_compute_range:`, `__sort_redefine_compute_range:`, and `\c__sort_max_length_int`.)

43.3 Protected user commands

`__sort_main:NNNn` Sorting happens in three steps. First store items in `\toks` registers ranging from `\l__sort_min_int` to `\l__sort_top_int - 1`, while checking that the list is not too long. If we reach the maximum length, that's an error; exit the group. Secondly, sort the array of `\toks` registers, using the user-defined sorting function: `__sort_level:` calls `__sort_compare:nn` as needed. Finally, unpack the `\toks` registers (now sorted) into the target `tl`, or into `\g__sort_internal_seq` for `seq` and `clist`. This is done by `__sort_seq:NNNNn` and `__sort_tl:NNn`.

```

3393 \cs_new_protected:Npn \__sort_main:NNNn #1#2#3#4
3394 {
3395   \__sort_disable_toksdef:
3396   \__sort_compute_range:
3397   \int_set_eq:NN \l__sort_top_int \l__sort_min_int
3398   #1 #3
3399   {
3400     \if_int_compare:w \l__sort_top_int = \l__sort_max_int
3401       \__sort_too_long_error:NNw #2 #3
3402     \fi:
3403     \tex_toks:D \l__sort_top_int {##1}
3404     \int_incr:N \l__sort_top_int
3405   }
3406   \int_set:Nn \l__sort_length_int
3407   { \l__sort_top_int - \l__sort_min_int }
3408   \cs_set:Npn \__sort_compare:nn ##1 ##2 {#4}
3409   \int_set:Nn \l__sort_block_int { 1 }
3410   \__sort_level:
3411 }

```

(End definition for `__sort_main:NNNn`.)

`\tl_sort:Nn` Call the main sorting function then unpack `\toks` registers outside the group into the target token list. The unpacking is done by `__sort_tl_toks:w`; registers are numbered from `\l__sort_min_int` to `\l__sort_top_int - 1`. For expansion behaviour we need

`\tl_gsort:Nn`

`\tl_gsort:cn`

`__sort_tl:NNn`

`__sort_tl_toks:w`

a couple of primitives. The `\tl_gclear:N` reduces memory usage. The `\prg_break_point:` is used by `__sort_main:NNNn` when the list is too long.

```

3412 \cs_new_protected:Npn \tl_sort:Nn { \__sort_tl:NNn \tl_set_eq:NN }
3413 \cs_generate_variant:Nn \tl_sort:Nn { c }
3414 \cs_new_protected:Npn \tl_gsort:Nn { \__sort_tl:NNn \tl_gset_eq:NN }
3415 \cs_generate_variant:Nn \tl_gsort:Nn { c }
3416 \cs_new_protected:Npn \__sort_tl:NNn #1#2#3
3417 {
3418   \group_begin:
3419     \__sort_main:NNNn \tl_map_inline:Nn \tl_map_break:n #2 {#3}
3420     \__kernel_tl_gset:Nx \g__sort_internal_tl
3421     { \__sort_tl_toks:w \l__sort_min_int ; }
3422   \group_end:
3423   #1 #2 \g__sort_internal_tl
3424   \tl_gclear:N \g__sort_internal_tl
3425   \prg_break_point:
3426 }
3427 \cs_new:Npn \__sort_tl_toks:w #1 ;
3428 {
3429   \if_int_compare:w #1 < \l__sort_top_int
3430     { \tex_the:D \tex_toks:D #1 }
3431     \exp_after:wN \__sort_tl_toks:w
3432     \int_value:w \int_eval:n { #1 + 1 } \exp_after:wN ;
3433   \fi:
3434 }

```

(End definition for `\tl_sort:Nn` and others. These functions are documented on page 116.)

<code>\seq_sort:Nn</code> <code>\seq_sort:cn</code> <code>\seq_gsort:Nn</code> <code>\seq_gsort:cn</code> <code>\clist_sort:Nn</code> <code>\clist_sort:cn</code> <code>\clist_gsort:Nn</code> <code>\clist_gsort:cn</code> <code>__sort_seq:NNNNn</code>	<p>Use the same general framework for seq and clist. Apply the general sorting code, then unpack <code>\toks</code> into <code>\g__sort_internal_seq</code>. Outside the group copy or convert (for clist) the data to the target variable. The <code>\seq_gclear:N</code> reduces memory usage. The <code>\prg_break_point:</code> is used by <code>__sort_main:NNNn</code> when the list is too long.</p> <pre> 3435 \cs_new_protected:Npn \seq_sort:Nn 3436 { __sort_seq:NNNNn \seq_map_inline:Nn \seq_map_break:n \seq_set_eq:NN } 3437 \cs_generate_variant:Nn \seq_sort:Nn { c } 3438 \cs_new_protected:Npn \seq_gsort:Nn 3439 { __sort_seq:NNNNn \seq_map_inline:Nn \seq_map_break:n \seq_gset_eq:NN } 3440 \cs_generate_variant:Nn \seq_gsort:Nn { c } 3441 \cs_new_protected:Npn \clist_sort:Nn 3442 { 3443 __sort_seq:NNNNn \clist_map_inline:Nn \clist_map_break:n 3444 \clist_set_from_seq:NN 3445 } 3446 \cs_generate_variant:Nn \clist_sort:Nn { c } 3447 \cs_new_protected:Npn \clist_gsort:Nn 3448 { 3449 __sort_seq:NNNNn \clist_map_inline:Nn \clist_map_break:n 3450 \clist_gset_from_seq:NN 3451 } 3452 \cs_generate_variant:Nn \clist_gsort:Nn { c } 3453 \cs_new_protected:Npn __sort_seq:NNNNn #1#2#3#4#5 3454 { 3455 \group_begin: 3456 __sort_main:NNNn #1 #2 #4 {#5} </pre>
--	--

```

3457 \seq_gset_from_inline_x:Nnn \g__sort_internal_seq
3458 {
3459   \int_step_function:nnN
3460     { \l__sort_min_int } { \l__sort_top_int - 1 }
3461 }
3462 { \tex_the:D \tex_toks:D ##1 }
3463 \group_end:
3464 #3 #4 \g__sort_internal_seq
3465 \seq_gclear:N \g__sort_internal_seq
3466 \prg_break_point:
3467 }

```

(End definition for `\seq_sort:Nn` and others. These functions are documented on page 147.)

43.4 Merge sort

`__sort_level:` This function is called once blocks of size `\l__sort_block_int` (initially 1) are each sorted. If the whole list fits in one block, then we are done (this also takes care of the case of an empty list or a list with one item). Otherwise, go through pairs of blocks starting from 0, then double the block size, and repeat.

```

3468 \cs_new_protected:Npn \__sort_level:
3469 {
3470   \if_int_compare:w \l__sort_block_int < \l__sort_length_int
3471     \l__sort_end_int \l__sort_min_int
3472     \__sort_merge_blocks:
3473     \tex_advance:D \l__sort_block_int \l__sort_block_int
3474     \exp_after:wN \__sort_level:
3475   \fi:
3476 }

```

(End definition for `__sort_level:.`)

`__sort_merge_blocks:` This function is called to merge a pair of blocks, starting at the last value of `\l__sort_end_int` (end-point of the previous pair of blocks). If shifting by one block to the right we reach the end of the list, then this pass has ended: the end of the list is sorted already. Otherwise, store the result of that shift in *A*, which indexes the first block starting from the top end. Then locate the end-point (maximum) of the second block: shift *end* upwards by one more block, but keeping it \leq *top*. Copy this upper block of `\toks` registers in registers above *length*, indexed by *C*: this is covered by `__sort_copy_block:.` Once this is done we are ready to do the actual merger using `__sort_merge_blocks_aux:`, after shifting *A*, *B* and *C* so that they point to the largest index in their respective ranges rather than pointing just beyond those ranges. Of course, once that pair of blocks is merged, move on to the next pair.

```

3477 \cs_new_protected:Npn \__sort_merge_blocks:
3478 {
3479   \l__sort_begin_int \l__sort_end_int
3480   \tex_advance:D \l__sort_end_int \l__sort_block_int
3481   \if_int_compare:w \l__sort_end_int < \l__sort_top_int
3482     \l__sort_A_int \l__sort_end_int
3483     \tex_advance:D \l__sort_end_int \l__sort_block_int
3484     \if_int_compare:w \l__sort_end_int > \l__sort_top_int
3485       \l__sort_end_int \l__sort_top_int

```

```

3486     \fi:
3487     \l__sort_B_int \l__sort_A_int
3488     \l__sort_C_int \l__sort_top_int
3489     \__sort_copy_block:
3490     \int_decr:N \l__sort_A_int
3491     \int_decr:N \l__sort_B_int
3492     \int_decr:N \l__sort_C_int
3493     \exp_after:wN \__sort_merge_blocks_aux:
3494     \exp_after:wN \__sort_merge_blocks:
3495     \fi:
3496 }

```

(End definition for __sort_merge_blocks:.)

__sort_copy_block: We wish to store a copy of the “upper” block of \toks registers, ranging between the initial value of \l__sort_B_int (included) and \l__sort_end_int (excluded) into a new range starting at the initial value of \l__sort_C_int, namely \l__sort_top_int.

```

3497 \cs_new_protected:Npn \__sort_copy_block:
3498 {
3499     \tex_toks:D \l__sort_C_int \tex_toks:D \l__sort_B_int
3500     \int_incr:N \l__sort_C_int
3501     \int_incr:N \l__sort_B_int
3502     \if_int_compare:w \l__sort_B_int = \l__sort_end_int
3503         \use_i:nn
3504     \fi:
3505     \__sort_copy_block:
3506 }

```

(End definition for __sort_copy_block:.)

__sort_merge_blocks_aux: At this stage, the first block starts at \l__sort_begin_int, and ends at \l__sort_A_int, and the second block starts at \l__sort_top_int and ends at \l__sort_C_int. The result of the merger is stored at positions indexed by \l__sort_B_int, which starts at \l__sort_end_int − 1 and decreases down to \l__sort_begin_int, covering the full range of the two blocks. In other words, we are building the merger starting with the largest values. The comparison function is defined to return either **swapped** or **same**. Of course, this means the arguments need to be given in the order they appear originally in the list.

```

3507 \cs_new_protected:Npn \__sort_merge_blocks_aux:
3508 {
3509     \exp_after:wN \__sort_compare:nn \exp_after:wN
3510     { \tex_the:D \tex_toks:D \exp_after:wN \l__sort_A_int \exp_after:wN }
3511     \exp_after:wN { \tex_the:D \tex_toks:D \l__sort_C_int }
3512     \prg_do_nothing:
3513     \__sort_return_mark:w
3514     \__sort_return_mark:w
3515     \s__sort_mark
3516     \__sort_return_none_error:
3517 }

```

(End definition for __sort_merge_blocks_aux:.)

`\sort_return_same:` Each comparison should call `\sort_return_same:` or `\sort_return_swapped:` exactly once. If neither is called, `__sort_return_none_error:` is called, since the `return_mark` removes tokens until `\s__sort_mark`. If one is called, the `return_mark` auxiliary removes everything except `__sort_return_same:w` (or its `swapped` analogue) followed by `__sort_return_none_error:`. Finally if two or more are called, `__sort_return_two_error:` ends up before any `__sort_return_mark:w`, so that it produces an error.

```

3518 \cs_new_protected:Npn \sort_return_same:
3519   #1 \__sort_return_mark:w #2 \s__sort_mark
3520   {
3521     #1
3522     #2
3523     \__sort_return_two_error:
3524     \__sort_return_mark:w
3525     \s__sort_mark
3526     \__sort_return_same:w
3527   }
3528 \cs_new_protected:Npn \sort_return_swapped:
3529   #1 \__sort_return_mark:w #2 \s__sort_mark
3530   {
3531     #1
3532     #2
3533     \__sort_return_two_error:
3534     \__sort_return_mark:w
3535     \s__sort_mark
3536     \__sort_return_swapped:w
3537   }
3538 \cs_new_protected:Npn \__sort_return_mark:w #1 \s__sort_mark { }
3539 \cs_new_protected:Npn \__sort_return_none_error:
3540   {
3541     \msg_error:nnxx { sort } { return-none }
3542     { \tex_the:D \tex_toks:D \l__sort_A_int }
3543     { \tex_the:D \tex_toks:D \l__sort_C_int }
3544     \__sort_return_same:w \__sort_return_none_error:
3545   }
3546 \cs_new_protected:Npn \__sort_return_two_error:
3547   {
3548     \msg_error:nnxx { sort } { return-two }
3549     { \tex_the:D \tex_toks:D \l__sort_A_int }
3550     { \tex_the:D \tex_toks:D \l__sort_C_int }
3551   }

```

(End definition for `\sort_return_same:` and others. These functions are documented on page 44.)

`__sort_return_same:w` If the comparison function returns `same`, then the second argument fed to `__sort_compare:nn` should remain to the right of the other one. Since we build the merger starting from the right, we copy that `\toks` register into the allotted range, then shift the pointers *B* and *C*, and go on to do one more step in the merger, unless the second block has been exhausted: then the remainder of the first block is already in the correct registers and we are done with merging those two blocks.

```

3552 \cs_new_protected:Npn \__sort_return_same:w #1 \__sort_return_none_error:
3553   {
3554     \tex_toks:D \l__sort_B_int \tex_toks:D \l__sort_C_int
3555     \int_decr:N \l__sort_B_int

```

```

3556     \int_decr:N \l__sort_C_int
3557     \if_int_compare:w \l__sort_C_int < \l__sort_top_int
3558         \use_i:nn
3559     \fi:
3560     \__sort_merge_blocks_aux:
3561 }

```

(End definition for __sort_return_same:w.)

__sort_return_swapped:w If the comparison function returns **swapped**, then the next item to add to the merger is the first argument, contents of the \toks register *A*. Then shift the pointers *A* and *B* to the left, and go for one more step for the merger, unless the left block was exhausted (*A* goes below the threshold). In that case, all remaining \toks registers in the second block, indexed by *C*, are copied to the merger by __sort_merge_blocks_end:.

```

3562 \cs_new_protected:Npn \__sort_return_swapped:w #1 \__sort_return_none_error:
3563 {
3564     \tex_toks:D \l__sort_B_int \tex_toks:D \l__sort_A_int
3565     \int_decr:N \l__sort_B_int
3566     \int_decr:N \l__sort_A_int
3567     \if_int_compare:w \l__sort_A_int < \l__sort_begin_int
3568         \__sort_merge_blocks_end: \use_i:nn
3569     \fi:
3570     \__sort_merge_blocks_aux:
3571 }

```

(End definition for __sort_return_swapped:w.)

__sort_merge_blocks_end: This function's task is to copy the \toks registers in the block indexed by *C* to the merger indexed by *B*. The end can equally be detected by checking when *B* reaches the threshold **begin**, or when *C* reaches **top**.

```

3572 \cs_new_protected:Npn \__sort_merge_blocks_end:
3573 {
3574     \tex_toks:D \l__sort_B_int \tex_toks:D \l__sort_C_int
3575     \int_decr:N \l__sort_B_int
3576     \int_decr:N \l__sort_C_int
3577     \if_int_compare:w \l__sort_B_int < \l__sort_begin_int
3578         \use_i:nn
3579     \fi:
3580     \__sort_merge_blocks_end:
3581 }

```

(End definition for __sort_merge_blocks_end:.)

43.5 Expandable sorting

Sorting expandably is very different from sorting and assigning to a variable. Since tokens cannot be stored, they must remain in the input stream, and be read through at every step. It is thus necessarily much slower (at best $O(n^2 \ln n)$) than non-expandable sorting functions ($O(n \ln n)$).

A prototypical version of expandable quicksort is as follows. If the argument has no item, return nothing, otherwise partition, using the first item as a pivot (argument #4 of __sort:nnNnn). The arguments of __sort:nnNnn are 1. items less than #4, 2. items greater or equal to #4, 3. comparison, 4. pivot, 5. next item to test. If #5 is the tail of

the list, call `\tl_sort:nN` on #1 and on #2, placing #4 in between; `\use:ff` expands the parts to make `\tl_sort:nN` f-expandable. Otherwise, compare #4 and #5 using #3. If they are ordered, place #5 amongst the “greater” items, otherwise amongst the “lesser” items, and continue partitioning.

```
\cs_new:Npn \tl_sort:nN #1#2
{
  \tl_if_blank:nF {#1}
  {
    \__sort:nnNnn { } { } #2
    #1 \q__sort_recursion_tail \q__sort_recursion_stop
  }
}
\cs_new:Npn \__sort:nnNnn #1#2#3#4#5
{
  \quark_if_recursion_tail_stop_do:nn {#5}
  { \use:ff { \tl_sort:nN {#1} #3 {#4} } { \tl_sort:nN {#2} #3 } }
  #3 {#4} {#5}
  { \__sort:nnNnn {#1} { #2 {#5} } #3 {#4} }
  { \__sort:nnNnn { #1 {#5} } {#2} #3 {#4} }
}
\cs_generate_variant:Nn \use:nn { ff }
```

There are quite a few optimizations available here: the code below is less legible, but more than twice as fast.

In the simple version of the code, `__sort:nnNnn` is called $O(n \ln n)$ times on average (the number of comparisons required by the quicksort algorithm). Hence most of our focus is on optimizing that function.

The first speed up is to avoid testing for the end of the list at every call to `__sort:nnNnn`. For this, the list is prepared by changing each *⟨item⟩* of the original token list into *⟨command⟩ {⟨item⟩}*, just like sequences are stored. We arrange things such that the *⟨command⟩* is the *⟨conditional⟩* provided by the user: the loop over the *⟨prepared tokens⟩* then looks like

```
\cs_new:Npn \__sort_loop:wNn ... #6#7
{
  #6 {⟨pivot⟩} {#7} ⟨loop big⟩ ⟨loop small⟩
  ⟨extra arguments⟩
}
\__sort_loop:wNn ... ⟨prepared tokens⟩
⟨end-loop⟩ {} \s__sort_stop
```

In this example, which matches the structure of `__sort_quick_split_i:NnnnnNn` and a few other functions below, the `__sort_loop:wNn` auxiliary normally receives the user’s *⟨conditional⟩* as #6 and an *⟨item⟩* as #7. This is compared to the *⟨pivot⟩* (the argument #5, not shown here), and the *⟨conditional⟩* leaves the *⟨loop big⟩* or *⟨loop small⟩* auxiliary, which both have the same form as `__sort_loop:wNn`, receiving the next pair *⟨conditional⟩ {⟨item⟩}* as #6 and #7. At the end, #6 is the *⟨end-loop⟩* function, which terminates the loop.

The second speed up is to minimize the duplicated tokens between the **true** and **false** branches of the conditional. For this, we introduce two versions of `__sort:nnNnn`,

which receive the new item as #1 and place it either into the list #2 of items less than the pivot #4 or into the list #3 of items greater or equal to the pivot.

```
\cs_new:Npn \__sort_i:nnnnNn #1#2#3#4#5#6
{
  #5 {#4} {#6} \__sort_ii:nnnnNn \__sort_i:nnnnNn
  {#6} { #2 {#1} } {#3} {#4}
}
\cs_new:Npn \__sort_ii:nnnnNn #1#2#3#4#5#6
{
  #5 {#4} {#6} \__sort_ii:nnnnNn \__sort_i:nnnnNn
  {#6} {#2} { #3 {#1} } {#4}
}
```

Note that the two functions have the form of `__sort_loop:wNn` above, receiving as #5 the conditional or a function to end the loop. In fact, the lists #2 and #3 must be made of pairs $\langle \text{conditional} \rangle \{ \langle \text{item} \rangle \}$, so we have to replace {#6} above by { #5 {#6} }, and {#1} by #1. The actual functions have one more argument, so all argument numbers are shifted compared to this code.

The third speed up is to avoid `\use:ff` using a continuation-passing style: `__sort_quick_split:NnNn` expects a list followed by `\s__sort_mark {<code>}`, and expands to $\langle \text{code} \rangle \langle \text{sorted list} \rangle$. Sorting the two parts of the list around the pivot is done with

```
\__sort_quick_split:NnNn #2 ... \s__sort_mark
{
  \__sort_quick_split:NnNn #1 ... \s__sort_mark {<code>}
  {<pivot>}
}
```

Items which are larger than the $\langle \text{pivot} \rangle$ are sorted, then placed after code that sorts the smaller items, and after the (braced) $\langle \text{pivot} \rangle$.

The fourth speed up is avoid the recursive call to `\tl_sort:nN` with an empty first argument. For this, we introduce functions similar to the `__sort_i:nnnnNn` of the last example, but aware of whether the list of $\langle \text{conditional} \rangle \{ \langle \text{item} \rangle \}$ read so far that are less than the pivot, and the list of those greater or equal, are empty or not: see `__sort_quick_split:NnNn` and functions defined below. Knowing whether the lists are empty or not is useless if we do not use distinct ending codes as appropriate. The splitting auxiliaries communicate to the $\langle \text{end-loop} \rangle$ function (that is initially placed after the “prepared” list) by placing a specific ending function, ignored when looping, but useful at the end. In fact, the $\langle \text{end-loop} \rangle$ function does nothing but place the appropriate ending function in front of all its arguments. The ending functions take care of sorting non-empty sublists, placing the pivot in between, and the continuation before.

The final change in fact slows down the code a little, but is required to avoid memory issues: schematically, when \TeX encounters

```
\use:n { \use:n { \use:n { ... } ... } ... }
```

the argument of the first `\use:n` is not completely read by the second `\use:n`, hence must remain in memory; then the argument of the second `\use:n` is not completely read when grabbing the argument of the third `\use:n`, hence must remain in memory, and so on. The memory consumption grows quadratically with the number of nested `\use:n`. In

practice, this means that we must read everything until a trailing `\s__sort_stop` once in a while, otherwise sorting lists of more than a few thousand items would exhaust a typical T_EX's memory.

`\tl_sort:nN`

`__sort_quick_prepare:Nnnn`
`__sort_quick_prepare_end:NNNnw`
`__sort_quick_cleanup:w`

The code within the `\exp_not:f` sorts the list, leaving in most cases a leading `\exp_not:f`, which stops the expansion, letting the result be return within `\exp_not:n`. We filter out the case of a list with no item, which would otherwise cause problems. Then prepare the token list #1 by inserting the conditional #2 before each item. The `prepare` auxiliary receives the conditional as #1, the prepared token list so far as #2, the next prepared item as #3, and the item after that as #4. The loop ends when #4 contains `\prg_break_point:`, then the `prepare_end` auxiliary finds the prepared token list as #4. The scene is then set up for `__sort_quick_split:NnNn`, which sorts the prepared list and perform the post action placed after `\s__sort_mark`, namely removing the trailing `\s__sort_stop` and `\s__sort_stop` and leaving `\exp_stop_f:` to stop f-expansion.

```

3582 \cs_new:Npn \tl_sort:nN #1#2
3583 {
3584   \exp_not:f
3585   {
3586     \tl_if_blank:nF {#1}
3587     {
3588       \__sort_quick_prepare:Nnnn #2 { } { }
3589       #1
3590       { \prg_break_point: \__sort_quick_prepare_end:NNNnw }
3591       \s__sort_stop
3592     }
3593   }
3594 }
3595 \cs_new:Npn \__sort_quick_prepare:Nnnn #1#2#3#4
3596 {
3597   \prg_break: #4 \prg_break_point:
3598   \__sort_quick_prepare:Nnnn #1 { #2 #3 } { #1 {#4} }
3599 }
3600 \cs_new:Npn \__sort_quick_prepare_end:NNNnw #1#2#3#4#5 \s__sort_stop
3601 {
3602   \__sort_quick_split:NnNn #4 \__sort_quick_end:nnTFNn { }
3603   \s__sort_mark { \__sort_quick_cleanup:w \exp_stop_f: }
3604   \s__sort_mark \s__sort_stop
3605 }
3606 \cs_new:Npn \__sort_quick_cleanup:w #1 \s__sort_mark \s__sort_stop {#1}

```

(End definition for `\tl_sort:nN` and others. This function is documented on page 116.)

`__sort_quick_split:NnNn`

`__sort_quick_only_i:NnnnnNn`
`__sort_quick_only_ii:NnnnnNn`
`__sort_quick_split_i:NnnnnNn`
`__sort_quick_split_ii:NnnnnNn`

The `only_i`, `only_ii`, `split_i` and `split_ii` auxiliaries receive a useless first argument, the new item #2 (that they append to either one of the next two arguments), the list #3 of items less than the pivot, bigger items #4, the pivot #5, a *function* #6, and an item #7. The *function* is the user's *conditional* except at the end of the list where it is `__sort_quick_end:nnTFNn`. The comparison is applied to the *pivot* and the *item*, and calls the `only_i` or `split_i` auxiliaries if the *item* is smaller, and the `only_ii` or `split_ii` auxiliaries otherwise. In both cases, the next auxiliary goes to work right away, with no intermediate expansion that would slow down operations. Note that the argument #2 left for the next call has the form *conditional* {*item*}, so that the lists #3 and #4 keep the right form to be fed to the next sorting function. The `split` auxiliary

differs from these in that it is missing three of the arguments, which would be empty, and its first argument is always the user's *conditional* rather than an ending function.

```

3607 \cs_new:Npn \__sort_quick_split:NnNn #1#2#3#4
3608 {
3609     #3 {#2} {#4} \__sort_quick_only_ii:NnnnnNn
3610     \__sort_quick_only_i:NnnnnNn
3611     \__sort_quick_single_end:nnnwnw
3612     { #3 {#4} } { } { } {#2}
3613 }
3614 \cs_new:Npn \__sort_quick_only_i:NnnnnNn #1#2#3#4#5#6#7
3615 {
3616     #6 {#5} {#7} \__sort_quick_split_ii:NnnnnNn
3617     \__sort_quick_only_i:NnnnnNn
3618     \__sort_quick_only_i_end:nnnwnw
3619     { #6 {#7} } { #3 #2 } { } {#5}
3620 }
3621 \cs_new:Npn \__sort_quick_only_ii:NnnnnNn #1#2#3#4#5#6#7
3622 {
3623     #6 {#5} {#7} \__sort_quick_only_ii:NnnnnNn
3624     \__sort_quick_split_i:NnnnnNn
3625     \__sort_quick_only_ii_end:nnnwnw
3626     { #6 {#7} } { } { #4 #2 } {#5}
3627 }
3628 \cs_new:Npn \__sort_quick_split_i:NnnnnNn #1#2#3#4#5#6#7
3629 {
3630     #6 {#5} {#7} \__sort_quick_split_ii:NnnnnNn
3631     \__sort_quick_split_i:NnnnnNn
3632     \__sort_quick_split_end:nnnwnw
3633     { #6 {#7} } { #3 #2 } {#4} {#5}
3634 }
3635 \cs_new:Npn \__sort_quick_split_ii:NnnnnNn #1#2#3#4#5#6#7
3636 {
3637     #6 {#5} {#7} \__sort_quick_split_ii:NnnnnNn
3638     \__sort_quick_split_i:NnnnnNn
3639     \__sort_quick_split_end:nnnwnw
3640     { #6 {#7} } {#3} { #4 #2 } {#5}
3641 }

```

(End definition for `__sort_quick_split:NnNn` and others.)

```

\__sort_quick_end:nnTFNn
  \__sort_quick_single_end:nnnwnw
  \__sort_quick_only_i_end:nnnwnw
  \__sort_quick_only_ii_end:nnnwnw
  \__sort_quick_split_end:nnnwnw

```

The `__sort_quick_end:nnTFNn` appears instead of the user's conditional, and receives as its arguments the pivot #1, a fake item #2, a `true` and a `false` branches #3 and #4, followed by an ending function #5 (one of the four auxiliaries here) and another copy #6 of the fake item. All those are discarded except the function #5. This function receives lists #1 and #2 of items less than or greater than the pivot #3, then a continuation code #5 just after `\s__sort_mark`. To avoid a memory problem described earlier, all of the ending functions read #6 until `\s__sort_stop` and place #6 back into the input stream. When the lists #1 and #2 are empty, the `single` auxiliary simply places the continuation #5 before the pivot {#3}. When #2 is empty, #1 is sorted and placed before the pivot {#3}, taking care to feed the continuation #5 as a continuation for the function sorting #1. When #1 is empty, #2 is sorted, and the continuation argument is used to place the continuation #5 and the pivot {#3} before the sorted result. Finally, when both

lists are non-empty, items larger than the pivot are sorted, then items less than the pivot, and the continuations are done in such a way to place the pivot in between.

```

3642 \cs_new:Npn \__sort_quick_end:nnTFNn #1#2#3#4#5#6 {#5}
3643 \cs_new:Npn \__sort_quick_single_end:nnnwnw #1#2#3#4 \s__sort_mark #5#6 \s__sort_stop
3644 { #5 {#3} #6 \s__sort_stop }
3645 \cs_new:Npn \__sort_quick_only_i_end:nnnwnw #1#2#3#4 \s__sort_mark #5#6 \s__sort_stop
3646 {
3647   \__sort_quick_split:NnNn #1
3648   \__sort_quick_end:nnTFNn { } \s__sort_mark {#5}
3649   {#3}
3650   #6 \s__sort_stop
3651 }
3652 \cs_new:Npn \__sort_quick_only_ii_end:nnnwnw #1#2#3#4 \s__sort_mark #5#6 \s__sort_stop
3653 {
3654   \__sort_quick_split:NnNn #2
3655   \__sort_quick_end:nnTFNn { } \s__sort_mark { #5 {#3} }
3656   #6 \s__sort_stop
3657 }
3658 \cs_new:Npn \__sort_quick_split_end:nnnwnw #1#2#3#4 \s__sort_mark #5#6 \s__sort_stop
3659 {
3660   \__sort_quick_split:NnNn #2 \__sort_quick_end:nnTFNn { } \s__sort_mark
3661   {
3662     \__sort_quick_split:NnNn #1
3663     \__sort_quick_end:nnTFNn { } \s__sort_mark {#5}
3664     {#3}
3665   }
3666   #6 \s__sort_stop
3667 }

```

(End definition for __sort_quick_end:nnTFNn and others.)

43.6 Messages

__sort_error: Bailing out of the sorting code is a bit tricky. It may not be safe to use a delimited argument, so instead we redefine many l3sort commands to be trivial, with __sort_level: jumping to the break point. This error recovery won't work in a group.

```

3668 \cs_new_protected:Npn \__sort_error:
3669 {
3670   \cs_set_eq:NN \__sort_merge_blocks_aux: \prg_do_nothing:
3671   \cs_set_eq:NN \__sort_merge_blocks: \prg_do_nothing:
3672   \cs_set_protected:Npn \__sort_level: { \group_end: \prg_break: }
3673 }

```

(End definition for __sort_error:.)

__sort_disable_toksdef: While sorting, \toksdef is locally disabled to prevent users from using \newtoks or similar commands in their comparison code: the \toks registers that would be assigned are in use by l3sort. In format mode, none of this is needed since there is no \toks allocator.

```

3674 \cs_new_protected:Npn \__sort_disable_toksdef:
3675 { \cs_set_eq:NN \toksdef \__sort_disabled_toksdef:n }
3676 \cs_new_protected:Npn \__sort_disabled_toksdef:n #1
3677 {

```

```

3678 \msg_error:nxx { sort } { toksdef }
3679 { \token_to_str:N #1 }
3680 \__sort_error:
3681 \tex_toksdef:D #1
3682 }
3683 \msg_new:nnnn { sort } { toksdef }
3684 { Allocation~of~\iow_char:N\ toks~registers~impossible~while~sorting. }
3685 {
3686   The~comparison~code~used~for~sorting~a~list~has~attempted~to~
3687   define~#1~as~a~new~\iow_char:N\ toks~register~using~
3688   \iow_char:N\ newtoks~
3689   or~a~similar~command.~The~list~will~not~be~sorted.
3690 }

```

(End definition for __sort_disable_toksdef: and __sort_disabled_toksdef:n.)

__sort_too_long_error:NNw When there are too many items in a sequence, this is an error, and we clean up properly the mapping over items in the list: break using the type-specific breaking function #1.

```

3691 \cs_new_protected:Npn \__sort_too_long_error:NNw #1#2 \fi:
3692 {
3693   \fi:
3694   \msg_error:nnxxx { sort } { too-large }
3695   { \token_to_str:N #2 }
3696   { \int_eval:n { \l__sort_true_max_int - \l__sort_min_int } }
3697   { \int_eval:n { \l__sort_top_int - \l__sort_min_int } }
3698   #1 \__sort_error:
3699 }
3700 \msg_new:nnnn { sort } { too-large }
3701 { The~list~#1~is~too~long~to~be~sorted~by~TeX. }
3702 {
3703   TeX~has~#2~toks~registers~still~available:~
3704   this~only~allows~to~sort~with~up~to~#3~
3705   items.~The~list~will~not~be~sorted.
3706 }

```

(End definition for __sort_too_long_error:NNw.)

```

3707 \msg_new:nnnn { sort } { return-none }
3708 { The~comparison~code~did~not~return. }
3709 {
3710   When~sorting~a~list,~the~code~to~compare~items~#1~and~#2~
3711   did~not~call~
3712   \iow_char:N\ sort_return_same: ~nor~
3713   \iow_char:N\ sort_return_swapped: .~
3714   Exactly~one~of~these~should~be~called.
3715 }
3716 \msg_new:nnnn { sort } { return-two }
3717 { The~comparison~code~returned~multiple~times. }
3718 {
3719   When~sorting~a~list,~the~code~to~compare~items~#1~and~#2~called~
3720   \iow_char:N\ sort_return_same: ~or~
3721   \iow_char:N\ sort_return_swapped: ~multiple~times.~
3722   Exactly~one~of~these~should~be~called.
3723 }
3724 \prop_gput:Nnn \g_msg_module_name_prop { sort } { LaTeX3 }
3725 \prop_gput:Nnn \g_msg_module_type_prop { sort } { }

```

3726 </package>

Chapter 44

l3tl-analysis implementation

3727 $\langle @@=tl \rangle$

44.1 Internal functions

$\backslash s_tl$ The format used to store token lists internally uses the scan mark $\backslash s_tl$ as a delimiter.

(End definition for $\backslash s_tl$.)

44.2 Internal format

The task of the `l3tl-analysis` module is to convert token lists to an internal format which allows us to extract all the relevant information about individual tokens (category code, character code), as well as reconstruct the token list quickly. This internal format is used in `l3regex` where we need to support arbitrary tokens, and it is used in conversion functions in `l3str-convert`, where we wish to support clusters of characters instead of single tokens.

We thus need a way to encode any $\langle token \rangle$ (even begin-group and end-group character tokens) in a way amenable to manipulating tokens individually. The best we can do is to find $\langle tokens \rangle$ which both `o`-expand and `x`-expand to the given $\langle token \rangle$. Collecting more information about the category code and character code is also useful for regular expressions, since most regexes are catcode-agnostic. The internal format thus takes the form of a succession of items of the form

$\langle tokens \rangle \backslash s_tl \langle catcode \rangle \langle char\ code \rangle \backslash s_tl$

The $\langle tokens \rangle$ `o`- and `x`-expand to the original token in the token list or to the cluster of tokens corresponding to one Unicode character in the given encoding (for `l3str-convert`). The $\langle catcode \rangle$ is given as a single hexadecimal digit, 0 for control sequences. The $\langle char\ code \rangle$ is given as a decimal number, -1 for control sequences.

Using delimited arguments lets us build the $\langle tokens \rangle$ progressively when doing an encoding conversion in `l3str-convert`. On the other hand, the delimiter $\backslash s_tl$ may not appear unbraced in $\langle tokens \rangle$. This is not a problem because we are careful to wrap control sequences in braces (as an argument to $\backslash exp_not:n$) when converting from a general token list to the internal format.

The current rule for converting a $\langle token \rangle$ to a balanced set of $\langle tokens \rangle$ which both `o`-expands and `x`-expands to it is the following.

- A control sequence `\cs` becomes `\exp_not:n { \cs } \s__tl 0 -1 \s__tl`.
- A begin-group character `{` becomes `\exp_after:wN { \if_false: } \fi: \s__tl 1 <char code> \s__tl`.
- An end-group character `}` becomes `\if_false: { \fi: } \s__tl 2 <char code> \s__tl`.
- A character with any other category code becomes `\exp_not:n {<character>} \s__tl <hex catcode> <char code> \s__tl`.

3728 `<*package>`

44.3 Variables and helper functions

`\s__tl` The scan mark `\s__tl` is used as a delimiter in the internal format. This is more practical than using a quark, because we would then need to control expansion much more carefully: compare `\int_value:w '#1 \s__tl` with `\int_value:w '#1 \exp_stop_f: \exp_not:N \q_mark` to extract a character code followed by the delimiter in an x-expansion.

3729 `\scan_new:N \s__tl`

(End definition for `\s__tl`.)

`\l__tl_analysis_token` The tokens in the token list are probed with the TeX primitive `\futurelet`. We use `\l__tl_analysis_token` in that construction. In some cases, we convert the following token to a string before probing it: then the token variable used is `\l__tl_analysis_char_token`. When getting tokens from the input stream we may need to look two tokens ahead, for which we use `\l__tl_analysis_next_token`.

3730 `\cs_new_eq:NN \l__tl_analysis_token ?`

3731 `\cs_new_eq:NN \l__tl_analysis_char_token ?`

3732 `\cs_new_eq:NN \l__tl_analysis_next_token ?`

(End definition for `\l__tl_analysis_token`, `\l__tl_analysis_char_token`, and `\l__tl_analysis_next_token`.)

`\l__tl_peek_code_tl` Holds some code to be run once the next token has been fully analysed in `\peek_analysis_map_inline:n`.

3733 `\tl_new:N \l__tl_peek_code_tl`

(End definition for `\l__tl_peek_code_tl`.)

`\c__tl_peek_catcodes_tl` A token list containing the character number 32 (space) with all possible category codes except 1 and 2 (begin-group and end-group). Why 32? Because some LuaTeX versions only allow creation of catcode 10 (space) tokens with this character code, and because even in other engines it is much easier to produce since `\char_generate:nn` refuses to produce spaces.

3734 `\group_begin:`

3735 `\char_set_active_eq:NN \ \scan_stop:`

3736 `\tl_const:Nx \c__tl_peek_catcodes_tl`

3737 `{`

3738 `\char_generate:nn { 32 } { 3 } 3`

3739 `\char_generate:nn { 32 } { 4 } 4`

3740 `# \char_generate:nn { 32 } { 6 } 6`

```

3741 \char_generate:nn { 32 } { 7 } 7
3742 \char_generate:nn { 32 } { 8 } 8
3743 \c_space_tl \token_to_str:N A
3744 \char_generate:nn { 32 } { 11 } \token_to_str:N B
3745 \char_generate:nn { 32 } { 12 } \token_to_str:N C
3746 \char_generate:nn { 32 } { 13 } \token_to_str:N D
3747 }
3748 \group_end:

```

(End definition for \c__tl_peek_catcodes_tl.)

`\l__tl_analysis_normal_int` The number of normal (N-type argument) tokens since the last special token.

```
3749 \int_new:N \l__tl_analysis_normal_int
```

(End definition for \l__tl_analysis_normal_int.)

`\l__tl_analysis_index_int` During the first pass, this is the index in the array being built. During the second pass, it is equal to the maximum index in the array from the first pass.

```
3750 \int_new:N \l__tl_analysis_index_int
```

(End definition for \l__tl_analysis_index_int.)

`\l__tl_analysis_nesting_int` Nesting depth of explicit begin-group and end-group characters during the first pass. This lets us detect the end of the token list without a reserved end-marker.

```
3751 \int_new:N \l__tl_analysis_nesting_int
```

(End definition for \l__tl_analysis_nesting_int.)

`\l__tl_analysis_type_int` When encountering special characters, we record their “type” in this integer.

```
3752 \int_new:N \l__tl_analysis_type_int
```

(End definition for \l__tl_analysis_type_int.)

`\g__tl_analysis_result_tl` The result of the conversion is stored in this token list, with a succession of items of the form

$\langle tokens \rangle \backslash s_tl \langle catcode \rangle \langle char\ code \rangle \backslash s_tl$

```
3753 \tl_new:N \g__tl_analysis_result_tl
```

(End definition for \g__tl_analysis_result_tl.)

`__tl_analysis_extract_charcode:`
`__tl_analysis_extract_charcode_aux:w` Extracting the character code from the meaning of `\l__tl_analysis_token`. This has no error checking, and should only be assumed to work for begin-group and end-group character tokens. It produces a number in the form ‘ $\langle char \rangle$ ’.

```

3754 \cs_new:Npn \__tl_analysis_extract_charcode:
3755 {
3756   \exp_after:wN \__tl_analysis_extract_charcode_aux:w
3757   \token_to_meaning:N \l__tl_analysis_token
3758 }
3759 \cs_new:Npn \__tl_analysis_extract_charcode_aux:w #1 ~ #2 ~ { ‘ }

```

(End definition for __tl_analysis_extract_charcode: and __tl_analysis_extract_charcode_aux:w.)

```

\__tl_analysis_cs_space_count:NN
\__tl_analysis_cs_space_count:w
\__tl_analysis_cs_space_count_end:w
Counts the number of spaces in the string representation of its second argument, as well
as the number of characters following the last space in that representation, and feeds the
two numbers as semicolon-delimited arguments to the first argument. When this function
is used, the escape character is printable and non-space.

3760 \cs_new:Npn \__tl_analysis_cs_space_count:NN #1 #2
3761 {
3762   \exp_after:wN #1
3763   \int_value:w \int_eval:w 0
3764   \exp_after:wN \__tl_analysis_cs_space_count:w
3765   \token_to_str:N #2
3766   \fi: \__tl_analysis_cs_space_count_end:w ; ~ !
3767 }
3768 \cs_new:Npn \__tl_analysis_cs_space_count:w #1 ~
3769 {
3770   \if_false: #1 #1 \fi:
3771   + 1
3772   \__tl_analysis_cs_space_count:w
3773 }
3774 \cs_new:Npn \__tl_analysis_cs_space_count_end:w ; #1 \fi: #2 !
3775 { \exp_after:wN ; \int_value:w \str_count_ignore_spaces:n {#1} ; }

(End definition for \__tl_analysis_cs_space_count:NN, \__tl_analysis_cs_space_count:w, and \__-
tl_analysis_cs_space_count_end:w.)

```

44.4 Plan of attack

Our goal is to produce a token list of the form roughly

```

⟨token 1⟩ \s__tl ⟨catcode 1⟩ ⟨char code 1⟩ \s__tl
⟨token 2⟩ \s__tl ⟨catcode 2⟩ ⟨char code 2⟩ \s__tl
... ⟨token N⟩ \s__tl ⟨catcode N⟩ ⟨char code N⟩ \s__tl

```

Most but not all tokens can be grabbed as an undelimited (N-type) argument by T_EX. The plan is to have a two pass system. In the first pass, locate special tokens, and store them in various `\toks` registers. In the second pass, which is done within an x-expanding assignment, normal tokens are taken in as N-type arguments, and special tokens are retrieved from the `\toks` registers, and removed from the input stream by some means. The whole process takes linear time, because we avoid building the result one item at a time.

We make the escape character printable (backslash, but this later oscillates between slash and backslash): this allows us to distinguish characters from control sequences.

A token has two characteristics: its **meaning**, and what it looks like for T_EX when it is in scanning mode (*e.g.*, when capturing parameters for a macro). For our purposes, we distinguish the following meanings:

- begin-group token (category code 1), either space (character code 32), or non-space;
- end-group token (category code 2), either space (character code 32), or non-space;
- space token (category code 10, character code 32);
- anything else (then the token is always an N-type argument).

The token itself can “look like” one of the following

- a non-active character, in which case its meaning is automatically that associated to its character code and category code, we call it “true” character;
- an active character;
- a control sequence.

The only tokens which are not valid N-type arguments are true begin-group characters, true end-group characters, and true spaces. We detect those characters by scanning ahead with `\futurelet`, then distinguishing true characters from control sequences set equal to them using the `\string` representation.

The second pass is a simple exercise in expandable loops.

`__tl_analysis:n` Everything is done within a group, and all definitions are local. We use `\group_align-safe_begin/end:` to avoid problems in case `__tl_analysis:n` is used within an alignment and its argument contains alignment tab tokens.

```

3776 \cs_new_protected:Npn \__tl_analysis:n #1
3777 {
3778   \group_begin:
3779   \group_align_safe_begin:
3780     \__tl_analysis_a:n {#1}
3781     \__tl_analysis_b:n {#1}
3782   \group_align_safe_end:
3783   \group_end:
3784 }
```

(End definition for `__tl_analysis:n`.)

44.5 Disabling active characters

`__tl_analysis_disable:n` Active characters can cause problems later on in the processing, so we provide a way to disable them, by setting them to `undefined`. Since Unicode contains too many characters to loop over all of them, we instead do this whenever we encounter a character. For `pTeX` and `upTeX` we skip characters beyond [0, 255] because `\lccode` only allows those values.

```

3785 \group_begin:
3786   \char_set_catcode_active:N \^^@
3787   \cs_new_protected:Npn \__tl_analysis_disable:n #1
3788   {
3789     \tex_lccode:D 0 = #1 \exp_stop_f:
3790     \tex_lowercase:D { \tex_let:D \^^@ } \tex_undefined:D
3791   }
3792   \bool_lazy_or:nnT
3793   { \sys_if_engine_ptex_p: }
3794   { \sys_if_engine_uptex_p: }
3795   {
3796     \cs_gset_protected:Npn \__tl_analysis_disable:n #1
3797     {
3798       \if_int_compare:w 256 > #1 \exp_stop_f:
3799       \tex_lccode:D 0 = #1 \exp_stop_f:
3800       \tex_lowercase:D { \tex_let:D \^^@ } \tex_undefined:D
3801     } \fi:
3802   }
3803 }
3804 \group_end:
```

(End definition for `_tl_analysis_disable:n`.)

44.6 First pass

The goal of this pass is to detect special (non-N-type) tokens, and count how many N-type tokens lie between special tokens. Also, we wish to store some representation of each special token in a `\toks` register.

We have 11 types of tokens:

1. a true non-space begin-group character;
2. a true space begin-group character;
3. a true non-space end-group character;
4. a true space end-group character;
5. a true space blank space character;
6. an active character;
7. any other true character;
8. a control sequence equal to a begin-group token (category code 1);
9. a control sequence equal to an end-group token (category code 2);
10. a control sequence equal to a space token (character code 32, category code 10);
11. any other control sequence.

Our first tool is `\futurelet`. This cannot distinguish case 8 from 1 or 2, nor case 9 from 3 or 4, nor case 10 from case 5. Those cases are later distinguished by applying the `\string` primitive to the following token, after possibly changing the escape character to ensure that a control sequence’s string representation cannot be mistaken for the true character.

In cases 6, 7, and 11, the following token is a valid N-type argument, so we grab it and distinguish the case of a character from a control sequence: in the latter case, `\str_tail:n {\token}` is non-empty, because the escape character is printable.

`_tl_analysis_a:n` We read tokens one by one using `\futurelet`. While performing the loop, we keep track of the number of true begin-group characters minus the number of true end-group characters in `\l_tl_analysis_nesting_int`. This reaches `-1` when we read the closing brace.

```
3805 \cs_new_protected:Npn \_tl_analysis_a:n #1
3806 {
3807   \_tl_analysis_disable:n { 32 }
3808   \int_set:Nn \tex_escapechar:D { 92 }
3809   \int_zero:N \l\_tl_analysis_normal_int
3810   \int_zero:N \l\_tl_analysis_index_int
3811   \int_zero:N \l\_tl_analysis_nesting_int
3812   \if_false: { \fi: \_tl_analysis_a_loop:w #1 }
3813   \int_decr:N \l\_tl_analysis_index_int
3814 }
```

(End definition for `_tl_analysis_a:n`.)

`_tl_analysis_a_loop:w` Read one character and check its type.

```
3815 \cs_new_protected:Npn \_tl\_analysis\_a\_loop:w
3816 { \tex_futurelet:D \l\_tl\_analysis\_token \_tl\_analysis\_a\_type:w }
```

(End definition for `_tl_analysis_a_loop:w`.)

`_tl_analysis_a_type:w` At this point, `\l_tl_analysis_token` holds the meaning of the following token. We store in `\l_tl_analysis_type_int` information about the meaning of the token ahead:

- 0 space token;
- 1 begin-group token;
- -1 end-group token;
- 2 other.

The values 0, 1, -1 correspond to how much a true such character changes the nesting level (2 is used only here, and is irrelevant later). Then call the auxiliary for each case. Note that nesting conditionals here is safe because we only skip over `\l_tl_analysis_token` if it matches with one of the character tokens (hence is not a primitive conditional).

```
3817 \cs_new_protected:Npn \_tl\_analysis\_a\_type:w
3818 {
3819   \l\_tl\_analysis\_type\_int =
3820   \if_meaning:w \l\_tl\_analysis\_token \c\_space\_token
3821     0
3822   \else:
3823     \if_catcode:w \exp\_not:N \l\_tl\_analysis\_token \c\_group\_begin\_token
3824       1
3825     \else:
3826       \if_catcode:w \exp\_not:N \l\_tl\_analysis\_token \c\_group\_end\_token
3827         - 1
3828       \else:
3829         2
3830     \fi:
3831   \fi:
3832   \fi:
3833   \exp\_stop\_f:
3834   \if_case:w \l\_tl\_analysis\_type\_int
3835     \exp\_after:wN \_tl\_analysis\_a\_space:w
3836   \or: \exp\_after:wN \_tl\_analysis\_a\_bgroup:w
3837   \or: \exp\_after:wN \_tl\_analysis\_a\_safe:N
3838   \else: \exp\_after:wN \_tl\_analysis\_a\_egroup:w
3839   \fi:
3840 }
```

(End definition for `_tl_analysis_a_type:w`.)

`_tl_analysis_a_space:w` In this branch, the following token's meaning is a blank space. Apply `\string` to that token: a true blank space gives a space, a control sequence gives a result starting with the escape character, an active character gives something else than a space since we disabled the space. We grab as `\l_tl_analysis_char_token` the first character of the string representation then test it in `_tl_analysis_a_space_test:w`. Also, since

`__tl_analysis_a_store:` expects the special token to be stored in the relevant `\toks` register, we do that. The extra `\exp_not:n` is unnecessary of course, but it makes the treatment of all tokens more homogeneous. If we discover that the next token was actually a control sequence or an active character instead of a true space, then we step the counter of normal tokens. We now have in front of us the whole string representation of the control sequence, including potential spaces; those will appear to be true spaces later in this pass. Hence, all other branches of the code in this first pass need to consider the string representation, so that the second pass does not need to test the meaning of tokens, only strings.

```

3841 \cs_new_protected:Npn \__tl_analysis_a_space:w
3842 {
3843   \tex_afterassignment:D \__tl_analysis_a_space_test:w
3844   \exp_after:wN \cs_set_eq:NN
3845   \exp_after:wN \l__tl_analysis_char_token
3846   \token_to_str:N
3847 }
3848 \cs_new_protected:Npn \__tl_analysis_a_space_test:w
3849 {
3850   \if_meaning:w \l__tl_analysis_char_token \c_space_token
3851     \tex_toks:D \l__tl_analysis_index_int { \exp_not:n { ~ } }
3852     \__tl_analysis_a_store:
3853   \else:
3854     \int_incr:N \l__tl_analysis_normal_int
3855   \fi:
3856   \__tl_analysis_a_loop:w
3857 }
```

(End definition for `__tl_analysis_a_space:w` and `__tl_analysis_a_space_test:w`.)

```

\__tl_analysis_a_bgroup:w
\__tl_analysis_a_egroup:w
\__tl_analysis_a_group:nw
\__tl_analysis_a_group_aux:w
  \tl_analysis_a_group_auxii:w
  \tl_analysis_a_group_test:w
```

The token is most likely a true character token with catcode 1 or 2, but it might be a control sequence, or an active character. Optimizing for the first case, we store in a `\toks` register some code that expands to that token. Since we will turn what follows into a string, we make sure the escape character is different from the current character code (by switching between solidus and backslash). To detect the special case of an active character let to the catcode 1 or 2 character with the same character code, we disable the active character with that character code and re-test: if the following token has become undefined we can in fact safely grab it. We are finally ready to turn what follows to a string and test it. This is one place where we need `\l__tl_analysis_char_token` to be a separate control sequence from `\l__tl_analysis_token`, to compare them.

```

3858 \group_begin:
3859   \char_set_catcode_group_begin:N \^^@ % {
3860   \cs_new_protected:Npn \__tl_analysis_a_bgroup:w
3861     { \__tl_analysis_a_group:nw { \exp_after:wN \^^@ \if_false: } \fi: } }
3862   \char_set_catcode_group_end:N \^^@
3863   \cs_new_protected:Npn \__tl_analysis_a_egroup:w
3864     { \__tl_analysis_a_group:nw { \if_false: { \fi: \^^@ } } % }
3865 \group_end:
3866 \cs_new_protected:Npn \__tl_analysis_a_group:nw #1
3867 {
3868   \tex_lccode:D 0 = \__tl_analysis_extract_charcode: \scan_stop:
3869   \tex_lowercase:D { \tex_toks:D \l__tl_analysis_index_int {#1} }
3870   \if_int_compare:w \tex_lccode:D 0 = \tex_escapechar:D
3871     \int_set:Nn \tex_escapechar:D { 139 - \tex_escapechar:D }
```

```

3872 \fi:
3873 \__tl_analysis_disable:n { \tex_lccode:D 0 }
3874 \tex_futurelet:D \l__tl_analysis_token \__tl_analysis_a_group_aux:w
3875 }
3876 \cs_new_protected:Npn \__tl_analysis_a_group_aux:w
3877 {
3878   \if_meaning:w \l__tl_analysis_token \tex_undefined:D
3879     \exp_after:wN \__tl_analysis_a_safe:N
3880   \else:
3881     \exp_after:wN \__tl_analysis_a_group_auxii:w
3882   \fi:
3883 }
3884 \cs_new_protected:Npn \__tl_analysis_a_group_auxii:w
3885 {
3886   \tex_afterassignment:D \__tl_analysis_a_group_test:w
3887   \exp_after:wN \cs_set_eq:NN
3888   \exp_after:wN \l__tl_analysis_char_token
3889   \token_to_str:N
3890 }
3891 \cs_new_protected:Npn \__tl_analysis_a_group_test:w
3892 {
3893   \if_charcode:w \l__tl_analysis_token \l__tl_analysis_char_token
3894     \__tl_analysis_a_store:
3895   \else:
3896     \int_incr:N \l__tl_analysis_normal_int
3897   \fi:
3898   \__tl_analysis_a_loop:w
3899 }

```

(End definition for `__tl_analysis_a_bgroup:w` and others.)

`__tl_analysis_a_store:` This function is called each time we meet a special token; at this point, the `\toks` register `\l__tl_analysis_index_int` holds a token list which expands to the given special token. Also, the value of `\l__tl_analysis_type_int` indicates which case we are in:

- -1 end-group character;
- 0 space character;
- 1 begin-group character.

We need to distinguish further the case of a space character (code 32) from other character codes, because those behave differently in the second pass. Namely, after testing the `\lccode` of 0 (which holds the present character code) we change the cases above to

- -2 space end-group character;
- -1 non-space end-group character;
- 0 space blank space character;
- 1 non-space begin-group character;
- 2 space begin-group character.

This has the property that non-space characters correspond to odd values of `\l__tl_analysis_type_int`. The number of normal tokens until here and the type of special token are packed into a `\skip` register. Finally, we check whether we reached the last closing brace, in which case we stop by disabling the looping function (locally).

```

3900 \cs_new_protected:Npn \__tl_analysis_a_store:
3901 {
3902   \tex_advance:D \l__tl_analysis_nesting_int \l__tl_analysis_type_int
3903   \if_int_compare:w \tex_lccode:D 0 = '\ \exp_stop_f:
3904     \tex_advance:D \l__tl_analysis_type_int \l__tl_analysis_type_int
3905   \fi:
3906   \tex_skip:D \l__tl_analysis_index_int
3907     = \l__tl_analysis_normal_int sp
3908     plus \l__tl_analysis_type_int sp \scan_stop:
3909   \int_incr:N \l__tl_analysis_index_int
3910   \int_zero:N \l__tl_analysis_normal_int
3911   \if_int_compare:w \l__tl_analysis_nesting_int = - \c_one_int
3912     \cs_set_eq:NN \__tl_analysis_a_loop:w \scan_stop:
3913   \fi:
3914 }

```

(End definition for `__tl_analysis_a_store:`)

```

\__tl_analysis_a_safe:N
\__tl_analysis_a_cs:ww

```

This should be the simplest case: since the upcoming token is safe, we can simply grab it in a second pass. If the token is a single character (including space), the `\if_charcode:w` test yields true; we disable a potentially active character (that could otherwise masquerade as the true character in the next pass) and we count one “normal” token. On the other hand, if the token is a control sequence, we should replace it by its string representation for compatibility with other code branches. Instead of slowly looping through the characters with the main code, we use the knowledge of how the second pass works: if the control sequence name contains no space, count that token as a number of normal tokens equal to its string length. If the control sequence contains spaces, they should be registered as special characters by increasing `\l__tl_analysis_index_int` (no need to carefully count character between each space), and all characters after the last space should be counted in the following sequence of “normal” tokens.

```

3915 \cs_new_protected:Npn \__tl_analysis_a_safe:N #1
3916 {
3917   \if_charcode:w
3918     \scan_stop:
3919     \exp_after:wN \use_none:n \token_to_str:N #1 \prg_do_nothing:
3920     \scan_stop:
3921     \exp_after:wN \use_i:nn
3922   \else:
3923     \exp_after:wN \use_ii:nn
3924   \fi:
3925   {
3926     \__tl_analysis_disable:n { '#1 }
3927     \int_incr:N \l__tl_analysis_normal_int
3928   }
3929   { \__tl_analysis_cs_space_count:NN \__tl_analysis_a_cs:ww #1 }
3930   \__tl_analysis_a_loop:w
3931 }
3932 \cs_new_protected:Npn \__tl_analysis_a_cs:ww #1; #2;
3933 {

```

```

3934 \if_int_compare:w #1 > \c_zero_int
3935 \tex_skip:D \l__tl_analysis_index_int
3936 = \int_eval:n { \l__tl_analysis_normal_int + 1 } sp \exp_stop_f:
3937 \tex_advance:D \l__tl_analysis_index_int #1 \exp_stop_f:
3938 \else:
3939 \tex_advance:D
3940 \fi:
3941 \l__tl_analysis_normal_int #2 \exp_stop_f:
3942 }

```

(End definition for `__tl_analysis_a_safe:N` and `__tl_analysis_a_cs:ww`.)

44.7 Second pass

The second pass is an exercise in expandable loops. All the necessary information is stored in `\skip` and `\toks` registers.

`__tl_analysis_b:n` Start the loop with the index 0. No need for an end-marker: the loop stops by itself when the last index is read. We repeatedly oscillate between reading long stretches of normal tokens, and reading special tokens.

```

3943 \cs_new_protected:Npn \__tl_analysis_b:n #1
3944 {
3945 \__kernel_tl_gset:Nx \g__tl_analysis_result_tl
3946 {
3947 \__tl_analysis_b_loop:w 0; #1
3948 \prg_break_point:
3949 }
3950 }
3951 \cs_new:Npn \__tl_analysis_b_loop:w #1;
3952 {
3953 \exp_after:wN \__tl_analysis_b_normals:ww
3954 \int_value:w \tex_skip:D #1 ; #1 ;
3955 }

```

(End definition for `__tl_analysis_b:n` and `__tl_analysis_b_loop:w`.)

`__tl_analysis_b_normals:ww` The first argument is the number of normal tokens which remain to be read, and the
`__tl_analysis_b_normal:wwN` second argument is the index in the array produced in the first step. A character's string representation is always one character long, while a control sequence is always longer (we have set the escape character to a printable value). In both cases, we leave `\exp_not:n` `{\token}` `\s__tl` in the input stream (after x-expansion). Here, `\exp_not:n` is used rather than `\exp_not:N` because `#3` could be a macro parameter character or could be `\s__tl` (which must be hidden behind braces in the result).

```

3956 \cs_new:Npn \__tl_analysis_b_normals:ww #1;
3957 {
3958 \if_int_compare:w #1 = \c_zero_int
3959 \__tl_analysis_b_special:w
3960 \fi:
3961 \__tl_analysis_b_normal:wwN #1;
3962 }
3963 \cs_new:Npn \__tl_analysis_b_normal:wwN #1; #2; #3
3964 {
3965 \exp_not:n { \exp_not:n { #3 } } \s__tl

```

```

3966 \if_charcode:w
3967 \scan_stop:
3968 \exp_after:wN \use_none:n \token_to_str:N #3 \prg_do_nothing:
3969 \scan_stop:
3970 \exp_after:wN \__tl_analysis_b_char:Nww
3971 \else:
3972 \exp_after:wN \__tl_analysis_b_cs:Nww
3973 \fi:
3974 #3 #1; #2;
3975 }

```

(End definition for __tl_analysis_b_normals:ww and __tl_analysis_b_normal:wwN.)

__tl_analysis_b_char:Nww If the normal token we grab is a character, leave $\langle catcode \rangle$ $\langle charcode \rangle$ followed by \s__tl in the input stream, and call __tl_analysis_b_normals:ww with its first argument decremented.

```

3976 \cs_new:Npx \__tl_analysis_b_char:Nww #1
3977 {
3978 \exp_not:N \if_meaning:w #1 \exp_not:N \tex_undefined:D
3979 \token_to_str:N D \exp_not:N \else:
3980 \exp_not:N \if_catcode:w #1 \c_catcode_other_token
3981 \token_to_str:N C \exp_not:N \else:
3982 \exp_not:N \if_catcode:w #1 \c_catcode_letter_token
3983 \token_to_str:N B \exp_not:N \else:
3984 \exp_not:N \if_catcode:w #1 \c_math_toggle_token 3
3985 \exp_not:N \else:
3986 \exp_not:N \if_catcode:w #1 \c_alignment_token 4
3987 \exp_not:N \else:
3988 \exp_not:N \if_catcode:w #1 \c_math_superscript_token 7
3989 \exp_not:N \else:
3990 \exp_not:N \if_catcode:w #1 \c_math_subscript_token 8
3991 \exp_not:N \else:
3992 \exp_not:N \if_catcode:w #1 \c_space_token
3993 \token_to_str:N A \exp_not:N \else:
3994 6
3995 \exp_not:n { \fi: \fi: \fi: \fi: \fi: \fi: \fi: }
3996 \exp_not:N \int_value:w '#1 \s__tl
3997 \exp_not:N \exp_after:wN \exp_not:N \__tl_analysis_b_normals:ww
3998 \exp_not:N \int_value:w \exp_not:N \int_eval:w - 1 +
3999 }

```

(End definition for __tl_analysis_b_char:Nww.)

__tl_analysis_b_cs:Nww __tl_analysis_b_cs_test:ww If the token we grab is a control sequence, leave 0 -1 (as category code and character code) in the input stream, followed by \s__tl, and call __tl_analysis_b_normals:ww with updated arguments.

```

4000 \cs_new:Npn \__tl_analysis_b_cs:Nww #1
4001 {
4002 0 -1 \s__tl
4003 \__tl_analysis_cs_space_count:NN \__tl_analysis_b_cs_test:ww #1
4004 }
4005 \cs_new:Npn \__tl_analysis_b_cs_test:ww #1 ; #2 ; #3 ; #4 ;
4006 {
4007 \exp_after:wN \__tl_analysis_b_normals:ww

```

```

4008 \int_value:w \int_eval:w
4009 \if_int_compare:w #1 = \c_zero_int
4010 #3
4011 \else:
4012 \tex_skip:D \int_eval:n { #4 + #1 } \exp_stop_f:
4013 \fi:
4014 - #2
4015 \exp_after:wN ;
4016 \int_value:w \int_eval:n { #4 + #1 } ;
4017 }

```

(End definition for `__tl_analysis_b_cs:Nww` and `__tl_analysis_b_cs_test:ww`.)

```

\__tl_analysis_b_special:w
\__tl_analysis_b_special_char:wN
\__tl_analysis_b_special_space:w

```

Here, `#1` is the current index in the array built in the first pass. Check now whether we reached the end (we shouldn't keep the trailing end-group character that marked the end of the token list in the first pass). Unpack the `\toks` register: when x-expanding again, we will get the special token. Then leave the category code in the input stream, followed by the character code, and call `__tl_analysis_b_loop:w` with the next index.

```

4018 \group_begin:
4019 \char_set_catcode_other:N A
4020 \cs_new:Npn \__tl_analysis_b_special:w
4021 \fi: \__tl_analysis_b_normal:wwN 0 ; #1 ;
4022 {
4023 \fi:
4024 \if_int_compare:w #1 = \l__tl_analysis_index_int
4025 \exp_after:wN \prg_break:
4026 \fi:
4027 \tex_the:D \tex_toks:D #1 \s__tl
4028 \if_case:w \tex_gluestretch:D \tex_skip:D #1 \exp_stop_f:
4029 \token_to_str:N A
4030 \or: 1
4031 \or: 1
4032 \else: 2
4033 \fi:
4034 \if_int_odd:w \tex_gluestretch:D \tex_skip:D #1 \exp_stop_f:
4035 \exp_after:wN \__tl_analysis_b_special_char:wN \int_value:w
4036 \else:
4037 \exp_after:wN \__tl_analysis_b_special_space:w \int_value:w
4038 \fi:
4039 \int_eval:n { 1 + #1 } \exp_after:wN ;
4040 \token_to_str:N
4041 }
4042 \group_end:
4043 \cs_new:Npn \__tl_analysis_b_special_char:wN #1 ; #2
4044 {
4045 \int_value:w '#2 \s__tl
4046 \__tl_analysis_b_loop:w #1 ;
4047 }
4048 \cs_new:Npn \__tl_analysis_b_special_space:w #1 ; ~
4049 {
4050 32 \s__tl
4051 \__tl_analysis_b_loop:w #1 ;
4052 }

```

(End definition for `_tl_analysis_b_special:w`, `_tl_analysis_b_special_char:wN`, and `_tl_analysis_b_special_space:w`.)

44.8 Mapping through the analysis

```
\tl_analysis_map_inline:nn
\tl_analysis_map_inline:Nn
  \_tl_analysis_map_inline_aux:Nn
  \_tl_analysis_map_inline_aux:nnn
```

First obtain the analysis of the token list into `\g_tl_analysis_result_tl`. To allow nested mappings, increase the nesting depth `\g_kernel_prg_map_int` (shared between all modules), then define the looping macro, which has a name specific to that nesting depth. That looping grabs the `\tokens`, `\catcode` and `\char code`; it checks for the end of the loop with `\use_none:n ##2`, normally empty, but which becomes `\tl_map_break:` at the end; it then performs the user's code `#2`, and loops by calling itself. When the loop ends, remember to decrease the nesting depth.

```
4053 \cs_new_protected:Npn \tl_analysis_map_inline:nn #1
4054 {
4055   \_tl_analysis:n {#1}
4056   \int_gincr:N \g_kernel_prg_map_int
4057   \exp_args:Nc \_tl_analysis_map_inline_aux:Nn
4058   { \_tl_analysis_map_inline_ \int_use:N \g_kernel_prg_map_int :wNw }
4059 }
4060 \cs_new_protected:Npn \tl_analysis_map_inline:Nn #1
4061 { \exp_args:No \tl_analysis_map_inline:nn #1 }
4062 \cs_new_protected:Npn \_tl_analysis_map_inline_aux:Nn #1#2
4063 {
4064   \cs_gset_protected:Npn #1 ##1 \s\_tl ##2 ##3 \s\_tl
4065   {
4066     \use\_none:n ##2
4067     \_tl_analysis_map_inline_aux:nnn {##1} {##3} {##2}
4068   }
4069   \cs_gset_protected:Npn \_tl_analysis_map_inline_aux:nnn ##1##2##3
4070   {
4071     #2
4072     #1
4073   }
4074   \exp_after:wN #1
4075   \g\_tl\_analysis\_result\_tl
4076   \s\_tl { ? \tl_map_break: } \s\_tl
4077   \prg_break_point:Nn \tl_map_break:
4078   { \int_gdecr:N \g_kernel_prg_map_int }
4079 }
```

(End definition for `\tl_analysis_map_inline:nn` and others. These functions are documented on page 45.)

44.9 Showing the results

```
\tl_analysis_show:N
\tl_analysis_log:N
\_tl_analysis_show:NNN
```

Add to `_tl_analysis:n` a third pass to display tokens to the terminal. If the token list variable is not defined, throw the same error as `\tl_show:N` by simply calling that function.

```
4080 \cs_new_protected:Npn \tl_analysis_show:N
4081 { \_tl_analysis_show:NNN \msg_show:nnxxxx \tl_show:N }
4082 \cs_new_protected:Npn \tl_analysis_log:N
```

```

4083 { \_tl\_analysis\_show:NNN \msg\_log:nnxxxx \tl\_log:N }
4084 \cs\_new\_protected:Npn \_tl\_analysis\_show:NNN #1#2#3
4085 {
4086   \tl\_if\_exist:NTF #3
4087   {
4088     \exp\_args:No \_tl\_analysis:n {#3}
4089     #1 { tl } { show-analysis }
4090     { \token\_to\_str:N #3 } { \_tl\_analysis\_show: } { } { }
4091   }
4092   { #2 #3 }
4093 }

```

(End definition for `\tl_analysis_show:N`, `\tl_analysis_log:N`, and `_tl_analysis_show:NNN`. These functions are documented on page 45.)

```

\tl\_analysis\_show:n No existence test needed here.
\tl\_analysis\_log:n
\_tl\_analysis\_show:Nn
4094 \cs\_new\_protected:Npn \tl\_analysis\_show:n
4095 { \_tl\_analysis\_show:Nn \msg\_show:nnxxxx }
4096 \cs\_new\_protected:Npn \tl\_analysis\_log:n
4097 { \_tl\_analysis\_show:Nn \msg\_log:nnxxxx }
4098 \cs\_new\_protected:Npn \_tl\_analysis\_show:Nn #1#2
4099 {
4100   \_tl\_analysis:n {#2}
4101   #1 { tl } { show-analysis } { } { \_tl\_analysis\_show: } { } { }
4102 }

```

(End definition for `\tl_analysis_show:n`, `\tl_analysis_log:n`, and `_tl_analysis_show:Nn`. These functions are documented on page 45.)

`_tl_analysis_show:` Here, #1 o- and x-expands to the token; #2 is the category code (one uppercase hexadecimal digit), 0 for control sequences; #3 is the character code, which we ignore. In the cases of control sequences and active characters, the meaning may overflow one line, and we want to truncate it. Those cases are thus separated out.

```

4103 \cs\_new:Npn \_tl\_analysis\_show:
4104 {
4105   \exp\_after:wN \_tl\_analysis\_show\_loop:wNw \g\_tl\_analysis\_result\_tl
4106   \s\_tl { ? \prg\_break: } \s\_tl
4107   \prg\_break\_point:
4108 }
4109 \cs\_new:Npn \_tl\_analysis\_show\_loop:wNw #1 \s\_tl #2 #3 \s\_tl
4110 {
4111   \use\_none:n #2
4112   \iow\_newline: > \use:nn { ~ } { ~ }
4113   \if\_int\_compare:w "#2 = \c\_zero\_int
4114     \exp\_after:wN \_tl\_analysis\_show\_cs:n
4115   \else:
4116     \if\_int\_compare:w "#2 = 13 \exp\_stop\_f:
4117     \exp\_after:wN \exp\_after:wN
4118     \exp\_after:wN \_tl\_analysis\_show\_active:n
4119   \else:
4120     \exp\_after:wN \exp\_after:wN
4121     \exp\_after:wN \_tl\_analysis\_show\_normal:n
4122   \fi:
4123   \fi:
4124   {#1}

```

```

4125   \_tl_analysis_show_loop:wNw
4126 }

```

(End definition for _tl_analysis_show: and _tl_analysis_show_loop:wNw.)

_tl_analysis_show_normal:n Non-active characters are a simple matter of printing the character, and its meaning. Our test suite checks that begin-group and end-group characters do not mess up T_EX's alignment status.

```

4127 \cs_new:Npn \_tl_analysis_show_normal:n #1
4128 {
4129   \exp_after:wN \token_to_str:N #1 ~
4130   ( \exp_after:wN \token_to_meaning:N #1 )
4131 }

```

(End definition for _tl_analysis_show_normal:n.)

_tl_analysis_show_value:N This expands to the value of #1 if it has any.

```

4132 \cs_new:Npn \_tl_analysis_show_value:N #1
4133 {
4134   \token_if_expandable:NF #1
4135   {
4136     \token_if_chardef:NTF #1 \prg_break: { }
4137     \token_if_mathchardef:NTF #1 \prg_break: { }
4138     \token_if_dim_register:NTF #1 \prg_break: { }
4139     \token_if_int_register:NTF #1 \prg_break: { }
4140     \token_if_skip_register:NTF #1 \prg_break: { }
4141     \token_if_toks_register:NTF #1 \prg_break: { }
4142     \use_none:nnn
4143     \prg_break_point:
4144     \use:n { \exp_after:wN = \tex_the:D #1 }
4145   }
4146 }

```

(End definition for _tl_analysis_show_value:N.)

_tl_analysis_show_cs:n Control sequences and active characters are printed in the same way, making sure not to go beyond the \l_iow_line_count_int. In case of an overflow, we replace the last characters by \c_tl_analysis_show_etc_str.

```

\_tl_analysis_show_active:n
\_tl_analysis_show_long:nn
\_tl_analysis_show_long_aux:nnnn
4147 \cs_new:Npn \_tl_analysis_show_cs:n #1
4148 { \exp_args:No \_tl_analysis_show_long:nn {#1} { control~sequence= } }
4149 \cs_new:Npn \_tl_analysis_show_active:n #1
4150 { \exp_args:No \_tl_analysis_show_long:nn {#1} { active~character= } }
4151 \cs_new:Npn \_tl_analysis_show_long:nn #1
4152 {
4153   \_tl_analysis_show_long_aux:oofn
4154   { \token_to_str:N #1 }
4155   { \token_to_meaning:N #1 }
4156   { \_tl_analysis_show_value:N #1 }
4157 }
4158 \cs_new:Npn \_tl_analysis_show_long_aux:nnnn #1#2#3#4
4159 {
4160   \int_compare:nNnTF
4161   { \str_count:n { #1 ~ ( #4 #2 #3 ) } }
4162   > { \l_iow_line_count_int - 3 }

```

```

4163     {
4164       \str_range:nnn { #1 ~ ( #4 #2 #3 ) } { 1 }
4165       {
4166         \l_iow_line_count_int - 3
4167         - \str_count:N \c__tl_analysis_show_etc_str
4168       }
4169       \c__tl_analysis_show_etc_str
4170     }
4171     { #1 ~ ( #4 #2 #3 ) }
4172   }
4173   \cs_generate_variant:Nn \__tl_analysis_show_long_aux:nnnn { oof }

```

(End definition for __tl_analysis_show_cs:n and others.)

44.10 Peeking ahead

\peek_analysis_map_break: The break statements use the general \prg_map_break:Nn.
\peek_analysis_map_break:n

```

4174 \cs_new:Npn \peek_analysis_map_break:
4175   { \prg_map_break:Nn \peek_analysis_map_break: { } }
4176 \cs_new:Npn \peek_analysis_map_break:n
4177   { \prg_map_break:Nn \peek_analysis_map_break: }

```

(End definition for \peek_analysis_map_break: and \peek_analysis_map_break:n. These functions are documented on page 194.)

\l__tl_peek_charcode_int

```

4178 \int_new:N \l__tl_peek_charcode_int

```

(End definition for \l__tl_peek_charcode_int.)

__tl_analysis_char_arg:Nw
 __tl_analysis_char_arg_aux:Nw

After a call to \futurelet \l__tl_analysis_token followed by a stringified character token (either explicit space or catcode other character), grab the argument and pass it to #1. We only need to do anything in the case of a space.

```

4179 \cs_new:Npn \__tl_analysis_char_arg:Nw
4180   {
4181     \if_meaning:w \l__tl_analysis_token \c_space_token
4182       \exp_after:wN \__tl_analysis_char_arg_aux:Nw
4183     \fi:
4184   }
4185 \cs_new:Npn \__tl_analysis_char_arg_aux:Nw #1 ~ { #1 { ~ } }

```

(End definition for __tl_analysis_char_arg:Nw and __tl_analysis_char_arg_aux:Nw.)

\peek_analysis_map_inline:n

__tl_peek_analysis_loop:NNn
 __tl_peek_analysis_test:
 __tl_peek_analysis_normal:N
 __tl_peek_analysis_cs:
 __tl_peek_analysis_char:N
 __tl_peek_analysis_char:nN
 __tl_peek_analysis_special:
 __tl_peek_analysis_retest:
 __tl_peek_analysis_next:
 __tl_peek_analysis_str:
 __tl_peek_analysis_str:w
 __tl_peek_analysis_str:n
 __tl_peek_analysis_active_str:n
 __tl_peek_analysis_explicit:n
 __tl_peek_analysis_escape:
 __tl_peek_analysis_collect:w
 __tl_peek_analysis_collect:n
 __tl_peek_analysis_collect_loop:
 __tl_peek_analysis_collect_test:

Save the user's code in a control sequence that is suitable for nested maps. We may wish to pass to this function an \outer control sequence or active character; for this we will undefine potentially-\outer tokens within a group, closed after the function receives its arguments. This user's code function also calls the loop auxiliary, and includes the trailing \prg_break_point:Nn for when the user wants to stop the loop. The loop auxiliary must remove that break point because it must look at the input stream.

```

4186 \cs_new_protected:Npn \peek_analysis_map_inline:n #1
4187   {
4188     \int_gincr:N \g__kernel_prg_map_int
4189     \cs_set_protected:cpn

```



```

4190     { __tl_analysis_map_ \int_use:N \g__kernel_prg_map_int :nnN }
4191     ##1##2##3
4192     {
4193       \group_end:
4194       #1
4195       \__tl_peek_analysis_loop:NNn
4196       \prg_break_point:Nn \peek_analysis_map_break: { }
4197     }
4198     \__tl_peek_analysis_loop:NNn ? ? ?
4199   }

```

The loop starts a group (closed by the user-code function defined above) with a normalized escape character, and checks if the next token is special or N-type.

```

4200 \cs_new_protected:Npn \__tl_peek_analysis_loop:NNn #1#2#3
4201 {
4202   \group_begin:
4203   \tl_set:Nx \l__tl_peek_code_tl
4204   {
4205     \exp_not:c
4206     { __tl_analysis_map_ \int_use:N \g__kernel_prg_map_int :nnN }
4207   }
4208   \int_set:Nn \tex_escapechar:D { '\ }
4209   \peek_after:Nw \__tl_peek_analysis_test:
4210 }
4211 \cs_new_protected:Npn \__tl_peek_analysis_test:
4212 {
4213   \if_int_odd:w
4214     \if_catcode:w \exp_not:N \l_peek_token { \c_zero_int \fi:
4215     \if_catcode:w \exp_not:N \l_peek_token } \c_zero_int \fi:
4216     \if_meaning:w \l_peek_token \c_space_token \c_zero_int \fi:
4217     \c_one_int
4218     \exp_after:wN \exp_after:wN
4219     \exp_after:wN \__tl_peek_analysis_normal:N
4220     \exp_after:wN \exp_not:N
4221   \else:
4222     \exp_after:wN \__tl_peek_analysis_special:
4223   \fi:
4224 }

```

Normal tokens are not too hard, but can be `\outer`, hence the `\exp_not:N` in the code above. If the token is expandable then it might be an `\outer` or a TeX conditional, so to be safe we set it to `\scan_stop:` (the assignment is local and stopped by the `\group_end:` upon calling the user's code). Then distinguish characters (including active ones and macro parameter characters) from control sequences (whose string representation is more than one character because the escape character is printable). For a control sequence call the user code with suitable arguments.

```

4225 \cs_new_protected:Npn \__tl_peek_analysis_normal:N #1
4226 {
4227   \exp_after:wN \reverse_if:N \exp_after:wN \if_meaning:w
4228   \exp_not:N #1 #1
4229   \tex_let:D #1 \scan_stop:
4230   \tl_put_right:Nn \l__tl_peek_code_tl { { \exp_not:N #1 } }
4231 \else:
4232   \tl_put_right:Nn \l__tl_peek_code_tl { { \exp_not:n {#1} } }

```

```

4233 \fi:
4234 \if_charcode:w
4235   \scan_stop:
4236   \exp_after:wN \use_none:n \token_to_str:N #1 \prg_do_nothing:
4237   \scan_stop:
4238   \exp_after:wN \__tl_peek_analysis_char:N
4239   \exp_after:wN #1
4240 \else:
4241   \exp_after:wN \__tl_peek_analysis_cs:
4242 \fi:
4243 }
4244 \cs_new_protected:Npn \__tl_peek_analysis_cs:
4245 { \l__tl_peek_code_tl { -1 } 0 }
4246 \cs_new_protected:Npn \__tl_peek_analysis_char:N #1
4247 {
4248   \char_set_lccode:nn { '#1 } { 32 }
4249   \tex_lowercase:D { \__tl_peek_analysis_char:nN {#1} } #1
4250 }
4251 \cs_new_protected:Npn \__tl_peek_analysis_char:nN #1#2
4252 {
4253   \cs_set_protected:Npn \__tl_tmp:w ##1 #1 ##2 ##3 \scan_stop:
4254   { \exp_args:No \l__tl_peek_code_tl { \int_value:w '#2 } ##2 }
4255   \exp_after:wN \__tl_tmp:w \c__tl_peek_catcodes_tl \scan_stop:
4256 }

```

For special characters the idea is to eventually act with `\token_to_str:N`, then pick up one by one the characters of this string representation until hitting the token that follows. First determine the character code of (the meaning of) the *token* (which we know is a special token), make sure the escape character is different from it, normalize the meanings of two active characters and the empty control sequence, and filter out these cases in `__tl_peek_analysis_retest:`.

```

4257 \cs_new_protected:Npn \__tl_peek_analysis_special:
4258 {
4259   \tex_let:D \l__tl_analysis_token = ~ \l_peek_token
4260   \int_set:Nn \l__tl_peek_charcode_int
4261   { \__tl_analysis_extract_charcode: }
4262   \if_int_compare:w \l__tl_peek_charcode_int = \tex_escapechar:D
4263   \int_set:Nn \tex_escapechar:D { '\ / }
4264 \fi:
4265 \char_set_active_eq:nN { \l__tl_peek_charcode_int } \scan_stop:
4266 \char_set_active_eq:nN { \tex_escapechar:D } \scan_stop:
4267 \cs_set_eq:cN { } \scan_stop:
4268 \tex_futurelet:D \l__tl_analysis_token
4269 \__tl_peek_analysis_retest:
4270 }
4271 \cs_new_protected:Npn \__tl_peek_analysis_retest:
4272 {
4273   \if_meaning:w \l__tl_analysis_token \scan_stop:
4274   \exp_after:wN \__tl_peek_analysis_normal:N
4275 \else:
4276   \exp_after:wN \__tl_peek_analysis_next:
4277 \fi:
4278 }

```

At this point we know the meaning of the *token* in the input stream is `\l_peek_`

token, either a space (32, 10) or a begin-group or end-group token (catcode 1 or 2), and we excluded a few cases that would be difficult later (empty control sequence, active character with the same character code as its meaning or as the escape character). Now look at the *<next token>* following it using a combination of `\afterassignment` and `\futurelet`. The syntax of this primitive is `\futurelet <peek token> <first token> <next token>`, and it sets *<peek token>* equal to *<next token>*. Traditionally, one takes *<first token>* to be some macro that regains control of the code and, e.g., analyses *<peek token>*. Here, both *<first token>* and *<next token>* are mostly unknown tokens in the input stream (but we know the *<first token>* has catcode 1, 2 or 10), where *<first token>* was already stored as `\l_peek_token`, and we regain control using `\afterassignment`, which inserts its argument after the assignment, hence after *<peek token>* but before *<first token>*.

```

4279 \cs_new_protected:Npn \__tl_peek_analysis_next:
4280 {
4281   \tl_if_empty:oT { \tex_the:D \tex_everyeof:D }
4282   { \tex_everyeof:D { \scan_stop: } }
4283   \tex_afterassignment:D \__tl_peek_analysis_str:
4284   \tex_futurelet:D \l__tl_analysis_next_token
4285 }

```

We then hit the *<first token>* with `\token_to_str:N` and grab characters until finding `\l__tl_analysis_next_token`. More precisely, by looking at the first character in the string representation of the *<first token>* we distinguish three cases: a stringified control sequence starts with the escape character; for an explicit character we find that same character; for an explicit character we find anything else (we made sure to exclude the case of an active character whose string representation coincides with the other two cases).

```

4286 \cs_new_protected:Npn \__tl_peek_analysis_str:
4287 {
4288   \exp_after:wN \tex_futurelet:D
4289   \exp_after:wN \l__tl_analysis_token
4290   \exp_after:wN \__tl_peek_analysis_str:w
4291   \token_to_str:N
4292 }
4293 \cs_new_protected:Npn \__tl_peek_analysis_str:w
4294 { \__tl_analysis_char_arg:Nw \__tl_peek_analysis_str:n }
4295 \cs_new_protected:Npn \__tl_peek_analysis_str:n #1
4296 {
4297   \int_case:nnF { '#1 }
4298   {
4299     { \l__tl_peek_charcode_int }
4300     { \__tl_peek_analysis_explicit:n {#1} }
4301     { \tex_escapechar:D } { \__tl_peek_analysis_escape: }
4302   }
4303   { \__tl_peek_analysis_active_str:n {#1} }
4304 }

```

When `#1` is a stringified active character we pass appropriate arguments to the user's code; thankfully `\char_generate:nn` can make active characters.

```

4305 \cs_new_protected:Npn \__tl_peek_analysis_active_str:n #1
4306 {
4307   \tl_put_right:Nx \l__tl_peek_code_tl
4308   {
4309     { \char_generate:nn { '#1 } { 13 } }
4310     { \int_value:w '#1 }

```

```

4311         \token_to_str:N D
4312     }
4313     \l__tl_peek_code_tl
4314 }

```

When #1 matches the character we had extracted from the meaning of `\l_peek_token`, the token was an explicit character, which can be a standard space, or a begin-group or end-group character with some character code. In the latter two cases we call `\char_generate:nn` with suitable arguments and put suitable `\if_false: \fi:` constructions to make the result balanced and such that o-expanding or x-expanding gives back a single (unbalanced) begin-group or end-group character.

```

4315 \cs_new_protected:Npn \__tl_peek_analysis_explicit:n #1
4316 {
4317     \tl_put_right:Nx \l__tl_peek_code_tl
4318     {
4319         \if_meaning:w \l_peek_token \c_space_token
4320         { ~ } { 32 } \token_to_str:N A
4321     \else:
4322         \if_catcode:w \l_peek_token \c_group_begin_token
4323         {
4324             \exp_not:N \exp_after:wN
4325             \char_generate:nn { '#1 } { 1 }
4326             \exp_not:N \if_false:
4327             \if_false: { \fi: }
4328             \exp_not:N \fi:
4329         }
4330         { \int_value:w '#1 }
4331         1
4332     \else:
4333     {
4334         \exp_not:N \if_false:
4335         { \if_false: } \fi:
4336         \exp_not:N \fi:
4337         \char_generate:nn { '#1 } { 2 }
4338     }
4339     { \int_value:w '#1 }
4340     2
4341     \fi:
4342     \fi:
4343 }
4344 \l__tl_peek_code_tl
4345 }

```

Finally there is the case of a special token whose string representation starts with an escape character, namely the token was a control sequence. In that case we could have grabbed the token directly as an N-type argument, but of course we couldn't know that until we had run all the various tests including stringifying the token. We are thus left with the hard work of picking up one by one the characters in the csname (being careful about spaces), until finding a token that matches the *<next token>* picked up earlier (which was not stringified), such that the control sequence that we found so far indeed has the expected meaning `\l_peek_token`. This comparison with `\l_peek_token` catches a reasonably common case like `\c_group_begin_token _` in which the trailing `_` has category code other: without comparison of the constructed csname with

`\l_peek_token` collection would stop at `\c`, which is wrong.

```

4346 \cs_new_protected:Npn \__tl_peek_analysis_escape:
4347 {
4348   \tl_clear:N \l__tl_internal_a_tl
4349   \tex_futurelet:D \l__tl_analysis_token
4350   \__tl_peek_analysis_collect:w
4351 }
4352 \cs_new_protected:Npn \__tl_peek_analysis_collect:w
4353 { \__tl_analysis_char_arg:Nw \__tl_peek_analysis_collect:n }
4354 \cs_new_protected:Npn \__tl_peek_analysis_collect:n #1
4355 {
4356   \tl_put_right:Nn \l__tl_internal_a_tl {#1}
4357   \__tl_peek_analysis_collect_loop:
4358 }
4359 \cs_new_protected:Npn \__tl_peek_analysis_collect_loop:
4360 {
4361   \tex_futurelet:D \l__tl_analysis_token
4362   \__tl_peek_analysis_collect_test:
4363 }
4364 \cs_new_protected:Npn \__tl_peek_analysis_collect_test:
4365 {
4366   \if_meaning:w \l__tl_analysis_token \l__tl_analysis_next_token
4367     \exp_after:wN \if_meaning:w \cs:w \l__tl_internal_a_tl \cs_end: \l_peek_token
4368     \__tl_peek_analysis_collect_end:NNN
4369   \fi:
4370   \fi:
4371   \__tl_peek_analysis_collect:w
4372 }

```

End by calling the user code with suitable arguments (here `#1`, `#2` are `\fi:`), which closes the group begun early on.

```

4373 \cs_new_protected:Npn \__tl_peek_analysis_collect_end:NNN #1#2#3
4374 {
4375   #1 #2
4376   \tl_put_right:Nx \l__tl_peek_code_tl
4377   {
4378     { \exp_not:N \exp_not:n { \exp_not:c { \l__tl_internal_a_tl } } }
4379     { -1 }
4380     0
4381   }
4382   \l__tl_peek_code_tl
4383 }

```

(End definition for `\peek_analysis_map_inline:n` and others. This function is documented on page [194](#).)

44.11 Messages

`\c__tl_analysis_show_etc_str` When a control sequence (or active character) and its meaning are too long to fit in one line of the terminal, the end is replaced by this token list.

```

4384 \tl_const:Nx \c__tl_analysis_show_etc_str % (
4385 { \token_to_str:N \ETC.) }

```

```

(End definition for \c__tl_analysis_show_etc_str.)

4386 \msg_new:nnn { tl } { show-analysis }
4387 {
4388   The~token~list~ \tl_if_empty:nF {#1} { #1 ~ }
4389   \tl_if_empty:nTF {#2}
4390     { is~empty }
4391     { contains~the~tokens: #2 }
4392 }
4393 \</package>

```

Chapter 45

l3regex implementation

4394 `<*package>`

4395 `<@@=regex>`

45.1 Plan of attack

Most regex engines use backtracking. This allows to provide very powerful features (back-references come to mind first), but it is costly, and raises the problem of catastrophic backtracking. Since `TeX` is not first and foremost a programming language, complicated code tends to run slowly, and we must use faster, albeit slightly more restrictive, techniques, coming from automata theory.

Given a regular expression of n characters, we do the following:

- (Compiling.) Analyse the regex, finding invalid input, and convert it to an internal representation.
- (Building.) Convert the compiled regex to a non-deterministic finite automaton (NFA) with $O(n)$ states which accepts precisely token lists matching that regex.
- (Matching.) Loop through the query token list one token (one “position”) at a time, exploring in parallel every possible path (“active thread”) through the NFA, considering active threads in an order determined by the quantifiers’ greediness.

We use the following vocabulary in the code comments (and in variable names).

- *Group*: index of the capturing group, -1 for non-capturing groups.
- *Position*: each token in the query is labelled by an integer `<position>`, with $\text{min_pos} - 1 \leq \text{<position>} \leq \text{max_pos}$. The lowest and highest positions $\text{min_pos} - 1$ and max_pos correspond to imaginary begin and end markers (with non-existent category code and character code). max_pos is only set quite late in the processing.
- *Query*: the token list to which we apply the regular expression.
- *State*: each state of the NFA is labelled by an integer `<state>` with $\text{min_state} \leq \text{<state>} < \text{max_state}$.
- *Active thread*: state of the NFA that is reached when reading the query token list for the matching. Those threads are ordered according to the greediness of quantifiers.

- *Step*: used when matching, starts at 0, incremented every time a character is read, and is not reset when searching for repeated matches. The integer `\l__regex_step_int` is a unique id for all the steps of the matching algorithm.

We use `l3intarray` to manipulate arrays of integers. We also abuse \TeX 's `\toks` registers, by accessing them directly by number rather than tying them to control sequence using the `\newtoks` allocation functions. Specifically, these arrays and `\toks` are used as follows. When building, `\toks<state>` holds the tests and actions to perform in the `<state>` of the NFA. When matching,

- `\g__regex_state_active_intarray` holds the last `<step>` in which each `<state>` was active.
- `\g__regex_thread_info_intarray` consists of blocks for each `<thread>` (with $\text{min_thread} \leq \langle thread \rangle < \text{max_thread}$). Each block has $1+2\backslash\l_regex_capturing_group_int$ entries: the `<state>` in which the `<thread>` currently is, followed by the beginnings of all submatches, and then the ends of all submatches. The `<threads>` are ordered starting from the best to the least preferred.
- `\g__regex_submatch_prev_intarray`, `\g__regex_submatch_begin_intarray` and `\g__regex_submatch_end_intarray` hold, for each submatch (as would be extracted by `\regex_extract_all:nnN`), the place where the submatch started to be looked for and its two end-points. For historical reasons, the minimum index is twice `max_state`, and the used registers go up to `\l__regex_submatch_int`. They are organized in blocks of `\l__regex_capturing_group_int` entries, each block corresponding to one match with all its submatches stored in consecutive entries.

When actually building the result,

- `\toks<position>` holds `<tokens>` which o- and x-expand to the `<position>`-th token in the query.
- `\g__regex_balance_intarray` holds the balance of begin-group and end-group character tokens which appear before that point in the token list.

The code is structured as follows. Variables are introduced in the relevant section. First we present some generic helper functions. Then comes the code for compiling a regular expression, and for showing the result of the compilation. The building phase converts a compiled regex to NFA states, and the automaton is run by the code in the following section. The only remaining brick is parsing the replacement text and performing the replacement. We are then ready for all the user functions. Finally, messages, and a little bit of tracing code.

45.2 Helpers

`__regex_int_eval:w` Access the primitive: performance is key here, so we do not use the slower route *via* `\int_eval:n`.

4396 `\cs_new_eq:NN __regex_int_eval:w \tex_numexpr:D`

(End definition for `__regex_int_eval:w`.)

`_regex_standard_escapechar:` Make the `\escapechar` into the standard backslash.

4397 `\cs_new_protected:Npn _regex_standard_escapechar:`

4398 `{ \int_set:Nn \tex_escapechar:D { '\ } }`

(End definition for `_regex_standard_escapechar:.`)

`_regex_toks_use:w` Unpack a `\toks` given its number.

```
4399 \cs_new:Npn \_regex_toks_use:w { \tex_the:D \tex_toks:D }
```

(End definition for `_regex_toks_use:w`.)

`_regex_toks_clear:N` Empty a `\toks` or set it to a value, given its number.

```
\_regex_toks_set:Nn 4400 \cs_new_protected:Npn \_regex_toks_clear:N #1
\_regex_toks_set:No 4401 { \_regex_toks_set:Nn #1 { } }
4402 \cs_new_eq:NN \_regex_toks_set:Nn \tex_toks:D
4403 \cs_new_protected:Npn \_regex_toks_set:No #1
4404 { \tex_toks:D #1 \exp_after:wN }
```

(End definition for `_regex_toks_clear:N` and `_regex_toks_set:Nn`.)

`_regex_toks_memcpy:NNn` Copy `#3` `\toks` registers from `#2` onwards to `#1` onwards, like C's `memcpy`.

```
4405 \cs_new_protected:Npn \_regex_toks_memcpy:NNn #1#2#3
4406 {
4407   \prg_replicate:nn {#3}
4408   {
4409     \tex_toks:D #1 = \tex_toks:D #2
4410     \int_incr:N #1
4411     \int_incr:N #2
4412   }
4413 }
```

(End definition for `_regex_toks_memcpy:NNn`.)

`_regex_toks_put_left:Nx` During the building phase we wish to add x-expanded material to `\toks`, either to the left
`_regex_toks_put_right:Nx` or to the right. The expansion is done “by hand” for optimization (these operations are
`_regex_toks_put_right:Nn` used quite a lot). The `Nn` version of `_regex_toks_put_right:Nx` is provided because
it is more efficient than x-expanding with `\exp_not:n`.

```
4414 \cs_new_protected:Npn \_regex_toks_put_left:Nx #1#2
4415 {
4416   \cs_set_nopar:Npx \_regex_tmp:w { #2 }
4417   \tex_toks:D #1 \exp_after:wN \exp_after:wN \exp_after:wN
4418   { \exp_after:wN \_regex_tmp:w \tex_the:D \tex_toks:D #1 }
4419 }
4420 \cs_new_protected:Npn \_regex_toks_put_right:Nx #1#2
4421 {
4422   \cs_set_nopar:Npx \_regex_tmp:w {#2}
4423   \tex_toks:D #1 \exp_after:wN
4424   { \tex_the:D \tex_toks:D \exp_after:wN #1 \_regex_tmp:w }
4425 }
4426 \cs_new_protected:Npn \_regex_toks_put_right:Nn #1#2
4427 { \tex_toks:D #1 \exp_after:wN { \tex_the:D \tex_toks:D #1 #2 } }
```

(End definition for `_regex_toks_put_left:Nx` and `_regex_toks_put_right:Nx`.)

`_regex_curr_cs_to_str:` Expands to the string representation of the token (known to be a control sequence) at
the current position `\l_regex_curr_pos_int`. It should only be used in x-expansion to
avoid losing a leading space.

```
4428 \cs_new:Npn \_regex_curr_cs_to_str:
```

```

4429 {
4430   \exp_after:wN \exp_after:wN \exp_after:wN \cs_to_str:N
4431   \l__regex_curr_token_tl
4432 }

```

(End definition for __regex_curr_cs_to_str:.)

```

\__regex_intarray_item:NnF Item of intarray, with a default value.
  \__regex_intarray_item_aux:nNF
4433 \cs_new:Npn \__regex_intarray_item:NnF #1#2
4434 { \exp_args:Nf \__regex_intarray_item_aux:nNF { \int_eval:n {#2} } #1 }
4435 \cs_new:Npn \__regex_intarray_item_aux:nNF #1#2
4436 {
4437   \if_int_compare:w #1 > \c_zero_int
4438     \exp_after:wN \use_i:nn
4439   \else:
4440     \exp_after:wN \use_ii:nn
4441   \fi:
4442   { \__kernel_intarray_item:Nn #2 {#1} }
4443 }

```

(End definition for __regex_intarray_item:NnF and __regex_intarray_item_aux:nNF.)

__regex_maplike_break: Analogous to \tl_map_break:, this correctly exits \tl_map_inline:nn and similar constructions and jumps to the matching \prg_break_point:Nn __regex_maplike_break: { }.

```

4444 \cs_new:Npn \__regex_maplike_break:
4445 { \prg_map_break:Nn \__regex_maplike_break: { } }

```

(End definition for __regex_maplike_break:.)

__regex_tl_odd_items:n Map through a token list one pair at a time, leaving the odd-numbered or even-numbered items (the first item is numbered 1).

```

\__regex_tl_even_items:n
  \__regex_tl_even_items_loop:nn
4446 \cs_new:Npn \__regex_tl_odd_items:n #1 { \__regex_tl_even_items:n { ? #1 } }
4447 \cs_new:Npn \__regex_tl_even_items:n #1
4448 {
4449   \__regex_tl_even_items_loop:nn #1 \q__regex_nil \q__regex_nil
4450   \prg_break_point:
4451 }
4452 \cs_new:Npn \__regex_tl_even_items_loop:nn #1#2
4453 {
4454   \__regex_use_none_delimit_by_q_nil:w #2 \prg_break: \q__regex_nil
4455   { \exp_not:n {#2} }
4456   \__regex_tl_even_items_loop:nn
4457 }

```

(End definition for __regex_tl_odd_items:n, __regex_tl_even_items:n, and __regex_tl_even_items_loop:nn.)

45.2.1 Constants and variables

__regex_tmp:w Temporary function used for various short-term purposes.

```

4458 \cs_new:Npn \__regex_tmp:w { }

```

(End definition for __regex_tmp:w.)

<pre> \l__regex_internal_a_tl \l__regex_internal_b_tl \l__regex_internal_a_int \l__regex_internal_b_int \l__regex_internal_c_int \l__regex_internal_bool \l__regex_internal_seq \g__regex_internal_tl </pre>	<p>Temporary variables used for various purposes.</p> <pre> 4459 \tl_new:N \l__regex_internal_a_tl 4460 \tl_new:N \l__regex_internal_b_tl 4461 \int_new:N \l__regex_internal_a_int 4462 \int_new:N \l__regex_internal_b_int 4463 \int_new:N \l__regex_internal_c_int 4464 \bool_new:N \l__regex_internal_bool 4465 \seq_new:N \l__regex_internal_seq 4466 \tl_new:N \g__regex_internal_tl </pre> <p>(End definition for \l__regex_internal_a_tl and others.)</p>
<pre> \l__regex_build_tl </pre>	<p>This temporary variable is specifically for use with the <code>tl_build</code> machinery.</p> <pre> 4467 \tl_new:N \l__regex_build_tl </pre> <p>(End definition for \l__regex_build_tl.)</p>
<pre> \c__regex_no_match_regex </pre>	<p>This regular expression matches nothing, but is still a valid regular expression. We could use a failing assertion, but I went for an empty class. It is used as the initial value for regular expressions declared using <code>\regex_new:N</code>.</p> <pre> 4468 \tl_const:Nn \c__regex_no_match_regex 4469 { 4470 __regex_branch:n 4471 { __regex_class:NnnnN \c_true_bool { } { 1 } { 0 } \c_true_bool } 4472 } </pre> <p>(End definition for \c__regex_no_match_regex.)</p>
<pre> \l__regex_balance_int </pre>	<p>During this phase, <code>\l__regex_balance_int</code> counts the balance of begin-group and end-group character tokens which appear before a given point in the token list. This variable is also used to keep track of the balance in the replacement text.</p> <pre> 4473 \int_new:N \l__regex_balance_int </pre> <p>(End definition for \l__regex_balance_int.)</p>

45.2.2 Testing characters

<pre> \c__regex_ascii_min_int \c__regex_ascii_max_control_int \c__regex_ascii_max_int </pre>	<pre> 4474 \int_const:Nn \c__regex_ascii_min_int { 0 } 4475 \int_const:Nn \c__regex_ascii_max_control_int { 31 } 4476 \int_const:Nn \c__regex_ascii_max_int { 127 } </pre> <p>(End definition for \c__regex_ascii_min_int, \c__regex_ascii_max_control_int, and \c__regex_ascii_max_int.)</p>
<pre> \c__regex_ascii_lower_int </pre>	<pre> 4477 \int_const:Nn \c__regex_ascii_lower_int { 'a - 'A } </pre> <p>(End definition for \c__regex_ascii_lower_int.)</p>

45.2.3 Internal auxiliaries

`\q__regex_recursion_stop` Internal recursion quarks.

```
4478 \quark_new:N \q__regex_recursion_stop
```

(End definition for `\q__regex_recursion_stop`.)

`\q__regex_nil` Internal quarks.

```
4479 \quark_new:N \q__regex_nil
```

(End definition for `\q__regex_nil`.)

`__regex_use_none_delimit_by_q_recursion_stop:w` Functions to gobble up to a quark.

`__regex_use_i_delimit_by_q_recursion_stop:nw`

`__regex_use_none_delimit_by_q_nil:w`

```
4480 \cs_new:Npn \__regex_use_none_delimit_by_q_recursion_stop:w
```

```
4481   #1 \q__regex_recursion_stop { }
```

```
4482 \cs_new:Npn \__regex_use_i_delimit_by_q_recursion_stop:nw
```

```
4483   #1 #2 \q__regex_recursion_stop {#1}
```

```
4484 \cs_new:Npn \__regex_use_none_delimit_by_q_nil:w #1 \q__regex_nil { }
```

(End definition for `__regex_use_none_delimit_by_q_recursion_stop:w`, `__regex_use_i_delimit_by_q_recursion_stop:nw`, and `__regex_use_none_delimit_by_q_nil:w`.)

`__regex_quark_if_nil_p:n` Branching quark conditional.

`__regex_quark_if_nil:nTF`

```
4485 \__kernel_quark_new_conditional:Nn \__regex_quark_if_nil:N { F }
```

(End definition for `__regex_quark_if_nil:nTF`.)

`__regex_break_point:TF`

`__regex_break_true:w`

When testing whether a character of the query token list matches a given character class in the regular expression, we often have to test it against several ranges of characters, checking if any one of those matches. This is done with a structure like

```

<test1> ... <test_n>
__regex_break_point:TF {<true code>} {<false code>}

```

If any of the tests succeeds, it calls `__regex_break_true:w`, which cleans up and leaves `<true code>` in the input stream. Otherwise, `__regex_break_point:TF` leaves the `<false code>` in the input stream.

```
4486 \cs_new_protected:Npn \__regex_break_true:w
```

```
4487   #1 \__regex_break_point:TF #2 #3 {#2}
```

```
4488 \cs_new_protected:Npn \__regex_break_point:TF #1 #2 { #2 }
```

(End definition for `__regex_break_point:TF` and `__regex_break_true:w`.)

`__regex_item_reverse:n`

This function makes showing regular expressions easier, and lets us define `\D` in terms of `\d` for instance. There is a subtlety: the end of the query is marked by `-2`, and thus matches `\D` and other negated properties; this case is caught by another part of the code.

```
4489 \cs_new_protected:Npn \__regex_item_reverse:n #1
```

```
4490   {
```

```
4491     #1
```

```
4492     \__regex_break_point:TF { } \__regex_break_true:w
```

```
4493   }
```

(End definition for `__regex_item_reverse:n`.)

_regex_item_caseful_equal:n
_regex_item_caseful_range:nn

Simple comparisons triggering _regex_break_true:w when true.

```

4494 \cs_new_protected:Npn \_regex_item_caseful_equal:n #1
4495 {
4496   \if_int_compare:w #1 = \l__regex_curr_char_int
4497   \exp_after:wN \_regex_break_true:w
4498   \fi:
4499 }
4500 \cs_new_protected:Npn \_regex_item_caseful_range:nn #1 #2
4501 {
4502   \reverse_if:N \if_int_compare:w #1 > \l__regex_curr_char_int
4503   \reverse_if:N \if_int_compare:w #2 < \l__regex_curr_char_int
4504   \exp_after:wN \exp_after:wN \exp_after:wN \_regex_break_true:w
4505   \fi:
4506   \fi:
4507 }

```

(End definition for _regex_item_caseful_equal:n and _regex_item_caseful_range:nn.)

_regex_item_caseless_equal:n
_regex_item_caseless_range:nn

For caseless matching, we perform the test both on the curr_char and on the case_changed_char. Before doing the second set of tests, we make sure that case_changed_char has been computed.

```

4508 \cs_new_protected:Npn \_regex_item_caseless_equal:n #1
4509 {
4510   \if_int_compare:w #1 = \l__regex_curr_char_int
4511   \exp_after:wN \_regex_break_true:w
4512   \fi:
4513   \__regex_maybe_compute_ccc:
4514   \if_int_compare:w #1 = \l__regex_case_changed_char_int
4515   \exp_after:wN \_regex_break_true:w
4516   \fi:
4517 }
4518 \cs_new_protected:Npn \_regex_item_caseless_range:nn #1 #2
4519 {
4520   \reverse_if:N \if_int_compare:w #1 > \l__regex_curr_char_int
4521   \reverse_if:N \if_int_compare:w #2 < \l__regex_curr_char_int
4522   \exp_after:wN \exp_after:wN \exp_after:wN \_regex_break_true:w
4523   \fi:
4524   \fi:
4525   \__regex_maybe_compute_ccc:
4526   \reverse_if:N \if_int_compare:w #1 > \l__regex_case_changed_char_int
4527   \reverse_if:N \if_int_compare:w #2 < \l__regex_case_changed_char_int
4528   \exp_after:wN \exp_after:wN \exp_after:wN \_regex_break_true:w
4529   \fi:
4530   \fi:
4531 }

```

(End definition for _regex_item_caseless_equal:n and _regex_item_caseless_range:nn.)

_regex_compute_case_changed_char:

This function is called when \l__regex_case_changed_char_int has not yet been computed. If the current character code is in the range [65,90] (upper-case), then add 32, making it lowercase. If it is in the lower-case letter range [97,122], subtract 32.

```

4532 \cs_new_protected:Npn \_regex_compute_case_changed_char:
4533 {
4534   \int_set_eq:NN \l__regex_case_changed_char_int \l__regex_curr_char_int

```

```

4535 \if_int_compare:w \l__regex_curr_char_int > 'Z \exp_stop_f:
4536 \if_int_compare:w \l__regex_curr_char_int > 'z \exp_stop_f: \else:
4537 \if_int_compare:w \l__regex_curr_char_int < 'a \exp_stop_f: \else:
4538 \int_sub:Nn \l__regex_case_changed_char_int
4539 { \c__regex_ascii_lower_int }
4540 \fi:
4541 \fi:
4542 \else:
4543 \if_int_compare:w \l__regex_curr_char_int < 'A \exp_stop_f: \else:
4544 \int_add:Nn \l__regex_case_changed_char_int
4545 { \c__regex_ascii_lower_int }
4546 \fi:
4547 \fi:
4548 \cs_set_eq:NN \__regex_maybe_compute_ccc: \prg_do_nothing:
4549 }
4550 \cs_new_eq:NN \__regex_maybe_compute_ccc: \__regex_compute_case_changed_char:

```

(End definition for __regex_compute_case_changed_char:.)

__regex_item_equal:n Those must always be defined to expand to a `caseful` (default) or `caseless` version, and not be protected: they must expand when compiling, to hard-code which tests are caseless or caseful.

```

4551 \cs_new_eq:NN \__regex_item_equal:n ?
4552 \cs_new_eq:NN \__regex_item_range:nn ?

```

(End definition for __regex_item_equal:n and __regex_item_range:nn.)

__regex_item_catcode:nT The argument is a sum of powers of 4 with exponents given by the allowed category codes (between 0 and 13). Dividing by a given power of 4 gives an odd result if and only if that category code is allowed. If the catcode does not match, then skip the character code tests which follow.

```

4553 \cs_new_protected:Npn \__regex_item_catcode:
4554 {
4555   "
4556   \if_case:w \l__regex_curr_catcode_int
4557     1      \or: 4      \or: 10      \or: 40
4558   \or: 100  \or:      \or: 1000    \or: 4000
4559   \or: 10000 \or:      \or: 100000  \or: 400000
4560   \or: 1000000 \or: 4000000 \else: 1*0
4561   \fi:
4562 }
4563 \cs_new_protected:Npn \__regex_item_catcode:nT #1
4564 {
4565   \if_int_odd:w \int_eval:n { #1 / \__regex_item_catcode: } \exp_stop_f:
4566   \exp_after:wN \use:n
4567   \else:
4568   \exp_after:wN \use_none:n
4569   \fi:
4570 }
4571 \cs_new_protected:Npn \__regex_item_catcode_reverse:nT #1#2
4572 { \__regex_item_catcode:nT {#1} { \__regex_item_reverse:n {#2} } }

```

(End definition for __regex_item_catcode:nT, __regex_item_catcode_reverse:nT, and __regex_item_catcode:.)

`__regex_item_exact:nn` This matches an exact $\langle category \rangle$ - $\langle character\ code \rangle$ pair, or an exact control sequence, more precisely one of several possible control sequences, separated by `\scan_stop:`.

```

4573 \cs_new_protected:Npn \__regex_item_exact:nn #1#2
4574 {
4575   \if_int_compare:w #1 = \l__regex_curr_catcode_int
4576   \if_int_compare:w #2 = \l__regex_curr_char_int
4577   \exp_after:wN \exp_after:wN \exp_after:wN \__regex_break_true:w
4578   \fi:
4579   \fi:
4580 }
4581 \cs_new_protected:Npn \__regex_item_exact_cs:n #1
4582 {
4583   \int_compare:nNnTF \l__regex_curr_catcode_int = 0
4584   {
4585     \__kernel_tl_set:Nx \l__regex_internal_a_tl
4586     { \scan_stop: \__regex_curr_cs_to_str: \scan_stop: }
4587     \tl_if_in:noTF { \scan_stop: #1 \scan_stop: }
4588     \l__regex_internal_a_tl
4589     { \__regex_break_true:w } { }
4590   }
4591   { }
4592 }

```

(End definition for `__regex_item_exact:nn` and `__regex_item_exact_cs:n`.)

`__regex_item_cs:n` Match a control sequence (the argument is a compiled regex). First test the catcode of the current token to be zero. Then perform the matching test, and break if the csname indeed matches.

```

4593 \cs_new_protected:Npn \__regex_item_cs:n #1
4594 {
4595   \int_compare:nNnTF \l__regex_curr_catcode_int = 0
4596   {
4597     \group_begin:
4598     \__regex_single_match:
4599     \__regex_disable_submatches:
4600     \__regex_build_for_cs:n {#1}
4601     \bool_set_eq:NN \l__regex_saved_success_bool
4602     \g__regex_success_bool
4603     \exp_args:Nx \__regex_match_cs:n { \__regex_curr_cs_to_str: }
4604     \if_meaning:w \c_true_bool \g__regex_success_bool
4605     \group_insert_after:N \__regex_break_true:w
4606     \fi:
4607     \bool_gset_eq:NN \g__regex_success_bool
4608     \l__regex_saved_success_bool
4609     \group_end:
4610   }
4611 }

```

(End definition for `__regex_item_cs:n`.)

45.2.4 Character property tests

`__regex_prop_d:` Character property tests for `\d`, `\W`, *etc.* These character properties are not affected by the `(?i)` option. The characters recognized by each one are as follows: `\d=[0-9]`,
`__regex_prop_h:`
`__regex_prop_s:`
`__regex_prop_v:`
`__regex_prop_w:`
`__regex_prop_N:`

`\w=[0-9A-Z_a-z]`, `\s=[_\^\^I\^J\^L\^M]`, `\h=[_\^\^I]`, `\v=[\^J-\^M]`, and the upper case counterparts match anything that the lower case does not match. The order in which the various tests appear is optimized for usual mostly lower case letter text.

```

4612 \cs_new_protected:Npn \__regex_prop_d:
4613 { \__regex_item_caseful_range:nn { '0 } { '9 } }
4614 \cs_new_protected:Npn \__regex_prop_h:
4615 {
4616   \__regex_item_caseful_equal:n { '\ }
4617   \__regex_item_caseful_equal:n { '\^I }
4618 }
4619 \cs_new_protected:Npn \__regex_prop_s:
4620 {
4621   \__regex_item_caseful_equal:n { '\ }
4622   \__regex_item_caseful_equal:n { '\^I }
4623   \__regex_item_caseful_equal:n { '\^J }
4624   \__regex_item_caseful_equal:n { '\^L }
4625   \__regex_item_caseful_equal:n { '\^M }
4626 }
4627 \cs_new_protected:Npn \__regex_prop_v:
4628 { \__regex_item_caseful_range:nn { '\^J } { '\^M } } % lf, vtab, ff, cr
4629 \cs_new_protected:Npn \__regex_prop_w:
4630 {
4631   \__regex_item_caseful_range:nn { 'a } { 'z }
4632   \__regex_item_caseful_range:nn { 'A } { 'Z }
4633   \__regex_item_caseful_range:nn { '0 } { '9 }
4634   \__regex_item_caseful_equal:n { '_' }
4635 }
4636 \cs_new_protected:Npn \__regex_prop_N:
4637 {
4638   \__regex_item_reverse:n
4639   { \__regex_item_caseful_equal:n { '\^J } }
4640 }

```

(End definition for `__regex_prop_d:` and others.)

```

\__regex_posix_alnum: POSIX properties. No surprise.
\__regex_posix_alpha: 4641 \cs_new_protected:Npn \__regex_posix_alnum:
\__regex_posix_ascii: 4642 { \__regex_posix_alpha: \__regex_posix_digit: }
\__regex_posix_blank: 4643 \cs_new_protected:Npn \__regex_posix_alpha:
\__regex_posix_cntrl: 4644 { \__regex_posix_lower: \__regex_posix_upper: }
\__regex_posix_digit: 4645 \cs_new_protected:Npn \__regex_posix_ascii:
\__regex_posix_graph: 4646 {
\__regex_posix_lower: 4647   \__regex_item_caseful_range:nn
\__regex_posix_print: 4648   \c__regex_ascii_min_int
\__regex_posix_punct: 4649   \c__regex_ascii_max_int
\__regex_posix_space: 4650 }
\__regex_posix_upper: 4651 \cs_new_eq:NN \__regex_posix_blank: \__regex_prop_h:
\__regex_posix_word: 4652 \cs_new_protected:Npn \__regex_posix_cntrl:
\__regex_posix_xdigit: 4653 {
4654   \__regex_item_caseful_range:nn
4655   \c__regex_ascii_min_int
4656   \c__regex_ascii_max_control_int
4657   \__regex_item_caseful_equal:n \c__regex_ascii_max_int
4658 }

```



```

4659 \cs_new_eq:NN \__regex_posix_digit: \__regex_prop_d:
4660 \cs_new_protected:Npn \__regex_posix_graph:
4661   { \__regex_item_caseful_range:nn { '!' } { '~ } }
4662 \cs_new_protected:Npn \__regex_posix_lower:
4663   { \__regex_item_caseful_range:nn { 'a' } { 'z' } }
4664 \cs_new_protected:Npn \__regex_posix_print:
4665   { \__regex_item_caseful_range:nn { '\' } { '~ } }
4666 \cs_new_protected:Npn \__regex_posix_punct:
4667   {
4668     \__regex_item_caseful_range:nn { '!' } { '/' }
4669     \__regex_item_caseful_range:nn { ':' } { '@' }
4670     \__regex_item_caseful_range:nn { '[' } { '[' }
4671     \__regex_item_caseful_range:nn { '{' } { '~' }
4672   }
4673 \cs_new_protected:Npn \__regex_posix_space:
4674   {
4675     \__regex_item_caseful_equal:n { '\' }
4676     \__regex_item_caseful_range:nn { '^I' } { '^M' }
4677   }
4678 \cs_new_protected:Npn \__regex_posix_upper:
4679   { \__regex_item_caseful_range:nn { 'A' } { 'Z' } }
4680 \cs_new_eq:NN \__regex_posix_word: \__regex_prop_w:
4681 \cs_new_protected:Npn \__regex_posix_xdigit:
4682   {
4683     \__regex_posix_digit:
4684     \__regex_item_caseful_range:nn { 'A' } { 'F' }
4685     \__regex_item_caseful_range:nn { 'a' } { 'f' }
4686   }

```

(End definition for `__regex_posix_alnum:` and others.)

45.2.5 Simple character escape

Before actually parsing the regular expression or the replacement text, we go through them once, converting `\n` to the character 10, *etc.* In this pass, we also convert any special character (`*`, `?`, `{`, *etc.*) or escaped alphanumeric character into a marker indicating that this was a special sequence, and replace escaped special characters and non-escaped alphanumeric characters by markers indicating that those were “raw” characters. The rest of the code can then avoid caring about escaping issues (those can become quite complex to handle in combination with ranges in character classes).

Usage: `__regex_escape_use:nnnn` *<inline 1>* *<inline 2>* *<inline 3>* *<{token list}>*
The *<token list>* is converted to a string, then read from left to right, interpreting backslashes as escaping the next character. Unescaped characters are fed to the function *<inline 1>*, and escaped characters are fed to the function *<inline 2>* within an *x*-expansion context (typically those functions perform some tests on their argument to decide how to output them). The escape sequences `\a`, `\e`, `\f`, `\n`, `\r`, `\t` and `\x` are recognized, and those are replaced by the corresponding character, then fed to *<inline 3>*. The result is then left in the input stream. Spaces are ignored unless escaped.

The conversion is done within an *x*-expanding assignment.

`__regex_escape_use:nnnn` The result is built in `\1__regex_internal_a_t1`, which is then left in the input stream. Tracing code is added as appropriate inside this token list. Go through `#4` once, applying `#1`, `#2`, or `#3` as relevant to each character (after de-escaping it).

```

4687 \cs_new_protected:Npn \__regex_escape_use:nnnn #1#2#3#4
4688 {
4689   \group_begin:
4690     \tl_clear:N \l__regex_internal_a_tl
4691     \cs_set:Npn \__regex_escape_unescaped:N ##1 { #1 }
4692     \cs_set:Npn \__regex_escape_escaped:N ##1 { #2 }
4693     \cs_set:Npn \__regex_escape_raw:N ##1 { #3 }
4694     \__regex_standard_escapechar:
4695     \__kernel_tl_gset:Nx \g__regex_internal_tl
4696       { \__kernel_str_to_other_fast:n {#4} }
4697     \tl_put_right:Nx \l__regex_internal_a_tl
4698       {
4699         \exp_after:wN \__regex_escape_loop:N \g__regex_internal_tl
4700         \scan_stop: \prg_break_point:
4701       }
4702     \exp_after:wN
4703     \group_end:
4704     \l__regex_internal_a_tl
4705   }

```

(End definition for __regex_escape_use:nnnn.)

__regex_escape_loop:N __regex_escape_loop:N reads one character: if it is special (space, backslash, or end-marker), perform the associated action, otherwise it is simply an unescaped character. After a backslash, the same is done, but unknown characters are “escaped”.

```

4706 \cs_new:Npn \__regex_escape_loop:N #1
4707 {
4708   \cs_if_exist_use:cF { __regex_escape_\token_to_str:N #1:w }
4709     { \__regex_escape_unescaped:N #1 }
4710   \__regex_escape_loop:N
4711 }
4712 \cs_new:cpn { __regex_escape_ \c_backslash_str :w }
4713   \__regex_escape_loop:N #1
4714 {
4715   \cs_if_exist_use:cF { __regex_escape_/token_to_str:N #1:w }
4716     { \__regex_escape_escaped:N #1 }
4717   \__regex_escape_loop:N
4718 }

```

(End definition for __regex_escape_loop:N and __regex_escape_\:w.)

__regex_escape_unescaped:N __regex_escape_escaped:N __regex_escape_raw:N Those functions are never called before being given a new meaning, so their definitions here don’t matter.

```

4719 \cs_new_eq:NN \__regex_escape_unescaped:N ?
4720 \cs_new_eq:NN \__regex_escape_escaped:N ?
4721 \cs_new_eq:NN \__regex_escape_raw:N ?

```

(End definition for __regex_escape_unescaped:N, __regex_escape_escaped:N, and __regex_escape_raw:N.)

__regex_escape_\scan_stop:w The loop is ended upon seeing the end-marker “break”, with an error if the string ended in a backslash. Spaces are ignored, and \a, \e, \f, \n, \r, \t take their meaning here.

```

\__regex_escape_/scan_stop:w
\__regex_escape_/a:w
\__regex_escape_/e:w
\__regex_escape_/f:w
\__regex_escape_/n:w
\__regex_escape_/r:w
\__regex_escape_/t:w
\__regex_escape_\:w

```

```

4722 \cs_new_eq:cN { __regex_escape_ \iow_char:N\scan_stop: :w } \prg_break:
4723 \cs_new:cpn { __regex_escape_/ \iow_char:N\scan_stop: :w }

```

```

4724 {
4725     \msg_expandable_error:nn { regex } { trailing-backslash }
4726     \prg_break:
4727 }
4728 \cs_new:cpn { __regex_escape_~:w } { }
4729 \cs_new:cpx { __regex_escape_/a:w }
4730 { \exp_not:N \__regex_escape_raw:N \iow_char:N \^^G }
4731 \cs_new:cpx { __regex_escape_/t:w }
4732 { \exp_not:N \__regex_escape_raw:N \iow_char:N \^^I }
4733 \cs_new:cpx { __regex_escape_/n:w }
4734 { \exp_not:N \__regex_escape_raw:N \iow_char:N \^^J }
4735 \cs_new:cpx { __regex_escape_/f:w }
4736 { \exp_not:N \__regex_escape_raw:N \iow_char:N \^^L }
4737 \cs_new:cpx { __regex_escape_/r:w }
4738 { \exp_not:N \__regex_escape_raw:N \iow_char:N \^^M }
4739 \cs_new:cpx { __regex_escape_/e:w }
4740 { \exp_not:N \__regex_escape_raw:N \iow_char:N \^^[ }

```

(End definition for __regex_escape_scan_stop::w and others.)

__regex_escape_/x:w
 __regex_escape_x_end:w
 __regex_escape_x_large:n

When \x is encountered, __regex_escape_x_test:N is responsible for grabbing some hexadecimal digits, and feeding the result to __regex_escape_x_end:w. If the number is too big interrupt the assignment and produce an error, otherwise call __regex_escape_raw:N on the corresponding character token.

```

4741 \cs_new:cpn { __regex_escape_/x:w } \__regex_escape_loop:N
4742 {
4743     \exp_after:wN \__regex_escape_x_end:w
4744     \int_value:w "0 \__regex_escape_x_test:N
4745 }
4746 \cs_new:Npn \__regex_escape_x_end:w #1 ;
4747 {
4748     \int_compare:nNnTF {#1} > \c_max_char_int
4749     {
4750         \msg_expandable_error:nnff { regex } { x-overflow }
4751         {#1} { \int_to_Hex:n {#1} }
4752     }
4753     {
4754         \exp_last_unbraced:Nf \__regex_escape_raw:N
4755         { \char_generate:nn {#1} { 12 } }
4756     }
4757 }

```

(End definition for __regex_escape_/x:w, __regex_escape_x_end:w, and __regex_escape_x_large:n.)

__regex_escape_x_test:N
 __regex_escape_x_testii:N

Find out whether the first character is a left brace (allowing any number of hexadecimal digits), or not (allowing up to two hexadecimal digits). We need to check for the end-of-string marker. Eventually, call either __regex_escape_x_loop:N or __regex_escape_x:N.

```

4758 \cs_new:Npn \__regex_escape_x_test:N #1
4759 {
4760     \if_meaning:w \scan_stop: #1
4761     \exp_after:wN \use_i:nnn \exp_after:wN ;
4762     \fi:
4763     \use:n

```

```

4764     {
4765         \if_charcode:w \c_space_token #1
4766         \exp_after:wN \__regex_escape_x_test:N
4767     \else:
4768         \exp_after:wN \__regex_escape_x_testii:N
4769         \exp_after:wN #1
4770     \fi:
4771 }
4772 }
4773 \cs_new:Npn \__regex_escape_x_testii:N #1
4774 {
4775     \if_charcode:w \c_left_brace_str #1
4776     \exp_after:wN \__regex_escape_x_loop:N
4777 \else:
4778     \__regex_hexadecimal_use:NTF #1
4779     { \exp_after:wN \__regex_escape_x:N }
4780     { ; \exp_after:wN \__regex_escape_loop:N \exp_after:wN #1 }
4781 \fi:
4782 }

```

(End definition for __regex_escape_x_test:N and __regex_escape_x_testii:N.)

__regex_escape_x:N This looks for the second digit in the unbraced case.

```

4783 \cs_new:Npn \__regex_escape_x:N #1
4784 {
4785     \if_meaning:w \scan_stop: #1
4786     \exp_after:wN \use_i:nnn \exp_after:wN ;
4787 \fi:
4788 \use:n
4789 {
4790     \__regex_hexadecimal_use:NTF #1
4791     { ; \__regex_escape_loop:N }
4792     { ; \__regex_escape_loop:N #1 }
4793 }
4794 }

```

(End definition for __regex_escape_x:N.)

__regex_escape_x_loop:N Grab hexadecimal digits, skip spaces, and at the end, check that there is a right brace,
 __regex_escape_x_loop_error: otherwise raise an error outside the assignment.

```

4795 \cs_new:Npn \__regex_escape_x_loop:N #1
4796 {
4797     \if_meaning:w \scan_stop: #1
4798     \exp_after:wN \use_ii:nnn
4799 \fi:
4800 \use_ii:nn
4801 { ; \__regex_escape_x_loop_error:n { } {#1} }
4802 {
4803     \__regex_hexadecimal_use:NTF #1
4804     { \__regex_escape_x_loop:N }
4805     {
4806         \token_if_eq_charcode:NNTF \c_space_token #1
4807         { \__regex_escape_x_loop:N }
4808     }

```

```

4809             ;
4810             \exp_after:wN
4811             \token_if_eq_charcode:NNTF \c_right_brace_str #1
4812             { \__regex_escape_loop:N }
4813             { \__regex_escape_x_loop_error:n {#1} }
4814         }
4815     }
4816 }
4817 }
4818 \cs_new:Npn \__regex_escape_x_loop_error:n #1
4819 {
4820     \msg_expandable_error:nnn { regex } { x-missing-rbrace } {#1}
4821     \__regex_escape_loop:N #1
4822 }

```

(End definition for __regex_escape_x_loop:N and __regex_escape_x_loop_error:.)

__regex_hexadecimal_use:NNTF TEX detects uppercase hexadecimal digits for us but not the lowercase letters, which we need to detect and replace by their uppercase counterpart.

```

4823 \prg_new_conditional:Npnn \__regex_hexadecimal_use:N #1 { TF }
4824 {
4825     \if_int_compare:w 1 < "1 \token_to_str:N #1 \exp_stop_f:
4826     #1 \prg_return_true:
4827     \else:
4828         \if_case:w
4829             \int_eval:n { \exp_after:wN ‘ \token_to_str:N #1 - ‘a }
4830             A
4831             \or: B
4832             \or: C
4833             \or: D
4834             \or: E
4835             \or: F
4836             \else:
4837                 \prg_return_false:
4838                 \exp_after:wN \use_none:n
4839             \fi:
4840             \prg_return_true:
4841         \fi:
4842     }

```

(End definition for __regex_hexadecimal_use:NNTF.)

__regex_char_if_alphanumeric:NNTF These two tests are used in the first pass when parsing a regular expression. That pass is responsible for finding escaped and non-escaped characters, and recognizing which ones have special meanings and which should be interpreted as “raw” characters. Namely,

- alphanumeric characters are “raw” if they are not escaped, and may have a special meaning when escaped;
- non-alphanumeric printable ASCII characters are “raw” if they are escaped, and may have a special meaning when not escaped;
- characters other than printable ASCII are always “raw”.

The code is ugly, and highly based on magic numbers and the ascii codes of characters. This is mostly unavoidable for performance reasons. Maybe the tests can be optimized a little bit more. Here, “alphanumeric” means 0–9, A–Z, a–z; “special” character means non-alphanumeric but printable ascii, from space (hex 20) to del (hex 7E).

```

4843 \prg_new_conditional:Npnn \__regex_char_if_special:N #1 { TF }
4844 {
4845   \if_int_compare:w '#1 > 'Z \exp_stop_f:
4846   \if_int_compare:w '#1 > 'z \exp_stop_f:
4847     \if_int_compare:w '#1 < \c__regex_ascii_max_int
4848     \prg_return_true: \else: \prg_return_false: \fi:
4849   \else:
4850     \if_int_compare:w '#1 < 'a \exp_stop_f:
4851     \prg_return_true: \else: \prg_return_false: \fi:
4852   \fi:
4853 \else:
4854   \if_int_compare:w '#1 > '9 \exp_stop_f:
4855   \if_int_compare:w '#1 < 'A \exp_stop_f:
4856   \prg_return_true: \else: \prg_return_false: \fi:
4857 \else:
4858   \if_int_compare:w '#1 < '0 \exp_stop_f:
4859   \if_int_compare:w '#1 < '\' \exp_stop_f:
4860   \prg_return_false: \else: \prg_return_true: \fi:
4861   \else: \prg_return_false: \fi:
4862 \fi:
4863 \fi:
4864 }
4865 \prg_new_conditional:Npnn \__regex_char_if_alphanumeric:N #1 { TF }
4866 {
4867   \if_int_compare:w '#1 > 'Z \exp_stop_f:
4868   \if_int_compare:w '#1 > 'z \exp_stop_f:
4869   \prg_return_false:
4870 \else:
4871   \if_int_compare:w '#1 < 'a \exp_stop_f:
4872   \prg_return_false: \else: \prg_return_true: \fi:
4873 \fi:
4874 \else:
4875   \if_int_compare:w '#1 > '9 \exp_stop_f:
4876   \if_int_compare:w '#1 < 'A \exp_stop_f:
4877   \prg_return_false: \else: \prg_return_true: \fi:
4878 \else:
4879   \if_int_compare:w '#1 < '0 \exp_stop_f:
4880   \prg_return_false: \else: \prg_return_true: \fi:
4881 \fi:
4882 \fi:
4883 }

```

(End definition for __regex_char_if_alphanumeric:N and __regex_char_if_special:N.)

45.3 Compiling

A regular expression starts its life as a string of characters. In this section, we convert it to internal instructions, resulting in a “compiled” regular expression. This compiled

expression is then turned into states of an automaton in the building phase. Compiled regular expressions consist of the following:

- `__regex_class:NnnnN` $\langle \text{boolean} \rangle \{ \langle \text{tests} \rangle \} \{ \langle \text{min} \rangle \} \{ \langle \text{more} \rangle \} \langle \text{lazyness} \rangle$
- `__regex_group:nnnN` $\{ \langle \text{branches} \rangle \} \{ \langle \text{min} \rangle \} \{ \langle \text{more} \rangle \} \langle \text{lazyness} \rangle$, also `__regex_group_no_capture:nnnN` and `__regex_group_resetting:nnnN` with the same syntax.
- `__regex_branch:n` $\{ \langle \text{contents} \rangle \}$
- `__regex_command_K:`
- `__regex_assertion:Nn` $\langle \text{boolean} \rangle \{ \langle \text{assertion test} \rangle \}$, where the $\langle \text{assertion test} \rangle$ is `__regex_b_test:` or `__regex_Z_test:` or `__regex_A_test:` or `__regex_G_test:`

Tests can be the following:

- `__regex_item_caseful_equal:n` $\{ \langle \text{char code} \rangle \}$
- `__regex_item_caseless_equal:n` $\{ \langle \text{char code} \rangle \}$
- `__regex_item_caseful_range:nn` $\{ \langle \text{min} \rangle \} \{ \langle \text{max} \rangle \}$
- `__regex_item_caseless_range:nn` $\{ \langle \text{min} \rangle \} \{ \langle \text{max} \rangle \}$
- `__regex_item_catcode:nT` $\{ \langle \text{catcode bitmap} \rangle \} \{ \langle \text{tests} \rangle \}$
- `__regex_item_catcode_reverse:nT` $\{ \langle \text{catcode bitmap} \rangle \} \{ \langle \text{tests} \rangle \}$
- `__regex_item_reverse:n` $\{ \langle \text{tests} \rangle \}$
- `__regex_item_exact:nn` $\{ \langle \text{catcode} \rangle \} \{ \langle \text{char code} \rangle \}$
- `__regex_item_exact_cs:n` $\{ \langle \text{csnames} \rangle \}$, more precisely given as $\langle \text{csname} \rangle \backslash \text{scan_stop:} \langle \text{csname} \rangle \backslash \text{scan_stop:} \langle \text{csname} \rangle$ and so on in a brace group.
- `__regex_item_cs:n` $\{ \langle \text{compiled regex} \rangle \}$

45.3.1 Variables used when compiling

`\l__regex_group_level_int` We make sure to open the same number of groups as we close.

```
4884 \int_new:N \l__regex_group_level_int
```

(End definition for `\l__regex_group_level_int`.)

`\l__regex_mode_int` While compiling, ten modes are recognized, labelled -63 , -23 , -6 , -2 , 0 , 2 , 3 , 6 , 23 , 63 .
`\c__regex_cs_in_class_mode_int` See section 45.3.3. We only define some of these as constants.

```

\c__regex_cs_mode_int
\c__regex_outer_mode_int
\c__regex_catcode_mode_int
\c__regex_class_mode_int
\c__regex_catcode_in_class_mode_int
4885 \int_new:N \l__regex_mode_int
4886 \int_const:Nn \c__regex_cs_in_class_mode_int { -6 }
4887 \int_const:Nn \c__regex_cs_mode_int { -2 }
4888 \int_const:Nn \c__regex_outer_mode_int { 0 }
4889 \int_const:Nn \c__regex_catcode_mode_int { 2 }
4890 \int_const:Nn \c__regex_class_mode_int { 3 }
4891 \int_const:Nn \c__regex_catcode_in_class_mode_int { 6 }
```

(End definition for `\l__regex_mode_int` and others.)

`\l__regex_catcodes_int` We wish to allow constructions such as `\c[^BE](. . \cL[a-z] . .)`, where the outer catcode test applies to the whole group, but is superseded by the inner catcode test. For this to work, we need to keep track of lists of allowed category codes: `\l__regex_catcodes_int` and `\l__regex_default_catcodes_int` are bitmaps, sums of 4^c , for all allowed catcodes c . The latter is local to each capturing group, and we reset `\l__regex_catcodes_int` to that value after each character or class, changing it only when encountering a `\c` escape. The boolean records whether the list of categories of a catcode test has to be inverted: compare `\c[^BE]` and `\c[BE]`.

```
4892 \int_new:N \l__regex_catcodes_int
4893 \int_new:N \l__regex_default_catcodes_int
4894 \bool_new:N \l__regex_catcodes_bool
```

(End definition for `\l__regex_catcodes_int`, `\l__regex_default_catcodes_int`, and `\l__regex_catcodes_bool`.)

`\c__regex_catcode_C_int` Constants: 4^c for each category, and the sum of all powers of 4.

```
4895 \int_const:Nn \c__regex_catcode_C_int { "1 }
4896 \int_const:Nn \c__regex_catcode_B_int { "4 }
4897 \int_const:Nn \c__regex_catcode_E_int { "10 }
4898 \int_const:Nn \c__regex_catcode_M_int { "40 }
4899 \int_const:Nn \c__regex_catcode_T_int { "100 }
4900 \int_const:Nn \c__regex_catcode_P_int { "1000 }
4901 \int_const:Nn \c__regex_catcode_U_int { "4000 }
4902 \int_const:Nn \c__regex_catcode_D_int { "10000 }
4903 \int_const:Nn \c__regex_catcode_S_int { "100000 }
4904 \int_const:Nn \c__regex_catcode_L_int { "400000 }
4905 \int_const:Nn \c__regex_catcode_O_int { "1000000 }
4906 \int_const:Nn \c__regex_catcode_A_int { "4000000 }
\c__regex_all_catcodes_int 4907 \int_const:Nn \c__regex_all_catcodes_int { "5515155 }
```

(End definition for `\c__regex_catcode_C_int` and others.)

`\l__regex_internal_regex` The compilation step stores its result in this variable.

```
4908 \cs_new_eq:NN \l__regex_internal_regex \c__regex_no_match_regex
```

(End definition for `\l__regex_internal_regex`.)

`\l__regex_show_prefix_seq` This sequence holds the prefix that makes up the line displayed to the user. The various items must be removed from the right, which is tricky with a token list, hence we use a sequence.

```
4909 \seq_new:N \l__regex_show_prefix_seq
```

(End definition for `\l__regex_show_prefix_seq`.)

`\l__regex_show_lines_int` A hack. To know whether a given class has a single item in it or not, we count the number of lines when showing the class.

```
4910 \int_new:N \l__regex_show_lines_int
```

(End definition for `\l__regex_show_lines_int`.)

45.3.2 Generic helpers used when compiling

`_regex_two_if_eq:NNNTF` Used to compare pairs of things like `_regex_compile_special:N ?` together. It's often inconvenient to get the catcodes of the character to match so we just compare the character code. Besides, the expanding behaviour of `\\if:w` is very useful as that means we can use `\\c_left_brace_str` and the like.

```

4911 \\prg_new_conditional:Npnn \\_regex_two_if_eq:NNNN #1#2#3#4 { TF }
4912 {
4913     \\if_meaning:w #1 #3
4914     \\if:w #2 #4
4915     \\prg_return_true:
4916     \\else:
4917     \\prg_return_false:
4918     \\fi:
4919     \\else:
4920     \\prg_return_false:
4921     \\fi:
4922 }
```

(End definition for _regex_two_if_eq:NNNTF.)

`_regex_get_digits:NTFw` If followed by some raw digits, collect them one by one in the integer variable `#1`, and
`_regex_get_digits_loop:w` take the true branch. Otherwise, take the false branch.

```

4923 \\cs_new_protected:Npn \\_regex_get_digits:NTFw #1#2#3#4#5
4924 {
4925     \\_regex_if_raw_digit:NNTF #4 #5
4926     { #1 = #5 \\_regex_get_digits_loop:nw {#2} }
4927     { #3 #4 #5 }
4928 }
4929 \\cs_new:Npn \\_regex_get_digits_loop:nw #1#2#3
4930 {
4931     \\_regex_if_raw_digit:NNTF #2 #3
4932     { #3 \\_regex_get_digits_loop:nw {#1} }
4933     { \\scan_stop: #1 #2 #3 }
4934 }
```

(End definition for _regex_get_digits:NTFw and _regex_get_digits_loop:w.)

`_regex_if_raw_digit:NNTF` Test used when grabbing digits for the `{m,n}` quantifier. It only accepts non-escaped digits.

```

4935 \\prg_new_conditional:Npnn \\_regex_if_raw_digit:NN #1#2 { TF }
4936 {
4937     \\if_meaning:w \\_regex_compile_raw:N #1
4938     \\if_int_compare:w 1 < 1 #2 \\exp_stop_f:
4939     \\prg_return_true:
4940     \\else:
4941     \\prg_return_false:
4942     \\fi:
4943     \\else:
4944     \\prg_return_false:
4945     \\fi:
4946 }
```

(End definition for _regex_if_raw_digit:NNTF.)

45.3.3 Mode

When compiling the NFA corresponding to a given regex string, we can be in ten distinct modes, which we label by some magic numbers:

- 6 `[\c{...}]` control sequence in a class,
- 2 `\c{...}` control sequence,
- 0 ... outer,
- 2 `\c...` catcode test,
- 6 `[\c...]` catcode test in a class,
- 63 `[\c{[...]}]` class inside mode -6,
- 23 `\c{[...]}` class inside mode -2,
- 3 `[...]` class inside mode 0,
- 23 `\c[...]` class inside mode 2,
- 63 `[\c[...]]` class inside mode 6.

This list is exhaustive, because `\c` escape sequences cannot be nested, and character classes cannot be nested directly. The choice of numbers is such as to optimize the most useful tests, and make transitions from one mode to another as simple as possible.

- Even modes mean that we are not directly in a character class. In this case, a left bracket appends 3 to the mode. In a character class, a right bracket changes the mode as $m \rightarrow (m - 15)/13$, truncated.
- Grouping, assertion, and anchors are allowed in non-positive even modes (0, -2, -6), and do not change the mode. Otherwise, they trigger an error.
- A left bracket is special in even modes, appending 3 to the mode; in those modes, quantifiers and the dot are recognized, and the right bracket is normal. In odd modes (within classes), the left bracket is normal, but the right bracket ends the class, changing the mode from m to $(m - 15)/13$, truncated; also, ranges are recognized.
- In non-negative modes, left and right braces are normal. In negative modes, however, left braces trigger a warning; right braces end the control sequence, going from -2 to 0 or -6 to 3, with error recovery for odd modes.
- Properties (such as the `\d` character class) can appear in any mode.

`__regex_if_in_class:TF` Test whether we are directly in a character class (at the innermost level of nesting). There, many escape sequences are not recognized, and special characters are normal. Also, for every raw character, we must look ahead for a possible raw dash.

```
4947 \cs_new:Npn \__regex_if_in_class:TF
4948 {
4949   \if_int_odd:w \l__regex_mode_int
4950     \exp_after:wN \use_i:nn
4951   \else:
4952     \exp_after:wN \use_ii:nn
4953   \fi:
4954 }
```

(End definition for `_regex_if_in_class:TF`.)

`_regex_if_in_cs:TF` Right braces are special only directly inside control sequences (at the inner-most level of nesting, not counting groups).

```

4955 \cs_new:Npn \_regex_if_in_cs:TF
4956   {
4957     \if_int_odd:w \l__regex_mode_int
4958       \exp_after:wN \use_ii:nn
4959     \else:
4960       \if_int_compare:w \l__regex_mode_int < \c__regex_outer_mode_int
4961         \exp_after:wN \exp_after:wN \exp_after:wN \use_i:nn
4962       \else:
4963         \exp_after:wN \exp_after:wN \exp_after:wN \use_ii:nn
4964       \fi:
4965     \fi:
4966   }

```

(End definition for `_regex_if_in_cs:TF`.)

`_regex_if_in_class_or_catcode:TF` Assertions are only allowed in modes 0, -2, and -6, *i.e.*, even, non-positive modes.

```

4967 \cs_new:Npn \_regex_if_in_class_or_catcode:TF
4968   {
4969     \if_int_odd:w \l__regex_mode_int
4970       \exp_after:wN \use_i:nn
4971     \else:
4972       \if_int_compare:w \l__regex_mode_int > \c__regex_outer_mode_int
4973         \exp_after:wN \exp_after:wN \exp_after:wN \use_i:nn
4974       \else:
4975         \exp_after:wN \exp_after:wN \exp_after:wN \use_ii:nn
4976       \fi:
4977     \fi:
4978   }

```

(End definition for `_regex_if_in_class_or_catcode:TF`.)

`_regex_if_within_catcode:TF` This test takes the true branch if we are in a catcode test, either immediately following it (modes 2 and 6) or in a class on which it applies (modes 23 and 63). This is used to tweak how left brackets behave in modes 2 and 6.

```

4979 \cs_new:Npn \_regex_if_within_catcode:TF
4980   {
4981     \if_int_compare:w \l__regex_mode_int > \c__regex_outer_mode_int
4982       \exp_after:wN \use_i:nn
4983     \else:
4984       \exp_after:wN \use_ii:nn
4985     \fi:
4986   }

```

(End definition for `_regex_if_within_catcode:TF`.)

`_regex_chk_c_allowed:T` The `\c` escape sequence is only allowed in modes 0 and 3, *i.e.*, not within any other `\c` escape sequence.

```

4987 \cs_new_protected:Npn \_regex_chk_c_allowed:T
4988   {
4989     \if_int_compare:w \l__regex_mode_int = \c__regex_outer_mode_int

```

```

4990     \exp_after:wN \use:n
4991 \else:
4992   \if_int_compare:w \l__regex_mode_int = \c__regex_class_mode_int
4993     \exp_after:wN \exp_after:wN \exp_after:wN \use:n
4994   \else:
4995     \msg_error:nn { regex } { c-bad-mode }
4996     \exp_after:wN \exp_after:wN \exp_after:wN \use_none:n
4997   \fi:
4998 \fi:
4999 }

```

(End definition for `__regex_chk_c_allowed:T`.)

`__regex_mode_quit:c`: This function changes the mode as it is needed just after a catcode test.

```

5000 \cs_new_protected:Npn \__regex_mode_quit:c:
5001 {
5002   \if_int_compare:w \l__regex_mode_int = \c__regex_catcode_mode_int
5003     \int_set_eq:NN \l__regex_mode_int \c__regex_outer_mode_int
5004   \else:
5005     \if_int_compare:w \l__regex_mode_int =
5006       \c__regex_catcode_in_class_mode_int
5007     \int_set_eq:NN \l__regex_mode_int \c__regex_class_mode_int
5008   \fi:
5009 \fi:
5010 }

```

(End definition for `__regex_mode_quit:c:.`)

45.3.4 Framework

`__regex_compile:w` Used when compiling a user regex or a regex for the `\c{...}` escape sequence within another regex. Start building a token list within a group (with x-expansion at the outset), and set a few variables (group level, catcodes), then start the first branch. At the end, make sure there are no dangling classes nor groups, close the last branch: we are done building `\l__regex_internal_regex`.

```

5011 \cs_new_protected:Npn \__regex_compile:w
5012 {
5013   \group_begin:
5014     \tl_build_begin:N \l__regex_build_tl
5015     \int_zero:N \l__regex_group_level_int
5016     \int_set_eq:NN \l__regex_default_catcodes_int
5017       \c__regex_all_catcodes_int
5018     \int_set_eq:NN \l__regex_catcodes_int \l__regex_default_catcodes_int
5019     \cs_set:Npn \__regex_item_equal:n { \__regex_item_caseful_equal:n }
5020     \cs_set:Npn \__regex_item_range:nn { \__regex_item_caseful_range:nn }
5021     \tl_build_put_right:Nn \l__regex_build_tl
5022       { \__regex_branch:n { \if_false: } \fi: }
5023   }
5024 \cs_new_protected:Npn \__regex_compile_end:
5025 {
5026   \__regex_if_in_class:TF
5027   {
5028     \msg_error:nn { regex } { missing-rbrack }
5029     \use:c { __regex_compile_]: }

```

```

5030     \prg_do_nothing: \prg_do_nothing:
5031   }
5032   { }
5033   \if_int_compare:w \l__regex_group_level_int > \c_zero_int
5034     \msg_error:nnx { regex } { missing-rparen }
5035     { \int_use:N \l__regex_group_level_int }
5036     \prg_replicate:nn
5037       { \l__regex_group_level_int }
5038     {
5039       \tl_build_put_right:Nn \l__regex_build_tl
5040       {
5041         \if_false: { \fi: }
5042         \if_false: { \fi: } { 1 } { 0 } \c_true_bool
5043       }
5044       \tl_build_end:N \l__regex_build_tl
5045       \exp_args:NNNo
5046       \group_end:
5047       \tl_build_put_right:Nn \l__regex_build_tl
5048       { \l__regex_build_tl }
5049     }
5050     \fi:
5051     \tl_build_put_right:Nn \l__regex_build_tl { \if_false: { \fi: } }
5052     \tl_build_end:N \l__regex_build_tl
5053     \exp_args:NNNx
5054     \group_end:
5055     \tl_set:Nn \l__regex_internal_regex { \l__regex_build_tl }
5056   }

```

(End definition for __regex_compile:w and __regex_compile_end:.)

__regex_compile:n The compilation is done between __regex_compile:w and __regex_compile_end:, starting in mode 0. Then __regex_escape_use:nnnn distinguishes special characters, escaped alphanumerics, and raw characters, interpreting \a, \x and other sequences. The 4 trailing \prg_do_nothing: are needed because some functions defined later look up to 4 tokens ahead. Before ending, make sure that any \c{...} is properly closed. No need to check that brackets are closed properly since __regex_compile_end: does that. However, catch the case of a trailing \cL construction.

```

5057 \cs_new_protected:Npn \__regex_compile:n #1
5058 {
5059   \__regex_compile:w
5060   \__regex_standard_escapechar:
5061   \int_set_eq:NN \l__regex_mode_int \c__regex_outer_mode_int
5062   \__regex_escape_use:nnnn
5063   {
5064     \__regex_char_if_special:NTF ##1
5065     \__regex_compile_special:N \__regex_compile_raw:N ##1
5066   }
5067   {
5068     \__regex_char_if_alphanumeric:NTF ##1
5069     \__regex_compile_escaped:N \__regex_compile_raw:N ##1
5070   }
5071   { \__regex_compile_raw:N ##1 }
5072   { #1 }
5073   \prg_do_nothing: \prg_do_nothing:

```

```

5074 \prg_do_nothing: \prg_do_nothing:
5075 \int_compare:nNnT \l__regex_mode_int = \c__regex_catcode_mode_int
5076 { \msg_error:nn { regex } { c-trailing } }
5077 \int_compare:nNnT \l__regex_mode_int < \c__regex_outer_mode_int
5078 {
5079   \msg_error:nn { regex } { c-missing-rbrace }
5080   \__regex_compile_end_cs:
5081   \prg_do_nothing: \prg_do_nothing:
5082   \prg_do_nothing: \prg_do_nothing:
5083 }
5084 \__regex_compile_end:
5085 }

```

(End definition for __regex_compile:n.)

`__regex_compile_use:n` Use a regex, regardless of whether it is given as a string (in which case we need to compile) or as a regex variable. This is used for `\regex_match_case:nn` and related functions to allow a mixture of explicit regex and regex variables.

```

5086 \cs_new_protected:Npn \__regex_compile_use:n #1
5087 {
5088   \tl_if_single_token:nT {#1}
5089   {
5090     \exp_after:wN \__regex_compile_use_aux:w
5091     \token_to_meaning:N #1 ~ \q__regex_nil
5092   }
5093   \__regex_compile:n {#1} \l__regex_internal_regex
5094 }
5095 \cs_new_protected:Npn \__regex_compile_use_aux:w #1 ~ #2 \q__regex_nil
5096 {
5097   \str_if_eq:nnT { #1 ~ } { macro:->\__regex_branch:n }
5098   { \use_ii:nnn }
5099 }

```

(End definition for __regex_compile_use:n.)

`__regex_compile_escaped:N` If the special character or escaped alphanumeric has a particular meaning in regexes, the corresponding function is used. Otherwise, it is interpreted as a raw character. We distinguish special characters from escaped alphanumeric characters because they behave differently when appearing as an end-point of a range.

`__regex_compile_special:N`

```

5100 \cs_new_protected:Npn \__regex_compile_special:N #1
5101 {
5102   \cs_if_exist_use:cF { __regex_compile_#1: }
5103   { \__regex_compile_raw:N #1 }
5104 }
5105 \cs_new_protected:Npn \__regex_compile_escaped:N #1
5106 {
5107   \cs_if_exist_use:cF { __regex_compile_/#1: }
5108   { \__regex_compile_raw:N #1 }
5109 }

```

(End definition for __regex_compile_escaped:N and __regex_compile_special:N.)

`__regex_compile_one:n` This is used after finding one “test”, such as `\d`, or a raw character. If that followed a catcode test (e.g., `\cL`), then restore the mode. If we are not in a class, then the test is

“standalone”, and we need to add `__regex_class:NnnnN` and search for quantifiers. In any case, insert the test, possibly together with a catcode test if appropriate.

```

5110 \cs_new_protected:Npn \__regex_compile_one:n #1
5111 {
5112   \__regex_mode_quit_c:
5113   \__regex_if_in_class:TF { }
5114   {
5115     \tl_build_put_right:Nn \l__regex_build_tl
5116     { \__regex_class:NnnnN \c_true_bool { \if_false: } \fi: }
5117   }
5118   \tl_build_put_right:Nx \l__regex_build_tl
5119   {
5120     \if_int_compare:w \l__regex_catcodes_int <
5121     \c__regex_all_catcodes_int
5122     \__regex_item_catcode:nT { \int_use:N \l__regex_catcodes_int }
5123     { \exp_not:N \exp_not:n {#1} }
5124     \else:
5125     \exp_not:N \exp_not:n {#1}
5126     \fi:
5127   }
5128   \int_set_eq:NN \l__regex_catcodes_int \l__regex_default_catcodes_int
5129   \__regex_if_in_class:TF { } { \__regex_compile_quantifier:w }
5130 }

```

(End definition for `__regex_compile_one:n`.)

`__regex_compile_abort_tokens:n`
`__regex_compile_abort_tokens:x`

This function places the collected tokens back in the input stream, each as a raw character. Spaces are not preserved.

```

5131 \cs_new_protected:Npn \__regex_compile_abort_tokens:n #1
5132 {
5133   \use:x
5134   {
5135     \exp_args:No \tl_map_function:nN { \tl_to_str:n {#1} }
5136     \__regex_compile_raw:N
5137   }
5138 }
5139 \cs_generate_variant:Nn \__regex_compile_abort_tokens:n { x }

```

(End definition for `__regex_compile_abort_tokens:n`.)

45.3.5 Quantifiers

`__regex_compile_if_quantifier:TFw`

This looks ahead and checks whether there are any quantifier (special character equal to either of `?+*{}`). This is useful for the `\u` and `\ur` escape sequences.

```

5140 \cs_new_protected:Npn \__regex_compile_if_quantifier:TFw #1#2#3#4
5141 {
5142   \token_if_eq_meaning:NNTF #3 \__regex_compile_special:N
5143   { \cs_if_exist:cTF { __regex_compile_quantifier_#4:w } }
5144   { \use_ii:nn }
5145   {#1} {#2} #3 #4
5146 }

```

(End definition for `__regex_compile_if_quantifier:TFw`.)

`_regex_compile_quantifier:w` This looks ahead and finds any quantifier (special character equal to either of `?+*{`).

```

5147 \cs_new_protected:Npn \_regex_compile_quantifier:w #1#2
5148 {
5149   \token_if_eq_meaning:NNTF #1 \_regex_compile_special:N
5150   {
5151     \cs_if_exist_use:cF { \_regex_compile_quantifier_#2:w }
5152     { \_regex_compile_quantifier_none: #1 #2 }
5153   }
5154   { \_regex_compile_quantifier_none: #1 #2 }
5155 }

```

(End definition for `_regex_compile_quantifier:w`.)

`_regex_compile_quantifier_none:` Those functions are called whenever there is no quantifier, or a braced construction is invalid (equivalent to no quantifier, and whatever characters were grabbed are left raw).
`_regex_compile_quantifier_abort:xNN`

```

5156 \cs_new_protected:Npn \_regex_compile_quantifier_none:
5157 {
5158   \tl_build_put_right:Nn \l__regex_build_tl
5159   { \if_false: { \fi: } { 1 } { 0 } \c_false_bool }
5160 }
5161 \cs_new_protected:Npn \_regex_compile_quantifier_abort:xNN #1#2#3
5162 {
5163   \_regex_compile_quantifier_none:
5164   \msg_warning:nxxx { regex } { invalid-quantifier } {#1} {#3}
5165   \_regex_compile_abort_tokens:x {#1}
5166   #2 #3
5167 }

```

(End definition for `_regex_compile_quantifier_none:` and `_regex_compile_quantifier_abort:xNN`.)

`_regex_compile_quantifier_lazyness:nnNN` Once the “main” quantifier (`?`, `*`, `+` or a braced construction) is found, we check whether it is lazy (followed by a question mark). We then add to the compiled regex a closing brace (ending `_regex_class:NnnnN` and friends), the start-point of the range, its end-point, and a boolean, true for lazy and false for greedy operators.

```

5168 \cs_new_protected:Npn \_regex_compile_quantifier_lazyness:nnNN #1#2#3#4
5169 {
5170   \_regex_two_if_eq:NNNTF #3 #4 \_regex_compile_special:N ?
5171   {
5172     \tl_build_put_right:Nn \l__regex_build_tl
5173     { \if_false: { \fi: } { #1 } { #2 } \c_true_bool }
5174   }
5175   {
5176     \tl_build_put_right:Nn \l__regex_build_tl
5177     { \if_false: { \fi: } { #1 } { #2 } \c_false_bool }
5178     #3 #4
5179   }
5180 }

```

(End definition for `_regex_compile_quantifier_lazyness:nnNN`.)

`_regex_compile_quantifier_?:w` For each “basic” quantifier, `?`, `*`, `+`, feed the correct arguments to `_regex_compile_`
`_regex_compile_quantifier_*:w` `quantifier_lazyness:nnNN`, `-1` means that there is no upper bound on the number of
`_regex_compile_quantifier_+:w` repetitions.

```

5181 \cs_new_protected:cpn { \_regex_compile_quantifier_?:w }

```



```

5182 { \_regex_compile_quantifier_lazyness:nnNN { 0 } { 1 } }
5183 \cs_new_protected:cpn { \_regex_compile_quantifier_*:w }
5184 { \_regex_compile_quantifier_lazyness:nnNN { 0 } { -1 } }
5185 \cs_new_protected:cpn { \_regex_compile_quantifier_+:w }
5186 { \_regex_compile_quantifier_lazyness:nnNN { 1 } { -1 } }

```

(End definition for _regex_compile_quantifier_?:w, _regex_compile_quantifier_*:w, and _regex_compile_quantifier_+:w.)

```

\_regex_compile_quantifier_{:w
\_regex_compile_quantifier_braced_auxi:w
\_regex_compile_quantifier_braced_auxii:w
\_regex_compile_quantifier_braced_auxiii:w

```

Three possible syntaxes: $\{\langle int \rangle\}$, $\{\langle int \rangle, \}$, or $\{\langle int \rangle, \langle int \rangle\}$. Any other syntax causes us to abort and put whatever we collected back in the input stream, as raw characters, including the opening brace. Grab a number into \l_regex_internal_a_int. If the number is followed by a right brace, the range is $[a, a]$. If followed by a comma, grab one more number, and call the_ii or_iii auxiliary. Those auxiliaries check for a closing brace, leading to the range $[a, \infty]$ or $[a, b]$, encoded as $\{a\}\{-1\}$ and $\{a\}\{b - a\}$.

```

5187 \cs_new_protected:cpn { \_regex_compile_quantifier_ \c_left_brace_str :w }
5188 {
5189   \_regex_get_digits:NTFw \l\_regex\_internal\_a\_int
5190   { \_regex_compile_quantifier_braced_auxi:w }
5191   { \_regex_compile_quantifier_abort:xNN { \c_left_brace_str } }
5192 }
5193 \cs_new_protected:Npn \_regex_compile_quantifier_braced_auxi:w #1#2
5194 {
5195   \str_case_e:nnF { #1 #2 }
5196   {
5197     { \_regex_compile_special:N \c_right_brace_str }
5198     {
5199       \exp_args:No \_regex_compile_quantifier_lazyness:nnNN
5200       { \int_use:N \l\_regex\_internal\_a\_int } { 0 }
5201     }
5202     { \_regex_compile_special:N , }
5203     {
5204       \_regex_get_digits:NTFw \l\_regex\_internal\_b\_int
5205       { \_regex_compile_quantifier_braced_auxiii:w }
5206       { \_regex_compile_quantifier_braced_auxii:w }
5207     }
5208   }
5209   {
5210     \_regex_compile_quantifier_abort:xNN
5211     { \c_left_brace_str \int_use:N \l\_regex\_internal\_a\_int }
5212     #1 #2
5213   }
5214 }
5215 \cs_new_protected:Npn \_regex_compile_quantifier_braced_auxii:w #1#2
5216 {
5217   \_regex_two_if_eq:NNNTF #1 #2 \_regex_compile_special:N \c_right_brace_str
5218   {
5219     \exp_args:No \_regex_compile_quantifier_lazyness:nnNN
5220     { \int_use:N \l\_regex\_internal\_a\_int } { -1 }
5221   }
5222   {
5223     \_regex_compile_quantifier_abort:xNN
5224     { \c_left_brace_str \int_use:N \l\_regex\_internal\_a\_int , }
5225     #1 #2

```

```

5226     }
5227 }
5228 \cs_new_protected:Npn \__regex_compile_quantifier_braced_auxiii:w #1#2
5229 {
5230     \__regex_two_if_eq:NNNTF #1 #2 \__regex_compile_special:N \c_right_brace_str
5231     {
5232         \if_int_compare:w \l__regex_internal_a_int >
5233         \l__regex_internal_b_int
5234         \msg_error:nnxx { regex } { backwards-quantifier }
5235         { \int_use:N \l__regex_internal_a_int }
5236         { \int_use:N \l__regex_internal_b_int }
5237         \int_zero:N \l__regex_internal_b_int
5238     \else:
5239         \int_sub:Nn \l__regex_internal_b_int \l__regex_internal_a_int
5240     \fi:
5241     \exp_args:Noo \__regex_compile_quantifier_lazyness:nnNN
5242     { \int_use:N \l__regex_internal_a_int }
5243     { \int_use:N \l__regex_internal_b_int }
5244 }
5245 {
5246     \__regex_compile_quantifier_abort:xNN
5247     {
5248         \c_left_brace_str
5249         \int_use:N \l__regex_internal_a_int ,
5250         \int_use:N \l__regex_internal_b_int
5251     }
5252     #1 #2
5253 }
5254 }

```

(End definition for `__regex_compile_quantifier_{:w}` and others.)

45.3.6 Raw characters

`__regex_compile_raw_error:N` Within character classes, and following catcode tests, some escaped alphanumeric sequences such as `\b` do not have any meaning. They are replaced by a raw character, after spitting out an error.

```

5255 \cs_new_protected:Npn \__regex_compile_raw_error:N #1
5256 {
5257     \msg_error:nnx { regex } { bad-escape } {#1}
5258     \__regex_compile_raw:N #1
5259 }

```

(End definition for `__regex_compile_raw_error:N`.)

`__regex_compile_raw:N` If we are in a character class and the next character is an unescaped dash, this denotes a range. Otherwise, the current character `#1` matches itself.

```

5260 \cs_new_protected:Npn \__regex_compile_raw:N #1#2#3
5261 {
5262     \__regex_if_in_class:TF
5263     {
5264         \__regex_two_if_eq:NNNTF #2 #3 \__regex_compile_special:N -
5265         { \__regex_compile_range:Nw #1 }
5266     }

```

```

5267         \_regex_compile_one:n
5268         { \_regex_item_equal:n { \int_value:w '#1 } }
5269         #2 #3
5270     }
5271 }
5272 {
5273     \_regex_compile_one:n
5274     { \_regex_item_equal:n { \int_value:w '#1 } }
5275     #2 #3
5276 }
5277 }

```

(End definition for _regex_compile_raw:N.)

_regex_compile_range:Nw
_regex_if_end_range:NNTF

We have just read a raw character followed by a dash; this should be followed by an end-point for the range. Valid end-points are: any raw character; any special character, except a right bracket. In particular, escaped characters are forbidden.

```

5278 \prg_new_protected_conditional:Npnn \_regex_if_end_range:NN #1#2 { TF }
5279 {
5280     \if_meaning:w \_regex_compile_raw:N #1
5281     \prg_return_true:
5282 }else:
5283     \if_meaning:w \_regex_compile_special:N #1
5284     \if_charcode:w ] #2
5285     \prg_return_false:
5286 }else:
5287     \prg_return_true:
5288 }fi:
5289 }else:
5290     \prg_return_false:
5291 }fi:
5292 }fi:
5293 }
5294 \cs_new_protected:Npn \_regex_compile_range:Nw #1#2#3
5295 {
5296     \_regex_if_end_range:NNTF #2 #3
5297     {
5298         \if_int_compare:w '#1 > '#3 \exp_stop_f:
5299         \msg_error:nnxx { regex } { range-backwards } {#1} {#3}
5300     }else:
5301         \tl_build_put_right:Nx \l__regex_build_tl
5302         {
5303             \if_int_compare:w '#1 = '#3 \exp_stop_f:
5304             \_regex_item_equal:n
5305         }else:
5306             \_regex_item_range:nn { \int_value:w '#1 }
5307             \fi:
5308             { \int_value:w '#3 }
5309         }
5310     }fi:
5311 }
5312 {
5313     \msg_warning:nnxx { regex } { range-missing-end }
5314     {#1} { \c_backslash_str #3 }

```

```

5315         \tl_build_put_right:Nx \l__regex_build_tl
5316         {
5317             \__regex_item_equal:n { \int_value:w '#1 \exp_stop_f: }
5318             \__regex_item_equal:n { \int_value:w '- \exp_stop_f: }
5319         }
5320         #2#3
5321     }
5322 }

```

(End definition for `__regex_compile_range:Nw` and `__regex_if_end_range:NNTF`.)

45.3.7 Character properties

`__regex_compile_.`: In a class, the dot has no special meaning. Outside, insert `__regex_prop_.`, which matches any character or control sequence, and refuses `-2` (end-marker).

```

5323 \cs_new_protected:cpx { __regex_compile_.: }
5324 {
5325     \exp_not:N \__regex_if_in_class:TF
5326     { \__regex_compile_raw:N . }
5327     { \__regex_compile_one:n \exp_not:c { __regex_prop_.: } }
5328 }
5329 \cs_new_protected:cpn { __regex_prop_.: }
5330 {
5331     \if_int_compare:w \l__regex_curr_char_int > - 2 \exp_stop_f:
5332     \exp_after:wN \__regex_break_true:w
5333     \fi:
5334 }

```

(End definition for `__regex_compile_.` and `__regex_prop_.`.)

`__regex_compile_/d:` The constants `__regex_prop_d:`, etc. hold a list of tests which match the corresponding character class, and jump to the `__regex_break_point:TF` marker. As for a normal character, we check for quantifiers.

```

5335 \cs_set_protected:Npn \__regex_tmp:w #1#2
5336 {
5337     \cs_new_protected:cpx { __regex_compile_/#1: }
5338     { \__regex_compile_one:n \exp_not:c { __regex_prop_#1: } }
5339     \cs_new_protected:cpx { __regex_compile_/#2: }
5340     {
5341         \__regex_compile_one:n
5342         { \__regex_item_reverse:n \exp_not:c { __regex_prop_#1: } }
5343     }
5344 }
5345 \__regex_tmp:w d D
5346 \__regex_tmp:w h H
5347 \__regex_tmp:w s S
5348 \__regex_tmp:w v V
5349 \__regex_tmp:w w W
5350 \cs_new_protected:cpn { __regex_compile_/N: }
5351 { \__regex_compile_one:n \__regex_prop_N: }

```

(End definition for `__regex_compile_/d:` and others.)

45.3.8 Anchoring and simple assertions

`__regex_compile_anchor_letter:NNN` In modes where assertions are forbidden, anchors such as `\A` produce an error (`\A` is invalid in classes); otherwise they add an `__regex_assertion:Nn` test as appropriate (the only negative assertion is `\B`). The test functions are defined later. The implementation for `$` and `^` is only different from `\A` etc because these are valid in a class.

```

5352 \cs_new_protected:Npn __regex_compile_anchor_letter:NNN #1#2#3
5353 {
5354   __regex_if_in_class_or_catcode:TF { __regex_compile_raw_error:N #1 }
5355   {
5356     \tl_build_put_right:Nn \l__regex_build_tl
5357     { __regex_assertion:Nn #2 {#3} }
5358   }
5359 }
5360 \cs_new_protected:cpn { __regex_compile_/A: }
5361 { __regex_compile_anchor_letter:NNN A \c_true_bool __regex_A_test: }
5362 \cs_new_protected:cpn { __regex_compile_/G: }
5363 { __regex_compile_anchor_letter:NNN G \c_true_bool __regex_G_test: }
5364 \cs_new_protected:cpn { __regex_compile_/Z: }
5365 { __regex_compile_anchor_letter:NNN Z \c_true_bool __regex_Z_test: }
5366 \cs_new_protected:cpn { __regex_compile_/z: }
5367 { __regex_compile_anchor_letter:NNN z \c_true_bool __regex_Z_test: }
5368 \cs_new_protected:cpn { __regex_compile_/b: }
5369 { __regex_compile_anchor_letter:NNN b \c_true_bool __regex_b_test: }
5370 \cs_new_protected:cpn { __regex_compile_/B: }
5371 { __regex_compile_anchor_letter:NNN B \c_false_bool __regex_b_test: }
5372 \cs_set_protected:Npn __regex_tmp:w #1#2
5373 {
5374   \cs_new_protected:cpn { __regex_compile_#1: }
5375   {
5376     __regex_if_in_class_or_catcode:TF { __regex_compile_raw:N #1 }
5377     {
5378       \tl_build_put_right:Nn \l__regex_build_tl
5379       { __regex_assertion:Nn \c_true_bool {#2} }
5380     }
5381   }
5382 }
5383 \exp_args:Nx __regex_tmp:w { \iow_char:N ^ } { __regex_A_test: }
5384 \exp_args:Nx __regex_tmp:w { \iow_char:N $ } { __regex_Z_test: }

```

(End definition for `__regex_compile_anchor_letter:NNN` and others.)

45.3.9 Character classes

`__regex_compile_]:` Outside a class, right brackets have no meaning. In a class, change the mode ($m \rightarrow (m - 15)/13$, truncated) to reflect the fact that we are leaving the class. Look for quantifiers, unless we are still in a class after leaving one (the case of `[... \cL[...] ...]`). quantifiers.

```

5385 \cs_new_protected:cpn { __regex_compile_]: }
5386 {
5387   __regex_if_in_class:TF
5388   {
5389     \if_int_compare:w \l__regex_mode_int >
5390     \c_regex_catcode_in_class_mode_int
5391     \tl_build_put_right:Nn \l__regex_build_tl { \if_false: { \fi: } }

```

```

5392     \fi:
5393     \tex_advance:D \l__regex_mode_int - 15 \exp_stop_f:
5394     \tex_divide:D \l__regex_mode_int 13 \exp_stop_f:
5395     \if_int_odd:w \l__regex_mode_int \else:
5396         \exp_after:wN \__regex_compile_quantifier:w
5397     \fi:
5398 }
5399 { \__regex_compile_raw:N ] }
5400 }

```

(End definition for __regex_compile_[:.))

__regex_compile_[: In a class, left brackets might introduce a POSIX character class, or mean nothing. Immediately following \c<category>, we must insert the appropriate catcode test, then parse the class; we pre-expand the catcode as an optimization. Otherwise (modes 0, -2 and -6) just parse the class. The mode is updated later.

```

5401 \cs_new_protected:cpn { \__regex_compile_[: }
5402 {
5403     \__regex_if_in_class:TF
5404     { \__regex_compile_class_posix_test:w }
5405     {
5406         \__regex_if_within_catcode:TF
5407         {
5408             \exp_after:wN \__regex_compile_class_catcode:w
5409             \int_use:N \l__regex_catcodes_int ;
5410         }
5411         { \__regex_compile_class_normal:w }
5412     }
5413 }

```

(End definition for __regex_compile_[:.))

__regex_compile_class_normal:w In the “normal” case, we insert __regex_class:NnnnN <boolean> in the compiled code. The <boolean> is true for positive classes, and false for negative classes, characterized by a leading ~. The auxiliary __regex_compile_class:TFNN also checks for a leading] which has a special meaning.

```

5414 \cs_new_protected:Npn \__regex_compile_class_normal:w
5415 {
5416     \__regex_compile_class:TFNN
5417     { \__regex_class:NnnnN \c_true_bool }
5418     { \__regex_class:NnnnN \c_false_bool }
5419 }

```

(End definition for __regex_compile_class_normal:w.))

__regex_compile_class_catcode:w This function is called for a left bracket in modes 2 or 6 (catcode test, and catcode test within a class). In mode 2 the whole construction needs to be put in a class (like single character). Then determine if the class is positive or negative, inserting __regex_item_catcode:nT or the reverse variant as appropriate, each with the current catcodes bitmap #1 as an argument, and reset the catcodes.

```

5420 \cs_new_protected:Npn \__regex_compile_class_catcode:w #1;
5421 {
5422     \if_int_compare:w \l__regex_mode_int = \c__regex_catcode_mode_int
5423         \tl_build_put_right:Nn \l__regex_build_tl

```

```

5424         { \_regex_class:NnnnN \c_true_bool { \if_false: } \fi: }
5425     \fi:
5426     \int_set_eq:NN \l__regex_catcodes_int \l__regex_default_catcodes_int
5427     \_regex_compile_class:TFNN
5428     { \_regex_item_catcode:nT {#1} }
5429     { \_regex_item_catcode_reverse:nT {#1} }
5430 }

```

(End definition for _regex_compile_class_catcode:w.)

_regex_compile_class:TFNN If the first character is ^, then the class is negative (use #2), otherwise it is positive (use
_regex_compile_class:NN #1). If the next character is a right bracket, then it should be changed to a raw one.

```

5431 \cs_new_protected:Npn \_regex_compile_class:TFNN #1#2#3#4
5432 {
5433     \l__regex_mode_int = \int_value:w \l__regex_mode_int 3 \exp_stop_f:
5434     \_regex_two_if_eq:NNNTF #3 #4 \_regex_compile_special:N ^
5435     {
5436         \tl_build_put_right:Nn \l__regex_build_tl { #2 { \if_false: } \fi: }
5437         \_regex_compile_class:NN
5438     }
5439     {
5440         \tl_build_put_right:Nn \l__regex_build_tl { #1 { \if_false: } \fi: }
5441         \_regex_compile_class:NN #3 #4
5442     }
5443 }
5444 \cs_new_protected:Npn \_regex_compile_class:NN #1#2
5445 {
5446     \token_if_eq_charcode:NNTF #2 ]
5447     { \_regex_compile_raw:N #2 }
5448     { #1 #2 }
5449 }

```

(End definition for _regex_compile_class:TFNN and _regex_compile_class:NN.)

_regex_compile_class_posix_test:w Here we check for a syntax such as [:alpha:]. We also detect [= and [. which have a
_regex_compile_class_posix:NNNNw meaning in POSIX regular expressions, but are not implemented in l3regex. In case we
_regex_compile_class_posix_loop:w see [:, grab raw characters until hopefully reaching :]. If that's missing, or the POSIX
_regex_compile_class_posix_end:w class is unknown, abort. If all is right, add the test to the current class, with an extra
_regex_item_reverse:n for negative classes.

```

5450 \cs_new_protected:Npn \_regex_compile_class_posix_test:w #1#2
5451 {
5452     \token_if_eq_meaning:NNT \_regex_compile_special:N #1
5453     {
5454         \str_case:nn { #2 }
5455         {
5456             : { \_regex_compile_class_posix:NNNNw }
5457             = {
5458                 \msg_warning:nnx { regex }
5459                 { posix-unsupported } { = }
5460             }
5461             . {
5462                 \msg_warning:nnx { regex }
5463                 { posix-unsupported } { . }
5464             }
5465         }
5466     }
5467 }

```

```

5465     }
5466   }
5467   \__regex_compile_raw:N [ #1 #2
5468 }
5469 \cs_new_protected:Npn \__regex_compile_class_posix:NNNNw #1#2#3#4#5#6
5470 {
5471   \__regex_two_if_eq:NNNTF #5 #6 \__regex_compile_special:N ^
5472   {
5473     \bool_set_false:N \l__regex_internal_bool
5474     \__kernel_tl_set:Nx \l__regex_internal_a_tl { \if_false: } \fi:
5475     \__regex_compile_class_posix_loop:w
5476   }
5477   {
5478     \bool_set_true:N \l__regex_internal_bool
5479     \__kernel_tl_set:Nx \l__regex_internal_a_tl { \if_false: } \fi:
5480     \__regex_compile_class_posix_loop:w #5 #6
5481   }
5482 }
5483 \cs_new:Npn \__regex_compile_class_posix_loop:w #1#2
5484 {
5485   \token_if_eq_meaning:NNTF \__regex_compile_raw:N #1
5486   { #2 \__regex_compile_class_posix_loop:w }
5487   { \if_false: { \fi: } \__regex_compile_class_posix_end:w #1 #2 }
5488 }
5489 \cs_new_protected:Npn \__regex_compile_class_posix_end:w #1#2#3#4
5490 {
5491   \__regex_two_if_eq:NNNTF #1 #2 \__regex_compile_special:N :
5492   { \__regex_two_if_eq:NNNTF #3 #4 \__regex_compile_special:N ] }
5493   { \use_ii:nn }
5494   {
5495     \cs_if_exist:cTF { __regex_posix_ \l__regex_internal_a_tl : }
5496     {
5497       \__regex_compile_one:n
5498       {
5499         \bool_if:NF \l__regex_internal_bool \__regex_item_reverse:n
5500         \exp_not:c { __regex_posix_ \l__regex_internal_a_tl : }
5501       }
5502     }
5503     {
5504       \msg_warning:nxx { regex } { posix-unknown }
5505       { \l__regex_internal_a_tl }
5506       \__regex_compile_abort_tokens:x
5507       {
5508         [: \bool_if:NF \l__regex_internal_bool { ^ }
5509         \l__regex_internal_a_tl :]
5510       }
5511     }
5512   }
5513   {
5514     \msg_error:nxxx { regex } { posix-missing-close }
5515     { [: \l__regex_internal_a_tl } { #2 #4 }
5516     \__regex_compile_abort_tokens:x { [: \l__regex_internal_a_tl }
5517     #1 #2 #3 #4
5518   }

```



```
5519 }
```

(End definition for `_regex_compile_class_posix_test:w` and others.)

45.3.10 Groups and alternations

```
\_regex_compile_group_begin:N
\_regex_compile_group_end:
```

The contents of a regex group are turned into compiled code in `\l__regex_build_tl`, which ends up with items of the form `_regex_branch:n {⟨concatenation⟩}`. This construction is done using `\tl_build_...` functions within a T_EX group, which automatically makes sure that options (case-sensitivity and default catcode) are reset at the end of the group. The argument #1 is `_regex_group:nnnN` or a variant thereof. A small subtlety to support `\cL(abc)` as a shorthand for `(\cLa\cLb\cLc)`: exit any pending catcode test, save the category code at the start of the group as the default catcode for that group, and make sure that the catcode is restored to the default outside the group.

```
5520 \cs_new_protected:Npn \_regex_compile_group_begin:N #1
5521 {
5522   \tl_build_put_right:Nn \l__regex_build_tl { #1 { \if_false: } \fi: }
5523   \_regex_mode_quit_c:
5524   \group_begin:
5525     \tl_build_begin:N \l__regex_build_tl
5526     \int_set_eq:NN \l__regex_default_catcodes_int \l__regex_catcodes_int
5527     \int_incr:N \l__regex_group_level_int
5528     \tl_build_put_right:Nn \l__regex_build_tl
5529       { \_regex_branch:n { \if_false: } \fi: }
5530   }
5531 \cs_new_protected:Npn \_regex_compile_group_end:
5532 {
5533   \if_int_compare:w \l__regex_group_level_int > \c_zero_int
5534     \tl_build_put_right:Nn \l__regex_build_tl { \if_false: { \fi: } }
5535     \tl_build_end:N \l__regex_build_tl
5536     \exp_args:NNNx
5537     \group_end:
5538     \tl_build_put_right:Nn \l__regex_build_tl { \l__regex_build_tl }
5539     \int_set_eq:NN \l__regex_catcodes_int \l__regex_default_catcodes_int
5540     \exp_after:wN \_regex_compile_quantifier:w
5541   \else:
5542     \msg_warning:nn { regex } { extra-rparen }
5543     \exp_after:wN \_regex_compile_raw:N \exp_after:wN )
5544   \fi:
5545 }
```

(End definition for `_regex_compile_group_begin:N` and `_regex_compile_group_end:.`)

```
\_regex_compile_(:
```

In a class, parentheses are not special. In a catcode test inside a class, a left parenthesis gives an error, to catch `[a\cL(bcd)e]`. Otherwise check for a `?`, denoting special groups, and run the code for the corresponding special group.

```
5546 \cs_new_protected:cpn { \_regex_compile_(: }
5547 {
5548   \_regex_if_in_class:TF { \_regex_compile_raw:N ( }
5549   {
5550     \if_int_compare:w \l__regex_mode_int =
5551       \c__regex_catcode_in_class_mode_int
```

```

5552         \msg_error:nn { regex } { c-lparen-in-class }
5553         \exp_after:wN \__regex_compile_raw:N \exp_after:wN (
5554         \else:
5555         \exp_after:wN \__regex_compile_lparen:w
5556         \fi:
5557     }
5558 }
5559 \cs_new_protected:Npn \__regex_compile_lparen:w #1#2#3#4
5560 {
5561     \__regex_two_if_eq:NNNTF #1 #2 \__regex_compile_special:N ?
5562     {
5563         \cs_if_exist_use:cF
5564         { \__regex_compile_special_group\_token_to_str:N #4 :w }
5565         {
5566             \msg_warning:nnx { regex } { special-group-unknown }
5567             { (? #4 }
5568             \__regex_compile_group_begin:N \__regex_group:nnnN
5569             \__regex_compile_raw:N ? #3 #4
5570         }
5571     }
5572     {
5573         \__regex_compile_group_begin:N \__regex_group:nnnN
5574         #1 #2 #3 #4
5575     }
5576 }

```

(End definition for __regex_compile(:.))

__regex_compile_|: In a class, the pipe is not special. Otherwise, end the current branch and open another one.

```

5577 \cs_new_protected:cpn { \__regex_compile_|: }
5578 {
5579     \__regex_if_in_class:TF { \__regex_compile_raw:N | }
5580     {
5581         \tl_build_put_right:Nn \l__regex_build_tl
5582         { \if_false: { \fi: } \__regex_branch:n { \if_false: } \fi: }
5583     }
5584 }

```

(End definition for __regex_compile_|.))

__regex_compile_): Within a class, parentheses are not special. Outside, close a group.

```

5585 \cs_new_protected:cpn { \__regex_compile_): }
5586 {
5587     \__regex_if_in_class:TF { \__regex_compile_raw:N ) }
5588     { \__regex_compile_group_end: }
5589 }

```

(End definition for __regex_compile_).))

_regex_compile_special_group::w Non-capturing, and resetting groups are easy to take care of during compilation; for those groups, the harder parts come when building.

```

5590 \cs_new_protected:cpn { \__regex_compile_special_group::w }
5591 { \__regex_compile_group_begin:N \__regex_group_no_capture:nnnN }
5592 \cs_new_protected:cpn { \__regex_compile_special_group_|:w }
5593 { \__regex_compile_group_begin:N \__regex_group_resetting:nnnN }

```

(End definition for `__regex_compile_special_group_i:w` and `__regex_compile_special_group_l:w`.)

`__regex_compile_special_group_i:w` The match can be made case-insensitive by setting the option with `(?i)`; the original
`__regex_compile_special_group_l:w` behaviour is restored by `(?-i)`. This is the only supported option.

```

5594 \cs_new_protected:Npn __regex_compile_special_group_i:w #1#2
5595 {
5596   __regex_two_if_eq:NNNTF #1 #2 __regex_compile_special:N )
5597   {
5598     \cs_set:Npn __regex_item_equal:n
5599       { __regex_item_caseless_equal:n }
5600     \cs_set:Npn __regex_item_range:nn
5601       { __regex_item_caseless_range:nn }
5602   }
5603   {
5604     \msg_warning:nnx { regex } { unknown-option } { (?i #2 }
5605     __regex_compile_raw:N (
5606     __regex_compile_raw:N ?
5607     __regex_compile_raw:N i
5608     #1 #2
5609   }
5610 }
5611 \cs_new_protected:cpn { __regex_compile_special_group_l:w } #1#2#3#4
5612 {
5613   __regex_two_if_eq:NNNTF #1 #2 __regex_compile_raw:N i
5614   { __regex_two_if_eq:NNNTF #3 #4 __regex_compile_special:N ) }
5615   { \use_ii:nn }
5616   {
5617     \cs_set:Npn __regex_item_equal:n
5618       { __regex_item_caseful_equal:n }
5619     \cs_set:Npn __regex_item_range:nn
5620       { __regex_item_caseful_range:nn }
5621   }
5622   {
5623     \msg_warning:nnx { regex } { unknown-option } { (?-#2#4 }
5624     __regex_compile_raw:N (
5625     __regex_compile_raw:N ?
5626     __regex_compile_raw:N -
5627     #1 #2 #3 #4
5628   }
5629 }

```

(End definition for `__regex_compile_special_group_i:w` and `__regex_compile_special_group_l:w`.)

45.3.11 Catcodes and csnames

`__regex_compile/c:` The `\c` escape sequence can be followed by a capital letter representing a character
`__regex_compile_c_test:NN` category, by a left bracket which starts a list of categories, or by a brace group holding
a regular expression for a control sequence name. Otherwise, raise an error.

```

5630 \cs_new_protected:cpn { __regex_compile/c: }
5631 { __regex_chk_c_allowed:T { __regex_compile_c_test:NN } }
5632 \cs_new_protected:Npn __regex_compile_c_test:NN #1#2
5633 {
5634   \token_if_eq_meaning:NNTF #1 __regex_compile_raw:N
5635   {

```

```

5636     \int_if_exist:cTF { c__regex_catcode_#2_int }
5637     {
5638         \int_set_eq:Nc \l__regex_catcodes_int
5639         { c__regex_catcode_#2_int }
5640         \l__regex_mode_int
5641         = \if_case:w \l__regex_mode_int
5642           \c__regex_catcode_mode_int
5643           \else:
5644             \c__regex_catcode_in_class_mode_int
5645           \fi:
5646         \token_if_eq_charcode:NNT C #2 { \__regex_compile_c_C:NN }
5647     }
5648 }
5649 { \cs_if_exist_use:cF { __regex_compile_c_#2:w } }
5650 {
5651     \msg_error:nmx { regex } { c-missing-category } {#2}
5652     #1 #2
5653 }
5654 }

```

(End definition for __regex_compile_/c: and __regex_compile_c_test:NN.)

__regex_compile_c_C:NN If \cC is not followed by . or (...) then complain because that construction cannot match anything, except in cases like \cC[\c{...}], where it has no effect.

```

5655 \cs_new_protected:Npn \__regex_compile_c_C:NN #1#2
5656 {
5657     \token_if_eq_meaning:NNTF #1 \__regex_compile_special:N
5658     {
5659         \token_if_eq_charcode:NNTF #2 .
5660         { \use_none:n }
5661         { \token_if_eq_charcode:NNTF #2 ( ) % )
5662         }
5663         { \use:n }
5664         { \msg_error:nnn { regex } { c-C-invalid } {#2} }
5665         #1 #2
5666     }

```

(End definition for __regex_compile_c_C:NN.)

__regex_compile_c[:w] When encountering \c[, the task is to collect uppercase letters representing character categories. First check for ^ which negates the list of category codes.

```

\__regex_compile_c_lbrack_loop:NN
\__regex_compile_c_lbrack_add:N
\__regex_compile_c_lbrack_end:
5667 \cs_new_protected:cpn { __regex_compile_c[:w] } #1#2
5668 {
5669     \l__regex_mode_int
5670     = \if_case:w \l__regex_mode_int
5671       \c__regex_catcode_mode_int
5672       \else:
5673         \c__regex_catcode_in_class_mode_int
5674       \fi:
5675     \int_zero:N \l__regex_catcodes_int
5676     \__regex_two_if_eq:NNNTF #1 #2 \__regex_compile_special:N ^
5677     {
5678         \bool_set_false:N \l__regex_catcodes_bool
5679         \__regex_compile_c_lbrack_loop:NN

```

```

5680     }
5681     {
5682         \bool_set_true:N \l__regex_catcodes_bool
5683         \__regex_compile_c_lbrack_loop:NN
5684         #1 #2
5685     }
5686 }
5687 \cs_new_protected:Npn \__regex_compile_c_lbrack_loop:NN #1#2
5688 {
5689     \token_if_eq_meaning:NNTF #1 \__regex_compile_raw:N
5690     {
5691         \int_if_exist:cTF { c__regex_catcode_#2_int }
5692         {
5693             \exp_args:Nc \__regex_compile_c_lbrack_add:N
5694             { c__regex_catcode_#2_int }
5695             \__regex_compile_c_lbrack_loop:NN
5696         }
5697     }
5698     {
5699         \token_if_eq_charcode:NNTF #2 ]
5700         { \__regex_compile_c_lbrack_end: }
5701     }
5702     {
5703         \msg_error:nxx { regex } { c-missing-rbrack } {#2}
5704         \__regex_compile_c_lbrack_end:
5705         #1 #2
5706     }
5707 }
5708 \cs_new_protected:Npn \__regex_compile_c_lbrack_add:N #1
5709 {
5710     \if_int_odd:w \int_eval:n { \l__regex_catcodes_int / #1 } \exp_stop_f:
5711     \else:
5712         \int_add:Nn \l__regex_catcodes_int {#1}
5713     \fi:
5714 }
5715 \cs_new_protected:Npn \__regex_compile_c_lbrack_end:
5716 {
5717     \if_meaning:w \c_false_bool \l__regex_catcodes_bool
5718     \int_set:Nn \l__regex_catcodes_int
5719     { \c__regex_all_catcodes_int - \l__regex_catcodes_int }
5720     \fi:
5721 }

```

(End definition for __regex_compile_c[:w and others.)

__regex_compile_c_{: The case of a left brace is easy, based on what we have done so far: in a group, compile the regular expression, after changing the mode to forbid nesting \c. Additionally, disable submatch tracking since groups don't escape the scope of \c{...}.

```

5722 \cs_new_protected:cpn { __regex_compile_c_ \c_left_brace_str :w }
5723 {
5724     \__regex_compile:w
5725     \__regex_disable_submatches:
5726     \l__regex_mode_int
5727     = \if_case:w \l__regex_mode_int

```

```

5728         \c__regex_cs_mode_int
5729     \else:
5730         \c__regex_cs_in_class_mode_int
5731     \fi:
5732 }

```

(End definition for __regex_compile_c{:.})

__regex_compile_{: We forbid unescaped left braces inside a \c{...} escape because they otherwise lead to the confusing question of whether the first right brace in \c{{}x} should end \c or whether one should match braces.

```

5733 \cs_new_protected:cpn { __regex_compile_ \c_left_brace_str : }
5734 {
5735     \__regex_if_in_cs:TF
5736     { \msg_error:nnn { regex } { cu-lbrace } { c } }
5737     { \exp_after:wN \__regex_compile_raw:N \c_left_brace_str }
5738 }

```

(End definition for __regex_compile_{:})

__regex_compile_{: Non-escaped right braces are only special if they appear when compiling the regular expression for a csname, but not within a class: \c{[{}]} matches the control sequences \{ and \}. So, end compiling the inner regex (this closes any dangling class or group). Then insert the corresponding test in the outer regex. As an optimization, if the control sequence test simply consists of several explicit possibilities (branches) then use __regex_item_exact_cs:n with an argument consisting of all possibilities separated by \scan_stop:.

```

5739 \flag_new:n { __regex_cs }
5740 \cs_new_protected:cpn { __regex_compile_ \c_right_brace_str : }
5741 {
5742     \__regex_if_in_cs:TF
5743     { \__regex_compile_end_cs: }
5744     { \exp_after:wN \__regex_compile_raw:N \c_right_brace_str }
5745 }
5746 \cs_new_protected:Npn \__regex_compile_end_cs:
5747 {
5748     \__regex_compile_end:
5749     \flag_clear:n { __regex_cs }
5750     \__kernel_tl_set:Nx \l__regex_internal_a_tl
5751     {
5752         \exp_after:wN \__regex_compile_cs_aux:Nn \l__regex_internal_regex
5753         \q__regex_nil \q__regex_nil \q__regex_recursion_stop
5754     }
5755     \exp_args:Nx \__regex_compile_one:n
5756     {
5757         \flag_if_raised:nTF { __regex_cs }
5758         { \__regex_item_cs:n { \exp_not:o \l__regex_internal_regex } }
5759         {
5760             \__regex_item_exact_cs:n
5761             { \tl_tail:N \l__regex_internal_a_tl }
5762         }
5763     }
5764 }
5765 \cs_new:Npn \__regex_compile_cs_aux:Nn #1#2

```

```

5766 {
5767   \cs_if_eq:NNTF #1 \__regex_branch:n
5768   {
5769     \scan_stop:
5770     \__regex_compile_cs_aux:NNnnnN #2
5771     \q__regex_nil \q__regex_nil \q__regex_nil
5772     \q__regex_nil \q__regex_nil \q__regex_nil \q__regex_recursion_stop
5773     \__regex_compile_cs_aux:Nn
5774   }
5775   {
5776     \__regex_quark_if_nil:NF #1 { \flag_raise_if_clear:n { __regex_cs } }
5777     \__regex_use_none_delimit_by_q_recursion_stop:w
5778   }
5779 }
5780 \cs_new:Npn \__regex_compile_cs_aux:NNnnnN #1#2#3#4#5#6
5781 {
5782   \bool_lazy_all:nTF
5783   {
5784     { \cs_if_eq_p:NN #1 \__regex_class:NnnnN }
5785     {#2}
5786     { \tl_if_head_eq_meaning_p:nN {#3} \__regex_item_caseful_equal:n }
5787     { \int_compare_p:nNn { \tl_count:n {#3} } = { 2 } }
5788     { \int_compare_p:nNn {#5} = { 0 } }
5789   }
5790   {
5791     \prg_replicate:nn {#4}
5792     { \char_generate:nn { \use_ii:nn #3 } {12} }
5793     \__regex_compile_cs_aux:NNnnnN
5794   }
5795   {
5796     \__regex_quark_if_nil:NF #1
5797     {
5798       \flag_raise_if_clear:n { __regex_cs }
5799       \__regex_use_i_delimit_by_q_recursion_stop:nw
5800     }
5801     \__regex_use_none_delimit_by_q_recursion_stop:w
5802   }
5803 }

```

(End definition for __regex_compile_}: and others.)

45.3.12 Raw token lists with \u

__regex_compile_/u: The \u escape is invalid in classes and directly following a catcode test. Otherwise test for a following r (for \ur), and call an auxiliary responsible for finding the variable name.

```

5804 \cs_new_protected:cpn { __regex_compile_/u: } #1#2
5805 {
5806   \__regex_if_in_class_or_catcode:TF
5807   { \__regex_compile_raw_error:N u #1 #2 }
5808   {
5809     \__regex_two_if_eq:NNNNTF #1 #2 \__regex_compile_raw:N r
5810     { \__regex_compile_u_brace:NNN \__regex_compile_ur_end: }
5811     { \__regex_compile_u_brace:NNN \__regex_compile_u_end: #1 #2 }
5812   }

```

```
5813 }
```

(End definition for `_regex_compile/u:.`)

`_regex_compile_u_brace:NNN` This enforces the presence of a left brace, then starts a loop to find the variable name.

```
5814 \cs_new:Npn \_regex_compile_u_brace:NNN #1#2#3
5815 {
5816   \_regex_two_if_eq:NNNTF #2 #3 \_regex_compile_special:N \c_left_brace_str
5817   {
5818     \tl_set:Nn \l_regex_internal_b_tl {#1}
5819     \_kernel_tl_set:Nx \l_regex_internal_a_tl { \if_false: } \fi:
5820     \_regex_compile_u_loop:NN
5821   }
5822   {
5823     \msg_error:nn { regex } { u-missing-lbrace }
5824     \token_if_eq_meaning:NNTF #1 \_regex_compile_ur_end:
5825     { \_regex_compile_raw:N u \_regex_compile_raw:N r }
5826     { \_regex_compile_raw:N u }
5827     #2 #3
5828   }
5829 }
```

(End definition for `_regex_compile_u_brace:NNN`.)

`_regex_compile_u_loop:NN` We collect the characters for the argument of `\u` within an x-expanding assignment. In principle we could just wait to encounter a right brace, but this is unsafe: if the right brace was missing, then we would reach the end-markers of the regex, and continue, leading to obscure fatal errors. Instead, we only allow raw and special characters, and stop when encountering a special right brace, any escaped character, or the end-marker.

```
5830 \cs_new:Npn \_regex_compile_u_loop:NN #1#2
5831 {
5832   \token_if_eq_meaning:NNTF #1 \_regex_compile_raw:N
5833   { #2 \_regex_compile_u_loop:NN }
5834   {
5835     \token_if_eq_meaning:NNTF #1 \_regex_compile_special:N
5836     {
5837       \exp_after:wN \token_if_eq_charcode:NNTF \c_right_brace_str #2
5838       { \if_false: { \fi: } \l_regex_internal_b_tl }
5839       {
5840         \if_charcode:w \c_left_brace_str #2
5841         \msg_expandable_error:nnn { regex } { cu-lbrace } { u }
5842         \else:
5843           #2
5844         \fi:
5845         \_regex_compile_u_loop:NN
5846       }
5847     }
5848     {
5849       \if_false: { \fi: }
5850       \msg_error:nnx { regex } { u-missing-rbrace } {#2}
5851       \l_regex_internal_b_tl
5852       #1 #2
5853     }
5854   }
5855 }
```


(End definition for `_regex_compile_u_loop:NN`.)

```

\_regex_compile_ur_end: For the \ur{...} construction, once we have extracted the variable's name, we replace
\_regex_compile_ur:n all groups by non-capturing groups in the compiled regex (passed as the argument of
\_regex_compile_ur_aux:w \_regex_compile_ur:n). If that has a single branch (namely \tl_if_empty:oTF is
false) and there is no quantifier, then simply insert the contents of this branch (obtained
by \use_ii:nn, which is expanded later). In all other cases, insert a non-capturing group
and look for quantifiers to determine the number of repetition etc.

5856 \cs_new_protected:Npn \_regex_compile_ur_end:
5857 {
5858   \group_begin:
5859   \cs_set:Npn \_regex_group:nnnN { \_regex_group_no_capture:nnnN }
5860   \cs_set:Npn \_regex_group_resetting:nnnN { \_regex_group_no_capture:nnnN }
5861   \exp_args:NNx
5862   \group_end:
5863   \_regex_compile_ur:n { \use:c { \l__regex_internal_a_tl } }
5864 }
5865 \cs_new_protected:Npn \_regex_compile_ur:n #1
5866 {
5867   \tl_if_empty:oTF { \_regex_compile_ur_aux:w #1 {} ? ? \q__regex_nil }
5868   { \_regex_compile_if_quantifier:TFw }
5869   { \use_i:nn }
5870   {
5871     \tl_build_put_right:Nn \l__regex_build_tl
5872     { \_regex_group_no_capture:nnnN { \if_false: } \fi: #1 }
5873     \_regex_compile_quantifier:w
5874   }
5875   { \tl_build_put_right:Nn \l__regex_build_tl { \use_ii:nn #1 } }
5876 }
5877 \cs_new:Npn \_regex_compile_ur_aux:w \_regex_branch:n #1#2#3 \q__regex_nil {#2}

```

(End definition for `_regex_compile_ur_end:`, `_regex_compile_ur:n`, and `_regex_compile_ur_aux:w`.)

```

\_regex_compile_u_end: Once we have extracted the variable's name, we check for quantifiers, in which case we
\_regex_compile_u_payload: set up a non-capturing group with a single branch. Inside this branch (we omit it and
the group if there is no quantifier), \_regex_compile_u_payload: puts the right tests
corresponding to the contents of the variable, which we store in \l__regex_internal_a_tl. The behaviour of \u then depends on whether we are within a \c{...} escape (in
this case, the variable is turned to a string), or not.

```

```

5878 \cs_new_protected:Npn \_regex_compile_u_end:
5879 {
5880   \_regex_compile_if_quantifier:TFw
5881   {
5882     \tl_build_put_right:Nn \l__regex_build_tl
5883     {
5884       \_regex_group_no_capture:nnnN { \if_false: } \fi:
5885       \_regex_branch:n { \if_false: } \fi:
5886     }
5887     \_regex_compile_u_payload:
5888     \tl_build_put_right:Nn \l__regex_build_tl { \if_false: { \fi: } }
5889     \_regex_compile_quantifier:w
5890   }

```

```

5891     { \_regex_compile_u_payload: }
5892   }
5893 \cs_new_protected:Npn \_regex_compile_u_payload:
5894 {
5895   \tl_set:Nv \l__regex_internal_a_tl { \l__regex_internal_a_tl }
5896   \if_int_compare:w \l__regex_mode_int = \c__regex_outer_mode_int
5897     \_regex_compile_u_not_cs:
5898   \else:
5899     \_regex_compile_u_in_cs:
5900   \fi:
5901 }

```

(End definition for _regex_compile_u_end: and _regex_compile_u_payload:.)

_regex_compile_u_in_cs: When \u appears within a control sequence, we convert the variable to a string with escaped spaces. Then for each character insert a class matching exactly that character, once.

```

5902 \cs_new_protected:Npn \_regex_compile_u_in_cs:
5903 {
5904   \__kernel_tl_gset:Nx \g__regex_internal_tl
5905   {
5906     \exp_args:No \__kernel_str_to_other_fast:n
5907     { \l__regex_internal_a_tl }
5908   }
5909   \tl_build_put_right:Nx \l__regex_build_tl
5910   {
5911     \tl_map_function:NN \g__regex_internal_tl
5912     \_regex_compile_u_in_cs_aux:n
5913   }
5914 }
5915 \cs_new:Npn \_regex_compile_u_in_cs_aux:n #1
5916 {
5917   \_regex_class:NnnN \c_true_bool
5918   { \_regex_item_caseful_equal:n { \int_value:w '#1 } }
5919   { 1 } { 0 } \c_false_bool
5920 }

```

(End definition for _regex_compile_u_in_cs:.)

_regex_compile_u_not_cs: In mode 0, the \u escape adds one state to the NFA for each token in \l__regex_internal_a_tl. If a given *<token>* is a control sequence, then insert a string comparison test, otherwise, _regex_item_exact:nn which compares catcode and character code.

```

5921 \cs_new_protected:Npn \_regex_compile_u_not_cs:
5922 {
5923   \tl_analysis_map_inline:Nn \l__regex_internal_a_tl
5924   {
5925     \tl_build_put_right:Nx \l__regex_build_tl
5926     {
5927       \_regex_class:NnnN \c_true_bool
5928       {
5929         \if_int_compare:w "##3 = \c_zero_int
5930           \_regex_item_exact_cs:n
5931           { \exp_after:wN \cs_to_str:N ##1 }
5932         \else:

```

```

5933         \__regex_item_exact:nn { \int_value:w "##3 } { ##2 }
5934         \fi:
5935     }
5936     { 1 } { 0 } \c_false_bool
5937 }
5938 }
5939 }

```

(End definition for __regex_compile_u_not_cs:.)

45.3.13 Other

__regex_compile_/K: The \K control sequence is currently the only “command”, which performs some action, rather than matching something. It is allowed in the same contexts as \b. At the compilation stage, we leave it as a single control sequence, defined later.

```

5940 \cs_new_protected:cpn { __regex_compile_/K: }
5941 {
5942     \int_compare:nNnTF \l__regex_mode_int = \c__regex_outer_mode_int
5943     { \tl_build_put_right:Nn \l__regex_build_tl { \__regex_command_K: } }
5944     { \__regex_compile_raw_error:N K }
5945 }

```

(End definition for __regex_compile_/K:.)

45.3.14 Showing regexes

Before showing a regex we check that it is “clean” in the sense that it has the correct internal structure. We do this (in the implementation of \regex_show:N and \regex_log:N) by comparing it with a cleaned-up version of the same regex. Along the way we also need similar functions for other types: all __regex_clean_⟨type⟩:n functions produce valid ⟨type⟩ tokens (bool, explicit integer, etc.) from arbitrary input, and the output coincides with the input if that was valid.

```

\__regex_clean_bool:n
\__regex_clean_int:n
\__regex_clean_int_aux:N
\__regex_clean_regex:n
\__regex_clean_regex_loop:w
\__regex_clean_branch:n
\__regex_clean_branch_loop:n
\__regex_clean_assertion:Nn
\__regex_clean_class:NnnN
\__regex_clean_group:nnnN
\__regex_clean_class:n
\__regex_clean_class_loop:nnn
\__regex_clean_exact_cs:n
\__regex_clean_exact_cs:w
5946 \cs_new:Npn \__regex_clean_bool:n #1
5947 {
5948     \tl_if_single:nTF {#1}
5949     { \bool_if:NTF #1 \c_true_bool \c_false_bool }
5950     { \c_true_bool }
5951 }
5952 \cs_new:Npn \__regex_clean_int:n #1
5953 {
5954     \tl_if_head_eq_meaning:nNTF {#1} -
5955     { - \exp_args:No \__regex_clean_int:n { \use_none:n #1 } }
5956     { \int_eval:n { 0 \str_map_function:nN {#1} \__regex_clean_int_aux:N } }
5957 }
5958 \cs_new:Npn \__regex_clean_int_aux:N #1
5959 {
5960     \if_int_compare:w 1 < 1 #1 ~
5961     #1
5962     \else:
5963         \exp_after:wN \str_map_break:
5964     \fi:
5965 }
5966 \cs_new:Npn \__regex_clean_regex:n #1

```

```

5967 {
5968     \__regex_clean_regex_loop:w #1
5969     \__regex_branch:n { \q_recursion_tail } \q_recursion_stop
5970 }
5971 \cs_new:Npn \__regex_clean_regex_loop:w #1 \__regex_branch:n #2
5972 {
5973     \quark_if_recursion_tail_stop:n {#2}
5974     \__regex_branch:n { \__regex_clean_branch:n {#2} }
5975     \__regex_clean_regex_loop:w
5976 }
5977 \cs_new:Npn \__regex_clean_branch:n #1
5978 {
5979     \__regex_clean_branch_loop:n #1
5980     ? ? ? ? ? \prg_break_point:
5981 }
5982 \cs_new:Npn \__regex_clean_branch_loop:n #1
5983 {
5984     \tl_if_single:nF {#1} { \prg_break: }
5985     \token_case_meaning:NnF #1
5986     {
5987         \__regex_command_K: { #1 \__regex_clean_branch_loop:n }
5988         \__regex_assertion:Nn { #1 \__regex_clean_assertion:Nn }
5989         \__regex_class:NnnN { #1 \__regex_clean_class:NnnN }
5990         \__regex_group:nnnN { #1 \__regex_clean_group:nnnN }
5991         \__regex_group_no_capture:nnnN { #1 \__regex_clean_group:nnnN }
5992         \__regex_group_resetting:nnnN { #1 \__regex_clean_group:nnnN }
5993     }
5994     { \prg_break: }
5995 }
5996 \cs_new:Npn \__regex_clean_assertion:Nn #1#2
5997 {
5998     \__regex_clean_bool:n {#1}
5999     \tl_if_single:nF {#2} { { \__regex_A_test: } \prg_break: }
6000     \token_case_meaning:NnTF #2
6001     {
6002         \__regex_A_test: { }
6003         \__regex_G_test: { }
6004         \__regex_Z_test: { }
6005         \__regex_b_test: { }
6006     }
6007     { {#2} }
6008     { { \__regex_A_test: } \prg_break: }
6009     \__regex_clean_branch_loop:n
6010 }
6011 \cs_new:Npn \__regex_clean_class:NnnN #1#2#3#4#5
6012 {
6013     \__regex_clean_bool:n {#1}
6014     { \__regex_clean_class:n {#2} }
6015     { \int_max:nn { 0 } { \__regex_clean_int:n {#3} } }
6016     { \int_max:nn { -1 } { \__regex_clean_int:n {#4} } }
6017     \__regex_clean_bool:n {#5}
6018     \__regex_clean_branch_loop:n
6019 }
6020 \cs_new:Npn \__regex_clean_group:nnnN #1#2#3#4

```

```

6021 {
6022   { \_regex_clean_regex:n {#1} }
6023   { \int_max:nn { 0 } { \_regex_clean_int:n {#2} } }
6024   { \int_max:nn { -1 } { \_regex_clean_int:n {#3} } }
6025   \_regex_clean_bool:n {#4}
6026   \_regex_clean_branch_loop:n
6027 }
6028 \cs_new:Npn \_regex_clean_class:n #1
6029 { \_regex_clean_class_loop:nnn #1 ????? \prg_break_point: }
6030 \cs_new:Npn \_regex_clean_class_loop:nnn #1#2#3
6031 {
6032   \tl_if_single:nF {#1} { \prg_break: }
6033   \token_case_meaning:NnTF #1
6034   {
6035     \_regex_item_cs:n { #1 { \_regex_clean_regex:n {#2} } }
6036     \_regex_item_exact_cs:n { #1 { \_regex_clean_exact_cs:n {#2} } }
6037     \_regex_item_caseful_equal:n { #1 { \_regex_clean_int:n {#2} } }
6038     \_regex_item_caseless_equal:n { #1 { \_regex_clean_int:n {#2} } }
6039     \_regex_item_reverse:n { #1 { \_regex_clean_class:n {#2} } }
6040   }
6041   { \_regex_clean_class_loop:nnn {#3} }
6042   {
6043     \token_case_meaning:NnTF #1
6044     {
6045       \_regex_item_caseful_range:nn { }
6046       \_regex_item_caseless_range:nn { }
6047       \_regex_item_exact:nn { }
6048     }
6049     { #1 { \_regex_clean_int:n {#2} } { \_regex_clean_int:n {#3} } }
6050     {
6051       \token_case_meaning:NnTF #1
6052       {
6053         \_regex_item_catcode:nT { }
6054         \_regex_item_catcode_reverse:nT { }
6055       }
6056       {
6057         #1 { \_regex_clean_int:n {#2} } { \_regex_clean_class:n {#3} }
6058         \_regex_clean_class_loop:nnn
6059       }
6060       { \prg_break: }
6061     }
6062   }
6063 }
6064 \cs_new:Npn \_regex_clean_exact_cs:n #1
6065 {
6066   \exp_last_unbraced:Nf \use_none:n
6067   {
6068     \_regex_clean_exact_cs:w #1
6069     \scan_stop: \q_recursion_tail \scan_stop:
6070     \q_recursion_stop
6071   }
6072 }
6073 \cs_new:Npn \_regex_clean_exact_cs:w #1 \scan_stop:
6074 {

```

```

6075 \quark_if_recursion_tail_stop:n {#1}
6076 \scan_stop: \tl_to_str:n {#1}
6077 \__regex_clean_exact_cs:w
6078 }

```

(End definition for __regex_clean_bool:n and others.)

__regex_show:N Within a group and within \tl_build_begin:N ... \tl_build_end:N we redefine all the function that can appear in a compiled regex, then run the regex. The result stored in \l__regex_internal_a_tl is then meant to be shown.

```

6079 \cs_new_protected:Npn \__regex_show:N #1
6080 {
6081   \group_begin:
6082     \tl_build_begin:N \l__regex_build_tl
6083     \cs_set_protected:Npn \__regex_branch:n
6084       {
6085         \seq_pop_right:NN \l__regex_show_prefix_seq
6086         \l__regex_internal_a_tl
6087         \__regex_show_one:n { +-branch }
6088         \seq_put_right:No \l__regex_show_prefix_seq
6089         \l__regex_internal_a_tl
6090         \use:n
6091       }
6092     \cs_set_protected:Npn \__regex_group:nnnN
6093       { \__regex_show_group_aux:nnnnN { } }
6094     \cs_set_protected:Npn \__regex_group_no_capture:nnnN
6095       { \__regex_show_group_aux:nnnnN { ~(no-capture) } }
6096     \cs_set_protected:Npn \__regex_group_resetting:nnnN
6097       { \__regex_show_group_aux:nnnnN { ~(resetting) } }
6098     \cs_set_eq:NN \__regex_class:NnnnN \__regex_show_class:NnnnN
6099     \cs_set_protected:Npn \__regex_command_K:
6100       { \__regex_show_one:n { reset-match~start~(\iow_char:N\K) } }
6101     \cs_set_protected:Npn \__regex_assertion:Nn ##1##2
6102       {
6103         \__regex_show_one:n
6104         { \bool_if:NF ##1 { negative~ } assertion:~##2 }
6105       }
6106     \cs_set:Npn \__regex_b_test: { word~boundary }
6107     \cs_set:Npn \__regex_Z_test: { anchor~at~end~(\iow_char:N\Z) }
6108     \cs_set:Npn \__regex_A_test: { anchor~at~start~(\iow_char:N\A) }
6109     \cs_set:Npn \__regex_G_test: { anchor~at~start~of~match~(\iow_char:N\G) }
6110     \cs_set_protected:Npn \__regex_item_caseful_equal:n ##1
6111       { \__regex_show_one:n { char~code~\__regex_show_char:n{##1} } }
6112     \cs_set_protected:Npn \__regex_item_caseful_range:nn ##1##2
6113       {
6114         \__regex_show_one:n
6115         { range~[\__regex_show_char:n{##1}, \__regex_show_char:n{##2}] }
6116       }
6117     \cs_set_protected:Npn \__regex_item_caseless_equal:n ##1
6118       { \__regex_show_one:n { char~code~\__regex_show_char:n{##1}~(caseless) } }
6119     \cs_set_protected:Npn \__regex_item_caseless_range:nn ##1##2
6120       {
6121         \__regex_show_one:n
6122         { Range~[\__regex_show_char:n{##1}, \__regex_show_char:n{##2}]~(caseless) }

```

```

6123     }
6124     \cs_set_protected:Npn \__regex_item_catcode:nT
6125     { \__regex_show_item_catcode:NnT \c_true_bool }
6126     \cs_set_protected:Npn \__regex_item_catcode_reverse:nT
6127     { \__regex_show_item_catcode:NnT \c_false_bool }
6128     \cs_set_protected:Npn \__regex_item_reverse:n
6129     { \__regex_show_scope:nn { Reversed~match } }
6130     \cs_set_protected:Npn \__regex_item_exact:nn ##1##2
6131     { \__regex_show_one:n { char~\__regex_show_char:n{##2},~catcode~##1 } }
6132     \cs_set_eq:NN \__regex_item_exact_cs:n \__regex_show_item_exact_cs:n
6133     \cs_set_protected:Npn \__regex_item_cs:n
6134     { \__regex_show_scope:nn { control~sequence } }
6135     \cs_set:cpn { __regex_prop_.: } { \__regex_show_one:n { any~token } }
6136     \seq_clear:N \l__regex_show_prefix_seq
6137     \__regex_show_push:n { ~ }
6138     \cs_if_exist_use:N #1
6139     \tl_build_end:N \l__regex_build_tl
6140     \exp_args:NNNo
6141     \group_end:
6142     \tl_set:Nn \l__regex_internal_a_tl { \l__regex_build_tl }
6143 }

```

(End definition for __regex_show:N.)

__regex_show_char:n Show a single character, together with its ascii representation if available. This could be extended to beyond ascii. It is not ideal for parentheses themselves.

```

6144 \cs_new:Npn \__regex_show_char:n #1
6145 {
6146     \int_eval:n {#1}
6147     \int_compare:nT { 32 <= #1 <= 126 }
6148     { ~ ( \char_generate:nn {#1} {12} ) }
6149 }

```

(End definition for __regex_show_char:n.)

__regex_show_one:n Every part of the final message go through this function, which adds one line to the output, with the appropriate prefix.

```

6150 \cs_new_protected:Npn \__regex_show_one:n #1
6151 {
6152     \int_incr:N \l__regex_show_lines_int
6153     \tl_build_put_right:Nx \l__regex_build_tl
6154     {
6155         \exp_not:N \iow_newline:
6156         \seq_map_function:NN \l__regex_show_prefix_seq \use:n
6157         #1
6158     }
6159 }

```

(End definition for __regex_show_one:n.)

__regex_show_push:n Enter and exit levels of nesting. The scope function prints its first argument as an “introduction”, then performs its second argument in a deeper level of nesting.

```

\__regex_show_pop:
\__regex_show_scope:nn
6160 \cs_new_protected:Npn \__regex_show_push:n #1
6161 { \seq_put_right:Nx \l__regex_show_prefix_seq { #1 ~ } }

```

```

6162 \cs_new_protected:Npn \__regex_show_pop:
6163 { \seq_pop_right:NN \l__regex_show_prefix_seq \l__regex_internal_a_tl }
6164 \cs_new_protected:Npn \__regex_show_scope:nn #1#2
6165 {
6166   \__regex_show_one:n {#1}
6167   \__regex_show_push:n { ~ }
6168   #2
6169   \__regex_show_pop:
6170 }

```

(End definition for __regex_show_push:n, __regex_show_pop:, and __regex_show_scope:nn.)

__regex_show_group_aux:nnnnN

We display all groups in the same way, simply adding a message, (no capture) or (resetting), to special groups. The odd \use_ii:nn avoids printing a spurious +-branch for the first branch.

```

6171 \cs_new_protected:Npn \__regex_show_group_aux:nnnnN #1#2#3#4#5
6172 {
6173   \__regex_show_one:n { ,-group~begin #1 }
6174   \__regex_show_push:n { | }
6175   \use_ii:nn #2
6176   \__regex_show_pop:
6177   \__regex_show_one:n
6178   { '-group~end \__regex_msg_repeated:nnN {#3} {#4} #5 }
6179 }

```

(End definition for __regex_show_group_aux:nnnnN.)

__regex_show_class:NnnnN

I'm entirely unhappy about this function: I couldn't find a way to test if a class is a single test. Instead, collect the representation of the tests in the class. If that had more than one line, write Match or Don't match on its own line, with the repeating information if any. Then the various tests on lines of their own, and finally a line. Otherwise, we need to evaluate the representation of the tests again (since the prefix is incorrect). That's clunky, but not too expensive, since it's only one test.

```

6180 \cs_set:Npn \__regex_show_class:NnnnN #1#2#3#4#5
6181 {
6182   \group_begin:
6183   \tl_build_begin:N \l__regex_build_tl
6184   \int_zero:N \l__regex_show_lines_int
6185   \__regex_show_push:n {~}
6186   #2
6187   \int_compare:nTF { \l__regex_show_lines_int = 0 }
6188   {
6189     \group_end:
6190     \__regex_show_one:n { \bool_if:NTF #1 { Fail } { Pass } }
6191   }
6192   {
6193     \bool_if:nTF
6194     { #1 && \int_compare_p:n { \l__regex_show_lines_int = 1 } }
6195     {
6196       \group_end:
6197       #2
6198       \tl_build_put_right:Nn \l__regex_build_tl
6199       { \__regex_msg_repeated:nnN {#3} {#4} #5 }
6200     }

```



```

6201     {
6202         \tl_build_end:N \l__regex_build_tl
6203         \exp_args:NNNo
6204         \group_end:
6205         \tl_set:Nn \l__regex_internal_a_tl \l__regex_build_tl
6206         \__regex_show_one:n
6207         {
6208             \bool_if:NTF #1 { Match } { Don't~match }
6209             \__regex_msg_repeated:nnN {#3} {#4} #5
6210         }
6211         \tl_build_put_right:Nx \l__regex_build_tl
6212         { \exp_not:o \l__regex_internal_a_tl }
6213     }
6214 }
6215 }

```

(End definition for `__regex_show_class:NnnnN`.)

`__regex_show_item_catcode:NnT` Produce a sequence of categories which the catcode bitmap #2 contains, and show it, indenting the tests on which this catcode constraint applies.

```

6216 \cs_new_protected:Npn \__regex_show_item_catcode:NnT #1#2
6217 {
6218     \seq_set_split:Nnn \l__regex_internal_seq { } { CBEMTPUDSLOA }
6219     \seq_set_filter:NNn \l__regex_internal_seq \l__regex_internal_seq
6220     { \int_if_odd_p:n { #2 / \int_use:c { c__regex_catcode_##1_int } } }
6221     \__regex_show_scope:nn
6222     {
6223         categories~
6224         \seq_map_function:NN \l__regex_internal_seq \use:n
6225         , ~
6226         \bool_if:NF #1 { negative~ } class
6227     }
6228 }

```

(End definition for `__regex_show_item_catcode:NnT`.)

`__regex_show_item_exact_cs:n`

```

6229 \cs_new_protected:Npn \__regex_show_item_exact_cs:n #1
6230 {
6231     \seq_set_split:Nnn \l__regex_internal_seq { \scan_stop: } {#1}
6232     \seq_set_map_x:NNn \l__regex_internal_seq
6233     \l__regex_internal_seq { \iow_char:N\##1 }
6234     \__regex_show_one:n
6235     { control~sequence~ \seq_use:Nn \l__regex_internal_seq { ~or~ } }
6236 }

```

(End definition for `__regex_show_item_exact_cs:n`.)

45.4 Building

45.4.1 Variables used while building

`\l__regex_min_state_int` The last state that was allocated is `\l__regex_max_state_int - 1`, so that `\l__regex_max_state_int` always points to a free state. The `min_state` variable is 1 to begin with, but gets shifted in nested calls to the matching code, namely in `\c{...}` constructions.

```

6237 \int_new:N \l__regex_min_state_int
6238 \int_set:Nn \l__regex_min_state_int { 1 }
6239 \int_new:N \l__regex_max_state_int

```

(End definition for `\l__regex_min_state_int` and `\l__regex_max_state_int`.)

`\l__regex_left_state_int` `\l__regex_right_state_int` `\l__regex_left_state_seq` `\l__regex_right_state_seq` Alternatives are implemented by branching from a `left` state into the various choices, then merging those into a `right` state. We store information about those states in two sequences. Those states are also used to implement group quantifiers. Most often, the left and right pointers only differ by 1.

```

6240 \int_new:N \l__regex_left_state_int
6241 \int_new:N \l__regex_right_state_int
6242 \seq_new:N \l__regex_left_state_seq
6243 \seq_new:N \l__regex_right_state_seq

```

(End definition for `\l__regex_left_state_int` and others.)

`\l__regex_capturing_group_int` `\l__regex_capturing_group_int` is the next ID number to be assigned to a capturing group. This starts at 0 for the group enclosing the full regular expression, and groups are counted in the order of their left parenthesis, except when encountering `resetting` groups.

```

6244 \int_new:N \l__regex_capturing_group_int

```

(End definition for `\l__regex_capturing_group_int`.)

45.4.2 Framework

This phase is about going from a compiled regex to an NFA. Each state of the NFA is stored in a `\toks`. The operations which can appear in the `\toks` are

- `__regex_action_start_wildcard:N` $\langle\textit{boolean}\rangle$ inserted at the start of the regular expression, where a `true` $\langle\textit{boolean}\rangle$ makes it unanchored.
- `__regex_action_success:` marks the exit state of the NFA.
- `__regex_action_cost:n` $\{\langle\textit{shift}\rangle\}$ is a transition from the current $\langle\textit{state}\rangle$ to $\langle\textit{state}\rangle + \langle\textit{shift}\rangle$, which consumes the current character: the target state is saved and will be considered again when matching at the next position.
- `__regex_action_free:n` $\{\langle\textit{shift}\rangle\}$, and `__regex_action_free_group:n` $\{\langle\textit{shift}\rangle\}$ are free transitions, which immediately perform the actions for the state $\langle\textit{state}\rangle + \langle\textit{shift}\rangle$ of the NFA. They differ in how they detect and avoid infinite loops. For now, we just need to know that the `group` variant must be used for transitions back to the start of a group.
- `__regex_action_submatch:nN` $\{\langle\textit{group}\rangle\}$ $\langle\textit{key}\rangle$ where the $\langle\textit{key}\rangle$ is `<` or `>` for the beginning or end of group numbered $\langle\textit{group}\rangle$. This causes the current position in the query to be stored as the $\langle\textit{key}\rangle$ submatch boundary.
- One of these actions, within a conditional.

We strive to preserve the following properties while building.

- The current capturing group is `capturing_group - 1`, and if a group opened now it would be labelled `capturing_group`.

- The last allocated state is `max_state - 1`, so `max_state` is a free state.
- The `left_state` points to a state to the left of the current group or of the last class.
- The `right_state` points to a newly created, empty state, with some transitions leading to it.
- The `left/right` sequences hold a list of the corresponding end-points of nested groups.

`__regex_build:n` The `n`-type function first compiles its argument. Reset some variables. Allocate two states, and put a wildcard in state 0 (transitions to state 1 and 0 state). Then build the regex within a (capturing) group numbered 0 (current value of `capturing_group`). Finally, if the match reaches the last state, it is successful. A `false` boolean for argument `#1` for the auxiliaries will suppress the wildcard and make the match anchored: used for `\peek_regex:nTF` and similar.

```

6245 \cs_new_protected:Npn \__regex_build:n
6246   { \__regex_build_aux:Nn \c_true_bool }
6247 \cs_new_protected:Npn \__regex_build:N
6248   { \__regex_build_aux:NN \c_true_bool }
6249 \cs_new_protected:Npn \__regex_build_aux:Nn #1#2
6250   {
6251     \__regex_compile:n {#2}
6252     \__regex_build_aux:NN #1 \l__regex_internal_regex
6253   }
6254 \cs_new_protected:Npn \__regex_build_aux:NN #1#2
6255   {
6256     \__regex_standard_escapechar:
6257     \int_zero:N \l__regex_capturing_group_int
6258     \int_set_eq:NN \l__regex_max_state_int \l__regex_min_state_int
6259     \__regex_build_new_state:
6260     \__regex_build_new_state:
6261     \__regex_toks_put_right:Nn \l__regex_left_state_int
6262     { \__regex_action_start_wildcard:N #1 }
6263     \__regex_group:nnnN {#2} { 1 } { 0 } \c_false_bool
6264     \__regex_toks_put_right:Nn \l__regex_right_state_int
6265     { \__regex_action_success: }
6266   }

```

(End definition for `__regex_build:n` and others.)

`\g__regex_case_int` Case number that was successfully matched in `\regex_match_case:nn` and related functions.

```

6267 \int_new:N \g__regex_case_int

```

(End definition for `\g__regex_case_int`.)

`\l__regex_case_max_group_int` The largest group number appearing in any of the *<regex>* in the argument of `\regex_match_case:nn` and related functions.

```

6268 \int_new:N \l__regex_case_max_group_int

```

(End definition for `\l__regex_case_max_group_int`.)

```

    \__regex_case_build:n See \__regex_build:n, but with a loop.
    \__regex_case_build:x
    \__regex_case_build_aux:Nn
    \__regex_case_build_loop:n

6269 \cs_new_protected:Npn \__regex_case_build:n #1
6270 {
6271     \__regex_case_build_aux:Nn \c_true_bool {#1}
6272     \int_gzero:N \g__regex_case_int
6273 }
6274 \cs_generate_variant:Nn \__regex_case_build:n { x }
6275 \cs_new_protected:Npn \__regex_case_build_aux:Nn #1#2
6276 {
6277     \__regex_standard_escapechar:
6278     \int_set_eq:NN \l__regex_max_state_int \l__regex_min_state_int
6279     \__regex_build_new_state:
6280     \__regex_build_new_state:
6281     \__regex_toks_put_right:Nn \l__regex_left_state_int
6282     { \__regex_action_start_wildcard:N #1 }
6283     %
6284     \__regex_build_new_state:
6285     \__regex_toks_put_left:Nx \l__regex_left_state_int
6286     { \__regex_action_submatch:nN { 0 } < }
6287     \__regex_push_lr_states:
6288     \int_zero:N \l__regex_case_max_group_int
6289     \int_gzero:N \g__regex_case_int
6290     \tl_map_inline:nn {#2}
6291     {
6292         \int_gincr:N \g__regex_case_int
6293         \__regex_case_build_loop:n {##1}
6294     }
6295     \int_set_eq:NN \l__regex_capturing_group_int \l__regex_case_max_group_int
6296     \__regex_pop_lr_states:
6297 }
6298 \cs_new_protected:Npn \__regex_case_build_loop:n #1
6299 {
6300     \int_set:Nn \l__regex_capturing_group_int { 1 }
6301     \__regex_compile_use:n {#1}
6302     \int_set:Nn \l__regex_case_max_group_int
6303     {
6304         \int_max:nn { \l__regex_case_max_group_int }
6305         { \l__regex_capturing_group_int }
6306     }
6307     \seq_pop:Nn \l__regex_right_state_seq \l__regex_internal_a_tl
6308     \int_set:Nn \l__regex_right_state_int \l__regex_internal_a_tl
6309     \__regex_toks_put_left:Nx \l__regex_right_state_int
6310     {
6311         \__regex_action_submatch:nN { 0 } >
6312         \int_gset:Nn \g__regex_case_int
6313         { \int_use:N \g__regex_case_int }
6314         \__regex_action_success:
6315     }
6316     \__regex_toks_clear:N \l__regex_max_state_int
6317     \seq_push:Nn \l__regex_right_state_seq
6318     { \int_use:N \l__regex_max_state_int }
6319     \int_incr:N \l__regex_max_state_int
6320 }

```

(End definition for `__regex_case_build:n`, `__regex_case_build_aux:Nn`, and `__regex_case_build_loop:n`.)

`__regex_build_for_cs:n` The matching code relies on some global intarray variables, but only uses a range of their entries. Specifically,

- `\g__regex_state_active_intarray` from `\l__regex_min_state_int` to `\l__regex_max_state_int`;

Here, in this nested call to the matching code, we need the new versions of this range to involve completely new entries of the intarray variables, so we begin by setting (the new) `\l__regex_min_state_int` to (the old) `\l__regex_max_state_int` to use higher entries.

When using a regex to match a cs, we don't insert a wildcard, we anchor at the end, and since we ignore submatches, there is no need to surround the expression with a group. However, for branches to work properly at the outer level, we need to put the appropriate `left` and `right` states in their sequence.

```

6321 \cs_new_protected:Npn \__regex_build_for_cs:n #1
6322 {
6323   \int_set_eq:NN \l__regex_min_state_int \l__regex_max_state_int
6324   \__regex_build_new_state:
6325   \__regex_build_new_state:
6326   \__regex_push_lr_states:
6327   #1
6328   \__regex_pop_lr_states:
6329   \__regex_toks_put_right:Nn \l__regex_right_state_int
6330   {
6331     \if_int_compare:w -2 = \l__regex_curr_char_int
6332       \exp_after:wN \__regex_action_success:
6333     \fi:
6334   }
6335 }
```

(End definition for `__regex_build_for_cs:n`.)

45.4.3 Helpers for building an nfa

`__regex_push_lr_states:` When building the regular expression, we keep track of pointers to the left-end and right-end of each group without help from TeX's grouping.

```

6336 \cs_new_protected:Npn \__regex_push_lr_states:
6337 {
6338   \seq_push:No \l__regex_left_state_seq
6339   { \int_use:N \l__regex_left_state_int }
6340   \seq_push:No \l__regex_right_state_seq
6341   { \int_use:N \l__regex_right_state_int }
6342 }
6343 \cs_new_protected:Npn \__regex_pop_lr_states:
6344 {
6345   \seq_pop:NN \l__regex_left_state_seq \l__regex_internal_a_tl
6346   \int_set:Nn \l__regex_left_state_int \l__regex_internal_a_tl
6347   \seq_pop:NN \l__regex_right_state_seq \l__regex_internal_a_tl
6348   \int_set:Nn \l__regex_right_state_int \l__regex_internal_a_tl
6349 }
```

(End definition for `_regex_push_lr_states:` and `_regex_pop_lr_states:.`)

`_regex_build_transition_left:NNN`
`_regex_build_transition_right:nNn`

Add a transition from #2 to #3 using the function #1. The `left` function is used for higher priority transitions, and the `right` function for lower priority transitions (which should be performed later). The signatures differ to reflect the differing usage later on. Both functions could be optimized.

```
6350 \cs_new_protected:Npn \_regex_build_transition_left:NNN #1#2#3
6351   { \_regex_toks_put_left:Nx #2 { #1 { \int_eval:n { #3 - #2 } } } }
6352 \cs_new_protected:Npn \_regex_build_transition_right:nNn #1#2#3
6353   { \_regex_toks_put_right:Nx #2 { #1 { \int_eval:n { #3 - #2 } } } }
```

(End definition for `_regex_build_transition_left:NNN` and `_regex_build_transition_right:nNn`.)

`_regex_build_new_state:`

Add a new empty state to the NFA. Then update the `left`, `right`, and `max` states, so that the `right` state is the new empty state, and the `left` state points to the previously “current” state.

```
6354 \cs_new_protected:Npn \_regex_build_new_state:
6355   {
6356     \_regex_toks_clear:N \l__regex_max_state_int
6357     \int_set_eq:NN \l__regex_left_state_int \l__regex_right_state_int
6358     \int_set_eq:NN \l__regex_right_state_int \l__regex_max_state_int
6359     \int_incr:N \l__regex_max_state_int
6360   }
```

(End definition for `_regex_build_new_state:.`)

`_regex_build_transitions_lazyness:NNNNN`

This function creates a new state, and puts two transitions starting from the old current state. The order of the transitions is controlled by #1, true for lazy quantifiers, and false for greedy quantifiers.

```
6361 \cs_new_protected:Npn \_regex_build_transitions_lazyness:NNNNN #1#2#3#4#5
6362   {
6363     \_regex_build_new_state:
6364     \_regex_toks_put_right:Nx \l__regex_left_state_int
6365     {
6366       \if_meaning:w \c_true_bool #1
6367         #2 { \int_eval:n { #3 - \l__regex_left_state_int } }
6368         #4 { \int_eval:n { #5 - \l__regex_left_state_int } }
6369       \else:
6370         #4 { \int_eval:n { #5 - \l__regex_left_state_int } }
6371         #2 { \int_eval:n { #3 - \l__regex_left_state_int } }
6372       \fi:
6373     }
6374   }
```

(End definition for `_regex_build_transitions_lazyness:NNNNN`.)

45.4.4 Building classes

`_regex_class:NnnnN`
`_regex_tests_action_cost:n`

The arguments are: $\langle \text{boolean} \rangle$ $\{ \langle \text{tests} \rangle \}$ $\{ \langle \text{min} \rangle \}$ $\{ \langle \text{more} \rangle \}$ $\langle \text{lazyness} \rangle$. First store the tests with a trailing `_regex_action_cost:n`, in the true branch of `_regex_break_point:TF` for positive classes, or the false branch for negative classes. The integer $\langle \text{more} \rangle$ is 0 for fixed repetitions, -1 for unbounded repetitions, and $\langle \text{max} \rangle - \langle \text{min} \rangle$ for a range of repetitions.

```

6375 \cs_new_protected:Npn \__regex_class:NnnnN #1#2#3#4#5
6376 {
6377   \cs_set:Npx \__regex_tests_action_cost:n ##1
6378   {
6379     \exp_not:n { \exp_not:n {#2} }
6380     \bool_if:NTF #1
6381     { \__regex_break_point:TF { \__regex_action_cost:n {##1} } { } }
6382     { \__regex_break_point:TF { } { \__regex_action_cost:n {##1} } }
6383   }
6384   \if_case:w - #4 \exp_stop_f:
6385     \__regex_class_repeat:n {#3}
6386   \or: \__regex_class_repeat:nN {#3} #5
6387   \else: \__regex_class_repeat:nnN {#3} {#4} #5
6388   \fi:
6389 }
6390 \cs_new:Npn \__regex_tests_action_cost:n { \__regex_action_cost:n }

```

(End definition for __regex_class:NnnnN and __regex_tests_action_cost:n.)

__regex_class_repeat:n This is used for a fixed number of repetitions. Build one state for each repetition, with a transition controlled by the tests that we have collected. That works just fine for #1 = 0 repetitions: nothing is built.

```

6391 \cs_new_protected:Npn \__regex_class_repeat:n #1
6392 {
6393   \prg_replicate:nn {#1}
6394   {
6395     \__regex_build_new_state:
6396     \__regex_build_transition_right:nNn \__regex_tests_action_cost:n
6397     \l__regex_left_state_int \l__regex_right_state_int
6398   }
6399 }

```

(End definition for __regex_class_repeat:n.)

__regex_class_repeat:nN This implements unbounded repetitions of a single class (e.g. the * and + quantifiers). If the minimum number #1 of repetitions is 0, then build a transition from the current state to itself governed by the tests, and a free transition to a new state (hence skipping the tests). Otherwise, call __regex_class_repeat:n for the code to match #1 repetitions, and add free transitions from the last state to the previous one, and to a new one. In both cases, the order of transitions is controlled by the lazyness boolean #2.

```

6400 \cs_new_protected:Npn \__regex_class_repeat:nN #1#2
6401 {
6402   \if_int_compare:w #1 = \c_zero_int
6403     \__regex_build_transitions_lazyness:NNNNN #2
6404     \__regex_action_free:n \l__regex_right_state_int
6405     \__regex_tests_action_cost:n \l__regex_left_state_int
6406   \else:
6407     \__regex_class_repeat:n {#1}
6408     \int_set_eq:NN \l__regex_internal_a_int \l__regex_left_state_int
6409     \__regex_build_transitions_lazyness:NNNNN #2
6410     \__regex_action_free:n \l__regex_right_state_int
6411     \__regex_action_free:n \l__regex_internal_a_int
6412   \fi:
6413 }

```

(End definition for `_regex_class_repeat:nnN`.)

`_regex_class_repeat:nnN` We want to build the code to match from `#1` to `#1 + #2` repetitions. Match `#1` repetitions (can be 0). Compute the final state of the next construction as `a`. Build `#2 > 0` states, each with a transition to the next state governed by the tests, and a transition to the final state `a`. The computation of `a` is safe because states are allocated in order, starting from `max_state`.

```

6414 \cs_new_protected:Npn \_regex_class_repeat:nnN #1#2#3
6415   {
6416     \_regex_class_repeat:n {#1}
6417     \int_set:Nn \l__regex_internal_a_int
6418       { \l__regex_max_state_int + #2 - 1 }
6419     \prg_replicate:nn { #2 }
6420     {
6421       \_regex_build_transitions_lazyness:NNNN #3
6422       \_regex_action_free:n          \l__regex_internal_a_int
6423       \_regex_tests_action_cost:n \l__regex_right_state_int
6424     }
6425   }

```

(End definition for `_regex_class_repeat:nnN`.)

45.4.5 Building groups

`_regex_group_aux:nnnnN` Arguments: $\{\langle label \rangle\} \{\langle contents \rangle\} \{\langle min \rangle\} \{\langle more \rangle\} \langle lazyness \rangle$. If $\langle min \rangle$ is 0, we need to add a state before building the group, so that the thread which skips the group does not also set the start-point of the submatch. After adding one more state, the `left_state` is the left end of the group, from which all branches stem, and the `right_state` is the right end of the group, and all branches end their course in that state. We store those two integers to be queried for each branch, we build the NFA states for the contents `#2` of the group, and we forget about the two integers. Once this is done, perform the repetition: either exactly `#3` times, or `#3` or more times, or between `#3` and `#3 + #4` times, with lazyness `#5`. The $\langle label \rangle$ `#1` is used for submatch tracking. Each of the three auxiliaries expects `left_state` and `right_state` to be set properly.

```

6426 \cs_new_protected:Npn \_regex_group_aux:nnnnN #1#2#3#4#5
6427   {
6428     \if_int_compare:w #3 = \c_zero_int
6429       \_regex_build_new_state:
6430       \_regex_build_transition_right:nNn \_regex_action_free_group:n
6431       \l__regex_left_state_int \l__regex_right_state_int
6432     \fi:
6433     \_regex_build_new_state:
6434     \_regex_push_lr_states:
6435     #2
6436     \_regex_pop_lr_states:
6437     \if_case:w - #4 \exp_stop_f:
6438       \_regex_group_repeat:nn {#1} {#3}
6439     \or:  \_regex_group_repeat:nnN {#1} {#3} #5
6440     \else: \_regex_group_repeat:nnnN {#1} {#3} {#4} #5
6441     \fi:
6442   }

```

(End definition for `_regex_group_aux:nnnnN`.)

`__regex_group:nnnN` Hand to `__regex_group_aux:nnnnN` the label of that group (expanded), and the group itself, with some extra commands to perform.

```

6443 \cs_new_protected:Npn \__regex_group:nnnN #1
6444 {
6445     \exp_args:No \__regex_group_aux:nnnnN
6446     { \int_use:N \l__regex_capturing_group_int }
6447     {
6448         \int_incr:N \l__regex_capturing_group_int
6449         #1
6450     }
6451 }
6452 \cs_new_protected:Npn \__regex_group_no_capture:nnnN
6453 { \__regex_group_aux:nnnnN { -1 } }

```

(End definition for `__regex_group:nnnN` and `__regex_group_no_capture:nnnN`.)

`__regex_group_resetting:nnnN` Again, hand the label `-1` to `__regex_group_aux:nnnnN`, but this time we work a little bit harder to keep track of the maximum group label at the end of any branch, and to reset the group number at each branch. This relies on the fact that a compiled regex always is a sequence of items of the form `__regex_branch:n {<branch>}`.

```

6454 \cs_new_protected:Npn \__regex_group_resetting:nnnN #1
6455 {
6456     \__regex_group_aux:nnnnN { -1 }
6457     {
6458         \exp_args:Noo \__regex_group_resetting_loop:nnNn
6459         { \int_use:N \l__regex_capturing_group_int }
6460         { \int_use:N \l__regex_capturing_group_int }
6461         #1
6462         { ?? \prg_break:n } { }
6463         \prg_break_point:
6464     }
6465 }
6466 \cs_new_protected:Npn \__regex_group_resetting_loop:nnNn #1#2#3#4
6467 {
6468     \use_none:nn #3 { \int_set:Nn \l__regex_capturing_group_int {#1} }
6469     \int_set:Nn \l__regex_capturing_group_int {#2}
6470     #3 {#4}
6471     \exp_args:Nf \__regex_group_resetting_loop:nnNn
6472     { \int_max:nn {#1} { \l__regex_capturing_group_int } }
6473     {#2}
6474 }

```

(End definition for `__regex_group_resetting:nnnN` and `__regex_group_resetting_loop:nnNn`.)

`__regex_branch:n` Add a free transition from the left state of the current group to a brand new state, starting point of this branch. Once the branch is built, add a transition from its last state to the right state of the group. The left and right states of the group are extracted from the relevant sequences.

```

6475 \cs_new_protected:Npn \__regex_branch:n #1
6476 {
6477     \__regex_build_new_state:
6478     \seq_get:NN \l__regex_left_state_seq \l__regex_internal_a_tl
6479     \int_set:Nn \l__regex_left_state_int \l__regex_internal_a_tl
6480     \__regex_build_transition_right:nNn \__regex_action_free:n

```

```

6481     \l__regex_left_state_int \l__regex_right_state_int
6482     #1
6483     \seq_get:NN \l__regex_right_state_seq \l__regex_internal_a_tl
6484     \__regex_build_transition_right:nNn \__regex_action_free:n
6485     \l__regex_right_state_int \l__regex_internal_a_tl
6486 }

```

(End definition for __regex_branch:n.)

__regex_group_repeat:nn This function is called to repeat a group a fixed number of times #2; if this is 0 we remove the group altogether (but don't reset the capturing_group label). Otherwise, the auxiliary __regex_group_repeat_aux:n copies #2 times the \toks for the group, and leaves internal_a pointing to the left end of the last repetition. We only record the submatch information at the last repetition. Finally, add a state at the end (the transition to it has been taken care of by the replicating auxiliary).

```

6487 \cs_new_protected:Npn \__regex_group_repeat:nn #1#2
6488 {
6489     \if_int_compare:w #2 = \c_zero_int
6490         \int_set:Nn \l__regex_max_state_int
6491         { \l__regex_left_state_int - 1 }
6492         \__regex_build_new_state:
6493     \else:
6494         \__regex_group_repeat_aux:n {#2}
6495         \__regex_group_submatches:nNN {#1}
6496         \l__regex_internal_a_int \l__regex_right_state_int
6497         \__regex_build_new_state:
6498     \fi:
6499 }

```

(End definition for __regex_group_repeat:nn.)

__regex_group_submatches:nNN This inserts in states #2 and #3 the code for tracking submatches of the group #1, unless inhibited by a label of -1.

```

6500 \cs_new_protected:Npn \__regex_group_submatches:nNN #1#2#3
6501 {
6502     \if_int_compare:w #1 > - \c_one_int
6503         \__regex_toks_put_left:Nx #2 { \__regex_action_submatch:nN {#1} < }
6504         \__regex_toks_put_left:Nx #3 { \__regex_action_submatch:nN {#1} > }
6505     \fi:
6506 }

```

(End definition for __regex_group_submatches:nNN.)

__regex_group_repeat_aux:n Here we repeat \toks ranging from left_state to max_state, #1 > 0 times. First add a transition so that the copies “chain” properly. Compute the shift c between the original copy and the last copy we want. Shift the right_state and max_state to their final values. We then want to perform c copy operations. At the end, b is equal to the max_state, and a points to the left of the last copy of the group.

```

6507 \cs_new_protected:Npn \__regex_group_repeat_aux:n #1
6508 {
6509     \__regex_build_transition_right:nNn \__regex_action_free:n
6510     \l__regex_right_state_int \l__regex_max_state_int
6511     \int_set_eq:NN \l__regex_internal_a_int \l__regex_left_state_int
6512     \int_set_eq:NN \l__regex_internal_b_int \l__regex_max_state_int

```

```

6513 \if_int_compare:w \int_eval:n {#1} > \c_one_int
6514 \int_set:Nn \l__regex_internal_c_int
6515 {
6516   ( #1 - 1 )
6517   * ( \l__regex_internal_b_int - \l__regex_internal_a_int )
6518 }
6519 \int_add:Nn \l__regex_right_state_int { \l__regex_internal_c_int }
6520 \int_add:Nn \l__regex_max_state_int { \l__regex_internal_c_int }
6521 \__regex_toks_memcpy:NNn
6522   \l__regex_internal_b_int
6523   \l__regex_internal_a_int
6524   \l__regex_internal_c_int
6525 \fi:
6526 }

```

(End definition for `__regex_group_repeat_aux:n`.)

`__regex_group_repeat:nnN` This function is called to repeat a group at least n times; the case $n = 0$ is very different from $n > 0$. Assume first that $n = 0$. Insert submatch tracking information at the start and end of the group, add a free transition from the right end to the “true” left state `a` (remember: in this case we had added an extra state before the left state). This forms the loop, which we break away from by adding a free transition from `a` to a new state.

Now consider the case $n > 0$. Repeat the group n times, chaining various copies with a free transition. Add submatch tracking only to the last copy, then add a free transition from the right end back to the left end of the last copy, either before or after the transition to move on towards the rest of the NFA. This transition can end up before submatch tracking, but that is irrelevant since it only does so when going again through the group, recording new matches. Finally, add a state; we already have a transition pointing to it from `__regex_group_repeat_aux:n`.

```

6527 \cs_new_protected:Npn \__regex_group_repeat:nnN #1#2#3
6528 {
6529   \if_int_compare:w #2 = \c_zero_int
6530     \__regex_group_submatches:nnN {#1}
6531     \l__regex_left_state_int \l__regex_right_state_int
6532     \int_set:Nn \l__regex_internal_a_int
6533     { \l__regex_left_state_int - 1 }
6534     \__regex_build_transition_right:nNn \__regex_action_free:n
6535     \l__regex_right_state_int \l__regex_internal_a_int
6536     \__regex_build_new_state:
6537     \if_meaning:w \c_true_bool #3
6538       \__regex_build_transition_left:NNN \__regex_action_free:n
6539       \l__regex_internal_a_int \l__regex_right_state_int
6540     \else:
6541       \__regex_build_transition_right:nNn \__regex_action_free:n
6542       \l__regex_internal_a_int \l__regex_right_state_int
6543     \fi:
6544   \else:
6545     \__regex_group_repeat_aux:n {#2}
6546     \__regex_group_submatches:nnN {#1}
6547     \l__regex_internal_a_int \l__regex_right_state_int
6548     \if_meaning:w \c_true_bool #3
6549       \__regex_build_transition_right:nNn \__regex_action_free_group:n
6550       \l__regex_right_state_int \l__regex_internal_a_int

```

```

6551     \else:
6552         \__regex_build_transition_left:NNN \__regex_action_free_group:n
6553         \l__regex_right_state_int \l__regex_internal_a_int
6554     \fi:
6555     \__regex_build_new_state:
6556 \fi:
6557 }

```

(End definition for __regex_group_repeat:nnN.)

__regex_group_repeat:nnnN

We wish to repeat the group between #2 and #2 + #3 times, with a laziness controlled by #4. We insert submatch tracking up front: in principle, we could avoid recording submatches for the first #2 copies of the group, but that forces us to treat specially the case #2 = 0. Repeat that group with submatch tracking #2 + #3 times (the maximum number of repetitions). Then our goal is to add #3 transitions from the end of the #2-th group, and each subsequent groups, to the end. For a lazy quantifier, we add those transitions to the left states, before submatch tracking. For the greedy case, we add the transitions to the right states, after submatch tracking and the transitions which go on with more repetitions. In the greedy case with #2 = 0, the transition which skips over all copies of the group must be added separately, because its starting state does not follow the normal pattern: we had to add it “by hand” earlier.

```

6558 \cs_new_protected:Npn \__regex_group_repeat:nnnN #1#2#3#4
6559 {
6560     \__regex_group_submatches:nnN {#1}
6561     \l__regex_left_state_int \l__regex_right_state_int
6562     \__regex_group_repeat_aux:n { #2 + #3 }
6563     \if_meaning:w \c_true_bool #4
6564     \int_set_eq:NN \l__regex_left_state_int \l__regex_max_state_int
6565     \prg_replicate:nn { #3 }
6566     {
6567         \int_sub:Nn \l__regex_left_state_int
6568         { \l__regex_internal_b_int - \l__regex_internal_a_int }
6569         \__regex_build_transition_left:NNN \__regex_action_free:n
6570         \l__regex_left_state_int \l__regex_max_state_int
6571     }
6572     \else:
6573     \prg_replicate:nn { #3 - 1 }
6574     {
6575         \int_sub:Nn \l__regex_right_state_int
6576         { \l__regex_internal_b_int - \l__regex_internal_a_int }
6577         \__regex_build_transition_right:nNn \__regex_action_free:n
6578         \l__regex_right_state_int \l__regex_max_state_int
6579     }
6580     \if_int_compare:w #2 = \c_zero_int
6581     \int_set:Nn \l__regex_right_state_int
6582     { \l__regex_left_state_int - 1 }
6583     \else:
6584     \int_sub:Nn \l__regex_right_state_int
6585     { \l__regex_internal_b_int - \l__regex_internal_a_int }
6586     \fi:
6587     \__regex_build_transition_right:nNn \__regex_action_free:n
6588     \l__regex_right_state_int \l__regex_max_state_int
6589 \fi:

```

```

6590     \__regex_build_new_state:
6591 }

```

(End definition for __regex_group_repeat:nnnN.)

45.4.6 Others

__regex_assertion:Nn Usage: __regex_assertion:Nn *<boolean>* {*<test>*}, where the *<test>* is either of the two other functions. Add a free transition to a new state, conditionally to the assertion test. The __regex_b_test: test is used by the \b and \B escape: check if the last character was a word character or not, and do the same to the current character. The __regex_G_test: boundary-markers of the string are non-word characters for this purpose.

```

6592 \cs_new_protected:Npn \__regex_assertion:Nn #1#2
6593 {
6594     \__regex_build_new_state:
6595     \__regex_toks_put_right:Nx \l__regex_left_state_int
6596     {
6597         \exp_not:n {#2}
6598         \__regex_break_point:TF
6599         \bool_if:NF #1 { { } }
6600         {
6601             \__regex_action_free:n
6602             {
6603                 \int_eval:n
6604                 { \l__regex_right_state_int - \l__regex_left_state_int }
6605             }
6606         }
6607         \bool_if:NT #1 { { } }
6608     }
6609 }
6610 \cs_new_protected:Npn \__regex_b_test:
6611 {
6612     \group_begin:
6613     \int_set_eq:NN \l__regex_curr_char_int \l__regex_last_char_int
6614     \__regex_prop_w:
6615     \__regex_break_point:TF
6616     { \group_end: \__regex_item_reverse:n \__regex_prop_w: }
6617     { \group_end: \__regex_prop_w: }
6618 }
6619 \cs_new_protected:Npn \__regex_Z_test:
6620 {
6621     \if_int_compare:w -2 = \l__regex_curr_char_int
6622     \exp_after:wN \__regex_break_true:w
6623     \fi:
6624 }
6625 \cs_new_protected:Npn \__regex_A_test:
6626 {
6627     \if_int_compare:w -2 = \l__regex_last_char_int
6628     \exp_after:wN \__regex_break_true:w
6629     \fi:
6630 }
6631 \cs_new_protected:Npn \__regex_G_test:
6632 {
6633     \if_int_compare:w \l__regex_curr_pos_int = \l__regex_start_pos_int

```

```

6634     \exp_after:wN \__regex_break_true:w
6635     \fi:
6636 }

```

(End definition for __regex_assertion:Nn and others.)

__regex_command_K: Change the starting point of the 0-th submatch (full match), and transition to a new state, pretending that this is a fresh thread.

```

6637 \cs_new_protected:Npn \__regex_command_K:
6638 {
6639     \__regex_build_new_state:
6640     \__regex_toks_put_right:Nx \l__regex_left_state_int
6641     {
6642         \__regex_action_submatch:nN { 0 } <
6643         \bool_set_true:N \l__regex_fresh_thread_bool
6644         \__regex_action_free:n
6645         {
6646             \int_eval:n
6647             { \l__regex_right_state_int - \l__regex_left_state_int }
6648         }
6649         \bool_set_false:N \l__regex_fresh_thread_bool
6650     }
6651 }

```

(End definition for __regex_command_K:.)

45.5 Matching

We search for matches by running all the execution threads through the NFA in parallel, reading one token of the query at each step. The NFA contains “free” transitions to other states, and transitions which “consume” the current token. For free transitions, the instruction at the new state of the NFA is performed immediately. When a transition consumes a character, the new state is appended to a list of “active states”, stored in \g__regex_thread_info_intarray (together with submatch information): this thread is made active again when the next token is read from the query. At every step (for each token in the query), we unpack that list of active states and the corresponding submatch props, and empty those.

If two paths through the NFA “collide” in the sense that they reach the same state after reading a given token, then they only differ in how they previously matched, and any future execution would be identical for both. (Note that this would be wrong in the presence of back-references.) Hence, we only need to keep one of the two threads: the thread with the highest priority. Our NFA is built in such a way that higher priority actions always come before lower priority actions, which makes things work.

The explanation in the previous paragraph may make us think that we simply need to keep track of which states were visited at a given step: after all, the loop generated when matching (a?)* against a is broken, isn’t it? No. The group first matches a, as it should, then repeats; it attempts to match a again but fails; it skips a, and finds out that this state has already been seen at this position in the query: the match stops. The capturing group is (wrongly) a. What went wrong is that a thread collided with itself, and the later version, which has gone through the group one more times with an empty match, should have a higher priority than not going through the group.

We solve this by distinguishing “normal” free transitions `__regex_action_free:n` from transitions `__regex_action_free_group:n` which go back to the start of the group. The former keeps threads unless they have been visited by a “completed” thread, while the latter kind of transition also prevents going back to a state visited by the current thread.

45.5.1 Variables used when matching

`\l__regex_min_pos_int`
`\l__regex_max_pos_int`
`\l__regex_curr_pos_int`
`\l__regex_start_pos_int`
`\l__regex_success_pos_int`

The tokens in the query are indexed from `min_pos` for the first to `max_pos - 1` for the last, and their information is stored in several arrays and `\toks` registers with those numbers. We match without backtracking, keeping all threads in lockstep at the `curr_pos` in the query. The starting point of the current match attempt is `start_pos`, and `success_pos`, updated whenever a thread succeeds, is used as the next starting position.

```
6652 \int_new:N \l__regex_min_pos_int
6653 \int_new:N \l__regex_max_pos_int
6654 \int_new:N \l__regex_curr_pos_int
6655 \int_new:N \l__regex_start_pos_int
6656 \int_new:N \l__regex_success_pos_int
```

(End definition for `\l__regex_min_pos_int` and others.)

`\l__regex_curr_char_int`
`\l__regex_curr_catcode_int`
`\l__regex_curr_token_tl`
`\l__regex_last_char_int`
`\l__regex_last_char_success_int`
`\l__regex_case_changed_char_int`

The character and category codes of the token at the current position and a token list expanding to that token; the character code of the token at the previous position; the character code of the token just before a successful match; and the character code of the result of changing the case of the current token (`A-Z`↔`a-z`). This last integer is only computed when necessary, and is otherwise `\c_max_int`. The `curr_char` variable is also used in various other phases to hold a character code.

```
6657 \int_new:N \l__regex_curr_char_int
6658 \int_new:N \l__regex_curr_catcode_int
6659 \tl_new:N \l__regex_curr_token_tl
6660 \int_new:N \l__regex_last_char_int
6661 \int_new:N \l__regex_last_char_success_int
6662 \int_new:N \l__regex_case_changed_char_int
```

(End definition for `\l__regex_curr_char_int` and others.)

`\l__regex_curr_state_int`

For every character in the token list, each of the active states is considered in turn. The variable `\l__regex_curr_state_int` holds the state of the NFA which is currently considered: transitions are then given as shifts relative to the current state.

```
6663 \int_new:N \l__regex_curr_state_int
```

(End definition for `\l__regex_curr_state_int`.)

`\l__regex_curr_submatches_tl`
`\l__regex_success_submatches_tl`

The submatches for the thread which is currently active are stored in the `curr_submatches` list, which is almost a comma list, but ends with a comma. This list is stored by `__regex_store_state:n` into an intarray variable, to be retrieved when matching at the next position. When a thread succeeds, this list is copied to `\l__regex_success_submatches_tl`: only the last successful thread remains there.

```
6664 \tl_new:N \l__regex_curr_submatches_tl
6665 \tl_new:N \l__regex_success_submatches_tl
```

(End definition for `\l__regex_curr_submatches_tl` and `\l__regex_success_submatches_tl`.)

`\l__regex_step_int` This integer, always even, is increased every time a character in the query is read, and not reset when doing multiple matches. We store in `\g__regex_state_active_intarray` the last step in which each $\langle state \rangle$ in the NFA was encountered. This lets us break infinite loops by not visiting the same state twice in the same step. In fact, the step we store is equal to `step` when we have started performing the operations of `\toks $\langle state \rangle$` , but not finished yet. However, once we finish, we store `step + 1` in `\g__regex_state_active_intarray`. This is needed to track submatches properly (see building phase). The `step` is also used to attach each set of submatch information to a given iteration (and automatically discard it when it corresponds to a past step).

```
6666 \int_new:N \l__regex_step_int
```

(End definition for `\l__regex_step_int`.)

`\l__regex_min_thread_int` `\l__regex_max_thread_int` All the currently active threads are kept in order of precedence in `\g__regex_thread_info_intarray` together with the corresponding submatch information. Data in this intarray is organized as blocks from `min_thread` (included) to `max_thread` (excluded). At the start of every step, the whole array is unpacked, so that the space can immediately be reused, and `max_thread` is reset to `min_thread`, effectively clearing the array.

```
6667 \int_new:N \l__regex_min_thread_int
```

```
6668 \int_new:N \l__regex_max_thread_int
```

(End definition for `\l__regex_min_thread_int` and `\l__regex_max_thread_int`.)

`\g__regex_state_active_intarray` `\g__regex_thread_info_intarray` `\g__regex_state_active_intarray` stores the last $\langle step \rangle$ in which each $\langle state \rangle$ was active. `\g__regex_thread_info_intarray` stores threads to be considered in the next step, more precisely the states in which these threads are.

```
6669 \intarray_new:Nn \g__regex_state_active_intarray { 65536 }
```

```
6670 \intarray_new:Nn \g__regex_thread_info_intarray { 65536 }
```

(End definition for `\g__regex_state_active_intarray` and `\g__regex_thread_info_intarray`.)

`\l__regex_matched_analysis_tl` `\l__regex_curr_analysis_tl` The list `\l__regex_curr_analysis_tl` consists of a brace group containing three brace groups corresponding to the current token, with the same syntax as `\tl_analysis_map_inline:nn`. The list `\l__regex_matched_analysis_tl` (constructed under the `tl-build` machinery) has one item for each token that has already been treated so far in a given match attempt: each item consists of three brace groups with the same syntax as `\tl_analysis_map_inline:nn`.

```
6671 \tl_new:N \l__regex_matched_analysis_tl
```

```
6672 \tl_new:N \l__regex_curr_analysis_tl
```

(End definition for `\l__regex_matched_analysis_tl` and `\l__regex_curr_analysis_tl`.)

`\l__regex_every_match_tl` Every time a match is found, this token list is used. For single matching, the token list is empty. For multiple matching, the token list is set to repeat the matching, after performing some operation which depends on the user function. See `__regex_single_match:` and `__regex_multi_match:n`.

```
6673 \tl_new:N \l__regex_every_match_tl
```

(End definition for `\l__regex_every_match_tl`.)


```
\l__regex_fresh_thread_bool
\l__regex_empty_success_bool
  \__regex_if_two_empty_matches:F
```

When doing multiple matches, we need to avoid infinite loops where each iteration matches the same empty token list. When an empty token list is matched, the next successful match of the same empty token list is suppressed. We detect empty matches by setting `\l__regex_fresh_thread_bool` to `true` for threads which directly come from the start of the regex or from the `\K` command, and testing that boolean whenever a thread succeeds. The function `__regex_if_two_empty_matches:F` is redefined at every match attempt, depending on whether the previous match was empty or not: if it was, then the function must cancel a purported success if it is empty and at the same spot as the previous match; otherwise, we definitely don't have two identical empty matches, so the function is `\use:n`.

```
6674 \bool_new:N \l__regex_fresh_thread_bool
6675 \bool_new:N \l__regex_empty_success_bool
6676 \cs_new_eq:NN \__regex_if_two_empty_matches:F \use:n
```

(End definition for `\l__regex_fresh_thread_bool`, `\l__regex_empty_success_bool`, and `__regex_if_two_empty_matches:F`.)

```
\g__regex_success_bool
\l__regex_saved_success_bool
\l__regex_match_success_bool
```

The boolean `\l__regex_match_success_bool` is true if the current match attempt was successful, and `\g__regex_success_bool` is true if there was at least one successful match. This is the only global variable in this whole module, but we would need it to be local when matching a control sequence with `\c{...}`. This is done by saving the global variable into `\l__regex_saved_success_bool`, which is local, hence not affected by the changes due to inner regex functions.

```
6677 \bool_new:N \g__regex_success_bool
6678 \bool_new:N \l__regex_saved_success_bool
6679 \bool_new:N \l__regex_match_success_bool
```

(End definition for `\g__regex_success_bool`, `\l__regex_saved_success_bool`, and `\l__regex_match_success_bool`.)

45.5.2 Matching: framework

```
\__regex_match:n
\__regex_match_cs:n
\__regex_match_init:
```

Initialize the variables that should be set once for each user function (even for multiple matches). Namely, the overall matching is not yet successful; none of the states should be marked as visited (`\g__regex_state_active_intarray`), and we start at step 0; we pretend that there was a previous match ending at the start of the query, which was not empty (to avoid smothering an empty match at the start). Once all this is set up, we are ready for the ride. Find the first match.

```
6680 \cs_new_protected:Npn \__regex_match:n #1
6681 {
6682   \__regex_match_init:
6683   \__regex_match_once_init:
6684   \tl_analysis_map_inline:nn {#1}
6685     { \__regex_match_one_token:nnN {##1} {##2} ##3 }
6686   \__regex_match_one_token:nnN { } { -2 } F
6687   \prg_break_point:Nn \__regex_maplike_break: { }
6688 }
6689 \cs_new_protected:Npn \__regex_match_cs:n #1
6690 {
6691   \int_set_eq:NN \l__regex_min_thread_int \l__regex_max_thread_int
6692   \__regex_match_init:
6693   \__regex_match_once_init:
```

```

6694 \str_map_inline:nn {#1}
6695 {
6696   \tl_if_blank:nTF {##1}
6697   { \__regex_match_one_token:nnN {##1} {'##1} A }
6698   { \__regex_match_one_token:nnN {##1} {'##1} C }
6699 }
6700 \__regex_match_one_token:nnN { } { -2 } F
6701 \prg_break_point:Nn \__regex_maplike_break: { }
6702 }
6703 \cs_new_protected:Npn \__regex_match_init:
6704 {
6705   \bool_gset_false:N \g__regex_success_bool
6706   \int_step_inline:nnn
6707   \l__regex_min_state_int { \l__regex_max_state_int - 1 }
6708   {
6709     \__kernel_intarray_gset:Nnn
6710     \g__regex_state_active_intarray {##1} { 1 }
6711   }
6712   \int_zero:N \l__regex_step_int
6713   \int_set:Nn \l__regex_min_pos_int { 2 }
6714   \int_set_eq:NN \l__regex_success_pos_int \l__regex_min_pos_int
6715   \int_set:Nn \l__regex_last_char_success_int { -2 }
6716   \tl_build_begin:N \l__regex_matched_analysis_tl
6717   \tl_clear:N \l__regex_curr_analysis_tl
6718   \int_set:Nn \l__regex_min_submatch_int { 1 }
6719   \int_set_eq:NN \l__regex_submatch_int \l__regex_min_submatch_int
6720   \bool_set_false:N \l__regex_empty_success_bool
6721 }

```

(End definition for __regex_match:n, __regex_match_cs:n, and __regex_match_init:.)

__regex_match_once_init: This function resets various variables used when finding one match. It is called before the loop through characters, and every time we find a match, before searching for another match (this is controlled by the `every_match` token list).

First initialize some variables: set the conditional which detects identical empty matches; this match attempt starts at the previous `success_pos`, is not yet successful, and has no submatches yet; clear the array of active threads, and put the starting state 0 in it. We are then almost ready to read our first token in the query, but we actually start one position earlier than the start because `__regex_match_one_token:nnN` increments `\l__regex_curr_pos_int` and saves `\l__regex_curr_char_int` as the `last_char` so that word boundaries can be correctly identified.

```

6722 \cs_new_protected:Npn \__regex_match_once_init:
6723 {
6724   \if_meaning:w \c_true_bool \l__regex_empty_success_bool
6725   \cs_set:Npn \__regex_if_two_empty_matches:F
6726   {
6727     \int_compare:nNnF
6728     \l__regex_start_pos_int = \l__regex_curr_pos_int
6729   }
6730   \else:
6731     \cs_set_eq:NN \__regex_if_two_empty_matches:F \use:n
6732   \fi:
6733   \int_set_eq:NN \l__regex_start_pos_int \l__regex_success_pos_int
6734   \bool_set_false:N \l__regex_match_success_bool

```

```

6735 \tl_set:Nx \l__regex_curr_submatches_tl
6736 { \prg_replicate:nn { 2 * \l__regex_capturing_group_int } { 0 , } }
6737 \int_set_eq:NN \l__regex_max_thread_int \l__regex_min_thread_int
6738 \__regex_store_state:n { \l__regex_min_state_int }
6739 \int_set:Nn \l__regex_curr_pos_int
6740 { \l__regex_start_pos_int - 1 }
6741 \int_set_eq:NN \l__regex_curr_char_int \l__regex_last_char_success_int
6742 \tl_build_get:NN \l__regex_matched_analysis_tl \l__regex_internal_a_tl
6743 \exp_args:NNf \__regex_match_once_init_aux:
6744 \tl_map_inline:nn
6745 { \exp_after:wN \l__regex_internal_a_tl \l__regex_curr_analysis_tl }
6746 { \__regex_match_one_token:nnN ##1 }
6747 \prg_break_point:Nn \__regex_maplike_break: { }
6748 }
6749 \cs_new_protected:Npn \__regex_match_once_init_aux:
6750 {
6751 \tl_build_clear:N \l__regex_matched_analysis_tl
6752 \tl_clear:N \l__regex_curr_analysis_tl
6753 }

```

(End definition for __regex_match_once_init:.)

__regex_single_match: For a single match, the overall success is determined by whether the only match attempt is a success. When doing multiple matches, the overall matching is successful as soon as any match succeeds. Perform the action #1, then find the next match.

__regex_multi_match:n

```

6754 \cs_new_protected:Npn \__regex_single_match:
6755 {
6756 \tl_set:Nn \l__regex_every_match_tl
6757 {
6758 \bool_gset_eq:NN
6759 \g__regex_success_bool
6760 \l__regex_match_success_bool
6761 \__regex_maplike_break:
6762 }
6763 }
6764 \cs_new_protected:Npn \__regex_multi_match:n #1
6765 {
6766 \tl_set:Nn \l__regex_every_match_tl
6767 {
6768 \if_meaning:w \c_false_bool \l__regex_match_success_bool
6769 \exp_after:wN \__regex_maplike_break:
6770 \fi:
6771 \bool_gset_true:N \g__regex_success_bool
6772 #1
6773 \__regex_match_once_init:
6774 }
6775 }

```

(End definition for __regex_single_match: and __regex_multi_match:n.)

__regex_match_one_token:nnN At each new position, set some variables and get the new character and category from the query. Then unpack the array of active threads, and clear it by resetting its length (max_thread). This results in a sequence of __regex_use_state_and_submatches:w <state>,<submatch-clist>; and we consider those states one by one in order. As soon

as a thread succeeds, exit the step, and, if there are threads to consider at the next position, and we have not reached the end of the string, repeat the loop. Otherwise, the last thread that succeeded is the match. We explain the `fresh_thread` business when describing `__regex_action_wildcard`:

```

6776 \cs_new_protected:Npn \__regex_match_one_token:nnN #1#2#3
6777 {
6778   \int_add:Nn \l__regex_step_int { 2 }
6779   \int_incr:N \l__regex_curr_pos_int
6780   \int_set_eq:NN \l__regex_last_char_int \l__regex_curr_char_int
6781   \cs_set_eq:NN \__regex_maybe_compute_ccc: \__regex_compute_case_changed_char:
6782   \tl_set:Nn \l__regex_curr_token_tl {#1}
6783   \int_set:Nn \l__regex_curr_char_int {#2}
6784   \int_set:Nn \l__regex_curr_catcode_int { "#3 }
6785   \tl_build_put_right:Nx \l__regex_matched_analysis_tl
6786     { \exp_not:o \l__regex_curr_analysis_tl }
6787   \tl_set:Nn \l__regex_curr_analysis_tl { { {#1} {#2} #3 } }
6788   \use:x
6789   {
6790     \int_set_eq:NN \l__regex_max_thread_int \l__regex_min_thread_int
6791     \int_step_function:nnN
6792       { \l__regex_min_thread_int }
6793       { \l__regex_max_thread_int - 1 }
6794     \__regex_match_one_active:n
6795   }
6796   \prg_break_point:
6797   \bool_set_false:N \l__regex_fresh_thread_bool
6798   \if_int_compare:w \l__regex_max_thread_int > \l__regex_min_thread_int
6799     \if_int_compare:w -2 < \l__regex_curr_char_int
6800       \exp_after:wN \exp_after:wN \exp_after:wN \use_none:n
6801     \fi:
6802   \fi:
6803   \l__regex_every_match_tl
6804 }
6805 \cs_new:Npn \__regex_match_one_active:n #1
6806 {
6807   \__regex_use_state_and_submatches:w
6808   \__kernel_intarray_range_to_clist:Nnn
6809     \g__regex_thread_info_intarray
6810     { 1 + #1 * (\l__regex_capturing_group_int * 2 + 1) }
6811     { (1 + #1) * (\l__regex_capturing_group_int * 2 + 1) }
6812   ;
6813 }

```

(End definition for `__regex_match_one_token:nnN` and `__regex_match_one_active:n`.)

45.5.3 Using states of the nfa

`__regex_use_state`: Use the current NFA instruction. The state is initially marked as belonging to the current **step**: this allows normal free transition to repeat, but group-repeating transitions won't. Once we are done exploring all the branches it spawned, the state is marked as **step + 1**: any thread hitting it at that point will be terminated.

```

6814 \cs_new_protected:Npn \__regex_use_state:
6815 {

```

```

6816     \__kernel_intarray_gset:Nnn \g__regex_state_active_intarray
6817     { \l__regex_curr_state_int } { \l__regex_step_int }
6818     \__regex_toks_use:w \l__regex_curr_state_int
6819     \__kernel_intarray_gset:Nnn \g__regex_state_active_intarray
6820     { \l__regex_curr_state_int }
6821     { \int_eval:n { \l__regex_step_int + 1 } }
6822 }

```

(End definition for __regex_use_state:.)

__regex_use_state_and_submatches:w

This function is called as one item in the array of active threads after that array has been unpacked for a new step. Update the `curr_state` and `curr_submatches` and use the state if it has not yet been encountered at this step.

```

6823 \cs_new_protected:Npn \__regex_use_state_and_submatches:w #1 , #2 ;
6824 {
6825     \int_set:Nn \l__regex_curr_state_int {#1}
6826     \if_int_compare:w
6827         \__kernel_intarray_item:Nn \g__regex_state_active_intarray
6828         { \l__regex_curr_state_int }
6829         < \l__regex_step_int
6830     \tl_set:Nn \l__regex_curr_submatches_tl { #2 , }
6831     \exp_after:wN \__regex_use_state:
6832     \fi:
6833     \scan_stop:
6834 }

```

(End definition for __regex_use_state_and_submatches:w.)

45.5.4 Actions when matching

__regex_action_start_wildcard:N

For an unanchored match, state 0 has a free transition to the next and a costly one to itself, to repeat at the next position. To catch repeated identical empty matches, we need to know if a successful thread corresponds to an empty match. The instruction resetting `\l__regex_fresh_thread_bool` may be skipped by a successful thread, hence we had to add it to `__regex_match_one_token:nnN` too.

```

6835 \cs_new_protected:Npn \__regex_action_start_wildcard:N #1
6836 {
6837     \bool_set_true:N \l__regex_fresh_thread_bool
6838     \__regex_action_free:n {1}
6839     \bool_set_false:N \l__regex_fresh_thread_bool
6840     \bool_if:NT #1 { \__regex_action_cost:n {0} }
6841 }

```

(End definition for __regex_action_start_wildcard:N.)

__regex_action_free:n
__regex_action_free_group:n
__regex_action_free_aux:nn

These functions copy a thread after checking that the NFA state has not already been used at this position. If not, store submatches in the new state, and insert the instructions for that state in the input stream. Then restore the old value of `\l__regex_curr_state_int` and of the current submatches. The two types of free transitions differ by how they test that the state has not been encountered yet: the `group` version is stricter, and will not use a state if it was used earlier in the current thread, hence forcefully breaking the loop, while the “normal” version will revisit a state even within the thread itself.

```

6842 \cs_new_protected:Npn \__regex_action_free:n

```

```

6843 { \__regex_action_free_aux:nn { > \l__regex_step_int \else: } }
6844 \cs_new_protected:Npn \__regex_action_free_group:n
6845 { \__regex_action_free_aux:nn { < \l__regex_step_int } }
6846 \cs_new_protected:Npn \__regex_action_free_aux:nn #1#2
6847 {
6848   \use:x
6849   {
6850     \int_add:Nn \l__regex_curr_state_int {#2}
6851     \exp_not:n
6852     {
6853       \if_int_compare:w
6854         \__kernel_intarray_item:Nn \g__regex_state_active_intarray
6855         { \l__regex_curr_state_int }
6856         #1
6857         \exp_after:wN \__regex_use_state:
6858         \fi:
6859       }
6860       \int_set:Nn \l__regex_curr_state_int
6861       { \int_use:N \l__regex_curr_state_int }
6862       \tl_set:Nn \exp_not:N \l__regex_curr_submatches_tl
6863       { \exp_not:o \l__regex_curr_submatches_tl }
6864     }
6865   }

```

(End definition for __regex_action_free:n, __regex_action_free_group:n, and __regex_action_free_aux:nn.)

__regex_action_cost:n A transition which consumes the current character and shifts the state by #1. The resulting state is stored in the appropriate array for use at the next position, and we also store the current submatches.

```

6866 \cs_new_protected:Npn \__regex_action_cost:n #1
6867 {
6868   \exp_args:Nx \__regex_store_state:n
6869   { \int_eval:n { \l__regex_curr_state_int + #1 } }
6870 }

```

(End definition for __regex_action_cost:n.)

__regex_store_state:n Put the given state and current submatch information in \g__regex_thread_info_intarray, and increment the length of the array.

```

6871 \cs_new_protected:Npn \__regex_store_state:n #1
6872 {
6873   \exp_args:No \__regex_store_submatches:nn
6874   \l__regex_curr_submatches_tl {#1}
6875   \int_incr:N \l__regex_max_thread_int
6876 }
6877 \cs_new_protected:Npn \__regex_store_submatches:nn #1#2
6878 {
6879   \__kernel_intarray_gset_range_from_clist:Nnn
6880   \g__regex_thread_info_intarray
6881   {
6882     \__regex_int_eval:w
6883     1 + \l__regex_max_thread_int *
6884     (\l__regex_capturing_group_int * 2 + 1)

```

```

6885     }
6886     { #2 , #1 }
6887 }

```

(End definition for `_regex_store_state:n` and `_regex_store_submatches:.`)

`_regex_disable_submatches:` Some user functions don't require tracking submatches. We get a performance improvement by simply defining the relevant functions to remove their argument and do nothing with it.

```

6888 \cs_new_protected:Npn \_regex_disable_submatches:
6889 {
6890     \cs_set_protected:Npn \_regex_store_submatches:n ##1 { }
6891     \cs_set_protected:Npn \_regex_action_submatch:nN ##1##2 { }
6892 }

```

(End definition for `_regex_disable_submatches:.`)

`_regex_action_submatch:nN` Update the current submatches with the information from the current position. Maybe a bottleneck.

```

\regex_action_submatch_aux:w
\regex_action_submatch_auxii:w
\regex_action_submatch_auxiii:w
\regex_action_submatch_auxiv:w
6893 \cs_new_protected:Npn \_regex_action_submatch:nN #1#2
6894 {
6895     \exp_after:wN \_regex_action_submatch_aux:w
6896     \l__regex_curr_submatches_tl ; {#1} #2
6897 }
6898 \cs_new_protected:Npn \_regex_action_submatch_aux:w #1 ; #2#3
6899 {
6900     \tl_set:Nx \l__regex_curr_submatches_tl
6901     {
6902         \prg_replicate:nn
6903         { #2 \if_meaning:w > #3 + \l__regex_capturing_group_int \fi: }
6904         { \_regex_action_submatch_auxii:w }
6905         \_regex_action_submatch_auxiii:w
6906         #1
6907     }
6908 }
6909 \cs_new:Npn \_regex_action_submatch_auxii:w
6910     #1 \_regex_action_submatch_auxiii:w #2 ,
6911     { #2 , #1 \_regex_action_submatch_auxiii:w }
6912 \cs_new:Npn \_regex_action_submatch_auxiii:w #1 ,
6913     { \int_use:N \l__regex_curr_pos_int , }

```

(End definition for `_regex_action_submatch:nN` and others.)

`_regex_action_success:` There is a successful match when an execution path reaches the last state in the NFA, unless this marks a second identical empty match. Then mark that there was a successful match; it is empty if it is “fresh”; and we store the current position and submatches. The current step is then interrupted with `\prg_break:`, and only paths with higher precedence are pursued further. The values stored here may be overwritten by a later success of a path with higher precedence.

```

6914 \cs_new_protected:Npn \_regex_action_success:
6915 {
6916     \_regex_if_two_empty_matches:F
6917     {
6918         \bool_set_true:N \l__regex_match_success_bool

```

```

6919         \bool_set_eq:NN \l__regex_empty_success_bool
6920         \l__regex_fresh_thread_bool
6921         \int_set_eq:NN \l__regex_success_pos_int \l__regex_curr_pos_int
6922         \int_set_eq:NN \l__regex_last_char_success_int \l__regex_last_char_int
6923         \tl_build_clear:N \l__regex_matched_analysis_tl
6924         \tl_set_eq:NN \l__regex_success_submatches_tl
6925         \l__regex_curr_submatches_tl
6926         \prg_break:
6927     }
6928 }

```

(End definition for `__regex_action_success:.`)

45.6 Replacement

45.6.1 Variables and helpers used in replacement

`\l__regex_replacement_csnames_int` The behaviour of closing braces inside a replacement text depends on whether a sequences `\c{` or `\u{` has been encountered. The number of “open” such sequences that should be closed by `}` is stored in `\l__regex_replacement_csnames_int`, and decreased by 1 by each `}`.

```

6929 \int_new:N \l__regex_replacement_csnames_int

```

(End definition for `\l__regex_replacement_csnames_int.`)

`\l__regex_replacement_category_tl` This sequence of letters is used to correctly restore categories in nested constructions such as `\cL(abc\cD(_)d)`.

`\l__regex_replacement_category_seq`

```

6930 \tl_new:N \l__regex_replacement_category_tl
6931 \seq_new:N \l__regex_replacement_category_seq

```

(End definition for `\l__regex_replacement_category_tl` and `\l__regex_replacement_category_seq.`)

`\g__regex_balance_tl` This token list holds the replacement text for `__regex_replacement_balance_one-match:n` while it is being built incrementally.

```

6932 \tl_new:N \g__regex_balance_tl

```

(End definition for `\g__regex_balance_tl.`)

`__regex_replacement_balance_one_match:n` This expects as an argument the first index of a set of entries in `\g__regex_submatch_begin_intarray` (and related arrays) which hold the submatch information for a given match. It can be used within an integer expression to obtain the brace balance incurred by performing the replacement on that match. This combines the braces lost by removing the match, braces added by all the submatches appearing in the replacement, and braces appearing explicitly in the replacement. Even though it is always redefined before use, we initialize it as for an empty replacement. An important property is that concatenating several calls to that function must result in a valid integer expression (hence a leading + in the actual definition).

```

6933 \cs_new:Npn \__regex_replacement_balance_one_match:n #1
6934 { - \__regex_submatch_balance:n {#1} }

```

(End definition for `__regex_replacement_balance_one_match:n.`)

`_regex_replacement_do_one_match:n` The input is the same as `__regex_replacement_balance_one_match:n`. This function is redefined to expand to the part of the token list from the end of the previous match to a given match, followed by the replacement text. Hence concatenating the result of this function with all possible arguments (one call for each match), as well as the range from the end of the last match to the end of the string, produces the fully replaced token list. The initialization does not matter, but (as an example) we set it as for an empty replacement.

```

6935 \cs_new:Npn \__regex_replacement_do_one_match:n #1
6936 {
6937   \__regex_query_range:nn
6938   { \__kernel_intarray_item:Nn \g__regex_submatch_prev_intarray {#1} }
6939   { \__kernel_intarray_item:Nn \g__regex_submatch_begin_intarray {#1} }
6940 }

```

(End definition for `_regex_replacement_do_one_match:n`.)

`_regex_replacement_exp_not:N` This function lets us navigate around the fact that the primitive `\exp_not:n` requires a braced argument. As far as I can tell, it is only needed if the user tries to include in the replacement text a control sequence set equal to a macro parameter character, such as `\c_parameter_token`. Indeed, within an x-expanding assignment, `\exp_not:N #` behaves as a single `#`, whereas `\exp_not:n {#}` behaves as a doubled `##`.

```

6941 \cs_new:Npn \__regex_replacement_exp_not:N #1 { \exp_not:n {#1} }

```

(End definition for `_regex_replacement_exp_not:N`.)

`_regex_replacement_exp_not:V` This is used for the implementation of `\u`, and it gets redefined for `\peek_regex_replace_once:nnTF`.

```

6942 \cs_new_eq:NN \__regex_replacement_exp_not:V \exp_not:V

```

(End definition for `_regex_replacement_exp_not:V`.)

45.6.2 Query and brace balance

`_regex_query_range:nn` When it is time to extract submatches from the token list, the various tokens are stored in `\toks` registers numbered from `\l__regex_min_pos_int` inclusive to `\l__regex_max_pos_int` exclusive. The function `_regex_query_range:nn {<min>} {<max>}` unpacks registers from the position `<min>` to the position `<max> - 1` included. Once this is expanded, a second x-expansion results in the actual tokens from the query. That second expansion is only done by user functions at the very end of their operation, after checking (and correcting) the brace balance first.

```

6943 \cs_new:Npn \__regex_query_range:nn #1#2
6944 {
6945   \exp_after:wN \__regex_query_range_loop:ww
6946   \int_value:w \__regex_int_eval:w #1 \exp_after:wN ;
6947   \int_value:w \__regex_int_eval:w #2 ;
6948   \prg_break_point:
6949 }
6950 \cs_new:Npn \__regex_query_range_loop:ww #1 ; #2 ;
6951 {
6952   \if_int_compare:w #1 < #2 \exp_stop_f:
6953   \else:
6954     \exp_after:wN \prg_break:
6955   \fi:

```

```

6956     \_regex_toks_use:w #1 \exp_stop_f:
6957     \exp_after:wN \_regex_query_range_loop:ww
6958     \int_value:w \_regex_int_eval:w #1 + 1 ; #2 ;
6959 }

```

(End definition for _regex_query_range:nn and _regex_query_range_loop:ww.)

_regex_query_submatch:n Find the start and end positions for a given submatch (of a given match).

```

6960 \cs_new:Npn \_regex_query_submatch:n #1
6961 {
6962     \_regex_query_range:nn
6963     { \_kernel_intarray_item:Nn \g__regex_submatch_begin_intarray {#1} }
6964     { \_kernel_intarray_item:Nn \g__regex_submatch_end_intarray {#1} }
6965 }

```

(End definition for _regex_query_submatch:n.)

_regex_submatch_balance:n Every user function must result in a balanced token list (unbalanced token lists cannot be stored by TeX). When we unpacked the query, we kept track of the brace balance, hence the contribution from a given range is the difference between the brace balances at the $\langle \text{max pos} \rangle$ and $\langle \text{min pos} \rangle$. These two positions are found in the corresponding “submatch” arrays.

```

6966 \cs_new_protected:Npn \_regex_submatch_balance:n #1
6967 {
6968     \int_eval:n
6969     {
6970         \_regex_intarray_item:NnF \g__regex_balance_intarray
6971         {
6972             \_kernel_intarray_item:Nn
6973             \g__regex_submatch_end_intarray {#1}
6974         }
6975         { 0 }
6976     -
6977     \_regex_intarray_item:NnF \g__regex_balance_intarray
6978     {
6979         \_kernel_intarray_item:Nn
6980         \g__regex_submatch_begin_intarray {#1}
6981     }
6982     { 0 }
6983 }
6984 }

```

(End definition for _regex_submatch_balance:n.)

45.6.3 Framework

_regex_replacement:n The replacement text is built incrementally. We keep track in \l__regex_balance_int of the balance of explicit begin- and end-group tokens and we store in \g__regex_balance_tl some code to compute the brace balance from submatches (see its description). Detect unescaped right braces, and escaped characters, with trailing \prg_do_nothing: because some of the later function look-ahead. Once the whole replacement text has been parsed, make sure that there is no open csname. Finally, define the balance_one_match and do_one_match functions.

```

6985 \cs_new_protected:Npn \__regex_replacement:n
6986 { \__regex_replacement_apply:Nn \__regex_replacement_set:n }
6987 \cs_new_protected:Npn \__regex_replacement_apply:Nn #1#2
6988 {
6989   \group_begin:
6990     \tl_build_begin:N \l__regex_build_tl
6991     \int_zero:N \l__regex_balance_int
6992     \tl_gclear:N \g__regex_balance_tl
6993     \__regex_escape_use:nnnn
6994     {
6995       \if_charcode:w \c_right_brace_str ##1
6996         \__regex_replacement_rbrace:N
6997       \else:
6998         \if_charcode:w \c_left_brace_str ##1
6999           \__regex_replacement_lbrace:N
7000         \else:
7001           \__regex_replacement_normal:n
7002         \fi:
7003       \fi:
7004       ##1
7005     }
7006     { \__regex_replacement_escaped:N ##1 }
7007     { \__regex_replacement_normal:n ##1 }
7008     {#2}
7009   \prg_do_nothing: \prg_do_nothing:
7010   \if_int_compare:w \l__regex_replacement_csnames_int > \c_zero_int
7011     \msg_error:nnx { regex } { replacement-missing-rbrace }
7012     { \int_use:N \l__regex_replacement_csnames_int }
7013     \tl_build_put_right:Nx \l__regex_build_tl
7014     { \prg_replicate:nn \l__regex_replacement_csnames_int \cs_end: }
7015   \fi:
7016   \seq_if_empty:NF \l__regex_replacement_category_seq
7017   {
7018     \msg_error:nnx { regex } { replacement-missing-rparen }
7019     { \seq_count:N \l__regex_replacement_category_seq }
7020     \seq_clear:N \l__regex_replacement_category_seq
7021   }
7022   \tl_gput_right:Nx \g__regex_balance_tl
7023   { + \int_use:N \l__regex_balance_int }
7024   \tl_build_end:N \l__regex_build_tl
7025   \exp_args:NNo
7026   \group_end:
7027   #1 \l__regex_build_tl
7028 }
7029 \cs_generate_variant:Nn \__regex_replacement:n { x }
7030 \cs_new_protected:Npn \__regex_replacement_set:n #1
7031 {
7032   \cs_set:Npn \__regex_replacement_do_one_match:n ##1
7033   {
7034     \__regex_query_range:nn
7035     {
7036       \__kernel_intarray_item:Nn
7037       \g__regex_submatch_prev_intarray {##1}
7038     }

```

```

7039         {
7040             \__kernel_intarray_item:Nn
7041             \g__regex_submatch_begin_intarray {##1}
7042         }
7043         #1
7044     }
7045     \exp_args:Nno \use:n
7046     { \cs_gset:Npn \__regex_replacement_balance_one_match:n ##1 }
7047     {
7048         \g__regex_balance_tl
7049         - \__regex_submatch_balance:n {##1}
7050     }
7051 }

```

(End definition for __regex_replacement:n, __regex_replacement_apply:Nn, and __regex_replacement_set:n.)

__regex_case_replacement:n
__regex_case_replacement:x

```

7052 \tl_new:N \g__regex_case_replacement_tl
7053 \tl_new:N \g__regex_case_balance_tl
7054 \cs_new_protected:Npn \__regex_case_replacement:n #1
7055 {
7056     \tl_gset:Nn \g__regex_case_balance_tl
7057     {
7058         \if_case:w
7059             \__kernel_intarray_item:Nn
7060             \g__regex_submatch_case_intarray {##1}
7061         }
7062     \tl_gset_eq:NN \g__regex_case_replacement_tl \g__regex_case_balance_tl
7063     \tl_map_tokens:nn {#1}
7064     { \__regex_replacement_apply:Nn \__regex_case_replacement_aux:n }
7065     \tl_gset:No \g__regex_balance_tl
7066     { \g__regex_case_balance_tl \fi: }
7067     \exp_args:No \__regex_replacement_set:n
7068     { \g__regex_case_replacement_tl \fi: }
7069 }
7070 \cs_generate_variant:Nn \__regex_case_replacement:n { x }
7071 \cs_new_protected:Npn \__regex_case_replacement_aux:n #1
7072 {
7073     \tl_gput_right:Nn \g__regex_case_replacement_tl { \or: #1 }
7074     \tl_gput_right:No \g__regex_case_balance_tl
7075     { \exp_after:wN \or: \g__regex_balance_tl }
7076 }

```

(End definition for __regex_case_replacement:n.)

__regex_replacement_put:n This gets redefined for \peek_regex_replace_once:nnTF.

```

7077 \cs_new_protected:Npn \__regex_replacement_put:n
7078 { \tl_build_put_right:Nn \l__regex_build_tl }

```

(End definition for __regex_replacement_put:n.)

__regex_replacement_normal:n
__regex_replacement_normal_aux:N

Most characters are simply sent to the output by \tl_build_put_right:Nn, unless a particular category code has been requested: then __regex_replacement_c_A:w or a similar auxiliary is called. One exception is right parentheses, which restore the category

code in place before the group started. Note that the sequence is non-empty there: it contains an empty entry corresponding to the initial value of `\l__regex_replacement_category_tl`. The argument `#1` is a single character (including the case of a catcode-other space). In case no specific catcode is requested, we took into account the current catcode regime (at the time the replacement is performed) as much as reasonable, with all impossible catcodes (escape, newline, etc.) being mapped to “other”.

```

7079 \cs_new_protected:Npn \__regex_replacement_normal:n #1
7080 {
7081   \int_compare:nNnTF { \l__regex_replacement_csnames_int } > 0
7082   { \exp_args:No \__regex_replacement_put:n { \token_to_str:N #1 } }
7083   {
7084     \tl_if_empty:NTF \l__regex_replacement_category_tl
7085     { \__regex_replacement_normal_aux:N #1 }
7086     { % (
7087       \token_if_eq_charcode:NNTF #1 )
7088       {
7089         \seq_pop:NN \l__regex_replacement_category_seq
7090         \l__regex_replacement_category_tl
7091       }
7092       {
7093         \use:c { __regex_replacement_c_ \l__regex_replacement_category_tl :w }
7094         ? #1
7095       }
7096     }
7097   }
7098 }
7099 \cs_new_protected:Npn \__regex_replacement_normal_aux:N #1
7100 {
7101   \token_if_eq_charcode:NNTF #1 \c_space_token
7102   { \__regex_replacement_c_S:w }
7103   {
7104     \exp_after:wN \exp_after:wN
7105     \if_case:w \tex_catcode:D ‘#1 \exp_stop_f:
7106       \__regex_replacement_c_O:w
7107       \or: \__regex_replacement_c_B:w
7108       \or: \__regex_replacement_c_E:w
7109       \or: \__regex_replacement_c_M:w
7110       \or: \__regex_replacement_c_T:w
7111       \or: \__regex_replacement_c_O:w
7112       \or: \__regex_replacement_c_P:w
7113       \or: \__regex_replacement_c_U:w
7114       \or: \__regex_replacement_c_D:w
7115       \or: \__regex_replacement_c_O:w
7116       \or: \__regex_replacement_c_S:w
7117       \or: \__regex_replacement_c_L:w
7118       \or: \__regex_replacement_c_O:w
7119       \or: \__regex_replacement_c_A:w
7120       \else: \__regex_replacement_c_O:w
7121       \fi:
7122     }
7123     ? #1
7124   }

```

(End definition for `__regex_replacement_normal:n` and `__regex_replacement_normal_aux:N`.)

`_regex_replacement_escaped:N` As in parsing a regular expression, we use an auxiliary built from `#1` if defined. Otherwise, check for escaped digits (standing from submatches from 0 to 9): anything else is a raw character.

```

7125 \cs_new_protected:Npn \_regex_replacement_escaped:N #1
7126 {
7127   \cs_if_exist_use:cF { \_regex_replacement_#1:w }
7128   {
7129     \if_int_compare:w 1 < 1#1 \exp_stop_f:
7130     \_regex_replacement_put_submatch:n {#1}
7131   \else:
7132     \_regex_replacement_normal:n {#1}
7133   \fi:
7134 }
7135 }

```

(End definition for `_regex_replacement_escaped:N`.)

45.6.4 Submatches

`_regex_replacement_put_submatch:n`
`_regex_replacement_put_submatch_aux:n` Insert a submatch in the replacement text. This is dropped if the submatch number is larger than the number of capturing groups. Unless the submatch appears inside a `\c{...}` or `\u{...}` construction, it must be taken into account in the brace balance. Later on, `##1` will be replaced by a pointer to the 0-th submatch for a given match. There is an `\exp_not:N` here as at the point-of-use of `\g__regex_balance_tl` there is an x-type expansion which is needed to get `##1` in correctly.

```

7136 \cs_new_protected:Npn \_regex_replacement_put_submatch:n #1
7137 {
7138   \if_int_compare:w #1 < \l__regex_capturing_group_int
7139   \_regex_replacement_put_submatch_aux:n {#1}
7140   \else:
7141     \msg_expandable_error:nnff { regex } { submatch-too-big }
7142     {#1} { \int_eval:n { \l__regex_capturing_group_int - 1 } }
7143   \fi:
7144 }
7145 \cs_new_protected:Npn \_regex_replacement_put_submatch_aux:n #1
7146 {
7147   \tl_build_put_right:Nn \l__regex_build_tl
7148   { \_regex_query_submatch:n { \int_eval:n { #1 + ##1 } } }
7149   \if_int_compare:w \l__regex_replacement_csnames_int = \c_zero_int
7150   \tl_gput_right:Nn \g__regex_balance_tl
7151   { + \_regex_submatch_balance:n { \int_eval:n { #1 + ##1 } } }
7152   \fi:
7153 }

```

(End definition for `_regex_replacement_put_submatch:n` and `_regex_replacement_put_submatch_aux:n`.)

`_regex_replacement_g:w`
`_regex_replacement_g_digits:NN` Grab digits for the `\g` escape sequence in a primitive assignment to the integer `\l__regex_internal_a_int`. At the end of the run of digits, check that it ends with a right brace.

```

7154 \cs_new_protected:Npn \_regex_replacement_g:w #1#2
7155 {
7156   \token_if_eq_meaning:NNTF #1 \_regex_replacement_lbrace:N

```

```

7157     { \l__regex_internal_a_int = \__regex_replacement_g_digits:NN }
7158     { \__regex_replacement_error:NNN g #1 #2 }
7159   }
7160 \cs_new:Npn \__regex_replacement_g_digits:NN #1#2
7161 {
7162   \token_if_eq_meaning:NNTF #1 \__regex_replacement_normal:n
7163   {
7164     \if_int_compare:w 1 < 1#2 \exp_stop_f:
7165       #2
7166       \exp_after:wN \use_i:nnn
7167       \exp_after:wN \__regex_replacement_g_digits:NN
7168     \else:
7169       \exp_stop_f:
7170       \exp_after:wN \__regex_replacement_error:NNN
7171       \exp_after:wN g
7172     \fi:
7173   }
7174   {
7175     \exp_stop_f:
7176     \if_meaning:w \__regex_replacement_rbrace:N #1
7177       \exp_args:No \__regex_replacement_put_submatch:n
7178       { \int_use:N \l__regex_internal_a_int }
7179       \exp_after:wN \use_none:nn
7180     \else:
7181       \exp_after:wN \__regex_replacement_error:NNN
7182       \exp_after:wN g
7183     \fi:
7184   }
7185   #1 #2
7186 }

```

(End definition for __regex_replacement_g:w and __regex_replacement_g_digits:NN.)

45.6.5 Csnames in replacement

__regex_replacement_c:w \c may only be followed by an unescaped character. If followed by a left brace, start a control sequence by calling an auxiliary common with \u. Otherwise test whether the category is known; if it is not, complain.

```

7187 \cs_new_protected:Npn \__regex_replacement_c:w #1#2
7188 {
7189   \token_if_eq_meaning:NNTF #1 \__regex_replacement_normal:n
7190   {
7191     \cs_if_exist:cTF { __regex_replacement_c_#2:w }
7192     { \__regex_replacement_cat:NNN #2 }
7193     { \__regex_replacement_error:NNN c #1#2 }
7194   }
7195   {
7196     \token_if_eq_meaning:NNTF #1 \__regex_replacement_lbrace:N
7197     { \__regex_replacement_cu_aux:Nw \__regex_replacement_exp_not:N }
7198     { \__regex_replacement_error:NNN c #1#2 }
7199   }
7200 }

```

(End definition for __regex_replacement_c:w.)

`_regex_replacement_cu_aux:Nw` Start a control sequence with `\cs:w`, protected from expansion by #1 (either `__regex_replacement_exp_not:N` or `\exp_not:V`), or turned to a string by `\tl_to_str:V` if inside another csname construction `\c` or `\u`. We use `\tl_to_str:V` rather than `\tl_to_str:N` to deal with integers and other registers.

```

7201 \cs_new_protected:Npn \_regex_replacement_cu_aux:Nw #1
7202 {
7203   \if_case:w \l__regex_replacement_csnames_int
7204     \tl_build_put_right:Nn \l__regex_build_tl
7205       { \exp_not:n { \exp_after:wN #1 \cs:w } }
7206   \else:
7207     \tl_build_put_right:Nn \l__regex_build_tl
7208       { \exp_not:n { \exp_after:wN \tl_to_str:V \cs:w } }
7209   \fi:
7210   \int_incr:N \l__regex_replacement_csnames_int
7211 }

```

(End definition for `_regex_replacement_cu_aux:Nw`.)

`__regex_replacement_u:w` Check that `\u` is followed by a left brace. If so, start a control sequence with `\cs:w`, which is then unpacked either with `\exp_not:V` or `\tl_to_str:V` depending on the current context.

```

7212 \cs_new_protected:Npn \__regex_replacement_u:w #1#2
7213 {
7214   \token_if_eq_meaning:NNTF #1 \__regex_replacement_lbrace:N
7215     { \_regex_replacement_cu_aux:Nw \_regex_replacement_exp_not:V }
7216     { \_regex_replacement_error:NNN u #1#2 }
7217 }

```

(End definition for `__regex_replacement_u:w`.)

`_regex_replacement_rbrace:N` Within a `\c{...}` or `\u{...}` construction, end the control sequence, and decrease the brace count. Otherwise, this is a raw right brace.

```

7218 \cs_new_protected:Npn \_regex_replacement_rbrace:N #1
7219 {
7220   \if_int_compare:w \l__regex_replacement_csnames_int > \c_zero_int
7221     \tl_build_put_right:Nn \l__regex_build_tl { \cs_end: }
7222     \int_decr:N \l__regex_replacement_csnames_int
7223   \else:
7224     \_regex_replacement_normal:n {#1}
7225   \fi:
7226 }

```

(End definition for `_regex_replacement_rbrace:N`.)

`__regex_replacement_lbrace:N` Within a `\c{...}` or `\u{...}` construction, this is forbidden. Otherwise, this is a raw left brace.

```

7227 \cs_new_protected:Npn \__regex_replacement_lbrace:N #1
7228 {
7229   \if_int_compare:w \l__regex_replacement_csnames_int > \c_zero_int
7230     \msg_error:nnn { regex } { cu-lbrace } { u }
7231   \else:
7232     \_regex_replacement_normal:n {#1}
7233   \fi:
7234 }

```

(End definition for `__regex_replacement_lbrace:N`.)

45.6.6 Characters in replacement

`_regex_replacement_cat:NNN` Here, `#1` is a letter among BEMTPUDSLOA and `#2#3` denote the next character. Complain if we reach the end of the replacement or if the construction appears inside `\\c{...}` or `\\u{...}`, and detect the case of a parenthesis. In that case, store the current category in a sequence and switch to a new one.

```

7235 \\cs_new_protected:Npn \\_regex_replacement_cat:NNN #1#2#3
7236 {
7237   \\token_if_eq_meaning:NNTF \\prg_do_nothing: #3
7238   { \\msg_error:nn { regex } { replacement-catcode-end } }
7239   {
7240     \\int_compare:nNnTF { \\l__regex_replacement_csnames_int } > 0
7241     {
7242       \\msg_error:nnnn
7243       { regex } { replacement-catcode-in-cs } {#1} {#3}
7244       #2 #3
7245     }
7246     {
7247       \\_regex_two_if_eq:NNNTF #2 #3 \\_regex_replacement_normal:n (
7248       {
7249         \\seq_push:NV \\l__regex_replacement_category_seq
7250         \\l__regex_replacement_category_tl
7251         \\tl_set:Nn \\l__regex_replacement_category_tl {#1}
7252       }
7253       {
7254         \\token_if_eq_meaning:NNT #2 \\_regex_replacement_escaped:N
7255         {
7256           \\_regex_char_if_alphanumeric:NTF #3
7257           {
7258             \\msg_error:nnnn
7259             { regex } { replacement-catcode-escaped }
7260             {#1} {#3}
7261           }
7262           { }
7263         }
7264         \\use:c { __regex_replacement_c_#1:w } #2 #3
7265       }
7266     }
7267   }
7268 }
```

(End definition for `_regex_replacement_cat:NNN`.)

We now need to change the category code of the null character many times, hence work in a group. The catcode-specific macros below are defined in alphabetical order; if you are trying to understand the code, start from the end of the alphabet as those categories are simpler than active or begin-group.

```

7269 \\group_begin:
```

`_regex_replacement_char:nNN` The only way to produce an arbitrary character-catcode pair is to use the `\\lowercase` or `\\uppercase` primitives. This is a wrapper for our purposes. The first argument is the null character with various catcodes. The second and third arguments are grabbed from the input stream: `#3` is the character whose character code to reproduce. We could use

`\char_generate:nn` but only for some catcodes (active characters and spaces are not supported).

```

7270 \cs_new_protected:Npn \__regex_replacement_char:nNN #1#2#3
7271 {
7272   \tex_lccode:D 0 = '#3 \scan_stop:
7273   \tex_lowercase:D { \__regex_replacement_put:n {#1} }
7274 }

```

(End definition for __regex_replacement_char:nNN.)

`__regex_replacement_c_A:w` For an active character, expansion must be avoided, twice because we later do two x-expansions, to unpack `\toks` for the query, and to expand their contents to tokens of the query.

```

7275 \char_set_catcode_active:N \^^@
7276 \cs_new_protected:Npn \__regex_replacement_c_A:w
7277 { \__regex_replacement_char:nNN { \exp_not:n { \exp_not:N \^^@ } } }

```

(End definition for __regex_replacement_c_A:w.)

`__regex_replacement_c_B:w` An explicit begin-group token increases the balance, unless within a `\c{...}` or `\u{...}` construction. Add the desired begin-group character, using the standard `\if_false:` trick. We eventually x-expand twice. The first time must yield a balanced token list, and the second one gives the bare begin-group token. The `\exp_after:wN` is not strictly needed, but is more consistent with l3tl-analysis.

```

7278 \char_set_catcode_group_begin:N \^^@
7279 \cs_new_protected:Npn \__regex_replacement_c_B:w
7280 {
7281   \if_int_compare:w \l__regex_replacement_csnames_int = \c_zero_int
7282   \int_incr:N \l__regex_balance_int
7283   \fi:
7284   \__regex_replacement_char:nNN
7285   { \exp_not:n { \exp_after:wN \^^@ \if_false: } \fi: } }
7286 }

```

(End definition for __regex_replacement_c_B:w.)

`__regex_replacement_c_C:w` This is not quite catcode-related: when the user requests a character with category “control sequence”, the one-character control symbol is returned. As for the active character, we prepare for two x-expansions.

```

7287 \cs_new_protected:Npn \__regex_replacement_c_C:w #1#2
7288 {
7289   \tl_build_put_right:Nn \l__regex_build_tl
7290   { \exp_not:N \__regex_replacement_exp_not:N \exp_not:c {#2} }
7291 }

```

(End definition for __regex_replacement_c_C:w.)

`__regex_replacement_c_D:w` Subscripts fit the mould: `\lowercase` the null byte with the correct category.

```

7292 \char_set_catcode_math_subscript:N \^^@
7293 \cs_new_protected:Npn \__regex_replacement_c_D:w
7294 { \__regex_replacement_char:nNN { \^^@ } }

```

(End definition for __regex_replacement_c_D:w.)

`_regex_replacement_c_E:w` Similar to the begin-group case, the second x-expansion produces the bare end-group token.

```

7295 \char_set_catcode_group_end:N \^^@
7296 \cs_new_protected:Npn \_regex_replacement_c_E:w
7297 {
7298   \if_int_compare:w \l__regex_replacement_csnames_int = \c_zero_int
7299     \int_decr:N \l__regex_balance_int
7300   \fi:
7301   \_regex_replacement_char:nNN
7302     { \exp_not:n { \if_false: { \fi: ^^@ } }
7303   }

```

(End definition for _regex_replacement_c_E:w.)

`_regex_replacement_c_L:w` Simply `\lowercase` a letter null byte to produce an arbitrary letter.

```

7304 \char_set_catcode_letter:N \^^@
7305 \cs_new_protected:Npn \_regex_replacement_c_L:w
7306 { \_regex_replacement_char:nNN { ^^@ } }

```

(End definition for _regex_replacement_c_L:w.)

`_regex_replacement_c_M:w` No surprise here, we lowercase the null math toggle.

```

7307 \char_set_catcode_math_toggle:N \^^@
7308 \cs_new_protected:Npn \_regex_replacement_c_M:w
7309 { \_regex_replacement_char:nNN { ^^@ } }

```

(End definition for _regex_replacement_c_M:w.)

`_regex_replacement_c_O:w` Lowercase an other null byte.

```

7310 \char_set_catcode_other:N \^^@
7311 \cs_new_protected:Npn \_regex_replacement_c_O:w
7312 { \_regex_replacement_char:nNN { ^^@ } }

```

(End definition for _regex_replacement_c_O:w.)

`_regex_replacement_c_P:w` For macro parameters, expansion is a tricky issue. We need to prepare for two x-expansions and passing through various macro definitions. Note that we cannot replace one `\exp_not:n` by doubling the macro parameter characters because this would misbehave if a mischievous user asks for `\c{\cP\#}`, since that macro parameter character would be doubled.

```

7313 \char_set_catcode_parameter:N \^^@
7314 \cs_new_protected:Npn \_regex_replacement_c_P:w
7315 {
7316   \_regex_replacement_char:nNN
7317     { \exp_not:n { \exp_not:n { ^^@^^@^^@^^@ } } }
7318 }

```

(End definition for _regex_replacement_c_P:w.)

`_regex_replacement_c_S:w` Spaces are normalized on input by TeX to have character code 32. It is in fact impossible to get a token with character code 0 and category code 10. Hence we use 32 instead of 0 as our base character.

```

7319 \cs_new_protected:Npn \_regex_replacement_c_S:w #1#2
7320 {

```

```

7321 \if_int_compare:w '#2 = \c_zero_int
7322 \msg_error:nn { regex } { replacement-null-space }
7323 \fi:
7324 \tex_lccode:D '\ = '#2 \scan_stop:
7325 \tex_lowercase:D { \__regex_replacement_put:n {~} }
7326 }

```

(End definition for __regex_replacement_c_S:w.)

__regex_replacement_c_T:w No surprise for alignment tabs here. Those are surrounded by the appropriate braces whenever necessary, hence they don't cause trouble in alignment settings.

```

7327 \char_set_catcode_alignment:N \^^@
7328 \cs_new_protected:Npn \__regex_replacement_c_T:w
7329 { \__regex_replacement_char:nNN { ^^@ } }

```

(End definition for __regex_replacement_c_T:w.)

__regex_replacement_c_U:w Simple call to __regex_replacement_char:nNN which lowercases the math superscript ^^@.

```

7330 \char_set_catcode_math_superscript:N \^^@
7331 \cs_new_protected:Npn \__regex_replacement_c_U:w
7332 { \__regex_replacement_char:nNN { ^^@ } }

```

(End definition for __regex_replacement_c_U:w.)

Restore the catcode of the null byte.

```

7333 \group_end:

```

45.6.7 An error

_regex_replacement_error:NNN Simple error reporting by calling one of the messages replacement-c, replacement-g, or replacement-u.

```

7334 \cs_new_protected:Npn \_regex_replacement_error:NNN #1#2#3
7335 {
7336   \msg_error:nnx { regex } { replacement-#1 } {#3}
7337   #2 #3
7338 }

```

(End definition for _regex_replacement_error:NNN.)

45.7 User functions

\regex_new:N Before being assigned a sensible value, a regex variable matches nothing.

```

7339 \cs_new_protected:Npn \regex_new:N #1
7340 { \cs_new_eq:NN #1 \c__regex_no_match_regex }

```

(End definition for \regex_new:N. This function is documented on page 54.)

\l_tmpa_regex The usual scratch space.

```

7341 \regex_new:N \l_tmpa_regex
7342 \regex_new:N \l_tmpb_regex
7343 \regex_new:N \g_tmpa_regex
7344 \regex_new:N \g_tmpb_regex

```

(End definition for `\l_tmpa_regex` and others. These variables are documented on page 59.)

`\regex_set:Nn` Compile, then store the result in the user variable with the appropriate assignment function.
`\regex_gset:Nn`
`\regex_const:Nn`

```

7345 \cs_new_protected:Npn \regex_set:Nn #1#2
7346 {
7347   \__regex_compile:n {#2}
7348   \tl_set_eq:NN #1 \l__regex_internal_regex
7349 }
7350 \cs_new_protected:Npn \regex_gset:Nn #1#2
7351 {
7352   \__regex_compile:n {#2}
7353   \tl_gset_eq:NN #1 \l__regex_internal_regex
7354 }
7355 \cs_new_protected:Npn \regex_const:Nn #1#2
7356 {
7357   \__regex_compile:n {#2}
7358   \tl_const:Nx #1 { \exp_not:o \l__regex_internal_regex }
7359 }

```

(End definition for `\regex_set:Nn`, `\regex_gset:Nn`, and `\regex_const:Nn`. These functions are documented on page 54.)

`\regex_show:n` User functions: the `n` variant requires compilation first. Then show the variable with some appropriate text. The auxiliary `__regex_show:N` is defined in a different section.
`\regex_log:n`

```

\__regex_show:Nn 7360 \cs_new_protected:Npn \regex_show:n { \__regex_show:Nn \msg_show:nnxxxx }
\regex_show:N 7361 \cs_new_protected:Npn \regex_log:n { \__regex_show:Nn \msg_log:nnxxxx }
\regex_log:N 7362 \cs_new_protected:Npn \__regex_show:Nn #1#2
\__regex_show:NN 7363 {
7364   \__regex_compile:n {#2}
7365   \__regex_show:N \l__regex_internal_regex
7366   #1 { regex } { show }
7367   { \tl_to_str:n {#2} } { }
7368   { \l__regex_internal_a_tl } { }
7369 }
7370 \cs_new_protected:Npn \regex_show:N { \__regex_show:NN \msg_show:nnxxxx }
7371 \cs_new_protected:Npn \regex_log:N { \__regex_show:NN \msg_log:nnxxxx }
7372 \cs_new_protected:Npn \__regex_show:NN #1#2
7373 {
7374   \__kernel_chk_tl_type:NnnT #2 { regex }
7375   { \exp_args:No \__regex_clean_regex:n {#2} }
7376   {
7377     \__regex_show:N #2
7378     #1 { regex } { show }
7379     { } { \token_to_str:N #2 }
7380     { \l__regex_internal_a_tl } { }
7381   }
7382 }

```

(End definition for `\regex_show:n` and others. These functions are documented on page 54.)

`\regex_match:nnTF` Those conditionals are based on a common auxiliary defined later. Its first argument
`\regex_match:NnTF` builds the NFA corresponding to the regex, and the second argument is the query token

list. Once we have performed the match, convert the resulting boolean to `\prg_return_true:` or false.

```

7383 \prg_new_protected_conditional:Npnn \regex_match:nn #1#2 { T , F , TF }
7384 {
7385     \__regex_if_match:nn { \__regex_build:n {#1} } {#2}
7386     \__regex_return:
7387 }
7388 \prg_new_protected_conditional:Npnn \regex_match:Nn #1#2 { T , F , TF }
7389 {
7390     \__regex_if_match:nn { \__regex_build:N #1 } {#2}
7391     \__regex_return:
7392 }
```

(End definition for `\regex_match:nnTF` and `\regex_match:NnTF`. These functions are documented on page 55.)

`\regex_count:nnN`
`\regex_count:NnN`

Again, use an auxiliary whose first argument builds the NFA.

```

7393 \cs_new_protected:Npn \regex_count:nnN #1
7394 { \__regex_count:nnN { \__regex_build:n {#1} } }
7395 \cs_new_protected:Npn \regex_count:NnN #1
7396 { \__regex_count:nnN { \__regex_build:N #1 } }
```

(End definition for `\regex_count:nnN` and `\regex_count:NnN`. These functions are documented on page 55.)

`\regex_match_case:nn`
`\regex_match_case:nnTF`

The auxiliary errors if #1 has an odd number of items, and otherwise it sets `\g__regex_case_int` according to which case was found (zero if not found). The `true` branch leaves the corresponding code in the input stream.

```

7397 \cs_new_protected:Npn \regex_match_case:nnTF #1#2#3
7398 {
7399     \__regex_match_case:nnTF {#1} {#2}
7400     {
7401         \tl_item:nn {#1} { 2 * \g__regex_case_int }
7402         #3
7403     }
7404 }
7405 \cs_new_protected:Npn \regex_match_case:nn #1#2
7406 { \regex_match_case:nnTF {#1} {#2} { } { } }
7407 \cs_new_protected:Npn \regex_match_case:nnT #1#2#3
7408 { \regex_match_case:nnTF {#1} {#2} {#3} { } }
7409 \cs_new_protected:Npn \regex_match_case:nnF #1#2
7410 { \regex_match_case:nnTF {#1} {#2} { } { } }
```

(End definition for `\regex_match_case:nnTF`. This function is documented on page 55.)

`\regex_extract_once:nnN`
`\regex_extract_once:nnNTF`
`\regex_extract_once:NnN`
`\regex_extract_once:NnNTF`
`\regex_extract_all:nnN`
`\regex_extract_all:nnNTF`
`\regex_extract_all:NnN`
`\regex_extract_all:NnNTF`
`\regex_replace_once:nnN`
`\regex_replace_once:nnNTF`
`\regex_replace_once:NnN`
`\regex_replace_once:NnNTF`
`\regex_replace_all:nnN`
`\regex_replace_all:nnNTF`
`\regex_replace_all:NnN`
`\regex_replace_all:NnNTF`
`\regex_split:nnN`
`\regex_split:nnNTF`
`\regex_split:NnN`

We define here 40 user functions, following a common pattern in terms of `:nnN` auxiliaries, defined in the coming subsections. The auxiliary is handed `__regex_build:n` or `__regex_build:N` with the appropriate regex argument, then all other necessary arguments (replacement text, token list, *etc.* The conditionals call `__regex_return:` to return either true or false once matching has been performed.

```

7411 \cs_set_protected:Npn \__regex_tmp:w #1#2#3
7412 {
7413     \cs_new_protected:Npn #2 ##1 { #1 { \__regex_build:n {##1} } }
7414     \cs_new_protected:Npn #3 ##1 { #1 { \__regex_build:N ##1 } }
```

```

7415 \prg_new_protected_conditional:Npnn #2 ##1##2##3 { T , F , TF }
7416 { #1 { \__regex_build:n {##1} } {##2} ##3 \__regex_return: }
7417 \prg_new_protected_conditional:Npnn #3 ##1##2##3 { T , F , TF }
7418 { #1 { \__regex_build:N ##1 } {##2} ##3 \__regex_return: }
7419 }
7420 \__regex_tmp:w \__regex_extract_once:nnN
7421 \regex_extract_once:nnN \regex_extract_once:NnN
7422 \__regex_tmp:w \__regex_extract_all:nnN
7423 \regex_extract_all:nnN \regex_extract_all:NnN
7424 \__regex_tmp:w \__regex_replace_once:nnN
7425 \regex_replace_once:nnN \regex_replace_once:NnN
7426 \__regex_tmp:w \__regex_replace_all:nnN
7427 \regex_replace_all:nnN \regex_replace_all:NnN
7428 \__regex_tmp:w \__regex_split:nnN \regex_split:nnN \regex_split:NnN

```

(End definition for `\regex_extract_once:nnNTF` and others. These functions are documented on page 56.)

`\regex_replace_case_once:nN`
`\regex_replace_case_once:nNTF`

If the input is bad (odd number of items) then take the false branch. Otherwise, use the same auxiliary as `\regex_replace_once:nnN`, but with more complicated code to build the automaton, and to find what replacement text to use. The `\tl_item:nn` is only expanded once we know the value of `\g__regex_case_int`, namely which case matched.

```

7429 \cs_new_protected:Npn \regex_replace_case_once:nNTF #1#2
7430 {
7431   \int_if_odd:nTF { \tl_count:n {#1} }
7432   {
7433     \msg_error:nnxxxx { regex } { case-odd }
7434     { \token_to_str:N \regex_replace_case_once:nN(TF) } { code }
7435     { \tl_count:n {#1} } { \tl_to_str:n {#1} }
7436     \use_ii:nn
7437   }
7438   {
7439     \__regex_replace_once_aux:nnN
7440     { \__regex_case_build:x { \__regex_tl_odd_items:n {#1} } }
7441     { \__regex_replacement:x { \tl_item:nn {#1} { 2 * \g__regex_case_int } } }
7442     #2
7443     \bool_if:NTF \g__regex_success_bool
7444   }
7445 }
7446 \cs_new_protected:Npn \regex_replace_case_once:nN #1#2
7447 { \regex_replace_case_once:nNTF {#1} {#2} { } { } }
7448 \cs_new_protected:Npn \regex_replace_case_once:nNT #1#2#3
7449 { \regex_replace_case_once:nNTF {#1} {#2} {#3} { } }
7450 \cs_new_protected:Npn \regex_replace_case_once:nNF #1#2
7451 { \regex_replace_case_once:nNTF {#1} {#2} { } }

```

(End definition for `\regex_replace_case_once:nNTF`. This function is documented on page 58.)

`\regex_case_replace_all:nN`
`\regex_case_replace_all:nNTF`

If the input is bad (odd number of items) then take the false branch. Otherwise, use the same auxiliary as `\regex_replace_all:nnN`, but with more complicated code to build the automaton, and to find what replacement text to use.

```

7452 \cs_new_protected:Npn \regex_replace_case_all:nNTF #1#2
7453 {
7454   \int_if_odd:nTF { \tl_count:n {#1} }

```

```

7455     {
7456         \msg_error:nnxxxx { regex } { case-odd }
7457         { \token_to_str:N \regex_replace_case_all:nN(TF) } { code }
7458         { \tl_count:n {#1} } { \tl_to_str:n {#1} }
7459         \use_ii:nn
7460     }
7461     {
7462         \__regex_replace_all_aux:nnN
7463         { \__regex_case_build:x { \__regex_tl_odd_items:n {#1} } }
7464         { \__regex_case_replacement:x { \__regex_tl_even_items:n {#1} } }
7465         #2
7466         \bool_if:NTF \g__regex_success_bool
7467     }
7468 }
7469 \cs_new_protected:Npn \regex_replace_case_all:nN #1#2
7470 { \regex_replace_case_all:nNTF {#1} {#2} { } { } }
7471 \cs_new_protected:Npn \regex_replace_case_all:nNT #1#2#3
7472 { \regex_replace_case_all:nNTF {#1} {#2} {#3} { } }
7473 \cs_new_protected:Npn \regex_replace_case_all:nNF #1#2
7474 { \regex_replace_case_all:nNTF {#1} {#2} { } }

```

(End definition for `\regex_case_replace_all:nNTF`. This function is documented on page ??.)

45.7.1 Variables and helpers for user functions

`\l__regex_match_count_int` The number of matches found so far is stored in `\l__regex_match_count_int`. This is only used in the `\regex_count:nnN` functions.

```

7475 \int_new:N \l__regex_match_count_int

```

(End definition for `\l__regex_match_count_int`.)

`__regex_begin` `__regex_end` Those flags are raised to indicate begin-group or end-group tokens that had to be added when extracting submatches.

```

7476 \flag_new:n { __regex_begin }
7477 \flag_new:n { __regex_end }

```

(End definition for `__regex_begin` and `__regex_end`.)

`\l__regex_min_submatch_int` `\l__regex_submatch_int` `\l__regex_zeroth_submatch_int` The end-points of each submatch are stored in two arrays whose index *submatch* ranges from `\l__regex_min_submatch_int` (inclusive) to `\l__regex_submatch_int` (exclusive). Each successful match comes with a 0-th submatch (the full match), and one match for each capturing group: submatches corresponding to the last successful match are labelled starting at `zeroth_submatch`. The entry `\l__regex_zeroth_submatch_int` in `\g__regex_submatch_prev_intarray` holds the position at which that match attempt started: this is used for splitting and replacements.

```

7478 \int_new:N \l__regex_min_submatch_int
7479 \int_new:N \l__regex_submatch_int
7480 \int_new:N \l__regex_zeroth_submatch_int

```

(End definition for `\l__regex_min_submatch_int`, `\l__regex_submatch_int`, and `\l__regex_zeroth_submatch_int`.)


```

\g_regex_submatch_prev_intarray 7481 \intarray_new:Nn \g_regex_submatch_prev_intarray { 65536 }
\g_regex_submatch_begin_intarray 7482 \intarray_new:Nn \g_regex_submatch_begin_intarray { 65536 }
\g_regex_submatch_end_intarray 7483 \intarray_new:Nn \g_regex_submatch_end_intarray { 65536 }
\g_regex_submatch_case_intarray 7484 \intarray_new:Nn \g_regex_submatch_case_intarray { 65536 }

(End definition for \g_regex_submatch_prev_intarray and others.)

\g_regex_balance_intarray The first thing we do when matching is to store the balance of begin-group/end-group
                           characters into \g_regex_balance_intarray.
                           7485 \intarray_new:Nn \g_regex_balance_intarray { 65536 }

(End definition for \g_regex_balance_intarray.)

\l__regex_added_begin_int Keep track of the number of left/right braces to add when performing a regex operation
\l__regex_added_end_int   such as a replacement.
                           7486 \int_new:N \l__regex_added_begin_int
                           7487 \int_new:N \l__regex_added_end_int

(End definition for \l__regex_added_begin_int and \l__regex_added_end_int.)

\__regex_return: This function triggers either \prg_return_false: or \prg_return_true: as appropriate
                  to whether a match was found or not. It is used by all user conditionals.
                  7488 \cs_new_protected:Npn \__regex_return:
                  7489 {
                  7490     \if_meaning:w \c_true_bool \g_regex_success_bool
                  7491         \prg_return_true:
                  7492     \else:
                  7493         \prg_return_false:
                  7494     \fi:
                  7495 }

(End definition for \__regex_return:.)

\__regex_query_set:n To easily extract subsets of the input once we found the positions at which to cut, store
\__regex_query_set_aux:nN the input tokens one by one into successive \toks registers. Also store the brace balance
                           (used to check for overall brace balance) in an array.
                           7496 \cs_new_protected:Npn \__regex_query_set:n #1
                           7497 {
                           7498     \int_zero:N \l__regex_balance_int
                           7499     \int_zero:N \l__regex_curr_pos_int
                           7500     \__regex_query_set_aux:nN { } F
                           7501     \tl_analysis_map_inline:nn {#1}
                           7502         { \__regex_query_set_aux:nN {##1} ##3 }
                           7503     \__regex_query_set_aux:nN { } F
                           7504     \int_set_eq:NN \l__regex_max_pos_int \l__regex_curr_pos_int
                           7505 }
                           7506 \cs_new_protected:Npn \__regex_query_set_aux:nN #1#2
                           7507 {
                           7508     \int_incr:N \l__regex_curr_pos_int
                           7509     \__regex_toks_set:Nn \l__regex_curr_pos_int {#1}
                           7510     \__kernel_intarray_gset:Nnn \g_regex_balance_intarray
                           7511         { \l__regex_curr_pos_int } { \l__regex_balance_int }

```

```

7512     \if_case:w "#2 \exp_stop_f:
7513     \or: \int_incr:N \l__regex_balance_int
7514     \or: \int_decr:N \l__regex_balance_int
7515     \fi:
7516 }

```

(End definition for __regex_query_set:n and __regex_query_set_aux:nN.)

45.7.2 Matching

__regex_if_match:nn We don't track submatches, and stop after a single match. Build the NFA with #1, and perform the match on the query #2.

```

7517 \cs_new_protected:Npn \__regex_if_match:nn #1#2
7518 {
7519     \group_begin:
7520     \__regex_disable_submatches:
7521     \__regex_single_match:
7522     #1
7523     \__regex_match:n {#2}
7524     \group_end:
7525 }

```

(End definition for __regex_if_match:nn.)

__regex_match_case:nnTF The code would get badly messed up if the number of items in #1 were not even, so we catch this case, then follow the same code as \regex_match:nnTF but using __regex_case_build:n and without returning a result.

```

7526 \cs_new_protected:Npn \__regex_match_case:nnTF #1#2
7527 {
7528     \int_if_odd:nTF { \tl_count:n {#1} }
7529     {
7530         \msg_error:nnxxxx { regex } { case-odd }
7531         { \token_to_str:N \regex_match_case:nn(TF) } { code }
7532         { \tl_count:n {#1} } { \tl_to_str:n {#1} }
7533         \use_ii:nn
7534     }
7535     {
7536         \__regex_if_match:nn
7537         { \__regex_case_build:x { \__regex_tl_odd_items:n {#1} } }
7538         {#2}
7539         \bool_if:NTF \g__regex_success_bool
7540     }
7541 }
7542 \cs_new:Npn \__regex_match_case_aux:nn #1#2 { \exp_not:n { {#1} } }

```

(End definition for __regex_match_case:nnTF and __regex_match_case_aux:nn.)

__regex_count:nnN Again, we don't care about submatches. Instead of aborting after the first "longest match" is found, we search for multiple matches, incrementing \l__regex_match_count_int every time to record the number of matches. Build the NFA and match. At the end, store the result in the user's variable.

```

7543 \cs_new_protected:Npn \__regex_count:nnN #1#2#3
7544 {
7545     \group_begin:

```

```

7546     \__regex_disable_submatches:
7547     \int_zero:N \l__regex_match_count_int
7548     \__regex_multi_match:n { \int_incr:N \l__regex_match_count_int }
7549     #1
7550     \__regex_match:n {#2}
7551     \exp_args:NNNo
7552     \group_end:
7553     \int_set:Nn #3 { \int_use:N \l__regex_match_count_int }
7554 }

```

(End definition for `__regex_count:nnN`.)

45.7.3 Extracting submatches

`__regex_extract_once:nnN` Match once or multiple times. After each match (or after the only match), extract the submatches using `__regex_extract:.` At the end, store the sequence containing all the submatches into the user variable #3 after closing the group.

```

7555 \cs_new_protected:Npn \__regex_extract_once:nnN #1#2#3
7556 {
7557     \group_begin:
7558     \__regex_single_match:
7559     #1
7560     \__regex_match:n {#2}
7561     \__regex_extract:
7562     \__regex_query_set:n {#2}
7563     \__regex_group_end_extract_seq:N #3
7564 }
7565 \cs_new_protected:Npn \__regex_extract_all:nnN #1#2#3
7566 {
7567     \group_begin:
7568     \__regex_multi_match:n { \__regex_extract: }
7569     #1
7570     \__regex_match:n {#2}
7571     \__regex_query_set:n {#2}
7572     \__regex_group_end_extract_seq:N #3
7573 }

```

(End definition for `__regex_extract_once:nnN` and `__regex_extract_all:nnN`.)

`__regex_split:nnN` Splitting at submatches is a bit more tricky. For each match, extract all submatches, and replace the zeroth submatch by the part of the query between the start of the match attempt and the start of the zeroth submatch. This is inhibited if the delimiter matched an empty token list at the start of this match attempt. After the last match, store the last part of the token list, which ranges from the start of the match attempt to the end of the query. This step is inhibited if the last match was empty and at the very end: decrement `\l__regex_submatch_int`, which controls which matches will be used.

```

7574 \cs_new_protected:Npn \__regex_split:nnN #1#2#3
7575 {
7576     \group_begin:
7577     \__regex_multi_match:n
7578     {
7579         \if_int_compare:w
7580             \l__regex_start_pos_int < \l__regex_success_pos_int

```

```

7581         \__regex_extract:
7582         \__kernel_intarray_gset:Nnn \g__regex_submatch_prev_intarray
7583         { \l__regex_zeroth_submatch_int } { 0 }
7584         \__kernel_intarray_gset:Nnn \g__regex_submatch_end_intarray
7585         { \l__regex_zeroth_submatch_int }
7586         {
7587             \__kernel_intarray_item:Nn \g__regex_submatch_begin_intarray
7588             { \l__regex_zeroth_submatch_int }
7589         }
7590         \__kernel_intarray_gset:Nnn \g__regex_submatch_begin_intarray
7591         { \l__regex_zeroth_submatch_int }
7592         { \l__regex_start_pos_int }
7593     \fi:
7594 }
7595 #1
7596 \__regex_match:n {#2}
7597 \__regex_query_set:n {#2}
7598 \__kernel_intarray_gset:Nnn \g__regex_submatch_prev_intarray
7599 { \l__regex_submatch_int } { 0 }
7600 \__kernel_intarray_gset:Nnn \g__regex_submatch_end_intarray
7601 { \l__regex_submatch_int }
7602 { \l__regex_max_pos_int }
7603 \__kernel_intarray_gset:Nnn \g__regex_submatch_begin_intarray
7604 { \l__regex_submatch_int }
7605 { \l__regex_start_pos_int }
7606 \int_incr:N \l__regex_submatch_int
7607 \if_meaning:w \c_true_bool \l__regex_empty_success_bool
7608     \if_int_compare:w \l__regex_start_pos_int = \l__regex_max_pos_int
7609     \int_decr:N \l__regex_submatch_int
7610 \fi:
7611 \fi:
7612 \__regex_group_end_extract_seq:N #3
7613 }

```

(End definition for __regex_split:nnN.)

__regex_group_end_extract_seq:N The end-points of submatches are stored as entries of two arrays from \l__regex_min_submatch_int to \l__regex_submatch_int (exclusive). Extract the relevant ranges into \g__regex_internal_tl, separated by __regex_tmp:w {. We keep track in the two flags __regex_begin and __regex_end of the number of begin-group or end-group tokens added to make each of these items overall balanced. At this step, }{ is counted as being balanced (same number of begin-group and end-group tokens). This problem is caught by __regex_extract_check:w, explained later. After complaining about any begin-group or end-group tokens we had to add, we are ready to construct the user's sequence outside the group.

```

7614 \cs_new_protected:Npn \__regex_group_end_extract_seq:N #1
7615 {
7616     \flag_clear:n { __regex_begin }
7617     \flag_clear:n { __regex_end }
7618     \cs_set_eq:NN \__regex_tmp:w \scan_stop:
7619     \__kernel_tl_gset:Nx \g__regex_internal_tl
7620     {
7621         \int_step_function:nnN { \l__regex_min_submatch_int }
7622         { \l__regex_submatch_int - 1 } \__regex_extract_seq_aux:n

```

```

7623     \__regex_tmp:w
7624   }
7625   \int_set:Nn \l__regex_added_begin_int
7626   { \flag_height:n { __regex_begin } }
7627   \int_set:Nn \l__regex_added_end_int
7628   { \flag_height:n { __regex_end } }
7629   \tex_afterassignment:D \__regex_extract_check:w
7630   \__kernel_tl_gset:Nx \g__regex_internal_tl
7631   { \g__regex_internal_tl \if_false: { \fi: } }
7632   \int_compare:nNnT
7633   { \l__regex_added_begin_int + \l__regex_added_end_int } > 0
7634   {
7635     \msg_error:nnxxx { regex } { result-unbalanced }
7636     { splitting~or~extracting~submatches }
7637     { \int_use:N \l__regex_added_begin_int }
7638     { \int_use:N \l__regex_added_end_int }
7639   }
7640   \group_end:
7641   \cs_set_eq:NN \__regex_tmp:w \__regex_extract_map_loop:w
7642   \seq_set_from_function:NnN #1
7643   { \__regex_extract_map:N } \exp_not:n
7644 }

```

(End definition for __regex_group_end_extract_seq:N.)

__regex_extract_seq_aux:n The :n auxiliary builds one item of the sequence of submatches. First compute the brace balance of the submatch, then extract the submatch from the query, adding the appropriate braces and raising a flag if the submatch is not balanced.

```

7645 \cs_new:Npn \__regex_extract_seq_aux:n #1
7646 {
7647   \__regex_tmp:w { }
7648   \exp_after:wN \__regex_extract_seq_aux:ww
7649   \int_value:w \__regex_submatch_balance:n {#1} ; #1;
7650 }
7651 \cs_new:Npn \__regex_extract_seq_aux:ww #1; #2;
7652 {
7653   \if_int_compare:w #1 < \c_zero_int
7654   \prg_replicate:nn {-#1}
7655   {
7656     \flag_raise:n { __regex_begin }
7657     \exp_not:n { { \if_false: } \fi: }
7658   }
7659   \fi:
7660   \__regex_query_submatch:n {#2}
7661   \if_int_compare:w #1 > \c_zero_int
7662   \prg_replicate:nn {#1}
7663   {
7664     \flag_raise:n { __regex_end }
7665     \exp_not:n { \if_false: { \fi: } }
7666   }
7667   \fi:
7668 }

```

(End definition for __regex_extract_seq_aux:n and __regex_extract_seq_aux:ww.)

```

\__regex_extract_check:w
\__regex_extract_check:n
  \__regex_extract_check_loop:w
\__regex_extract_check_end:w

```

In `__regex_group_end_extract_seq:N` we had to expand `\g__regex_internal_tl` to turn `\if_false:` constructions into actual begin-group and end-group tokens. This is done with a `__kernel_tl_gset:Nx` assignment, and `__regex_extract_check:w` is run immediately after this assignment ends, thanks to the `\afterassignment` primitive. If all of the items were properly balanced (enough begin-group tokens before end-group tokens, so `}{` is not) then `__regex_extract_check:w` is called just before the closing brace of the `__kernel_tl_gset:Nx` (thanks to our sneaky `\if_false: { \fi: }` construction), and finds that there is nothing left to expand. If any of the items is unbalanced, the assignment gets ended early by an extra end-group token, and our check finds more tokens needing to be expanded in a new `__kernel_tl_gset:Nx` assignment. We need to add a begin-group and an end-group tokens to the unbalanced item, namely to the last item found so far, which we reach through a loop.

```

7669 \cs_new_protected:Npn \__regex_extract_check:w
7670 {
7671   \exp_after:wN \__regex_extract_check:n
7672   \exp_after:wN { \if_false: } \fi:
7673 }
7674 \cs_new_protected:Npn \__regex_extract_check:n #1
7675 {
7676   \tl_if_empty:nF {#1}
7677   {
7678     \int_incr:N \l__regex_added_begin_int
7679     \int_incr:N \l__regex_added_end_int
7680     \tex_afterassignment:D \__regex_extract_check:w
7681     \__kernel_tl_gset:Nx \g__regex_internal_tl
7682     {
7683       \exp_after:wN \__regex_extract_check_loop:w
7684       \g__regex_internal_tl
7685       \__regex_tmp:w \__regex_extract_check_end:w
7686       #1
7687     }
7688   }
7689 }
7690 \cs_new:Npn \__regex_extract_check_loop:w #1 \__regex_tmp:w #2
7691 {
7692   #2
7693   \exp_not:o {#1}
7694   \__regex_tmp:w { }
7695   \__regex_extract_check_loop:w \prg_do_nothing:
7696 }

```

Arguments of `__regex_extract_check_end:w` are: `#1` is the part of the item before the extra end-group token; `#2` is junk; `#3` is `\prg_do_nothing:` followed by the not-yet-expanded part of the item after the extra end-group token. In the replacement text, the first brace and the `\if_false: { \fi: }` construction are the added begin-group and end-group tokens (the latter being not-yet expanded, just like `#3`), while the closing brace after `\exp_not:o {#1}` replaces the extra end-group token that had ended the assignment early. In particular this means that the character code of that end-group token is lost.

```

7697 \cs_new:Npn \__regex_extract_check_end:w
7698   \exp_not:o #1#2 \__regex_extract_check_loop:w #3 \__regex_tmp:w
7699 {
7700   { \exp_not:o {#1} }
7701   #3

```

```

7702     \if_false: { \fi: }
7703     \__regex_tmp:w
7704 }

```

(End definition for __regex_extract_check:w and others.)

__regex_extract_map:N This receives a seq internal function and maps it over all items in \g__regex_internal_tl. This token list takes the form __regex_tmp:w {} <item₁> __regex_tmp:w {} <item₂> ... __regex_tmp:w, and the calling code has set __regex_tmp:w equal to __regex_extract_map_loop:w. The loop is otherwise pretty standard, with \prg_do_nothing: to avoid losing braces.

```

7705 \cs_new:Npn \__regex_extract_map:N #1
7706 {
7707     \exp_after:wN \__regex_extract_map_aux:NNn
7708     \exp_after:wN #1
7709     \g__regex_internal_tl \use_none:nnn
7710 }
7711 \cs_new:Npn \__regex_extract_map_aux:NNn #1#2#3
7712 { #3 #2 #1 \prg_do_nothing: }
7713 \cs_new:Npn \__regex_extract_map_loop:w #1#2 \__regex_tmp:w #3
7714 {
7715     \exp_after:wN #1 \exp_after:wN {#2}
7716     #3 \__regex_extract_map_loop:w #1 \prg_do_nothing:
7717 }

```

(End definition for __regex_extract_map:N, __regex_extract_map_aux:NNn, and __regex_extract_map_loop:w.)

__regex_extract: Our task here is to store the list of end-points of submatches, and store them in appropriate array entries, from \l__regex_zeroth_submatch_int upwards. First, we store in \g__regex_submatch_prev_intarray the position at which the match attempt started. We extract the rest from the comma list \l__regex_success_submatches_tl, which starts with entries to be stored in \g__regex_submatch_begin_intarray and continues with entries for \g__regex_submatch_end_intarray.

```

7718 \cs_new_protected:Npn \__regex_extract:
7719 {
7720     \if_meaning:w \c_true_bool \g__regex_success_bool
7721     \int_set_eq:NN \l__regex_zeroth_submatch_int \l__regex_submatch_int
7722     \prg_replicate:nn \l__regex_capturing_group_int
7723     {
7724         \__kernel_intarray_gset:Nnn \g__regex_submatch_prev_intarray
7725         { \l__regex_submatch_int } { 0 }
7726         \__kernel_intarray_gset:Nnn \g__regex_submatch_case_intarray
7727         { \l__regex_submatch_int } { 0 }
7728         \int_incr:N \l__regex_submatch_int
7729     }
7730     \__kernel_intarray_gset:Nnn \g__regex_submatch_prev_intarray
7731     { \l__regex_zeroth_submatch_int } { \l__regex_start_pos_int }
7732     \__kernel_intarray_gset:Nnn \g__regex_submatch_case_intarray
7733     { \l__regex_zeroth_submatch_int } { \g__regex_case_int }
7734     \int_zero:N \l__regex_internal_a_int
7735     \exp_after:wN \__regex_extract_aux:w \l__regex_success_submatches_tl
7736     \prg_break_point: \__regex_use_none_delimit_by_q_recursion_stop:w ,
7737     \q__regex_recursion_stop

```

```

7738     \fi:
7739   }
7740   \cs_new_protected:Npn \__regex_extract_aux:w #1 ,
7741   {
7742     \prg_break: #1 \prg_break_point:
7743     \if_int_compare:w \l__regex_internal_a_int < \l__regex_capturing_group_int
7744       \__kernel_intarray_gset:Nnn \g__regex_submatch_begin_intarray
7745       { \__regex_int_eval:w \l__regex_zeroth_submatch_int + \l__regex_internal_a_int } {#1}
7746     \else:
7747       \__kernel_intarray_gset:Nnn \g__regex_submatch_end_intarray
7748       { \__regex_int_eval:w \l__regex_zeroth_submatch_int + \l__regex_internal_a_int - \l__regex_capturing_group_int }
7749     \fi:
7750     \int_incr:N \l__regex_internal_a_int
7751     \__regex_extract_aux:w
7752   }

```

(End definition for __regex_extract: and __regex_extract_aux:w.)

45.7.4 Replacement

```

\__regex_replace_once:nnN
  \__regex_replace_once_aux:nnN

```

Build the NFA and the replacement functions, then find a single match. If the match failed, simply exit the group. Otherwise, we do the replacement. Extract submatches. Compute the brace balance corresponding to replacing this match by the replacement (this depends on submatches). Prepare the replaced token list: the replacement function produces the tokens from the start of the query to the start of the match and the replacement text for this match; we need to add the tokens from the end of the match to the end of the query. Finally, store the result in the user's variable after closing the group: this step involves an additional x-expansion, and checks that braces are balanced in the final result.

```

7753 \cs_new_protected:Npn \__regex_replace_once:nnN #1#2
7754 { \__regex_replace_once_aux:nnN {#1} { \__regex_replacement:n {#2} } }
7755 \cs_new_protected:Npn \__regex_replace_once_aux:nnN #1#2#3
7756 {
7757   \group_begin:
7758   \__regex_single_match:
7759   #1
7760   \exp_args:No \__regex_match:n {#3}
7761   \bool_if:NTF \g__regex_success_bool
7762   {
7763     \__regex_extract:
7764     \exp_args:No \__regex_query_set:n {#3}
7765     #2
7766     \int_set:Nn \l__regex_balance_int
7767     {
7768       \__regex_replacement_balance_one_match:n
7769       { \l__regex_zeroth_submatch_int }
7770     }
7771     \__kernel_tl_set:Nx \l__regex_internal_a_tl
7772     {
7773       \__regex_replacement_do_one_match:n
7774       { \l__regex_zeroth_submatch_int }
7775       \__regex_query_range:nn
7776       {
7777         \__kernel_intarray_item:Nn \g__regex_submatch_end_intarray

```



```

7778         { \l__regex_zeroth_submatch_int }
7779     }
7780     { \l__regex_max_pos_int }
7781 }
7782 \__regex_group_end_replace:N #3
7783 }
7784 { \group_end: }
7785 }

```

(End definition for __regex_replace_once:nnN and __regex_replace_once_aux:nnN.)

__regex_replace_all:nnN Match multiple times, and for every match, extract submatches and additionally store the position at which the match attempt started. The entries from \l__regex_min_submatch_int to \l__regex_submatch_int hold information about submatches of every match in order; each match corresponds to \l__regex_capturing_group_int consecutive entries. Compute the brace balance corresponding to doing all the replacements: this is the sum of brace balances for replacing each match. Join together the replacement texts for each match (including the part of the query before the match), and the end of the query.

```

7786 \cs_new_protected:Npn \__regex_replace_all:nnN #1#2
7787 { \__regex_replace_all_aux:nnN {#1} { \__regex_replacement:n {#2} } }
7788 \cs_new_protected:Npn \__regex_replace_all_aux:nnN #1#2#3
7789 {
7790     \group_begin:
7791     \__regex_multi_match:n { \__regex_extract: }
7792     #1
7793     \exp_args:No \__regex_match:n {#3}
7794     \exp_args:No \__regex_query_set:n {#3}
7795     #2
7796     \int_set:Nn \l__regex_balance_int
7797     {
7798         0
7799         \int_step_function:nnnN
7800         { \l__regex_min_submatch_int }
7801         \l__regex_capturing_group_int
7802         { \l__regex_submatch_int - 1 }
7803         \__regex_replacement_balance_one_match:n
7804     }
7805     \__kernel_tl_set:Nx \l__regex_internal_a_tl
7806     {
7807         \int_step_function:nnnN
7808         { \l__regex_min_submatch_int }
7809         \l__regex_capturing_group_int
7810         { \l__regex_submatch_int - 1 }
7811         \__regex_replacement_do_one_match:n
7812         \__regex_query_range:nn
7813         \l__regex_start_pos_int \l__regex_max_pos_int
7814     }
7815     \__regex_group_end_replace:N #3
7816 }

```

(End definition for __regex_replace_all:nnN.)

__regex_group_end_replace:N At this stage \l__regex_internal_a_tl (x-expands to the desired result). Guess from \l__regex_balance_int the number of braces to add before or after the result then
 __regex_group_end_replace_try:
 __regex_group_end_replace_check:w
 __regex_group_end_replace_check:n

try expanding. The simplest case is when `\l__regex_internal_a_tl` together with the braces we insert via `\prg_replicate:nn` give a balanced result, and the assignment ends at the `\if_false: { \fi: }` construction: then `__regex_group_end_replace_check:w` sees that there is no material left and we successfully found the result. The harder case is that expanding `\l__regex_internal_a_tl` may produce extra closing braces and end the assignment early. Then we grab the remaining code using; importantly, what follows has not yet been expanded so that `__regex_group_end_replace_check:n` grabs everything until the last brace in `__regex_group_end_replace_try:`, letting us try again with an extra surrounding pair of braces.

```

7817 \cs_new_protected:Npn \__regex_group_end_replace:N #1
7818 {
7819   \int_set:Nn \l__regex_added_begin_int
7820     { \int_max:nn { - \l__regex_balance_int } { 0 } }
7821   \int_set:Nn \l__regex_added_end_int
7822     { \int_max:nn { \l__regex_balance_int } { 0 } }
7823   \__regex_group_end_replace_try:
7824   \int_compare:nNnT { \l__regex_added_begin_int + \l__regex_added_end_int } > 0
7825     {
7826       \msg_error:nnxxx { regex } { result-unbalanced }
7827       { replacing } { \int_use:N \l__regex_added_begin_int }
7828       { \int_use:N \l__regex_added_end_int }
7829     }
7830   \group_end:
7831   \tl_set_eq:NN #1 \g__regex_internal_tl
7832 }
7833 \cs_new_protected:Npn \__regex_group_end_replace_try:
7834 {
7835   \tex_afterassignment:D \__regex_group_end_replace_check:w
7836   \__kernel_tl_gset:Nx \g__regex_internal_tl
7837   {
7838     \prg_replicate:nn { \l__regex_added_begin_int } { { \if_false: } \fi: }
7839     \l__regex_internal_a_tl
7840     \prg_replicate:nn { \l__regex_added_end_int } { { \if_false: { \fi: } }
7841       \if_false: { \fi: }
7842     }
7843   }
7844 \cs_new_protected:Npn \__regex_group_end_replace_check:w
7845 {
7846   \exp_after:wN \__regex_group_end_replace_check:n
7847   \exp_after:wN { \if_false: } \fi:
7848 }
7849 \cs_new_protected:Npn \__regex_group_end_replace_check:n #1
7850 {
7851   \tl_if_empty:nF {#1}
7852   {
7853     \int_incr:N \l__regex_added_begin_int
7854     \int_incr:N \l__regex_added_end_int
7855     \__regex_group_end_replace_try:
7856   }
7857 }

```

(End definition for `__regex_group_end_replace:N` and others.)

45.7.5 Peeking ahead

```

\l__regex_peek_true_tl True/false code arguments of \peek_regex:nTF or similar.
\l__regex_peek_false_tl
7858 \tl_new:N \l__regex_peek_true_tl
7859 \tl_new:N \l__regex_peek_false_tl

(End definition for \l__regex_peek_true_tl and \l__regex_peek_false_tl.)

\l__regex_replacement_tl When peeking in \peek_regex_replace_once:nnTF we need to store the replacement
text.
7860 \tl_new:N \l__regex_replacement_tl

(End definition for \l__regex_replacement_tl.)

\l__regex_input_tl Stores each token found as \__regex_input_item:n {<tokens>}, where the <tokens> o-
\__regex_input_item:n expand to the token found, as for \tl_analysis_map_inline:nn.
7861 \tl_new:N \l__regex_input_tl
7862 \cs_new_eq:NN \__regex_input_item:n ?

(End definition for \l__regex_input_tl and \__regex_input_item:n.)

\peek_regex:nTF The T and F functions just call the corresponding TF function. The four TF functions differ
\peek_regex:NTF along two axes: whether to remove the token or not, distinguished by using \__regex_-
\peek_regex_remove_once:nTF peek_end: or \__regex_peek_remove_end:n (the latter case needs an argument, as we
\peek_regex_remove_once:NTF will see), and whether the regex has to be compiled or is already in an N-type variable,
distinguished by calling \__regex_build_aux:Nn or \__regex_build_aux:NN. The first
argument of these functions is \c_false_bool to indicate that there should be no implicit
insertion of a wildcard at the start of the pattern: otherwise the code would keep looking
further into the input stream until matching the regex.

7863 \cs_new_protected:Npn \peek_regex:nTF #1
7864 {
7865   \__regex_peek:nnTF
7866   { \__regex_build_aux:Nn \c_false_bool {#1} }
7867   { \__regex_peek_end: }
7868 }
7869 \cs_new_protected:Npn \peek_regex:nT #1#2
7870 { \peek_regex:nTF {#1} {#2} { } }
7871 \cs_new_protected:Npn \peek_regex:nF #1 { \peek_regex:nTF {#1} { } }
7872 \cs_new_protected:Npn \peek_regex:NTF #1
7873 {
7874   \__regex_peek:nnTF
7875   { \__regex_build_aux:NN \c_false_bool #1 }
7876   { \__regex_peek_end: }
7877 }
7878 \cs_new_protected:Npn \peek_regex:NT #1#2
7879 { \peek_regex:NTF #1 {#2} { } }
7880 \cs_new_protected:Npn \peek_regex:NF #1 { \peek_regex:NTF {#1} { } }
7881 \cs_new_protected:Npn \peek_regex_remove_once:nTF #1
7882 {
7883   \__regex_peek:nnTF
7884   { \__regex_build_aux:Nn \c_false_bool {#1} }
7885   { \__regex_peek_remove_end:n {##1} }
7886 }
7887 \cs_new_protected:Npn \peek_regex_remove_once:nT #1#2

```

```

7888 { \peek_regex_remove_once:nTF {#1} {#2} { } }
7889 \cs_new_protected:Npn \peek_regex_remove_once:nF #1
7890 { \peek_regex_remove_once:nTF {#1} { } }
7891 \cs_new_protected:Npn \peek_regex_remove_once:NTF #1
7892 {
7893   \__regex_peek:nnTF
7894     { \__regex_build_aux:NN \c_false_bool #1 }
7895     { \__regex_peek_remove_end:n {##1} }
7896 }
7897 \cs_new_protected:Npn \peek_regex_remove_once:NT #1#2
7898 { \peek_regex_remove_once:NTF #1 {#2} { } }
7899 \cs_new_protected:Npn \peek_regex_remove_once:NF #1
7900 { \peek_regex_remove_once:NTF #1 { } }

```

(End definition for `\peek_regex:nTF` and others. These functions are documented on page 194.)

`__regex_peek:nnTF` Store the user's true/false codes (plus `\group_end:`) into two token lists. Then build the automaton with `#1`, without submatch tracking, and aiming for a single match. Then start matching by setting up a few variables like for any regex matching like `\regex_match:nnTF`, with the addition of `\l__regex_input_tl` that keeps track of the tokens seen, to reinsert them at the end. Instead of `\tl_analysis_map_inline:nn` on the input, we call `\peek_analysis_map_inline:n` to go through tokens in the input stream. Since `__regex_match_one_token:nnN` calls `__regex_maplike_break:` we need to catch that and break the `\peek_analysis_map_inline:n` loop instead.

```

7901 \cs_new_protected:Npn \__regex_peek:nnTF #1
7902 {
7903   \__regex_peek_aux:nnTF
7904   {
7905     \__regex_disable_submatches:
7906     #1
7907   }
7908 }
7909 \cs_new_protected:Npn \__regex_peek_aux:nnTF #1#2#3#4
7910 {
7911   \group_begin:
7912   \tl_set:Nn \l__regex_peek_true_tl { \group_end: #3 }
7913   \tl_set:Nn \l__regex_peek_false_tl { \group_end: #4 }
7914   \__regex_single_match:
7915   #1
7916   \__regex_match_init:
7917   \tl_build_clear:N \l__regex_input_tl
7918   \__regex_match_once_init:
7919   \peek_analysis_map_inline:n
7920   {
7921     \tl_build_put_right:Nn \l__regex_input_tl
7922     { \__regex_input_item:n {##1} }
7923     \__regex_match_one_token:nnN {##1} {##2} ##3
7924     \use_none:nnn
7925     \prg_break_point:Nn \__regex_maplike_break:
7926     { \peek_analysis_map_break:n {#2} }
7927   }
7928 }

```

(End definition for `__regex_peek:nnTF` and `__regex_peek_aux:nnTF`.)

`__regex_peek_end:` Once the regex matches (or permanently fails to match) we call `__regex_peek_end:`, or `__regex_peek_remove_end:n` with argument the last token seen. For `\peek_regex:nTF` we reinsert tokens seen by calling `__regex_peek_reinsert:N` regardless of the result of the match. For `\peek_regex_remove_once:nTF` we reinsert the tokens seen only if the match failed; otherwise we just reinsert the tokens `#1`, with one expansion. To be more precise, `#1` consists of tokens that o-expand and x-expand to the last token seen, for example it is `\exp_not:N <cs>` for a control sequence. This means that just doing `\exp_after:wN \l__regex_peek_true_tl #1` would be unsafe because the expansion of `<cs>` would be suppressed.

```

7929 \cs_new_protected:Npn \__regex_peek_end:
7930 {
7931   \bool_if:NTF \g__regex_success_bool
7932     { \__regex_peek_reinsert:N \l__regex_peek_true_tl }
7933     { \__regex_peek_reinsert:N \l__regex_peek_false_tl }
7934 }
7935 \cs_new_protected:Npn \__regex_peek_remove_end:n #1
7936 {
7937   \bool_if:NTF \g__regex_success_bool
7938     { \exp_args:NNo \use:nn \l__regex_peek_true_tl {#1} }
7939     { \__regex_peek_reinsert:N \l__regex_peek_false_tl }
7940 }

```

(End definition for `__regex_peek_end:` and `__regex_peek_remove_end:n`.)

`__regex_peek_reinsert:N` Insert the true/false code `#1`, followed by the tokens found, which were stored in `\l__regex_input_tl`. For this, loop through that token list using `__regex_reinsert_item:n`, which expands `#1` once to get a single token, and jumps over it to expand what follows, with suitable `\exp:w` and `\exp_end:.` We cannot just use `\use:e` on the whole token list because the result may be unbalanced, which would stop the primitive prematurely, or let it continue beyond where we would like.

```

7941 \cs_new_protected:Npn \__regex_peek_reinsert:N #1
7942 {
7943   \tl_build_end:N \l__regex_input_tl
7944   \cs_set_eq:NN \__regex_input_item:n \__regex_reinsert_item:n
7945   \exp_after:wN #1 \exp:w \l__regex_input_tl \exp_end:
7946 }
7947 \cs_new_protected:Npn \__regex_reinsert_item:n #1
7948 {
7949   \exp_after:wN \exp_after:wN
7950   \exp_after:wN \exp_end:
7951   \exp_after:wN \exp_after:wN
7952   #1
7953   \exp:w
7954 }

```

(End definition for `__regex_peek_reinsert:N` and `__regex_reinsert_item:n`.)

`\peek_regex_replace_once:nn` Similar to `\peek_regex:nTF` above.

```

7955 \cs_new_protected:Npn \peek_regex_replace_once:nnTF #1
7956 { \__regex_peek_replace:nnTF { \__regex_build_aux:Nn \c_false_bool {#1} } }
7957 \cs_new_protected:Npn \peek_regex_replace_once:nnT #1#2#3
7958 { \peek_regex_replace_once:nnTF {#1} {#2} {#3} { } }
7959 \cs_new_protected:Npn \peek_regex_replace_once:nnF #1#2

```

```

7960 { \peek_regex_replace_once:nnTF {#1} {#2} { } }
7961 \cs_new_protected:Npn \peek_regex_replace_once:nn #1#2
7962 { \peek_regex_replace_once:nnTF {#1} {#2} { } { } }
7963 \cs_new_protected:Npn \peek_regex_replace_once:NnTF #1
7964 { \__regex_peek_replace:nnTF { \__regex_build_aux:NN \c_false_bool #1 } }
7965 \cs_new_protected:Npn \peek_regex_replace_once:NnT #1#2#3
7966 { \peek_regex_replace_once:NnTF #1 {#2} {#3} { } }
7967 \cs_new_protected:Npn \peek_regex_replace_once:NnF #1#2
7968 { \peek_regex_replace_once:NnTF #1 {#2} { } }
7969 \cs_new_protected:Npn \peek_regex_replace_once:Nn #1#2
7970 { \peek_regex_replace_once:NnTF #1 {#2} { } { } }

```

(End definition for `\peek_regex_replace_once:nnTF` and `\peek_regex_replace_once:NnTF`. These functions are documented on page 195.)

`__regex_peek_replace:nnTF` Same as `__regex_peek:nnTF` (used for `\peek_regex:nTF` above), but without disabling submatches, and with a different end. The replacement text #2 is stored, to be analyzed later.

```

7971 \cs_new_protected:Npn \__regex_peek_replace:nnTF #1#2
7972 {
7973   \tl_set:Nn \l__regex_replacement_tl {#2}
7974   \__regex_peek_aux:nnTF {#1} { \__regex_peek_replace_end: }
7975 }

```

(End definition for `__regex_peek_replace:nnTF`.)

`__regex_peek_replace_end:` If the match failed `__regex_peek_reinsert:N` reinserts the tokens found. Otherwise, finish storing the submatch information using `__regex_extract:`, and store the input into `\toks`. Redefine a few auxiliaries to change slightly their expansion behaviour as explained below. Analyse the replacement text with `__regex_replacement:n`, which as usual defines `__regex_replacement_do_one_match:n` to insert the tokens from the start of the match attempt to the beginning of the match, followed by the replacement text. The `\use:x` expands for instance the trailing `__regex_query_range:nn` down to a sequence of `__regex_reinsert_item:n {⟨tokens⟩}` where `⟨tokens⟩` o-expand to a single token that we want to insert. After x-expansion, `\use:x` does `\use:n`, so we have `\exp_after:wN \l__regex_peek_true_tl \exp:w ... \exp_end:`. This is set up such as to obtain `\l__regex_peek_true_tl` followed by the replaced tokens (possibly unbalanced) in the input stream.

```

7976 \cs_new_protected:Npn \__regex_peek_replace_end:
7977 {
7978   \bool_if:NTF \g__regex_success_bool
7979   {
7980     \__regex_extract:
7981     \__regex_query_set_from_input_tl:
7982     \cs_set_eq:NN \__regex_replacement_put:n \__regex_peek_replacement_put:n
7983     \cs_set_eq:NN \__regex_replacement_put_submatch_aux:n
7984     \__regex_peek_replacement_put_submatch_aux:n
7985     \cs_set_eq:NN \__regex_input_item:n \__regex_reinsert_item:n
7986     \cs_set_eq:NN \__regex_replacement_exp_not:N \__regex_peek_replacement_token:n
7987     \cs_set_eq:NN \__regex_replacement_exp_not:V \__regex_peek_replacement_var:N
7988     \exp_args:No \__regex_replacement:n { \l__regex_replacement_tl }
7989     \use:x
7990     {

```

```

7991         \exp_not:n { \exp_after:wN \l__regex_peek_true_tl \exp:w }
7992         \__regex_replacement_do_one_match:n
7993         { \l__regex_zeroth_submatch_int }
7994         \__regex_query_range:nn
7995         {
7996             \__kernel_intarray_item:Nn \g__regex_submatch_end_intarray
7997             { \l__regex_zeroth_submatch_int }
7998         }
7999         { \l__regex_max_pos_int }
8000         \exp_end:
8001     }
8002 }
8003 { \__regex_peek_reinsert:N \l__regex_peek_false_tl }
8004 }

```

(End definition for __regex_peek_replace_end:.)

__regex_query_set_from_input_tl:
__regex_query_set_item:n

The input was stored into \l__regex_input_tl as successive items __regex_input_item:n {⟨tokens⟩}. Store that in successive \toks. It's not clear whether the empty entries before and after are both useful.

```

8005 \cs_new_protected:Npn \__regex_query_set_from_input_tl:
8006 {
8007     \tl_build_end:N \l__regex_input_tl
8008     \int_zero:N \l__regex_curr_pos_int
8009     \cs_set_eq:NN \__regex_input_item:n \__regex_query_set_item:n
8010     \__regex_query_set_item:n { }
8011     \l__regex_input_tl
8012     \__regex_query_set_item:n { }
8013     \int_set_eq:NN \l__regex_max_pos_int \l__regex_curr_pos_int
8014 }
8015 \cs_new_protected:Npn \__regex_query_set_item:n #1
8016 {
8017     \int_incr:N \l__regex_curr_pos_int
8018     \__regex_toks_set:Nn \l__regex_curr_pos_int { \__regex_input_item:n {#1} }
8019 }

```

(End definition for __regex_query_set_from_input_tl: and __regex_query_set_item:n.)

__regex_peek_replacement_put:n

While building the replacement function __regex_replacement_do_one_match:n, we often want to put simple material, given as #1, whose x-expansion o-expands to a single token. Normally we can just add the token to \l__regex_build_tl, but for \peek_regex_replace_once:nnTF we eventually want to do some strange expansion that is basically using \exp_after:wN to jump through numerous tokens (we cannot use x-expansion like for \regex_replace_once:nnNTF because it is ok for the result to be unbalanced since we insert it in the input stream rather than storing it. When within a csname we don't do any such shenanigan because \cs:w ... \cs_end: does all the expansion we need.

```

8020 \cs_new_protected:Npn \__regex_peek_replacement_put:n #1
8021 {
8022     \if_case:w \l__regex_replacement_csnames_int
8023     \tl_build_put_right:Nn \l__regex_build_tl
8024     { \exp_not:N \__regex_reinsert_item:n {#1} }
8025     \else:
8026     \tl_build_put_right:Nn \l__regex_build_tl {#1}

```

```

8027     \fi:
8028 }

```

(End definition for `_regex_peek_replacement_put:n`.)

`_regex_peek_replacement_token:n` When hit with `\exp:w`, `_regex_peek_replacement_token:n {⟨token⟩}` stops `\exp_end:` and does `\exp_after:wN ⟨token⟩ \exp:w` to continue expansion after it.

```

8029 \cs_new_protected:Npn \_regex_peek_replacement_token:n #1
8030 { \exp_after:wN \exp_end: \exp_after:wN #1 \exp:w }

```

(End definition for `_regex_peek_replacement_token:n`.)

`_regex_peek_replacement_put_submatch_aux:n` While analyzing the replacement we also have to insert submatches found in the query. Since query items `_regex_input_item:n {⟨tokens⟩}` expand correctly only when surrounded by `\exp:w ... \exp_end:`, and since these expansion controls are not there within csnames (because `\cs:w ... \cs_end:` make them unnecessary in most cases), we have to put `\exp:w` and `\exp_end:` by hand here.

```

8031 \cs_new_protected:Npn \_regex_peek_replacement_put_submatch_aux:n #1
8032 {
8033     \if_case:w \l__regex_replacement_csnames_int
8034     \tl_build_put_right:Nn \l__regex_build_tl
8035     { \_regex_query_submatch:n { \int_eval:n { #1 + ##1 } } }
8036     \else:
8037     \tl_build_put_right:Nn \l__regex_build_tl
8038     { \exp:w \_regex_query_submatch:n { \int_eval:n { #1 + ##1 } } \exp_end: }
8039     \fi:
8040 }

```

(End definition for `_regex_peek_replacement_put_submatch_aux:n`.)

`_regex_peek_replacement_var:N` This is used for `\u` outside csnames. It makes sure to continue expansion with `\exp:w` before expanding the variable `#1` and stopping the `\exp:w` that precedes.

```

8041 \cs_new_protected:Npn \_regex_peek_replacement_var:N #1
8042 {
8043     \exp_after:wN \exp_last_unbraced:NV
8044     \exp_after:wN \exp_end:
8045     \exp_after:wN #1
8046     \exp:w
8047 }

```

(End definition for `_regex_peek_replacement_var:N`.)

45.8 Messages

Messages for the preparsing phase.

```

8048 \use:x
8049 {
8050     \msg_new:nnn { regex } { trailing-backslash }
8051     { Trailing~'\iow_char:N\}'~in-regex-or-replacement. }
8052     \msg_new:nnn { regex } { x-missing-rbrace }
8053     {
8054         Missing~brace~'\iow_char:N\}'~in-regex~
8055         '...~\iow_char:N\~x~\iow_char:N\{...##1'.

```



```

8056     }
8057     \msg_new:nnn { regex } { x-overflow }
8058     {
8059         Character~code~##1~too~large~in~
8060         \iow_char:N\ \x\iow_char:N\{##2\iow_char:N\}~regex.
8061     }
8062 }

```

Invalid quantifier.

```

8063 \msg_new:nnnn { regex } { invalid-quantifier }
8064 { Braced~quantifier~'~#1'~may~not~be~followed~by~'~#2'. }
8065 {
8066     The~character~'~#2'~is~invalid~in~the~braced~quantifier~'~#1'.~
8067     The~only~valid~quantifiers~are~'*',~'?',~'+',~'{<int>}',~
8068     '{<min>}',~and~'{<min>,<max>}',~optionally~followed~by~'?''.
8069 }

```

Messages for missing or extra closing brackets and parentheses, with some fancy singular/plural handling for the case of parentheses.

```

8070 \msg_new:nnnn { regex } { missing-rbrack }
8071 { Missing~right~bracket~inserted~in~regular~expression. }
8072 {
8073     LaTeX~was~given~a~regular~expression~where~a~character~class~
8074     was~started~with~'[',~but~the~matching~']'~is~missing.
8075 }
8076 \msg_new:nnnn { regex } { missing-rparen }
8077 {
8078     Missing~right~
8079     \int_compare:nTF { #1 = 1 } { parenthesis } { parentheses } ~
8080     inserted~in~regular~expression.
8081 }
8082 {
8083     LaTeX~was~given~a~regular~expression~with~\int_eval:n {#1} ~
8084     more~left~parentheses~than~right~parentheses.
8085 }
8086 \msg_new:nnnn { regex } { extra-rparen }
8087 { Extra~right~parenthesis~ignored~in~regular~expression. }
8088 {
8089     LaTeX~came~across~a~closing~parenthesis~when~no~submatch~group~
8090     was~open.~The~parenthesis~will~be~ignored.
8091 }

```

Some escaped alphanumerics are not allowed everywhere.

```

8092 \msg_new:nnnn { regex } { bad-escape }
8093 {
8094     Invalid~escape~'\iow_char:N\ \#1'~
8095     \__regex_if_in_cs:TF { within~a~control~sequence. }
8096     {
8097         \__regex_if_in_class:TF
8098         { in~a~character~class. }
8099         { following~a~category~test. }
8100     }
8101 }
8102 {
8103     The~escape~sequence~'\iow_char:N\ \#1'~may~not~appear~

```

```

8104     \_regex_if_in_cs:TF
8105     {
8106         within-a-control-sequence-test-introduced-by~
8107         '\iow_char:N\\c\iow_char:N{\'.
8108     }
8109     {
8110         \_regex_if_in_class:TF
8111         { within-a-character-class~ }
8112         { following-a-category-test-such-as~'\iow_char:N\\cL'~ }
8113         because-it-does-not-match-exactly-one-character.
8114     }
8115 }

```

Range errors.

```

8116 \msg_new:nnnn { regex } { range-missing-end }
8117 { Invalid-end-point-for-range~'#1-#2'~in-character-class. }
8118 {
8119     The-end-point~'#2'~of-the-range~'#1-#2'~may-not-serve-as-an~
8120     end-point-for-a-range:-alphanumeric-characters-should-not-be~
8121     escaped,~and-non-alphanumeric-characters-should-be-escaped.
8122 }
8123 \msg_new:nnnn { regex } { range-backwards }
8124 { Range~'[#1-#2]'~out-of-order~in-character-class. }
8125 {
8126     In-ranges-of-characters~'[x-y]'~appearing-in-character-classes,~
8127     the-first-character-code-must-not-be-larger-than-the-second.~
8128     Here,~'#1'~has-character-code~\int_eval:n {'#1},~while~
8129     '#2'~has-character-code~\int_eval:n {'#2}.
8130 }

```

Errors related to \c and \u.

```

8131 \msg_new:nnnn { regex } { c-bad-mode }
8132 { Invalid-nested~'\iow_char:N\\c'~escape-in-regular-expression. }
8133 {
8134     The~'\iow_char:N\\c'~escape-cannot-be-used-within~
8135     a-control-sequence-test~'\iow_char:N\\c{...}'~
8136     nor-another-category-test.~
8137     To-combine-several-category-tests,~use~'\iow_char:N\\c[...]'~.
8138 }
8139 \msg_new:nnnn { regex } { c-C-invalid }
8140 { '\iow_char:N\\cC'~should-be-followed-by~'.'~or~'(',~not~'#1'. }
8141 {
8142     The~'\iow_char:N\\cC'~construction-restricts-the-next-item-to-be-a~
8143     control-sequence-or-the-next-group-to-be-made-of-control-sequences.~
8144     It-only-makes-sense-to-follow-it-by~'.'~or-by-a-group.
8145 }
8146 \msg_new:nnnn { regex } { cu-lbrace }
8147 { Left-braces-must-be-escaped-in~'\iow_char:N\\#1{...}'~. }
8148 {
8149     Constructions-such-as~'\iow_char:N\\#1{...~\iow_char:N{...}'~are~
8150     not-allowed-and-should-be-replaced-by~
8151     '\iow_char:N\\#1{...~\token_to_str:N{...}'~.
8152 }
8153 \msg_new:nnnn { regex } { c-lparen-in-class }
8154 { Catcode-test-cannot-apply-to-group-in-character-class }

```

```

8155 {
8156   Construction~such~as~'\iow_char:N\cL(abc)'\~are~not~allowed~inside~a~
8157   class~'[...]\~because~classes~do~not~match~multiple~characters~at~once.
8158 }
8159 \msg_new:nnnn { regex } { c-missing-rbrace }
8160 { Missing~right~brace~inserted~for~'\iow_char:N\c'\~escape. }
8161 {
8162   LaTeX~was~given~a~regular~expression~where~a~
8163   '\iow_char:N\c\iow_char:N\{...\~construction~was~not~ended~
8164   with~a~closing~brace~'\iow_char:N\}' .
8165 }
8166 \msg_new:nnnn { regex } { c-missing-rbrack }
8167 { Missing~right~bracket~inserted~for~'\iow_char:N\c'\~escape. }
8168 {
8169   A~construction~'\iow_char:N\c[...\~appears~in~a~
8170   regular~expression,~but~the~closing~'\~is~not~present.
8171 }
8172 \msg_new:nnnn { regex } { c-missing-category }
8173 { Invalid~character~'#1'\~following~'\iow_char:N\c'\~escape. }
8174 {
8175   In~regular~expressions,~the~'\iow_char:N\c'\~escape~sequence~
8176   may~only~be~followed~by~a~left~brace,~a~left~bracket,~or~a~
8177   capital~letter~representing~a~character~category,~namely~
8178   one~of~'ABCDELMOPSTU'.
8179 }
8180 \msg_new:nnnn { regex } { c-trailing }
8181 { Trailing~category~code~escape~'\iow_char:N\c'... }
8182 {
8183   A~regular~expression~ends~with~'\iow_char:N\c'\~followed~
8184   by~a~letter.~It~will~be~ignored.
8185 }
8186 \msg_new:nnnn { regex } { u-missing-lbrace }
8187 { Missing~left~brace~following~'\iow_char:N\u'\~escape. }
8188 {
8189   The~'\iow_char:N\u'\~escape~sequence~must~be~followed~by~
8190   a~brace~group~with~the~name~of~the~variable~to~use.
8191 }
8192 \msg_new:nnnn { regex } { u-missing-rbrace }
8193 { Missing~right~brace~inserted~for~'\iow_char:N\u'\~escape. }
8194 {
8195   LaTeX~
8196   \str_if_eq:eeTF { } {#2}
8197   { reached~the~end~of~the~string~ }
8198   { encountered~an~escaped~alphanumeric~character '\iow_char:N\#2'\~ }
8199   when~parsing~the~argument~of~an~
8200   '\iow_char:N\u\iow_char:N\{...\}'~escape.
8201 }

```

Errors when encountering the POSIX syntax [:...:].

```

8202 \msg_new:nnnn { regex } { posix-unsupported }
8203 { POSIX~collating~element~'[#1 ~ #1]'\~not~supported. }
8204 {
8205   The~'[.foo.]'\~and~'[=bar=]'\~syntaxes~have~a~special~meaning~
8206   in~POSIX~regular~expressions.~This~is~not~supported~by~LaTeX.~
8207   Maybe~you~forgot~to~escape~a~left~bracket~in~a~character~class?

```

```

8208 }
8209 \msg_new:nnnn { regex } { posix-unknown }
8210 { POSIX~class~'[:#1:]'~unknown. }
8211 {
8212   '[:#1:]'~is~not~among~the~known~POSIX~classes~
8213   '[:alnum:]',~'[:alpha:]',~'[:ascii:]',~'[:blank:]',~
8214   '[:cntrl:]',~'[:digit:]',~'[:graph:]',~'[:lower:]',~
8215   '[:print:]',~'[:punct:]',~'[:space:]',~'[:upper:]',~
8216   '[:word:]',~and~'[:xdigit:]'.
8217 }
8218 \msg_new:nnnn { regex } { posix-missing-close }
8219 { Missing~closing~':'~for~POSIX~class. }
8220 { The~POSIX~syntax~'#1'~must~be~followed~by~':'',~not~'#2'. }

```

In various cases, the result of a `l3regex` operation can leave us with an unbalanced token list, which we must re-balance by adding begin-group or end-group character tokens.

```

8221 \msg_new:nnnn { regex } { result-unbalanced }
8222 { Missing~brace~inserted~when~#1. }
8223 {
8224   LaTeX~was~asked~to~do~some~regular~expression~operation,~
8225   and~the~resulting~token~list~would~not~have~the~same~number~
8226   of~begin~group~and~end~group~tokens.~Braces~were~inserted:~
8227   #2~left,~#3~right.
8228 }

```

Error message for unknown options.

```

8229 \msg_new:nnnn { regex } { unknown-option }
8230 { Unknown~option~'#1'~for~regular~expressions. }
8231 {
8232   The~only~available~option~is~'case-insensitive',~toggled~by~
8233   '(?i)''~and~'(?-i)'.
8234 }
8235 \msg_new:nnnn { regex } { special-group-unknown }
8236 { Unknown~special~group~'#1...''~in~a~regular~expression. }
8237 {
8238   The~only~valid~constructions~starting~with~'(?''~are~
8239   '(:~...~)',~'(?|~...~)',~'(?i)',~and~'(?-i)'.
8240 }

```

Errors in the replacement text.

```

8241 \msg_new:nnnn { regex } { replacement-c }
8242 { Misused~'\iow_char:N\c'~command~in~a~replacement~text. }
8243 {
8244   In~a~replacement~text,~the~'\iow_char:N\c'~escape~sequence~
8245   can~be~followed~by~one~of~the~letters~'ABCDELMPSTU'~
8246   or~a~brace~group,~not~by~'#1'.
8247 }
8248 \msg_new:nnnn { regex } { replacement-u }
8249 { Misused~'\iow_char:N\u'~command~in~a~replacement~text. }
8250 {
8251   In~a~replacement~text,~the~'\iow_char:N\u'~escape~sequence~
8252   must~be~followed~by~a~brace~group~holding~the~name~of~the~
8253   variable~to~use.
8254 }

```

```

8255 \msg_new:nnnn { regex } { replacement-g }
8256 {
8257     Missing~brace~for~the~'\iow_char:N\g'~construction~
8258     in~a~replacement~text.
8259 }
8260 {
8261     In~the~replacement~text~for~a~regular~expression~search,~
8262     submatches~are~represented~either~as~'\iow_char:N \g{dd..d}',~
8263     or~'\d',~where~'d'~are~single~digits.~Here,~a~brace~is~missing.
8264 }
8265 \msg_new:nnnn { regex } { replacement-catcode-end }
8266 {
8267     Missing~character~for~the~'\iow_char:N\c<category><character>'~
8268     construction~in~a~replacement~text.
8269 }
8270 {
8271     In~a~replacement~text,~the~'\iow_char:N\c'~escape~sequence~
8272     can~be~followed~by~one~of~the~letters~'ABCDELMOPTU'~representing~
8273     the~character~category.~Then,~a~character~must~follow.~LaTeX~
8274     reached~the~end~of~the~replacement~when~looking~for~that.
8275 }
8276 \msg_new:nnnn { regex } { replacement-catcode-escaped }
8277 {
8278     Escaped~letter~or~digit~after~category~code~in~replacement~text.
8279 }
8280 {
8281     In~a~replacement~text,~the~'\iow_char:N\c'~escape~sequence~
8282     can~be~followed~by~one~of~the~letters~'ABCDELMOPTU'~representing~
8283     the~character~category.~Then,~a~character~must~follow,~not~
8284     '\iow_char:N\#2'.
8285 }
8286 \msg_new:nnnn { regex } { replacement-catcode-in-cs }
8287 {
8288     Category~code~'\iow_char:N\c#1#3'~ignored~inside~
8289     '\iow_char:N\c\{...\}'~in~a~replacement~text.
8290 }
8291 {
8292     In~a~replacement~text,~the~category~codes~of~the~argument~of~
8293     '\iow_char:N\c\{...\}'~are~ignored~when~building~the~control~
8294     sequence~name.
8295 }
8296 \msg_new:nnnn { regex } { replacement-null-space }
8297 { TeX~cannot~build~a~space~token~with~character~code~0. }
8298 {
8299     You~asked~for~a~character~token~with~category~space,~
8300     and~character~code~0,~for~instance~through~
8301     '\iow_char:N\cS\iow_char:N\x00'.~
8302     This~specific~case~is~impossible~and~will~be~replaced~
8303     by~a~normal~space.
8304 }
8305 \msg_new:nnnn { regex } { replacement-missing-rbrace }
8306 { Missing~right~brace~inserted~in~replacement~text. }
8307 {
8308     There~ \int_compare:nTF { #1 = 1 } { was } { were } ~ #1~

```

```

8309     missing~right~\int_compare:nTF { #1 = 1 } { brace } { braces } .
8310 }
8311 \msg_new:nnnn { regex } { replacement-missing-rparen }
8312 { Missing~right~parenthesis~inserted~in~replacement~text. }
8313 {
8314     There~ \int_compare:nTF { #1 = 1 } { was } { were } ~ #1~
8315     missing~right~
8316     \int_compare:nTF { #1 = 1 } { parenthesis } { parentheses } .
8317 }
8318 \msg_new:nnn { regex } { submatch-too-big }
8319 { Submatch~#1~used~but~regex~only~has~#2~group(s) }

Some escaped alphanumerics are not allowed everywhere.

8320 \msg_new:nnnn { regex } { backwards-quantifier }
8321 { Quantifier~"#{1,#2}"~is~backwards. }
8322 { The~values~given~in~a~quantifier~must~be~in~order. }

Used in user commands, and when showing a regex.

8323 \msg_new:nnnn { regex } { case-odd }
8324 { #1~with~odd~number~of~items }
8325 {
8326     There~must~be~a~#2~part~for~each~regex:~
8327     found~odd~number~of~items~(#{3})~in~\
8328     \iow_indent:n {#4}
8329 }
8330 \msg_new:nnn { regex } { show }
8331 {
8332     >~Compiled~regex~
8333     \tl_if_empty:nTF {#1} { variable~ #2 } { {#1} } :
8334     #3
8335 }
8336 \prop_gput:Nnn \g_msg_module_name_prop { regex } { LaTeX3 }
8337 \prop_gput:Nnn \g_msg_module_type_prop { regex } { }

```

`__regex_msg_repeated:nnN` This is not technically a message, but seems related enough to go there. The arguments are: #1 is the minimum number of repetitions; #2 is the number of allowed extra repetitions (−1 for infinite number), and #3 tells us about laziness.

```

8338 \cs_new:Npn \__regex_msg_repeated:nnN #1#2#3
8339 {
8340     \str_if_eq:eeF { #1 #2 } { 1 0 }
8341     {
8342         , ~ repeated ~
8343         \int_case:nnF {#2}
8344         {
8345             { -1 } { #1~or~more~times,~\bool_if:NTF #3 { lazy } { greedy } }
8346             { 0 } { #1~times }
8347         }
8348         {
8349             between~#1~and~\int_eval:n {#1+#2}~times,~
8350             \bool_if:NTF #3 { lazy } { greedy }
8351         }
8352     }
8353 }

```

(End definition for `__regex_msg_repeated:nnN`.)

45.9 Code for tracing

There is a more extensive implementation of tracing in the l3trial package l3trace. Function names are a bit different but could be merged.

```

\__regex_trace_push:nnN
\__regex_trace_pop:nnN
  \__regex_trace:nnx

```

Here #1 is the module name (regex) and #2 is typically 1. If the module's current tracing level is less than #2 show nothing, otherwise write #3 to the terminal.

```

8354 \cs_new_protected:Npn \__regex_trace_push:nnN #1#2#3
8355   { \__regex_trace:nnx {#1} {#2} { entering~ \token_to_str:N #3 } }
8356 \cs_new_protected:Npn \__regex_trace_pop:nnN #1#2#3
8357   { \__regex_trace:nnx {#1} {#2} { leaving~ \token_to_str:N #3 } }
8358 \cs_new_protected:Npn \__regex_trace:nnx #1#2#3
8359   {
8360     \int_compare:nNnF
8361       { \int_use:c { g__regex_trace_#1_int } } < {#2}
8362       { \iow_term:x { Trace:~#3 } }
8363   }

```

(End definition for __regex_trace_push:nnN, __regex_trace_pop:nnN, and __regex_trace:nnx.)

```

\g__regex_trace_regex_int

```

No tracing when that is zero.

```

8364 \int_new:N \g__regex_trace_regex_int

```

(End definition for \g__regex_trace_regex_int.)

```

\__regex_trace_states:n

```

This function lists the contents of all states of the NFA, stored in \toks from 0 to \l__regex_max_state_int (excluded).

```

8365 \cs_new_protected:Npn \__regex_trace_states:n #1
8366   {
8367     \int_step_inline:nnn
8368       \l__regex_min_state_int
8369       { \l__regex_max_state_int - 1 }
8370     {
8371       \__regex_trace:nnx { regex } {#1}
8372       { \iow_char:N \\toks ##1 = { \__regex_toks_use:w ##1 } }
8373     }
8374   }

```

(End definition for __regex_trace_states:n.)

```

8375 </package>

```

Chapter 46

l3prg implementation

The following test files are used for this code: *m3prg001.lvt, m3prg002.lvt, m3prg003.lvt.*

```
8376 \*package\
```

46.1 Primitive conditionals

`\if_bool:N` Those two primitive TeX conditionals are synonyms. `\if_bool:N` is defined in `l3basics`, as it's needed earlier to define quark test functions.

```
8377 \cs_new_eq:NN \if_predicate:w \tex_ifodd:D
```

(End definition for `\if_bool:N` and `\if_predicate:w`. These functions are documented on page 70.)

46.2 Defining a set of conditional functions

These are all defined in `l3basics`, as they are needed “early”. This is just a reminder!

(End definition for `\prg_set_conditional:Npnn` and others. These functions are documented on page 62.)

46.3 The boolean data type

```
8378 @@=bool\
```

Boolean variables have to be initiated when they are created. Other than that there is not much to say here.

```
8379 \cs_new_protected:Npn \bool_new:N #1 { \cs_new_eq:NN #1 \c_false_bool }
8380 \cs_generate_variant:Nn \bool_new:N { c }
```

(End definition for `\bool_new:N`. This function is documented on page 65.)

`\bool_const:Nn` A merger between `\tl_const:Nn` and `\bool_set:Nn`.

```
\bool_const:cn
8381 \cs_new_protected:Npn \bool_const:Nn #1#2
8382 {
8383   \__kernel_chk_if_free_cs:N #1
8384   \tex_global:D \tex_chardef:D #1 = \bool_if_p:n {#2}
8385 }
8386 \cs_generate_variant:Nn \bool_const:Nn { c }
```


(End definition for `\bool_const:Nn`. This function is documented on page 65.)

`\bool_set_true:N` Setting is already pretty easy. When `check-declarations` is active, the definitions are
`\bool_set_true:c` patched to make sure the boolean exists. This is needed because booleans are not based
`\bool_gset_true:N` on token lists nor on T_EX registers.
`\bool_gset_true:c`
`\bool_set_false:N`
`\bool_set_false:c`
`\bool_gset_false:N`
`\bool_gset_false:c`

```

8387 \cs_new_protected:Npn \bool_set_true:N #1
8388   { \cs_set_eq:NN #1 \c_true_bool }
8389 \cs_new_protected:Npn \bool_set_false:N #1
8390   { \cs_set_eq:NN #1 \c_false_bool }
8391 \cs_new_protected:Npn \bool_gset_true:N #1
8392   { \cs_gset_eq:NN #1 \c_true_bool }
8393 \cs_new_protected:Npn \bool_gset_false:N #1
8394   { \cs_gset_eq:NN #1 \c_false_bool }
8395 \cs_generate_variant:Nn \bool_set_true:N { c }
8396 \cs_generate_variant:Nn \bool_set_false:N { c }
8397 \cs_generate_variant:Nn \bool_gset_true:N { c }
8398 \cs_generate_variant:Nn \bool_gset_false:N { c }

```

(End definition for `\bool_set_true:N` and others. These functions are documented on page 65.)

`\bool_set_eq:NN` The usual copy code. While it would be cleaner semantically to copy the `\cs_set_eq:NN`
`\bool_set_eq:cN` family of functions, we copy `\tl_set_eq:NN` because that has the correct checking code.
`\bool_set_eq:Nc`
`\bool_set_eq:cc`
`\bool_gset_eq:NN`
`\bool_gset_eq:cN`
`\bool_gset_eq:Nc`
`\bool_gset_eq:cc`

```

8399 \cs_new_eq:NN \bool_set_eq:NN \tl_set_eq:NN
8400 \cs_new_eq:NN \bool_gset_eq:NN \tl_gset_eq:NN
8401 \cs_generate_variant:Nn \bool_set_eq:NN { Nc, cN, cc }
8402 \cs_generate_variant:Nn \bool_gset_eq:NN { Nc, cN, cc }

```

(End definition for `\bool_set_eq:NN` and `\bool_gset_eq:NN`. These functions are documented on page 65.)

`\bool_set:Nn` This function evaluates a boolean expression and assigns the first argument the meaning
`\bool_set:cn` `\c_true_bool` or `\c_false_bool`. Again, we include some checking code. It is important
`\bool_gset:Nn` to evaluate the expression before applying the `\chardef` primitive, because that primitive
`\bool_gset:cn` sets the left-hand side to `\scan_stop:` before looking for the right-hand side.

```

8403 \cs_new_protected:Npn \bool_set:Nn #1#2
8404   {
8405     \exp_last_unbraced:NNNf
8406       \tex_chardef:D #1 = { \bool_if_p:n {#2} }
8407   }
8408 \cs_new_protected:Npn \bool_gset:Nn #1#2
8409   {
8410     \exp_last_unbraced:NNNNf
8411       \tex_global:D \tex_chardef:D #1 = { \bool_if_p:n {#2} }
8412   }
8413 \cs_generate_variant:Nn \bool_set:Nn { c }
8414 \cs_generate_variant:Nn \bool_gset:Nn { c }

```

(End definition for `\bool_set:Nn` and `\bool_gset:Nn`. These functions are documented on page 65.)

46.4 Internal auxiliaries

`\q__bool_recursion_tail` Internal recursion quarks.

`\q__bool_recursion_stop`

```

8415 \quark_new:N \q__bool_recursion_tail
8416 \quark_new:N \q__bool_recursion_stop

```

(End definition for `\q__bool_recursion_tail` and `\q__bool_recursion_stop`.)

`__bool_use_i_delimit_by_q_recursion_stop:nw` Functions to gobble up to a quark.

```

8417 \cs_new:Npn \__bool_use_i_delimit_by_q_recursion_stop:nw
8418   #1 #2 \q__bool_recursion_stop {#1}

```

(End definition for `__bool_use_i_delimit_by_q_recursion_stop:nw`.)

`__bool_if_recursion_tail_stop_do:nn` Functions to query recursion quarks.

```

8419 \__kernel_quark_new_test:N \__bool_if_recursion_tail_stop_do:nn

```

(End definition for `__bool_if_recursion_tail_stop_do:nn`.)

`\bool_if_p:N` Straight forward here. We could optimize here if we wanted to as the boolean can just be input directly.

`\bool_if_p:c`

`\bool_if:N \overline{TF}`

`\bool_if:c \overline{TF}`

```

8420 \prg_new_conditional:Npnn \bool_if:N #1 { p , T , F , TF }
8421 {
8422   \if_bool:N #1
8423     \prg_return_true:
8424   \else:
8425     \prg_return_false:
8426   \fi:
8427 }
8428 \prg_generate_conditional_variant:Nnn \bool_if:N { c } { p , T , F , TF }

```

(End definition for `\bool_if:N \overline{TF}` . This function is documented on page 65.)

`\bool_to_str:N` Expands to true or false with category code letter.

`\bool_to_str:c`

`\bool_to_str:n`

```

8429 \cs_new:Npn \bool_to_str:N #1 { \bool_if:N $\overline{TF}$  #1 { true } { false } }
8430 \cs_generate_variant:Nn \bool_to_str:N { c }
8431 \cs_new:Npn \bool_to_str:n #1 { \bool_if:n $\overline{TF}$  {#1} { true } { false } }

```

(End definition for `\bool_to_str:N` and `\bool_to_str:n`. These functions are documented on page 65.)

`\bool_show:n` Show the truth value of the boolean.

`\bool_log:n`

```

8432 \cs_new_protected:Npn \bool_show:n
8433 { \msg_show_eval:Nn \bool_to_str:n }
8434 \cs_new_protected:Npn \bool_log:n
8435 { \msg_log_eval:Nn \bool_to_str:n }

```

(End definition for `\bool_show:n` and `\bool_log:n`. These functions are documented on page 66.)

`\bool_show:N` Show the truth value of the boolean, as true or false.

`\bool_show:c`

`\bool_log:N`

`\bool_log:c`

`__bool_show:NN`

```

8436 \cs_new_protected:Npn \bool_show:N { \__bool_show:NN \tl_show:n }
8437 \cs_generate_variant:Nn \bool_show:N { c }
8438 \cs_new_protected:Npn \bool_log:N { \__bool_show:NN \tl_log:n }
8439 \cs_generate_variant:Nn \bool_log:N { c }
8440 \cs_new_protected:Npn \__bool_show:NN #1#2
8441 {

```

```

8442     \__kernel_chk_defined:NT #2
8443     {
8444         \token_case_meaning:NnF #2
8445         {
8446             \c_true_bool { \exp_args:Nx #1 { \token_to_str:N #2 = true } }
8447             \c_false_bool { \exp_args:Nx #1 { \token_to_str:N #2 = false } }
8448         }
8449         {
8450             \msg_error:nxxxx { kernel } { bad-type }
8451             { \token_to_str:N #2 } { \token_to_meaning:N #2 } { bool }
8452         }
8453     }
8454 }

```

(End definition for `\bool_show:N`, `\bool_log:N`, and `__bool_show:NN`. These functions are documented on page 66.)

`\l_tmpa_bool` A few booleans just if you need them.

```

\l_tmpb_bool 8455 \bool_new:N \l_tmpa_bool
\g_tmpa_bool 8456 \bool_new:N \l_tmpb_bool
\g_tmpb_bool 8457 \bool_new:N \g_tmpa_bool
8458 \bool_new:N \g_tmpb_bool

```

(End definition for `\l_tmpa_bool` and others. These variables are documented on page 66.)

`\bool_if_exist_p:N` Copies of the `cs` functions defined in `l3basics`.

```

\bool_if_exist_p:c 8459 \prg_new_eq_conditional:NNn \bool_if_exist:N \cs_if_exist:N
\bool_if_exist:N\TF 8460 { TF , T , F , p }
\bool_if_exist:c\TF 8461 \prg_new_eq_conditional:NNn \bool_if_exist:c \cs_if_exist:c
8462 { TF , T , F , p }

```

(End definition for `\bool_if_exist:N\TF`. This function is documented on page 66.)

46.5 Boolean expressions

`\bool_if_p:n` Evaluating the truth value of a list of predicates is done using an input syntax somewhat similar to the one found in other programming languages with `(` and `)` for grouping, `!` for logical “Not”, `&&` for logical “And” and `||` for logical “Or”. However, they perform eager evaluation. We shall use the terms Not, And, Or, Open and Close for these operations.

Any expression is terminated by a Close operation. Evaluation happens from left to right in the following manner using a `GetNext` function:

- If an Open is seen, start evaluating a new expression using the `Eval` function and call `GetNext` again.
- If a Not is seen, remove the `!` and call a `GetNext` function with the logic reversed.
- If none of the above, reinsert the token found (this is supposed to be a predicate function) in front of an `Eval` function, which evaluates it to the boolean value $\langle true \rangle$ or $\langle false \rangle$.

The `Eval` function then contains a post-processing operation which grabs the instruction following the predicate. This is either And, Or or Close. In each case the truth value is used to determine where to go next. The following situations can arise:

⟨true⟩And Current truth value is true, logical And seen, continue with GetNext to examine truth value of next boolean (sub-)expression.

⟨false⟩And Current truth value is false, logical And seen, stop using the values of predicates within this sub-expression until the next Close. Then return *⟨false⟩*.

⟨true⟩Or Current truth value is true, logical Or seen, stop using the values of predicates within this sub-expression until the nearest Close. Then return *⟨true⟩*.

⟨false⟩Or Current truth value is false, logical Or seen, continue with GetNext to examine truth value of next boolean (sub-)expression.

⟨true⟩Close Current truth value is true, Close seen, return *⟨true⟩*.

⟨false⟩Close Current truth value is false, Close seen, return *⟨false⟩*.

```

8463 \prg_new_conditional:Npnn \bool_if:n #1 { T , F , TF }
8464 {
8465   \if_predicate:w \bool_if_p:n {#1}
8466   \prg_return_true:
8467   \else:
8468   \prg_return_false:
8469   \fi:
8470 }
```

(End definition for `\bool_if:nTF`. This function is documented on page 67.)

```

\bool_if_p:n
__bool_if_p:n
__bool_if_p_aux:w
```

To speed up the case of a single predicate, `f-expand` and check whether the result is one token (possibly surrounded by spaces), which must be `\c_true_bool` or `\c_false_bool`. We use a version of `\tl_if_single:nTF` optimized for speed since we know that an empty `#1` is an error. The auxiliary `__bool_if_p_aux:w` removes the trailing parenthesis and gets rid of any space. For the general case, first issue a `\group_align_safe_begin:` as we are using `&&` as syntax shorthand for the And operation and we need to hide it for `TeX`. This group is closed after `__bool_get_next:NN` returns `\c_true_bool` or `\c_false_bool`. That function requires the trailing parenthesis to know where the expression ends.

```

8471 \cs_new:Npn \bool_if_p:n { \exp_args:Nf __bool_if_p:n }
8472 \cs_new:Npn __bool_if_p:n #1
8473 {
8474   \tl_if_empty:oT { \use_none:nn #1 . } { __bool_if_p_aux:w }
8475   \group_align_safe_begin:
8476   \exp_after:wN
8477   \group_align_safe_end:
8478   \exp:w \exp_end_continue_f:w % (
8479   __bool_get_next:NN \use_i:nnnn #1 )
8480 }
8481 \cs_new:Npn __bool_if_p_aux:w #1 \use_i:nnnn #2#3 {#2}
```

(End definition for `\bool_if_p:n`, `__bool_if_p:n`, and `__bool_if_p_aux:w`. This function is documented on page 67.)

```
__bool_get_next:NN
```

The GetNext operation. Its first argument is `\use_i:nnnn`, `\use_ii:nnnn`, `\use_iii:nnnn`, or `\use_iv:nnnn` (we call these “states”). In the first state, this function eventually expand to the truth value `\c_true_bool` or `\c_false_bool` of the expression

which follows until the next unmatched closing parenthesis. For instance “`_bool_get_next:NN \use_i:nnnn \c_true_bool && \c_true_bool)`” (including the closing parenthesis) expands to `\c_true_bool`. In the second state (after a `!`) the logic is reversed. We call these two states “normal” and the next two “skipping”. In the third state (after `\c_true_bool||`) it always returns `\c_true_bool`. In the fourth state (after `\c_false_bool&&`) it always returns `\c_false_bool` and also stops when encountering `||`, not only parentheses. This code itself is a switch: if what follows is neither `!` nor `(`, we assume it is a predicate.

```

8482 \cs_new:Npn \_bool\_get\_next:NN #1#2
8483 {
8484   \use:c
8485   {
8486     \_bool\_
8487     \if\_meaning:w !#2 ! \else: \if\_meaning:w (#2 ( \else: p \fi: \fi:
8488     :Nw
8489   }
8490   #1 #2
8491 }

```

(End definition for `_bool_get_next:NN`.)

`_bool_!:Nw` The Not operation reverses the logic: it discards the `!` token and calls the `GetNext` operation with the appropriate first argument. Namely the first and second states are interchanged, but after `\c_true_bool||` or `\c_false_bool&&` the `!` is ignored.

```

8492 \cs_new:cpn { \_bool\_!:Nw } #1#2
8493 {
8494   \exp\_after:wN \_bool\_get\_next:NN
8495   #1 \use\_ii:nnnn \use\_i:nnnn \use\_iii:nnnn \use\_iv:nnnn
8496 }

```

(End definition for `_bool_!:Nw`.)

`_bool_(:Nw` The Open operation starts a sub-expression after discarding the open parenthesis. This is done by calling `GetNext` (which eventually discards the corresponding closing parenthesis), with a post-processing step which looks for And, Or or Close after the group.

```

8497 \cs_new:cpn { \_bool\_(:Nw } #1#2
8498 {
8499   \exp\_after:wN \_bool\_choose:NNN \exp\_after:wN #1
8500   \int\_value:w \_bool\_get\_next:NN \use\_i:nnnn
8501 }

```

(End definition for `_bool_(:Nw`.)

`_bool_p:Nw` If what follows `GetNext` is neither `!` nor `(`, evaluate the predicate using the primitive `\int_value:w`. The canonical `true` and `false` values have numerical values 1 and 0 respectively. Look for And, Or or Close afterwards.

```

8502 \cs_new:cpn { \_bool\_p:Nw } #1
8503 { \exp\_after:wN \_bool\_choose:NNN \exp\_after:wN #1 \int\_value:w }

```

(End definition for `_bool_p:Nw`.)

_bool_choose:NNN The arguments are #1: a function such as \use_i:nnnn, #2: 0 or 1 encoding the current truth value, #3: the next operation, And, Or or Close. We distinguish three cases according to a combination of #1 and #2. Case 2 is when #1 is \use_iii:nnnn (state 3), namely after \c_true_bool ||. Case 1 is when #1 is \use_i:nnnn and #2 is true or when #1 is \use_ii:nnnn and #2 is false, for instance for !\c_false_bool. Case 0 includes the same with true/false interchanged and the case where #1 is \use_iv:nnnn namely after \c_false_bool &&.

bool|_0: When seeing) the current subexpression is done, leave the appropriate boolean.
bool|_1: When seeing & in case 0 go into state 4, equivalent to having seen \c_false_bool &&.
bool|_2: In case 1, namely when the argument is true and we are in a normal state continue in the normal state 1. In case 2, namely when skipping alternatives in an Or, continue in the same state. When seeing | in case 0, continue in a normal state; in particular stop skipping for \c_false_bool && because that binds more tightly than ||. In the other two cases start skipping for \c_true_bool ||.

```

8504 \cs_new:Npn \_bool_choose:NNN #1#2#3
8505 {
8506   \use:c
8507   {
8508     __bool_ \token_to_str:N #3 _
8509     #1 #2 { \if_meaning:w 0 #2 1 \else: 0 \fi: } 2 0 :
8510   }
8511 }
8512 \cs_new:cpn { __bool_)_0: } { \c_false_bool }
8513 \cs_new:cpn { __bool_)_1: } { \c_true_bool }
8514 \cs_new:cpn { __bool_)_2: } { \c_true_bool }
8515 \cs_new:cpn { __bool_&_0: } & { \_bool_get_next:NN \use_iv:nnnn }
8516 \cs_new:cpn { __bool_&_1: } & { \_bool_get_next:NN \use_i:nnnn }
8517 \cs_new:cpn { __bool_&_2: } & { \_bool_get_next:NN \use_iii:nnnn }
8518 \cs_new:cpn { __bool_|_0: } | { \_bool_get_next:NN \use_i:nnnn }
8519 \cs_new:cpn { __bool_|_1: } | { \_bool_get_next:NN \use_iii:nnnn }
8520 \cs_new:cpn { __bool_|_2: } | { \_bool_get_next:NN \use_iii:nnnn }

```

(End definition for _bool_choose:NNN and others.)

\bool_lazy_all_p:n Go through the list of expressions, stopping whenever an expression is false. If the end is reached without finding any false expression, then the result is true.

\bool_lazy_all:nTF

```

\_bool_lazy_all:n
8521 \cs_new:Npn \bool_lazy_all_p:n #1
8522 { \_bool_lazy_all:n #1 \q__bool_recursion_tail \q__bool_recursion_stop }
8523 \prg_new_conditional:Npnn \bool_lazy_all:n #1 { T , F , TF }
8524 {
8525   \if_predicate:w \bool_lazy_all_p:n {#1}
8526   \prg_return_true:
8527   \else:
8528   \prg_return_false:
8529   \fi:
8530 }
8531 \cs_new:Npn \_bool_lazy_all:n #1
8532 {
8533   \_bool_if_recursion_tail_stop_do:nn {#1} { \c_true_bool }
8534   \bool_if:nF {#1}
8535   { \_bool_use_i_delimit_by_q_recursion_stop:nw { \c_false_bool } }
8536   \_bool_lazy_all:n
8537 }

```

(End definition for `\bool_lazy_all:nTF` and `__bool_lazy_all:n`. This function is documented on page 68.)

`\bool_lazy_and_p:nn` Only evaluate the second expression if the first is `true`. Note that #2 must be removed as an argument, not just by skipping to the `\else:` branch of the conditional since #2 may contain unbalanced `TEX` conditionals.

`\bool_lazy_and:nnTF`

```

8538 \prg_new_conditional:Npnn \bool_lazy_and:nn #1#2 { p , T , F , TF }
8539 {
8540   \if_predicate:w
8541     \bool_if:nTF {#1} { \bool_if_p:n {#2} } { \c_false_bool }
8542     \prg_return_true:
8543   \else:
8544     \prg_return_false:
8545   \fi:
8546 }
```

(End definition for `\bool_lazy_and:nnTF`. This function is documented on page 68.)

`\bool_lazy_any_p:n` Go through the list of expressions, stopping whenever an expression is `true`. If the end is reached without finding any `true` expression, then the result is `false`.

`\bool_lazy_any:nTF`

`__bool_lazy_any:n`

```

8547 \cs_new:Npn \bool_lazy_any_p:n #1
8548 { \__bool_lazy_any:n #1 \q_bool_recursion_tail \q_bool_recursion_stop }
8549 \prg_new_conditional:Npnn \bool_lazy_any:n #1 { T , F , TF }
8550 {
8551   \if_predicate:w \bool_lazy_any_p:n {#1}
8552     \prg_return_true:
8553   \else:
8554     \prg_return_false:
8555   \fi:
8556 }
8557 \cs_new:Npn \__bool_lazy_any:n #1
8558 {
8559   \__bool_if_recursion_tail_stop_do:nn {#1} { \c_false_bool }
8560   \bool_if:nT {#1}
8561     { \__bool_use_i_delimit_by_q_recursion_stop:nw { \c_true_bool } }
8562   \__bool_lazy_any:n
8563 }
```

(End definition for `\bool_lazy_any:nTF` and `__bool_lazy_any:n`. This function is documented on page 68.)

`\bool_lazy_or_p:nn` Only evaluate the second expression if the first is `false`.

`\bool_lazy_or:nnTF`

```

8564 \prg_new_conditional:Npnn \bool_lazy_or:nn #1#2 { p , T , F , TF }
8565 {
8566   \if_predicate:w
8567     \bool_if:nTF {#1} { \c_true_bool } { \bool_if_p:n {#2} }
8568     \prg_return_true:
8569   \else:
8570     \prg_return_false:
8571   \fi:
8572 }
```

(End definition for `\bool_lazy_or:nnTF`. This function is documented on page 68.)

`\bool_not_p:n` The Not variant just reverses the outcome of `\bool_if_p:n`. Can be optimized but this is nice and simple and according to the implementation plan. Not even particularly useful to have it when the infix notation is easier to use.

```
8573 \cs_new:Npn \bool_not_p:n #1 { \bool_if_p:n { ! ( #1 ) } }
```

(End definition for `\bool_not_p:n`. This function is documented on page 68.)

`\bool_xor_p:nn` Exclusive or. If the boolean expressions have same truth value, return `false`, otherwise
`\bool_xor:nnTF` return `true`.

```
8574 \prg_new_conditional:Npnn \bool_xor:nn #1#2 { p , T , F , TF }
8575 {
8576   \bool_if:nT {#1} \reverse_if:N
8577   \if_predicate:w \bool_if_p:n {#2}
8578   \prg_return_true:
8579   \else:
8580   \prg_return_false:
8581   \fi:
8582 }
```

(End definition for `\bool_xor:nnTF`. This function is documented on page 68.)

46.6 Logical loops

`\bool_while_do:Nn` A while loop where the boolean is tested before executing the statement. The “while”
`\bool_while_do:cn` version executes the code as long as the boolean is true; the “until” version executes the
`\bool_until_do:Nn` code as long as the boolean is false.
`\bool_until_do:cn`

```
8583 \cs_new:Npn \bool_while_do:Nn #1#2
8584 { \bool_if:NT #1 { #2 \bool_while_do:Nn #1 {#2} } }
8585 \cs_new:Npn \bool_until_do:Nn #1#2
8586 { \bool_if:NF #1 { #2 \bool_until_do:Nn #1 {#2} } }
8587 \cs_generate_variant:Nn \bool_while_do:Nn { c }
8588 \cs_generate_variant:Nn \bool_until_do:Nn { c }
```

(End definition for `\bool_while_do:Nn` and `\bool_until_do:Nn`. These functions are documented on page 69.)

`\bool_do_while:Nn` A do-while loop where the body is performed at least once and the boolean is tested
`\bool_do_while:cn` after executing the body. Otherwise identical to the above functions.

```
8589 \cs_new:Npn \bool_do_while:Nn #1#2
8590 { #2 \bool_if:NT #1 { \bool_do_while:Nn #1 {#2} } }
8591 \cs_new:Npn \bool_do_until:Nn #1#2
8592 { #2 \bool_if:NF #1 { \bool_do_until:Nn #1 {#2} } }
8593 \cs_generate_variant:Nn \bool_do_while:Nn { c }
8594 \cs_generate_variant:Nn \bool_do_until:Nn { c }
```

(End definition for `\bool_do_while:Nn` and `\bool_do_until:Nn`. These functions are documented on page 69.)

`\bool_while_do:nn` Loop functions with the test either before or after the first body expansion.

```
\bool_do_while:nn
\bool_until_do:nn
\bool_do_until:nn
8595 \cs_new:Npn \bool_while_do:nn #1#2
8596 {
8597   \bool_if:nT {#1}
8598   {
```



```

8599         #2
8600         \bool_while_do:nn {#1} {#2}
8601     }
8602 }
8603 \cs_new:Npn \bool_do_while:nn #1#2
8604 {
8605     #2
8606     \bool_if:nT {#1} { \bool_do_while:nn {#1} {#2} }
8607 }
8608 \cs_new:Npn \bool_until_do:nn #1#2
8609 {
8610     \bool_if:nF {#1}
8611     {
8612         #2
8613         \bool_until_do:nn {#1} {#2}
8614     }
8615 }
8616 \cs_new:Npn \bool_do_until:nn #1#2
8617 {
8618     #2
8619     \bool_if:nF {#1} { \bool_do_until:nn {#1} {#2} }
8620 }

```

(End definition for `\bool_while_do:nn` and others. These functions are documented on page 69.)

46.7 Producing multiple copies

```
8621 (@@=prg)
```

```
\prg_replicate:nn
```

This function uses a cascading csname technique by David Kastrup (who else :-)

```
\__prg_replicate:N
```

The idea is to make the input 25 result in first adding five, and then 20 copies of the code to be replicated. The technique uses cascading csnames which means that we start building several csnames so we end up with a list of functions to be called in reverse order. This is important here (and other places) because it means that we can for instance make the function that inserts five copies of something to also hand down ten to the next function in line. This is exactly what happens here: in the example with 25 then the next function is the one that inserts two copies but it sees the ten copies handed down by the previous function. In order to avoid the last function to insert say, 100 copies of the original argument just to gobble them again we define separate functions to be inserted first. These functions also close the expansion of `\exp:w`, which ensures that `\prg_replicate:nn` only requires two steps of expansion.

This function has one flaw though: Since it constantly passes down ten copies of its previous argument it severely affects the main memory once you start demanding hundreds of thousands of copies. Now I don't think this is a real limitation for any ordinary use, and if necessary, it is possible to write `\prg_replicate:nn {1000} { \prg_replicate:nn {1000} {<code>} }`. An alternative approach is to create a string of m's with `\exp:w` which can be done with just four macros but that method has its own problems since it can exhaust the string pool. Also, it is considerably slower than what we use here so the few extra csnames are well spent I would say.

```
8622 \cs_new:Npn \prg_replicate:nn #1
```

```
8623 {
```

```
8624     \exp:w
```

```
\__prg_replicate_first_8:n
```

```
\__prg_replicate_first_9:n
```

```

8625     \exp_after:wN \__prg_replicate_first:N
8626     \int_value:w \int_eval:n {#1}
8627     \cs_end:
8628 }
8629 \cs_new:Npn \__prg_replicate:N #1
8630 { \cs:w \__prg_replicate_#1 :n \__prg_replicate:N }
8631 \cs_new:Npn \__prg_replicate_first:N #1
8632 { \cs:w \__prg_replicate_first_#1 :n \__prg_replicate:N }

```

Then comes all the functions that do the hard work of inserting all the copies. The first function takes :n as a parameter.

```

8633 \cs_new:Npn \__prg_replicate_ :n #1 { \cs_end: }
8634 \cs_new:cpn { \__prg_replicate_0:n } #1
8635 { \cs_end: {#1#1#1#1#1#1#1#1#1#1} }
8636 \cs_new:cpn { \__prg_replicate_1:n } #1
8637 { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1 }
8638 \cs_new:cpn { \__prg_replicate_2:n } #1
8639 { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1#1 }
8640 \cs_new:cpn { \__prg_replicate_3:n } #1
8641 { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1#1#1 }
8642 \cs_new:cpn { \__prg_replicate_4:n } #1
8643 { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1#1#1#1 }
8644 \cs_new:cpn { \__prg_replicate_5:n } #1
8645 { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1#1#1#1#1 }
8646 \cs_new:cpn { \__prg_replicate_6:n } #1
8647 { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1#1#1#1#1#1 }
8648 \cs_new:cpn { \__prg_replicate_7:n } #1
8649 { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1#1#1#1#1#1#1 }
8650 \cs_new:cpn { \__prg_replicate_8:n } #1
8651 { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1#1#1#1#1#1#1#1 }
8652 \cs_new:cpn { \__prg_replicate_9:n } #1
8653 { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1#1#1#1#1#1#1#1#1 }

```

Users shouldn't ask for something to be replicated once or even not at all but...

```

8654 \cs_new:cpn { \__prg_replicate_first_-:n } #1
8655 {
8656     \exp_end:
8657     \msg_expandable_error:nn { prg } { negative-replication }
8658 }
8659 \cs_new:cpn { \__prg_replicate_first_0:n } #1 { \exp_end: }
8660 \cs_new:cpn { \__prg_replicate_first_1:n } #1 { \exp_end: #1 }
8661 \cs_new:cpn { \__prg_replicate_first_2:n } #1 { \exp_end: #1#1 }
8662 \cs_new:cpn { \__prg_replicate_first_3:n } #1 { \exp_end: #1#1#1 }
8663 \cs_new:cpn { \__prg_replicate_first_4:n } #1 { \exp_end: #1#1#1#1 }
8664 \cs_new:cpn { \__prg_replicate_first_5:n } #1 { \exp_end: #1#1#1#1#1 }
8665 \cs_new:cpn { \__prg_replicate_first_6:n } #1 { \exp_end: #1#1#1#1#1#1 }
8666 \cs_new:cpn { \__prg_replicate_first_7:n } #1 { \exp_end: #1#1#1#1#1#1#1 }
8667 \cs_new:cpn { \__prg_replicate_first_8:n } #1 { \exp_end: #1#1#1#1#1#1#1#1 }
8668 \cs_new:cpn { \__prg_replicate_first_9:n } #1
8669 { \exp_end: #1#1#1#1#1#1#1#1#1#1 }

```

(End definition for \prg_replicate:nn and others. This function is documented on page 69.)

46.8 Detecting T_EX's mode

`\mode_if_vertical_p:` For testing vertical mode. Strikes me here on the bus with David, that as long as we are just talking about returning true and false states, we can just use the primitive conditionals for this and gobbling the `\exp_end:` in the input stream. However this requires knowledge of the implementation so we keep things nice and clean and use the return statements.

```
8670 \prg_new_conditional:Npnn \mode_if_vertical: { p , T , F , TF }
8671 { \if_mode_vertical: \prg_return_true: \else: \prg_return_false: \fi: }
```

(End definition for `\mode_if_vertical:TF`. This function is documented on page 70.)

`\mode_if_horizontal_p:` For testing horizontal mode.

```
\mode_if_horizontal:TF
8672 \prg_new_conditional:Npnn \mode_if_horizontal: { p , T , F , TF }
8673 { \if_mode_horizontal: \prg_return_true: \else: \prg_return_false: \fi: }
```

(End definition for `\mode_if_horizontal:TF`. This function is documented on page 70.)

`\mode_if_inner_p:` For testing inner mode.

```
\mode_if_inner:TF
8674 \prg_new_conditional:Npnn \mode_if_inner: { p , T , F , TF }
8675 { \if_mode_inner: \prg_return_true: \else: \prg_return_false: \fi: }
```

(End definition for `\mode_if_inner:TF`. This function is documented on page 70.)

`\mode_if_math_p:` For testing math mode. At the beginning of an alignment cell, this should be used only inside a non-expandable function.

```
\mode_if_math:TF
8676 \prg_new_conditional:Npnn \mode_if_math: { p , T , F , TF }
8677 { \if_mode_math: \prg_return_true: \else: \prg_return_false: \fi: }
```

(End definition for `\mode_if_math:TF`. This function is documented on page 70.)

46.9 Internal programming functions

`\group_align_safe_begin:` T_EX's alignment structures present many problems. As Knuth says himself in *T_EX: The Program*: “It’s sort of a miracle whenever `\halign` or `\valign` work, [...]” One problem relates to commands that internally issue a `\cr` but also peek ahead for the next character for use in, say, an optional argument. If the next token happens to be a `&` with category code 4 we get some sort of weird error message because the underlying `\futurelet` stores the token at the end of the alignment template. This could be a `&_4` giving a message like `! Misplaced \cr.` or even worse: it could be the `\endtemplate` token causing even more trouble! To solve this we have to open a special group so that T_EX still thinks it’s on safe ground but at the same time we don’t want to introduce any brace group that may find its way to the output. The following functions help with this by using behaviour documented only in Appendix D of *The T_EXbook*... In short evaluating ‘{ and ‘} as numbers will not change the counter T_EX uses to keep track of its state in an alignment, whereas gobbling a brace using `\if_false:` will affect T_EX’s state without producing any real group. We place the `\if_false: { \fi:` part at that place so that the successive expansions of `\group_align_safe_begin/end:` are always brace balanced.

```
8678 \group_begin:
8679 \tex_catcode:D ‘^@ = 2 \exp_stop_f:
8680 \cs_new:Npn \group_align_safe_begin:
8681 { \exp:w \if_false: { \fi: ‘^@ \exp_stop_f: }
```

```

8682 \group_end:
8683 \cs_new:Npn \group_align_safe_end:
8684   { \if_int_compare:w '{ = \c_zero_int } \fi: }

```

(End definition for \group_align_safe_begin: and \group_align_safe_end:. These functions are documented on page 71.)

\g__kernel_prg_map_int A nesting counter for mapping.

```

8685 \int_new:N \g__kernel_prg_map_int

```

(End definition for \g__kernel_prg_map_int.)

\prg_break_point:Nn These are defined in l3basics, as they are needed “early”. This is just a reminder that is the case!
\prg_map_break:Nn

(End definition for \prg_break_point:Nn and \prg_map_break:Nn. These functions are documented on page 70.)

\prg_break_point: Also done in l3basics.

\prg_break:
\prg_break:n

(End definition for \prg_break_point:, \prg_break:, and \prg_break:n. These functions are documented on page 71.)

```

8686 \</package>

```

Chapter 47

l3sys implementation

```
8687 <@@=sys>
```

47.1 Kernel code

```
8688 <*package>
8689 <*tex>
```

47.1.1 Detecting the engine

`__sys_const:nn` Set the T, F, TF, p forms of #1 to be constants equal to the result of evaluating the boolean expression #2.

```
8690 \cs_new_protected:Npn \__sys_const:nn #1#2
8691 {
8692   \bool_if:nTF {#2}
8693   {
8694     \cs_new_eq:cN { #1 :T } \use:n
8695     \cs_new_eq:cN { #1 :F } \use_none:n
8696     \cs_new_eq:cN { #1 :TF } \use_i:nn
8697     \cs_new_eq:cN { #1 _p: } \c_true_bool
8698   }
8699   {
8700     \cs_new_eq:cN { #1 :T } \use_none:n
8701     \cs_new_eq:cN { #1 :F } \use:n
8702     \cs_new_eq:cN { #1 :TF } \use_ii:nn
8703     \cs_new_eq:cN { #1 _p: } \c_false_bool
8704   }
8705 }
```

(End definition for __sys_const:nn.)

`\sys_if_engine luatex_p:` Set up the engine tests on the basis exactly one test should be true. Mainly a case of looking for the appropriate marker primitive.

```
\sys_if_engine luatex:TF
\sys_if_engine pdftex_p:
\sys_if_engine pdftex:TF
  \sys_if_engine ptex_p:
  \sys_if_engine ptex:TF
\sys_if_engine uptex_p:
\sys_if_engine uptex:TF
\sys_if_engine xetex_p:
\sys_if_engine xetex:TF
  \c_sys_engine_str
```

```
8706 \str_const:Nx \c_sys_engine_str
8707 {
8708   \cs_if_exist:NT \tex luatexversion:D { luatex }
8709   \cs_if_exist:NT \tex pdftexversion:D { pdftex }
8710   \cs_if_exist:NT \tex kanjiskip:D
8711 }
```

```

8712     \cs_if_exist:NTF \tex_enablecjktoken:D
8713         { uptex }
8714         { ptex }
8715     }
8716     \cs_if_exist:NT \tex_XeTeXversion:D { xetex }
8717 }
8718 \tl_map_inline:nn { { luatex } { pdftex } { ptex } { uptex } { xetex } }
8719 {
8720     \__sys_const:nn { sys_if_engine_ #1 }
8721     { \str_if_eq_p:Vn \c_sys_engine_str {#1} }
8722 }

```

(End definition for `\sys_if_engine luatex:TF` and others. These functions are documented on page 73.)

`\c_sys_engine_exec_str`
`\c_sys_engine_format_str`

Take the functions defined above, and set up the engine and format names. `\c_sys_engine_exec_str` differs from `\c_sys_engine_str` as it is the *actual* engine name, not a “filtered” version. It differs for `ptex` and `uptex`, which have a leading `e`, and for `luatex`, because L^AT_EX uses the LuaH^BT_EX engine.

`\c_sys_engine_format_str` is quite similar to `\c_sys_engine_str`, except that it differentiates `pdflatex` from `latex` (which is `pdfTEX` in DVI mode). This differentiation, however, is reliable only if the user doesn’t change `\tex_pdfoutput:D` before loading this code.

```

8723 \group_begin:
8724     \cs_set_eq:NN \lua_now:e \tex_directlua:D
8725     \str_const:Nx \c_sys_engine_exec_str
8726     {
8727         \sys_if_engine_pdftex:T { pdf }
8728         \sys_if_engine_xetex:T { xe }
8729         \sys_if_engine_ptex:T { ep }
8730         \sys_if_engine_uptex:T { eup }
8731         \sys_if_engine_luatex:T
8732         {
8733             lua \lua_now:e
8734             {
8735                 if (pcall(require, 'luaharfbuzz')) then ~
8736                     tex.print("hb") ~
8737                 end
8738             }
8739         }
8740         tex
8741     }
8742 \group_end:
8743 \str_const:Nx \c_sys_engine_format_str
8744 {
8745     \cs_if_exist:NTF \fmtname
8746     {
8747         \bool_lazy_or:nnTF
8748         { \str_if_eq_p:Vn \fmtname { plain } }
8749         { \str_if_eq_p:Vn \fmtname { LaTeX2e } }
8750         {
8751             \sys_if_engine_pdftex:T
8752             { \int_compare:nNnT { \tex_pdfoutput:D } = { 1 } { pdf } }
8753             \sys_if_engine_xetex:T { xe }

```

```

8754         \sys_if_engine_ptex:T { p }
8755         \sys_if_engine_uptex:T { up }
8756         \sys_if_engine_luatex:T
8757         {
8758             \int_compare:nNnT { \tex_pdfoutput:D } = { 0 } { dvi }
8759             lua
8760         }
8761         \str_if_eq:VnTF \fmtname { LaTeX2e }
8762         { latex }
8763         {
8764             \bool_lazy_and:nnT
8765             { \sys_if_engine_pdftex_p: }
8766             { \int_compare_p:nNn { \tex_pdfoutput:D } = { 0 } }
8767             { e }
8768             tex
8769         }
8770     }
8771     { \fmtname }
8772 }
8773 { unknown }
8774 }

```

(End definition for `\c_sys_engine_exec_str` and `\c_sys_engine_format_str`. These variables are documented on page 73.)

47.1.2 Randomness

This candidate function is placed there because `\sys_if_rand_exist:TF` is used in `l3fp-rand`.

`\sys_if_rand_exist_p:` Currently, randomness exists under pdfTeX, LuaTeX, pTeX and upTeX.

```

\sys_if_rand_exist:TF
8775   \__sys_const:nn { sys_if_rand_exist }
8776   { \cs_if_exist_p:N \tex_uniformdeviate:D }

```

(End definition for `\sys_if_rand_exist:TF`. This function is documented on page 306.)

47.1.3 Platform

`\sys_if_platform_unix_p:` Setting these up requires the file module (file lookup), so is actually implemented there.

`\sys_if_platform_unix:TF`

`\sys_if_platform_windows_p:` (End definition for `\sys_if_platform_unix:TF`, `\sys_if_platform_windows:TF`, and `\c_sys_platform_str`. These functions are documented on page 74.)

`\sys_if_platform_windows:TF`

`\c_sys_platform_str`

47.1.4 Configurations

`\sys_load_backend:n` Loading the backend code is pretty simply: check that the backend is valid, then load it up.

`__sys_load_backend_check:N`

`\c_sys_backend_str`

```

8777 \cs_new_protected:Npn \sys_load_backend:n #1
8778 {
8779     \sys_finalise:
8780     \str_if_exist:NTF \c_sys_backend_str
8781     {
8782         \str_if_eq:VnF \c_sys_backend_str {#1}
8783         { \msg_error:nn { sys } { backend-set } }

```

```

8784     }
8785     {
8786         \tl_if_blank:nF {#1}
8787         { \tl_gset:Nn \g__sys_backend_tl {#1} }
8788         \__sys_load_backend_check:N \g__sys_backend_tl
8789         \str_const:Nx \c_sys_backend_str { \g__sys_backend_tl }
8790         \__kernel_sys_configuration_load:n
8791         { l3backend- \c_sys_backend_str }
8792     }
8793 }
8794 \cs_new_protected:Npn \__sys_load_backend_check:N #1
8795 {
8796     \sys_if_engine_xetex:TF
8797     {
8798         \str_case:VnF #1
8799         {
8800             { dvisvgm } { }
8801             { xdvipdfmx } { \tl_gset:Nn #1 { xetex } }
8802             { xetex } { }
8803         }
8804         {
8805             \msg_error:nnxx { sys } { wrong-backend }
8806             #1 { xetex }
8807             \tl_gset:Nn #1 { xetex }
8808         }
8809     }
8810     {
8811         \sys_if_output_pdf:TF
8812         {
8813             \str_if_eq:VnTF #1 { pdfmode }
8814             {
8815                 \sys_if_engine luatex:TF
8816                 { \tl_gset:Nn #1 { luatex } }
8817                 { \tl_gset:Nn #1 { pdftex } }
8818             }
8819             {
8820                 \bool_lazy_or:nnF
8821                 { \str_if_eq_p:Vn #1 { luatex } }
8822                 { \str_if_eq_p:Vn #1 { pdftex } }
8823                 {
8824                     \msg_error:nnxx { sys } { wrong-backend }
8825                     #1 { \sys_if_engine luatex:TF { luatex } { pdftex } }
8826                     \sys_if_engine luatex:TF
8827                     { \tl_gset:Nn #1 { luatex } }
8828                     { \tl_gset:Nn #1 { pdftex } }
8829                 }
8830             }
8831         }
8832         {
8833             \str_case:VnF #1
8834             {
8835                 { dvipdfmx } { }
8836                 { dvips } { }
8837                 { dvisvgm } { }

```



```

8838         }
8839         {
8840             \msg_error:nnxx { sys } { wrong-backend }
8841             #1 { dvips }
8842             \tl_gset:Nn #1 { dvips }
8843         }
8844     }
8845 }
8846 }

```

(End definition for `\sys_load_backend:n`, `__sys_load_backend_check:N`, and `\c_sys_backend_str`. These functions are documented on page 75.)

`\g__sys_debug_bool`

```

8847 \bool_new:N \g__sys_debug_bool

```

(End definition for `\g__sys_debug_bool`.)

`\sys_load_debug:` Simple.

```

8848 \cs_new_protected:Npn \sys_load_debug:
8849 {
8850     \bool_if:NF \g__sys_debug_bool
8851     { \__kernel_sys_configuration_load:n { l3debug } }
8852     \bool_gset_true:N \g__sys_debug_bool
8853 }

```

(End definition for `\sys_load_debug:`. This function is documented on page 76.)

47.1.5 Access to the shell

`\l__sys_internal_tl`

```

8854 \tl_new:N \l__sys_internal_tl

```

(End definition for `\l__sys_internal_tl`.)

`\c__sys_marker_tl` The same idea as the marker for rescanning token lists.

```

8855 \tl_const:Nx \c__sys_marker_tl { : \token_to_str:N : }

```

(End definition for `\c__sys_marker_tl`.)

`\sys_get_shell:nnNF` Setting using a shell is at this level just a slightly specialised file operation, with an additional check for quotes, as these are not supported.

`\sys_get_shell:nnN`
`__sys_get:nnN`
`__sys_get_do:Nw`

```

8856 \cs_new_protected:Npn \sys_get_shell:nnN #1#2#3
8857 {
8858     \sys_get_shell:nnNF {#1} {#2} #3
8859     { \tl_set:Nn #3 { \q_no_value } }
8860 }
8861 \prg_new_protected_conditional:Npnn \sys_get_shell:nnN #1#2#3 { T , F , TF }
8862 {
8863     \sys_if_shell:TF
8864     { \exp_args:No \__sys_get:nnN { \tl_to_str:n {#1} } {#2} #3 }
8865     { \prg_return_false: }
8866 }
8867 \cs_new_protected:Npn \__sys_get:nnN #1#2#3
8868 {

```

```

8869 \tl_if_in:nnTF {#1} { " }
8870 {
8871   \msg_error:nnx
8872   { kernel } { quote-in-shell } {#1}
8873   \prg_return_false:
8874 }
8875 {
8876   \group_begin:
8877   \if_false: { \fi:
8878     \int_set_eq:NN \tex_tracingnesting:D \c_zero_int
8879     \exp_args:No \tex_everyeof:D { \c__sys_marker_tl }
8880     #2 \scan_stop:
8881     \exp_after:wN \__sys_get_do:Nw
8882     \exp_after:wN #3
8883     \exp_after:wN \prg_do_nothing:
8884     \tex_input:D | "#1" \scan_stop:
8885     \if_false: } \fi:
8886     \prg_return_true:
8887   }
8888 }
8889 \exp_args:Nno \use:nn
8890 { \cs_new_protected:Npn \__sys_get_do:Nw #1#2 }
8891 { \c__sys_marker_tl }
8892 {
8893   \group_end:
8894   \tl_set:No #1 {#2}
8895 }

```

(End definition for `\sys_get_shell:nnTF` and others. These functions are documented on page 74.)

`\c__sys_shell_stream_int` This is not needed for LuaTeX: shell escape there isn't done using a TeX interface.

```

8896 \sys_if_engine luatex:F
8897 { \int_const:Nn \c__sys_shell_stream_int { 18 } }

```

(End definition for `\c__sys_shell_stream_int`.)

`\sys_shell_now:n` Execute commands through shell escape immediately.
`__sys_shell_now:e` For LuaTeX, we use a pseudo-primitive to do the actual work.

```

8898 </tex>
8899 <*lua>
8900 do
8901   local os_exec = os.execute
8902
8903   local function shellescape(cmd)
8904     local status,msg = os_exec(cmd)
8905     if status == nil then
8906       write_nl("log","runsystem(" .. cmd .. ")...(" .. msg .. ")\n")
8907     elseif status == 0 then
8908       write_nl("log","runsystem(" .. cmd .. ")...executed\n")
8909     else
8910       write_nl("log","runsystem(" .. cmd .. ")...failed " .. (msg or "") .. "\n")
8911     end
8912   end
8913   luacmd("__sys_shell_now:e", function()

```

```

8914     shellescape(scan_string())
8915   end, "global", "protected")
8916 </lua>

8917 <*tex>
8918 \sys_if_engine luatex:TF
8919 {
8920   \cs_new_protected:Npn \sys_shell_now:n #1
8921     { \__sys_shell_now:e { \exp_not:n {#1} } }
8922 }
8923 {
8924   \cs_new_protected:Npn \sys_shell_now:n #1
8925     { \iow_now:Nn \c__sys_shell_stream_int {#1} }
8926 }
8927 \cs_generate_variant:Nn \sys_shell_now:n { x }
8928 </tex>

```

(End definition for `\sys_shell_now:n` and `__sys_shell_now:e`. This function is documented on page 75.)

`\sys_shell_shipout:n` Execute commands through shell escape at shipout.
`__sys_shell_shipout:e` For LuaTeX, we use the same helper as above but delayed using a `late_lua` whatsit. Creating a `late_lua` whatsit works a bit different if we are running under ConTeXt.

```

8929 <*lua>
8930   local new_latelua = nodes and nodes.nuts and nodes.nuts.pool and nodes.nuts.pool.latelua
8931   local whatsit_id = node.id'whatsit'
8932   local latelua_sub = node.subtype'late_lua'
8933   local node_new = node.direct.new
8934   local setfield = node.direct.setwhatsitfield or node.direct.setfield
8935   return function(f)
8936     local n = node_new(whatsit_id, latelua_sub)
8937     setfield(n, 'data', f)
8938     return n
8939   end
8940 end)()
8941 local node_write = node.direct.write
8942
8943 luacmd("__sys_shell_shipout:e", function()
8944   local cmd = scan_string()
8945   node_write(new_latelua(function() shellescape(cmd) end))
8946 end, "global", "protected")
8947 end
8948 </lua>

8949 <*tex>
8950 \sys_if_engine luatex:TF
8951 {
8952   \cs_new_protected:Npn \sys_shell_shipout:n #1
8953     { \__sys_shell_shipout:e { \exp_not:n {#1} } }
8954 }
8955 {
8956   \cs_new_protected:Npn \sys_shell_shipout:n #1
8957     { \iow_shipout:Nn \c__sys_shell_stream_int {#1} }
8958 }
8959 \cs_generate_variant:Nn \sys_shell_shipout:n { x }

```

(End definition for `\sys_shell_shipout:n` and `__sys_shell_shipout:e`. This function is documented on page 75.)

47.2 Dynamic (every job) code

```

\sys_everyjob:
__sys_everyjob:n
\g__sys_everyjob_tl
8960 \cs_new_protected:Npn \sys_everyjob:
8961 {
8962   \tl_use:N \g__sys_everyjob_tl
8963   \tl_gclear:N \g__sys_everyjob_tl
8964 }
8965 \cs_new_protected:Npn \__sys_everyjob:n #1
8966 { \tl_gput_right:Nn \g__sys_everyjob_tl {#1} }
8967 \tl_new:N \g__sys_everyjob_tl

```

(End definition for `\sys_everyjob:`, `__sys_everyjob:n`, and `\g__sys_everyjob_tl`. This function is documented on page ??.)

47.2.1 The name of the job

`\c_sys_jobname_str` Inherited from the L^AT_EX3 name for the primitive. This *has* to be the primitive as it's set in `\everyjob`. If the user does

```
pdflatex \input some-file-name
```

then `\everyjob` is inserted *before* `\jobname` is changed from `texput`, and thus we would have the wrong result.

```

8968 \__sys_everyjob:n
8969 { \cs_new_eq:NN \c_sys_jobname_str \tex_jobname:D }

```

(End definition for `\c_sys_jobname_str`. This variable is documented on page 72.)

47.2.2 Time and date

`\c_sys_minute_int` `\c_sys_hour_int` `\c_sys_day_int` `\c_sys_month_int` `\c_sys_year_int` Copies of the information provided by T_EX. There is a lot of defensive code in package mode: someone may have moved the primitives, and they can only be recovered if we have `\primitive` and it is working correctly. For IniT_EX of course that is all redundant but does no harm.

```

8970 \__sys_everyjob:n
8971 {
8972   \group_begin:
8973   \cs_set:Npn \__sys_tmp:w #1
8974   {
8975     \str_if_eq:eeTF { \cs_meaning:N #1 } { \token_to_str:N #1 }
8976     { #1 }
8977     {
8978       \cs_if_exist:NTF \tex_primitive:D
8979       {
8980         \bool_lazy_and:nnTF
8981         { \sys_if_engine_xetex_p: }
8982         {
8983           \int_compare_p:nNn
8984           { \exp_after:wN \use_none:n \tex_XeTeXrevision:D }

```

```

8985             < { 99999 }
8986         }
8987         { 0 }
8988         { \tex_primitive:D #1 }
8989     }
8990     { 0 }
8991 }
8992 }
8993 \int_const:Nn \c_sys_minute_int
8994 { \int_mod:nn { \__sys_tmp:w \time } { 60 } }
8995 \int_const:Nn \c_sys_hour_int
8996 { \int_div_truncate:nn { \__sys_tmp:w \time } { 60 } }
8997 \int_const:Nn \c_sys_day_int { \__sys_tmp:w \day }
8998 \int_const:Nn \c_sys_month_int { \__sys_tmp:w \month }
8999 \int_const:Nn \c_sys_year_int { \__sys_tmp:w \year }
9000 \group_end:
9001 }

```

(End definition for `\c_sys_minute_int` and others. These variables are documented on page 72.)

47.2.3 Random numbers

`\sys_rand_seed:` Unpack the primitive. When random numbers are not available, we return zero after an error (and incidentally make sure the number of expansions needed is the same as with random numbers available).

```

9002 \__sys_everyjob:n
9003 {
9004     \sys_if_rand_exist:TF
9005     { \cs_new:Npn \sys_rand_seed: { \tex_the:D \tex_randomseed:D } }
9006     {
9007         \cs_new:Npn \sys_rand_seed:
9008         {
9009             \int_value:w
9010             \msg_expandable_error:nnn { kernel } { fp-no-random }
9011             { \sys_rand_seed: }
9012             \c_zero_int
9013         }
9014     }
9015 }

```

(End definition for `\sys_rand_seed:`. This function is documented on page 74.)

`\sys_gset_rand_seed:n` The primitive always assigns the seed globally.

```

9016 \__sys_everyjob:n
9017 {
9018     \sys_if_rand_exist:TF
9019     {
9020         \cs_new_protected:Npn \sys_gset_rand_seed:n #1
9021         { \tex_setrandomseed:D \int_eval:n {#1} \exp_stop_f: }
9022     }
9023     {
9024         \cs_new_protected:Npn \sys_gset_rand_seed:n #1
9025         {
9026             \msg_error:nnn { kernel } { fp-no-random }

```

```

9027         { \sys_gset_rand_seed:n {#1} }
9028     }
9029 }
9030 }

```

(End definition for `\sys_gset_rand_seed:n`. This function is documented on page 74.)

```

\sys_timer: In LuaTeX, create a pseudo-primitive, otherwise try to locate the real primitive. The
\__sys_elapsedtime: elapsed time will be available if this succeeds.
\sys_if_timer_exist_p:
\sys_if_timer_exist:TF
9031 </tex>
9032 (*lua)
9033   local gettimeofday = os.gettimeofday
9034   local epoch = gettimeofday() - os.clock()
9035   local write = tex.write
9036   local tointeger = math.tointeger
9037   luacmd('__sys_elapsedtime:', function()
9038     write(tointeger((gettimeofday() - epoch)*65536 // 1))
9039   end, 'global')
9040 </lua>
9041 (*tex)
9042 \sys_if_engine luatex:TF
9043 {
9044   \cs_new:Npn \sys_timer:
9045     { \__sys_elapsedtime: }
9046 }
9047 {
9048   \cs_if_exist:NTF \tex_elapsedtime:D
9049     {
9050       \cs_new:Npn \sys_timer:
9051         { \int_value:w \tex_elapsedtime:D }
9052     }
9053     {
9054       \msg_new:nnnn { kernel } { no-elapsed-time }
9055       { No-clock-detected-for~#1. }
9056       { The-current-engine-provides-no-way-to-access-the-system-time. }
9057       \cs_new:Npn \sys_timer:
9058         {
9059           \int_value:w
9060           \msg_expandable_error:nnn { kernel } { no-elapsed-time }
9061           { \sys_timer: }
9062           \c_zero_int
9063         }
9064     }
9065 }
9066 \__sys_const:nn { sys_if_timer_exist }
9067 { \cs_if_exist_p:N \tex_elapsedtime:D || \cs_if_exist_p:N \__sys_elapsedtime: }

```

(End definition for `\sys_timer:`, `__sys_elapsedtime:`, and `\sys_if_timer_exist:TF`. These functions are documented on page 73.)

47.2.4 Access to the shell

`\c_sys_shell_escape_int` Expose the engine's shell escape status to the user.

```

9068 \__sys_everyjob:n

```

```

9069 {
9070   \int_const:Nn \c_sys_shell_escape_int
9071   {
9072     \sys_if_engine luatex:TF
9073     {
9074       \tex_directlua:D
9075       { tex.sprint(status.shell_escape-or-os.execute()) }
9076     }
9077     { \tex_shellescape:D }
9078   }
9079 }

```

(End definition for `\c_sys_shell_escape_int`. This variable is documented on page 75.)

`\sys_if_shell_p:` Performs a check for whether shell escape is enabled. The first set of functions returns true if either of restricted or unrestricted shell escape is enabled, while the other two sets of functions return true in only one of these two cases.

```

\sys_if_shell:TF
\sys_if_shell_unrestricted_p:
\sys_if_shell_unrestricted:TF
\sys_if_shell_restricted_p:
\sys_if_shell_restricted:TF
9080 \__sys_everyjob:n
9081 {
9082   \__sys_const:nn { sys_if_shell }
9083   { \int_compare_p:nNn \c_sys_shell_escape_int > 0 }
9084   \__sys_const:nn { sys_if_shell_unrestricted }
9085   { \int_compare_p:nNn \c_sys_shell_escape_int = 1 }
9086   \__sys_const:nn { sys_if_shell_restricted }
9087   { \int_compare_p:nNn \c_sys_shell_escape_int = 2 }
9088 }

```

(End definition for `\sys_if_shell:TF`, `\sys_if_shell_unrestricted:TF`, and `\sys_if_shell_restricted:TF`. These functions are documented on page 75.)

47.2.5 Held over from `l3file`

`\g_file_curr_name_str` See comments about `\c_sys_jobname_str`: here, as soon as there is file input/output, things get “tided up”.

```

9089 \__sys_everyjob:n
9090 { \cs_gset_eq:NN \g_file_curr_name_str \tex_jobname:D }

```

(End definition for `\g_file_curr_name_str`. This variable is documented on page 93.)

47.3 Last-minute code

`\sys_finalise:` A simple hook to finalise the system-dependent layer. This is forced by the backend loader, which is forced by the main loader, so we do not need to include that here.

```

\__sys_finalise:n
\g__sys_finalise_tl
9091 \cs_new_protected:Npn \sys_finalise:
9092 {
9093   \sys_everyjob:
9094   \tl_use:N \g__sys_finalise_tl
9095   \tl_gclear:N \g__sys_finalise_tl
9096 }
9097 \cs_new_protected:Npn \__sys_finalise:n #1
9098 { \tl_gput_right:Nn \g__sys_finalise_tl {#1} }
9099 \tl_new:N \g__sys_finalise_tl

```

(End definition for `\sys_finalise:`, `__sys_finalise:n`, and `\g__sys_finalise_tl`. This function is documented on page 76.)

47.3.1 Detecting the output

`\sys_if_output_dvi_p:` This is a simple enough concept: the two views here are complementary.

```

\sys_if_output_dvi:TF
\sys_if_output_pdf_p:
\sys_if_output_pdf:TF
\c_sys_output_str
\c_sys_output_str
9100 \__sys_finalise:n
9101 {
9102   \str_const:Nx \c_sys_output_str
9103   {
9104     \int_compare:nNnTF
9105       { \cs_if_exist_use:NF \tex_pdfoutput:D { 0 } } > { 0 }
9106       { pdf }
9107       { dvi }
9108   }
9109   \__sys_const:nn { sys_if_output_dvi }
9110   { \str_if_eq_p:Vn \c_sys_output_str { dvi } }
9111   \__sys_const:nn { sys_if_output_pdf }
9112   { \str_if_eq_p:Vn \c_sys_output_str { pdf } }
9113 }

```

(End definition for `\sys_if_output_dvi:TF`, `\sys_if_output_pdf:TF`, and `\c_sys_output_str`. These functions are documented on page 73.)

47.3.2 Configurations

`\g__sys_backend_tl` As the backend has to be checked and possibly adjusted, the approach here is to create a variable and use that in a one-shot to set a constant.

```

9114 \tl_new:N \g__sys_backend_tl
9115 \__sys_finalise:n
9116 {
9117   \__kernel_tl_gset:Nx \g__sys_backend_tl
9118   {
9119     \sys_if_engine_xetex:TF
9120     { xetex }
9121     {
9122       \sys_if_output_pdf:TF
9123       {
9124         \sys_if_engine_pdftex:TF
9125         { pdftex }
9126         { luatex }
9127       }
9128       { dvips }
9129     }
9130   }
9131 }

```

If there is a class option set, and recognised, we pick it up: these will over-ride anything set automatically but will themselves be over-written if there is a package option.

```

9132 \__sys_finalise:n
9133 {
9134   \cs_if_exist:NT \@classoptionslist
9135   {
9136     \cs_if_eq:NNF \@classoptionslist \scan_stop:
9137     {
9138       \clist_map_inline:Nn \@classoptionslist
9139       {

```



```

9140         \str_case:nnT {#1}
9141         {
9142             { dvipdfmx }
9143             { \tl_gset:Nn \g__sys_backend_tl { dvipdfmx } }
9144             { dvips }
9145             { \tl_gset:Nn \g__sys_backend_tl { dvips } }
9146             { dvisvgm }
9147             { \tl_gset:Nn \g__sys_backend_tl { dvisvgm } }
9148             { pdftex }
9149             { \tl_gset:Nn \g__sys_backend_tl { pdfmode } }
9150             { xetex }
9151             { \tl_gset:Nn \g__sys_backend_tl { xdvipdfmx } }
9152         }
9153         { \clist_remove_all:Nn \@unusedoptionlist {#1} }
9154     }
9155 }
9156 }
9157 }

```

(End definition for \g__sys_backend_tl.)

```

9158 </tex>
9159 </package>

```

Chapter 48

l3msg implementation

```
9160 ⟨*package⟩
9161 ⟨@@=msg⟩

\l__msg_internal_tl A general scratch for the module.
9162 \tl_new:N \l__msg_internal_tl
(End definition for \l__msg_internal_tl.)

\l__msg_name_str Used to save module info when creating messages.
\l__msg_text_str
9163 \str_new:N \l__msg_name_str
9164 \str_new:N \l__msg_text_str
(End definition for \l__msg_name_str and \l__msg_text_str.)
```

48.1 Internal auxiliaries

```
\s__msg_mark Internal scan marks.
\s__msg_stop
9165 \scan_new:N \s__msg_mark
9166 \scan_new:N \s__msg_stop
(End definition for \s__msg_mark and \s__msg_stop.)

\_msg_use_none_delimit_by_s_stop:w Functions to gobble up to a scan mark.
9167 \cs_new:Npn \_msg_use_none_delimit_by_s_stop:w #1 \s__msg_stop { }
(End definition for \_msg_use_none_delimit_by_s_stop:w.)
```

48.2 Creating messages

Messages are created and used separately, so there two parts to the code here. First, a mechanism for creating message text. This is pretty simple, as there is not actually a lot to do.

```
\c__msg_text_prefix_tl Locations for the text of messages.
\c__msg_more_text_prefix_tl
9168 \tl_const:Nn \c__msg_text_prefix_tl { msg~text~>~ }
9169 \tl_const:Nn \c__msg_more_text_prefix_tl { msg~extra~text~>~ }
```

(End definition for \c__msg_text_prefix_tl and \c__msg_more_text_prefix_tl.)

\msg_if_exist:p:nn Test whether the control sequence containing the message text exists or not.
\msg_if_exist:nnTF

```

9170 \prg_new_conditional:Npnn \msg_if_exist:nn #1#2 { p , T , F , TF }
9171 {
9172     \cs_if_exist:cTF { \c__msg_text_prefix_tl #1 / #2 }
9173     { \prg_return_true: } { \prg_return_false: }
9174 }

```

(End definition for \msg_if_exist:nnTF. This function is documented on page 78.)

__msg_chk_if_free:nn This auxiliary is similar to __kernel_chk_if_free_cs:N, and is used when defining messages with \msg_new:nnnn.

```

9175 \cs_new_protected:Npn \__msg_chk_free:nn #1#2
9176 {
9177     \msg_if_exist:nnT {#1} {#2}
9178     {
9179         \msg_error:nnxx { msg } { already-defined }
9180         {#1} {#2}
9181     }
9182 }

```

(End definition for __msg_chk_if_free:nn.)

\msg_new:nnnn Setting a message simply means saving the appropriate text into two functions. A sanity check first.

\msg_new:nnn
\msg_gset:nnnn
\msg_gset:nnn
\msg_set:nnnn
\msg_set:nnn

```

9183 \cs_new_protected:Npn \msg_new:nnnn #1#2
9184 {
9185     \__msg_chk_free:nn {#1} {#2}
9186     \msg_gset:nnnn {#1} {#2}
9187 }
9188 \cs_new_protected:Npn \msg_new:nnn #1#2#3
9189 { \msg_new:nnnn {#1} {#2} {#3} { } }
9190 \cs_new_protected:Npn \msg_set:nnnn #1#2#3#4
9191 {
9192     \cs_set:cpn { \c__msg_text_prefix_tl #1 / #2 }
9193     ##1##2##3##4 {#3}
9194     \cs_set:cpn { \c__msg_more_text_prefix_tl #1 / #2 }
9195     ##1##2##3##4 {#4}
9196 }
9197 \cs_new_protected:Npn \msg_set:nnn #1#2#3
9198 { \msg_set:nnnn {#1} {#2} {#3} { } }
9199 \cs_new_protected:Npn \msg_gset:nnnn #1#2#3#4
9200 {
9201     \cs_gset:cpn { \c__msg_text_prefix_tl #1 / #2 }
9202     ##1##2##3##4 {#3}
9203     \cs_gset:cpn { \c__msg_more_text_prefix_tl #1 / #2 }
9204     ##1##2##3##4 {#4}
9205 }
9206 \cs_new_protected:Npn \msg_gset:nnn #1#2#3
9207 { \msg_gset:nnnn {#1} {#2} {#3} { } }

```

(End definition for \msg_new:nnnn and others. These functions are documented on page 78.)

48.3 Messages: support functions and text

```

\c__msg_coding_error_text_tl
\c__msg_continue_text_tl
\c__msg_critical_text_tl
\c__msg_fatal_text_tl
\c__msg_help_text_tl
\c__msg_no_info_text_tl
\c__msg_on_line_text_tl
\c__msg_return_text_tl
\c__msg_trouble_text_tl

9208 \tl_const:Nn \c__msg_coding_error_text_tl
9209 {
9210   This-is-a-coding-error.
9211   \\ \\
9212 }
9213 \tl_const:Nn \c__msg_continue_text_tl
9214 { Type~<return>~to~continue }
9215 \tl_const:Nn \c__msg_critical_text_tl
9216 { Reading~the~current~file~'\g_file_curr_name_str'~will~stop. }
9217 \tl_const:Nn \c__msg_fatal_text_tl
9218 { This-is-a-fatal-error:~LaTeX~will~abort. }
9219 \tl_const:Nn \c__msg_help_text_tl
9220 { For~immediate-help~type-H~<return> }
9221 \tl_const:Nn \c__msg_no_info_text_tl
9222 {
9223   LaTeX~does~not~know~anything~more~about~this~error,~sorry.
9224   \c__msg_return_text_tl
9225 }
9226 \tl_const:Nn \c__msg_on_line_text_tl { on-line }
9227 \tl_const:Nn \c__msg_return_text_tl
9228 {
9229   \\ \\
9230   Try~typing~<return>~to~proceed.
9231   \\
9232   If~that~doesn't~work,~type-X~<return>~to~quit.
9233 }
9234 \tl_const:Nn \c__msg_trouble_text_tl
9235 {
9236   \\ \\
9237   More~errors~will~almost~certainly~follow: \\
9238   the~LaTeX~run~should~be~aborted.
9239 }

```

(End definition for `\c__msg_coding_error_text_tl` and others.)

`\msg_line_number:` For writing the line number nicely. `\msg_line_context:` was set up earlier, so this is not new.

```

9240 \cs_new:Npn \msg_line_number: { \int_use:N \tex_inputlineno:D }
9241 \cs_gset:Npn \msg_line_context:
9242 {
9243   \c__msg_on_line_text_tl
9244   \c_space_tl
9245   \msg_line_number:
9246 }

```

(End definition for `\msg_line_number:` and `\msg_line_context:`. These functions are documented on page 79.)

48.4 Showing messages: low level mechanism

`_msg_interrupt:Nnnn`
`_msg_no_more_text:nnnn`

The low-level interruption macro is rather opaque, unfortunately. Depending on the availability of more information there is a choice of how to set up the further help. We feed the extra help text and the message itself to a wrapping auxiliary, in this order because we must first setup T_EX's `\errhelp` register before issuing an `\errmessage`. To deal with the various cases of critical or fatal errors with and without help text, there is a bit of argument-passing to do.

```

9247 \cs_new_protected:Npn \_msg_interrupt:NnnnN #1#2#3#4#5
9248 {
9249   \str_set:Nx \l_msg_text_str { #1 {#2} }
9250   \str_set:Nx \l_msg_name_str { \msg_module_name:n {#2} }
9251   \cs_if_eq:cNTF
9252     { \c__msg_more_text_prefix_tl #2 / #3 }
9253     \_msg_no_more_text:nnnn
9254     {
9255       \_msg_interrupt_wrap:nnn
9256       { \use:c { \c__msg_text_prefix_tl #2 / #3 } #4 }
9257       { \c__msg_continue_text_tl }
9258       {
9259         \c__msg_no_info_text_tl
9260         \tl_if_empty:NF #5
9261         { \\ \\ #5 }
9262       }
9263     }
9264     {
9265       \_msg_interrupt_wrap:nnn
9266       { \use:c { \c__msg_text_prefix_tl #2 / #3 } #4 }
9267       { \c__msg_help_text_tl }
9268       {
9269         \use:c { \c__msg_more_text_prefix_tl #2 / #3 } #4
9270         \tl_if_empty:NF #5
9271         { \\ \\ #5 }
9272       }
9273     }
9274   }
9275   \cs_new:Npn \_msg_no_more_text:nnnn #1#2#3#4 { }
```

(End definition for `_msg_interrupt:Nnnn` and `_msg_no_more_text:nnnn`.)

`_msg_interrupt_wrap:nnn`
`_msg_interrupt_text:n`
`_msg_interrupt_more_text:n`

First setup T_EX's `\errhelp` register with the extra help #1, then build a nice-looking error message with #2. Everything is done using x-type expansion as the new line markers are different for the two type of text and need to be correctly set up. The auxiliary `_msg_interrupt_more_text:n` receives its argument as a line-wrapped string, which is thus unaffected by expansion. We have to split the main text into two parts as only the “message” itself is wrapped with a leader: the generic help is wrapped at full width. We also have to allow for the two characters used by `\errmessage` itself.

```

9276 \cs_new_protected:Npn \_msg_interrupt_wrap:nnn #1#2#3
9277 {
9278   \iow_wrap:nnnN { \\ #3 } { } { } \_msg_interrupt_more_text:n
9279   \group_begin:
9280     \int_sub:Nn \l_iow_line_count_int { 2 }
9281     \iow_wrap:nxnN { \l_msg_text_str : ~ #1 }
```

```

9282     {
9283       ( \l__msg_name_str )
9284       \prg_replicate:nn
9285       {
9286         \str_count:N \l__msg_text_str
9287         - \str_count:N \l__msg_name_str
9288         + 2
9289       }
9290       { ~ }
9291     }
9292     { } \__msg_interrupt_text:n
9293     \iow_wrap:nnnN { \l__msg_internal_tl \\ \\ #2 } { } { }
9294     \__msg_interrupt:n
9295   }
9296   \cs_new_protected:Npn \__msg_interrupt_text:n #1
9297   {
9298     \group_end:
9299     \tl_set:Nn \l__msg_internal_tl {#1}
9300   }
9301   \cs_new_protected:Npn \__msg_interrupt_more_text:n #1
9302   { \exp_args:Nx \tex_errhelp:D { #1 \iow_newline: } }

```

(End definition for __msg_interrupt_wrap:nnn, __msg_interrupt_text:n, and __msg_interrupt_more_text:n.)

`__msg_interrupt:n` The business end of the process starts by producing some visual separation of the message from the main part of the log. The error message needs to be printed with everything made “invisible”: T_EX’s own information involves the macro in which `\errmessage` is called, and the end of the argument of the `\errmessage`, including the closing brace. We use an active `!` to call the `\errmessage` primitive, and end its argument with `\use_none:n` `{\spaces}` which fills the output with spaces. Two trailing closing braces are turned into spaces to hide them as well. The group in which we alter the definition of the active `!` is closed before producing the message: this ensures that tokens inserted by typing `I` in the command-line are inserted after the message is entirely cleaned up.

The `__kernel_iow_with:Nnn` auxiliary, defined in `l3file`, expects an *integer variable*, an integer *value*, and some *code*. It runs the *code* after ensuring that the *integer variable* takes the given *value*, then restores the former value of the *integer variable* if needed. We use it to ensure that the `\newlinechar` is 10, as needed for `\iow_newline:` to work, and that `\errorcontextlines` is `-1`, to avoid showing irrelevant context. Note that restoring the former value of these integers requires inserting tokens after the `\errmessage`, which go in the way of tokens which could be inserted by the user. This is unavoidable.

```

9303 \group_begin:
9304   \char_set_lccode:nn { 38 } { 32 } % &
9305   \char_set_lccode:nn { 46 } { 32 } % .
9306   \char_set_lccode:nn { 123 } { 32 } % {
9307   \char_set_lccode:nn { 125 } { 32 } % }
9308   \char_set_catcode_active:N \&
9309   \tex_lowercase:D
9310   {
9311     \group_end:
9312     \cs_new_protected:Npn \__msg_interrupt:n #1
9313     {

```

```

9314 \iow_term:n { }
9315 \__kernel_iow_with:Nnn \tex_newlinechar:D { '\^^J }
9316 {
9317   \__kernel_iow_with:Nnn \tex_errorcontextlines:D { -1 }
9318   {
9319     \group_begin:
9320     \cs_set_protected:Npn &
9321     {
9322       \tex_errmessage:D
9323       {
9324         #1
9325         \use_none:n
9326         { ..... }
9327       }
9328     }
9329     \exp_after:wN
9330     \group_end:
9331     &
9332     }
9333   }
9334 }
9335 }

```

(End definition for `_msg_interrupt:n`.)

48.5 Displaying messages

L^AT_EX is handling error messages and so the T_EX ones are disabled.

```

9336 \int_gset:Nn \tex_errorcontextlines:D { -1 }

```

`\msg_fatal_text:n` A function for issuing messages: both the text and order could in principle vary. The module name may be empty for kernel messages, hence the slightly contorted code path for a space.

```

\msg_critical_text:n
\msg_error_text:n
\msg_warning_text:n
\msg_info_text:n
\__msg_text:nn
\__msg_text:n
9337 \cs_new:Npn \msg_fatal_text:n #1
9338 {
9339   Fatal ~
9340   \msg_error_text:n {#1}
9341 }
9342 \cs_new:Npn \msg_critical_text:n #1
9343 {
9344   Critical ~
9345   \msg_error_text:n {#1}
9346 }
9347 \cs_new:Npn \msg_error_text:n #1
9348 { \__msg_text:nn {#1} { Error } }
9349 \cs_new:Npn \msg_warning_text:n #1
9350 { \__msg_text:nn {#1} { Warning } }
9351 \cs_new:Npn \msg_info_text:n #1
9352 { \__msg_text:nn {#1} { Info } }
9353 \cs_new:Npn \__msg_text:nn #1#2
9354 {
9355   \exp_args:Nf \__msg_text:n { \msg_module_type:n {#1} }
9356   \exp_args:Nf \__msg_text:n { \msg_module_name:n {#1} }

```

```

9357     #2
9358   }
9359   \cs_new:Npn \_msg_text:n #1
9360   {
9361     \tl_if_blank:nF {#1}
9362     { #1 ~ }
9363   }

```

(End definition for `\msg_fatal_text:n` and others. These functions are documented on page 79.)

`\g_msg_module_name_prop` For storing public module information: the kernel data is set up in advance.
`\g_msg_module_type_prop`

```

9364   \prop_new:N \g_msg_module_name_prop
9365   \prop_gput:Nnn \g_msg_module_name_prop { LaTeX } { LaTeX3 }
9366   \prop_new:N \g_msg_module_type_prop
9367   \prop_gput:Nnn \g_msg_module_type_prop { LaTeX } { }

```

(End definition for `\g_msg_module_name_prop` and `\g_msg_module_type_prop`. These variables are documented on page 78.)

`\msg_module_type:n` Contextual footer information, with the potential to give modules an alternative name.

```

9368   \cs_new:Npn \msg_module_type:n #1
9369   {
9370     \prop_if_in:NnTF \g_msg_module_type_prop {#1}
9371     { \prop_item:Nn \g_msg_module_type_prop {#1} }
9372     { Package }
9373   }

```

(End definition for `\msg_module_type:n`. This function is documented on page 78.)

`\msg_module_name:n` Contextual footer information, with the potential to give modules an alternative name.
`\msg_see_documentation_text:n`

```

9374   \cs_new:Npn \msg_module_name:n #1
9375   {
9376     \prop_if_in:NnTF \g_msg_module_name_prop {#1}
9377     { \prop_item:Nn \g_msg_module_name_prop {#1} }
9378     {#1}
9379   }
9380   \cs_new:Npn \msg_see_documentation_text:n #1
9381   {
9382     See-the~ \msg_module_name:n {#1} ~
9383     documentation-for-further-information.
9384   }

```

(End definition for `\msg_module_name:n` and `\msg_see_documentation_text:n`. These functions are documented on page 78.)

`_msg_class_new:nn`

```

9385   \group_begin:
9386   \cs_set_protected:Npn \_msg_class_new:nn #1#2
9387   {
9388     \prop_new:c { l_msg_redirect_ #1 _prop }
9389     \cs_new_protected:cpn { _msg_ #1 _code:nnnnnn }
9390     ##1##2##3##4##5##6 {#2}
9391     \cs_new_protected:cpn { msg_ #1 :nnnnnn } ##1##2##3##4##5##6
9392     {
9393       \use:x

```



```

9394         {
9395             \exp_not:n { \_msg_use:nnnnnn {#1} {##1} {##2} }
9396             { \tl_to_str:n {##3} } { \tl_to_str:n {##4} }
9397             { \tl_to_str:n {##5} } { \tl_to_str:n {##6} }
9398         }
9399     }
9400     \cs_new_protected:cpx { msg_ #1 :nnnnn } ##1##2##3##4##5
9401     { \exp_not:c { msg_ #1 :nnnnnn } {##1} {##2} {##3} {##4} {##5} { } }
9402     \cs_new_protected:cpx { msg_ #1 :nnnn } ##1##2##3##4
9403     { \exp_not:c { msg_ #1 :nnnnnn } {##1} {##2} {##3} {##4} { } { } }
9404     \cs_new_protected:cpx { msg_ #1 :nnn } ##1##2##3
9405     { \exp_not:c { msg_ #1 :nnnnnn } {##1} {##2} {##3} { } { } { } }
9406     \cs_new_protected:cpx { msg_ #1 :nn } ##1##2
9407     { \exp_not:c { msg_ #1 :nnnnnn } {##1} {##2} { } { } { } { } }
9408     \cs_new_protected:cpx { msg_ #1 :nnxxx } ##1##2##3##4##5##6
9409     {
9410         \use:x
9411         {
9412             \exp_not:N \exp_not:n
9413             { \exp_not:c { msg_ #1 :nnnnnn } {##1} {##2} }
9414             {##3} {##4} {##5} {##6}
9415         }
9416     }
9417     \cs_new_protected:cpx { msg_ #1 :nnxxx } ##1##2##3##4##5
9418     { \exp_not:c { msg_ #1 :nnxxx } {##1} {##2} {##3} {##4} {##5} { } }
9419     \cs_new_protected:cpx { msg_ #1 :nnxx } ##1##2##3##4
9420     { \exp_not:c { msg_ #1 :nnxxx } {##1} {##2} {##3} {##4} { } { } }
9421     \cs_new_protected:cpx { msg_ #1 :nnx } ##1##2##3
9422     { \exp_not:c { msg_ #1 :nnxxx } {##1} {##2} {##3} { } { } { } }
9423 }

```

(End definition for `_msg_class_new:nn`.)

`\msg_fatal:nnnnnn` For fatal errors, after the error message `TEX` bails out. We force a bail out rather than using `\end` as this means it does not matter if we are in a context where normally the run cannot end.

```

\msg_fatal:nnxxx
\msg_fatal:nnnnn
\msg_fatal:nnxx
\msg_fatal:nnx
\msg_fatal:nnn
\msg_fatal:nnx
\msg_fatal:nn
\_msg_fatal_exit:
9424 \_msg_class_new:nn { fatal }
9425 {
9426     \_msg_interrupt:NnnnN
9427     \msg_fatal_text:n {#1} {#2}
9428     { {#3} {#4} {#5} {#6} }
9429     \c__msg_fatal_text_tl
9430     \_msg_fatal_exit:
9431 }
9432 \cs_new_protected:Npn \_msg_fatal_exit:
9433 {
9434     \tex_batchmode:D
9435     \tex_read:D -1 to \l__msg_internal_tl
9436 }

```

(End definition for `\msg_fatal:nnnnnn` and others. These functions are documented on page 80.)

`\msg_critical:nnnnnn` Not quite so bad: just end the current file.

```

\msg_critical:nnxxx
\msg_critical:nnnnn
\msg_critical:nnxxx
\msg_critical:nnnn
\msg_critical:nnxx
\msg_critical:nnn
\msg_critical:nnx
\msg_critical:nn
9437 \_msg_class_new:nn { critical }

```

```

9438 {
9439   \__msg_interrupt:NnnnN
9440   \msg_critical_text:n {#1} {#2}
9441   { {#3} {#4} {#5} {#6} }
9442   \c__msg_critical_text_tl
9443   \tex_endinput:D
9444 }

```

(End definition for `\msg_critical:nnnnnn` and others. These functions are documented on page 81.)

`\msg_error:nnnnnn` For an error, the interrupt routine is called. We check if there is a “more text” by
`\msg_error:nnxxxx` comparing that control sequence with a permanently empty text. We have to undefine
`\msg_error:nnnnnn` the bootstrap versions here.

```

\msg_error:nnxxxx 9445 \cs_undefine:N \msg_error:nnxxx
\msg_error:nnnnn 9446 \cs_undefine:N \msg_error:nnx
\msg_error:nnxxx 9447 \cs_undefine:N \msg_error:nn
\msg_error:nnnn 9448 \__msg_class_new:nn { error }
\msg_error:nnx 9449 {
\msg_error:nn 9450   \__msg_interrupt:NnnnN
9451   \msg_error_text:n {#1} {#2}
9452   { {#3} {#4} {#5} {#6} }
9453   \c_empty_tl
9454 }

```

(End definition for `\msg_error:nnnnnn` and others. These functions are documented on page 81.)

`__msg_info_aux:NNnnnnnn` Warnings and information messages have no decoration. Warnings are printed to the
`\msg_warning:nnnnnn` terminal while information can either go to the log or both log and terminal.

```

\msg_warning:nnxxxx 9455 \cs_new_protected:Npn \__msg_info_aux:NNnnnnnn #1#2#3#4#5#6#7#8
\msg_warning:nnnnnn 9456 {
\msg_warning:nnxxx 9457   \str_set:Nx \l__msg_text_str { #2 {#3} }
\msg_warning:nnnn 9458   \str_set:Nx \l__msg_name_str { \msg_module_name:n {#3} }
\msg_warning:nnxx 9459   #1 { }
\msg_warning:nnnn 9460   \iow_wrap:nxnN
\msg_warning:nnx 9461   {
\msg_warning:nn 9462     \l__msg_text_str : ~
\msg_note:nnnnnn 9463     \use:c { \c__msg_text_prefix_tl #3 / #4 } {#5} {#6} {#7} {#8}
\msg_note:nnxxxx 9464   }
\msg_note:nnnnnn 9465   {
\msg_note:nnxxx 9466     ( \l__msg_name_str )
\msg_note:nnnn 9467     \prg_replicate:nn
\msg_note:nnxx 9468     {
\msg_note:nnx 9469       \str_count:N \l__msg_text_str
\msg_note:nnnn 9470       - \str_count:N \l__msg_name_str
\msg_note:nnx 9471     }
\msg_note:nn 9472     { ~ }
\msg_info:nnnnnn 9473   }
\msg_info:nnxxxx 9474   { } #1
\msg_info:nnnnnn 9475   #1 { }
\msg_info:nnxxxx 9476 }
\msg_info:nnxxxx 9477 \__msg_class_new:nn { warning }
\msg_info:nnnn 9478 {
\msg_info:nnxx 9479   \__msg_info_aux:NNnnnnnn \iow_term:n \msg_warning_text:n
\msg_info:nnnn 9480   {#1} {#2} {#3} {#4} {#5} {#6}
\msg_info:nnx
\msg_info:nn

```

```

9481     }
9482     \_msg\_class\_new:nn { note }
9483     {
9484         \_msg\_info\_aux:NNnnnnnn \iow\_term:n \msg\_info\_text:n
9485         {#1} {#2} {#3} {#4} {#5} {#6}
9486     }
9487     \_msg\_class\_new:nn { info }
9488     {
9489         \_msg\_info\_aux:NNnnnnnn \iow\_log:n \msg\_info\_text:n
9490         {#1} {#2} {#3} {#4} {#5} {#6}
9491     }

```

(End definition for _msg_info_aux:NNnnnnnn and others. These functions are documented on page 81.)

\msg_log:nnnnnn “Log” data is very similar to information, but with no extras added. “Term” is used for communicating with the user through the terminal, like diagnostic messages, and debugging. This is similar to “log” messages, but uses the terminal output.

```

\msg\_log:nnxxxx
\msg\_log:nnnnn
\msg\_log:nnxxx
\msg\_log:nnnn
\msg\_log:nnxx
\msg\_log:nnn
\msg\_log:nnx
\msg\_log:nn
\msg\_term:nnnnnn
\msg\_term:nnxxxx
\msg\_term:nnnnn
\msg\_term:nnxxx
\msg\_term:nnnn
\msg\_term:nnxx
\msg\_term:nnn

```

```

9492     \_msg\_class\_new:nn { log }
9493     {
9494         \iow\_wrap:nnnN
9495         { \use:c { \c\_msg\_text\_prefix\_tl #1 / #2 } {#3} {#4} {#5} {#6} }
9496         { } { } \iow\_log:n
9497     }
9498     \_msg\_class\_new:nn { term }
9499     {
9500         \iow\_wrap:nnnN
9501         { \use:c { \c\_msg\_text\_prefix\_tl #1 / #2 } {#3} {#4} {#5} {#6} }
9502         { } { } \iow\_term:n
9503     }

```

(End definition for \msg_log:nnnnnn and others. These functions are documented on page 82.)

\msg_term:nnx The none message type is needed so that input can be gobbled.

```

\msg\_none:nnnnnn
\msg\_term:nn
\msg\_none:nnxxxx
\msg\_none:nnnnn
\msg\_none:nnxxx

```

```

9504     \_msg\_class\_new:nn { none } { }

```

(End definition for \msg_none:nnnnnn and others. These functions are documented on page 82.)

\msg_none:nnnn The show message type is used for \seq_show:N and similar complicated data structures. Wrap the given text with a trailing dot (important later) then pass it to _msg_show:n. If there is \>~ (or if the whole thing starts with >~) we split there, print the first part and show the second part using \showtokens (the \exp_after:wN ensure a nice display). Note that this primitive adds a leading >~ and trailing dot. That is why we included a trailing dot before wrapping and removed it afterwards. If there is no \>~ do the same but with an empty second part which adds a spurious but inevitable >~.

```

\msg\_show:nnxx
\msg\_show:nnn
\msg\_show:nnx
\msg\_show:nn
\_msg\_show:n
\_msg\_show:w
\_msg\_show\_dot:w
\_msg\_show:nn

```

```

9505     \_msg\_class\_new:nn { show }
9506     {
9507         \iow\_wrap:nnnN
9508         { \use:c { \c\_msg\_text\_prefix\_tl #1 / #2 } {#3} {#4} {#5} {#6} }
9509         { } { } \_msg\_show:n
9510     }
9511     \cs\_new\_protected:Npn \_msg\_show:n #1
9512     {
9513         \tl\_if\_in:nnTF { ^~J #1 } { ^~J > ~ }

```

```

9514     {
9515         \tl_if_in:nnTF { #1 \s__msg_mark } { . \s__msg_mark }
9516         { \__msg_show_dot:w } { \__msg_show:w }
9517         ^^J #1 \s__msg_stop
9518     }
9519     { \__msg_show:nn { ? #1 } { } }
9520 }
9521 \cs_new:Npn \__msg_show_dot:w #1 ^^J > ~ #2 . \s__msg_stop
9522 { \__msg_show:nn {#1} {#2} }
9523 \cs_new:Npn \__msg_show:w #1 ^^J > ~ #2 \s__msg_stop
9524 { \__msg_show:nn {#1} {#2} }
9525 \cs_new_protected:Npn \__msg_show:nn #1#2
9526 {
9527     \tl_if_empty:nF {#1}
9528     { \exp_args:No \iow_term:n { \use_none:n #1 } }
9529     \tl_set:Nn \l__msg_internal_tl {#2}
9530     \__kernel_iow_with:Nnn \tex_newlinechar:D { 10 }
9531     {
9532         \__kernel_iow_with:Nnn \tex_errorcontextlines:D { -1 }
9533         {
9534             \tex_showtokens:D \exp_after:wN \exp_after:wN \exp_after:wN
9535             { \exp_after:wN \l__msg_internal_tl }
9536         }
9537     }
9538 }

```

(End definition for `\msg_show:nnnnnn` and others. These functions are documented on page 83.)

End the group to eliminate `__msg_class_new:nn`.

```

9539 \group_end:

```

`__msg_class_chk_exist:nT` Checking that a message class exists. We build this from `\cs_if_free:cTF` rather than `\cs_if_exist:cTF` because that avoids reading the second argument earlier than necessary.

```

9540 \cs_new:Npn \__msg_class_chk_exist:nT #1
9541 {
9542     \cs_if_free:cTF { __msg_ #1 _code:nnnnnn }
9543     { \msg_error:nnx { msg } { class-unknown } {#1} }
9544 }

```

(End definition for `__msg_class_chk_exist:nT`.)

`\l__msg_class_tl` Support variables needed for the redirection system.
`\l__msg_current_class_tl`

```

9545 \tl_new:N \l__msg_class_tl
9546 \tl_new:N \l__msg_current_class_tl

```

(End definition for `\l__msg_class_tl` and `\l__msg_current_class_tl`.)

`\l__msg_redirect_prop` For redirection of individually-named messages

```

9547 \prop_new:N \l__msg_redirect_prop

```

(End definition for `\l__msg_redirect_prop`.)

`\l__msg_hierarchy_seq` During redirection, split the message name into a sequence: `{/module/submodule}`, `{/module}`, and `{}`.

```

9548 \seq_new:N \l__msg_hierarchy_seq

```

(End definition for \l__msg_hierarchy_seq.)

\l__msg_class_loop_seq Classes encountered when following redirections to check for loops.

```
9549 \seq_new:N \l__msg_class_loop_seq
```

(End definition for \l__msg_class_loop_seq.)

__msg_use:nnnnnnn

Actually using a message is a multi-step process. First, some safety checks on the message and class requested. The code and arguments are then stored to avoid passing them around. The assignment to __msg_use_code: is similar to \tl_set:Nn. The message is eventually produced with whatever \l__msg_class_tl is when __msg_use_code: is called. Here is also a good place to suppress tracing output if the trace package is loaded since all (non-expandable) messages go through this auxiliary.

```
\__msg_use_redirect_name:n
\__msg_use_hierarchy:nwN
\__msg_use_redirect_module:n
\__msg_use_code:

9550 \cs_new_protected:Npn \__msg_use:nnnnnnn #1#2#3#4#5#6#7
9551 {
9552   \cs_if_exist_use:N \conditionally@traceoff
9553   \msg_if_exist:nnTF {#2} {#3}
9554   {
9555     \__msg_class_chk_exist:nT {#1}
9556     {
9557       \tl_set:Nn \l__msg_current_class_tl {#1}
9558       \cs_set_protected:Npx \__msg_use_code:
9559       {
9560         \exp_not:n
9561         {
9562           \use:c { __msg_ \l__msg_class_tl _code:nnnnnn }
9563           {#2} {#3} {#4} {#5} {#6} {#7}
9564         }
9565       }
9566       \__msg_use_redirect_name:n { #2 / #3 }
9567     }
9568   }
9569   { \msg_error:nnxx { msg } { unknown } {#2} {#3} }
9570   \cs_if_exist_use:N \conditionally@traceon
9571 }
9572 \cs_new_protected:Npn \__msg_use_code: { }
```

The first check is for a individual message redirection. If this applies then no further redirection is attempted. Otherwise, split the message name into $\langle module \rangle$, $\langle submodule \rangle$ and $\langle message \rangle$ (with an arbitrary number of slashes), and store $\{/module/submodule\}$, $\{/module\}$ and $\{\}$ into \l__msg_hierarchy_seq. We then map through this sequence, applying the most specific redirection.

```
9573 \cs_new_protected:Npn \__msg_use_redirect_name:n #1
9574 {
9575   \prop_get:NnNTF \l__msg_redirect_prop { / #1 } \l__msg_class_tl
9576   { \__msg_use_code: }
9577   {
9578     \seq_clear:N \l__msg_hierarchy_seq
9579     \__msg_use_hierarchy:nwN { }
9580     #1 \s__msg_mark \__msg_use_hierarchy:nwN
9581     / \s__msg_mark \__msg_use_none_delimit_by_s_stop:w
9582     \s__msg_stop
9583     \__msg_use_redirect_module:n { }
9584   }
```

```

9585     }
9586 \cs_new_protected:Npn \__msg_use_hierarchy:nwwN #1#2 / #3 \s__msg_mark #4
9587 {
9588     \seq_put_left:Nn \l__msg_hierarchy_seq {#1}
9589     #4 { #1 / #2 } #3 \s__msg_mark #4
9590 }

```

At this point, the items of `\l__msg_hierarchy_seq` are the various levels at which we should look for a redirection. Redirections which are less specific than the argument of `__msg_use_redirect_module:n` are not attempted. This argument is empty for a class redirection, */module* for a module redirection, *etc.* Loop through the sequence to find the most specific redirection, with module `##1`. The loop is interrupted after testing for a redirection for `##1` equal to the argument `#1` (least specific redirection allowed). When a redirection is found, break the mapping, then if the redirection targets the same class, output the code with that class, and otherwise set the target as the new current class, and search for further redirections. Those redirections should be at least as specific as `##1`.

```

9591 \cs_new_protected:Npn \__msg_use_redirect_module:n #1
9592 {
9593     \seq_map_inline:Nn \l__msg_hierarchy_seq
9594     {
9595         \prop_get:cnTF { l__msg_redirect_ \l__msg_current_class_tl _prop }
9596         {##1} \l__msg_class_tl
9597         {
9598             \seq_map_break:n
9599             {
9600                 \tl_if_eq:NNTF \l__msg_current_class_tl \l__msg_class_tl
9601                 { \__msg_use_code: }
9602                 {
9603                     \tl_set_eq:NN \l__msg_current_class_tl \l__msg_class_tl
9604                     \__msg_use_redirect_module:n {##1}
9605                 }
9606             }
9607         }
9608         {
9609             \str_if_eq:nnT {##1} {#1}
9610             {
9611                 \tl_set_eq:NN \l__msg_class_tl \l__msg_current_class_tl
9612                 \seq_map_break:n { \__msg_use_code: }
9613             }
9614         }
9615     }
9616 }

```

(End definition for `__msg_use:nnnnnnn` and others.)

\msg_redirect_name:nnn Named message always use the given class even if that class is redirected further. An empty target class cancels any existing redirection for that message.

```

9617 \cs_new_protected:Npn \msg_redirect_name:nnn #1#2#3
9618 {
9619     \tl_if_empty:nTF {#3}
9620     { \prop_remove:Nn \l__msg_redirect_prop { / #1 / #2 } }
9621     {
9622         \__msg_class_chk_exist:nT {#3}

```

```

9623         { \prop_put:Nnn \l__msg_redirect_prop { / #1 / #2 } {#3} }
9624     }
9625 }

```

(End definition for `\msg_redirect_name:nnn`. This function is documented on page 84.)

```

\msg_redirect_class:nn
\msg_redirect_module:nnn
  \__msg_redirect:nnn
\__msg_redirect_loop_chk:nnn
\__msg_redirect_loop_list:n

```

If the target class is empty, eliminate the corresponding redirection. Otherwise, add the redirection. We must then check for a loop: as an initialization, we start by storing the initial class in `\l__msg_current_class_tl`.

```

9626 \cs_new_protected:Npn \msg_redirect_class:nn
9627   { \__msg_redirect:nnn { } }
9628 \cs_new_protected:Npn \msg_redirect_module:nnn #1
9629   { \__msg_redirect:nnn { / #1 } }
9630 \cs_new_protected:Npn \__msg_redirect:nnn #1#2#3
9631   {
9632     \__msg_class_chk_exist:nT {#2}
9633     {
9634       \tl_if_empty:nTF {#3}
9635       { \prop_remove:cn { l__msg_redirect_ #2 _prop } {#1} }
9636       {
9637         \__msg_class_chk_exist:nT {#3}
9638         {
9639           \prop_put:cnn { l__msg_redirect_ #2 _prop } {#1} {#3}
9640           \tl_set:Nn \l__msg_current_class_tl {#2}
9641           \seq_clear:N \l__msg_class_loop_seq
9642           \__msg_redirect_loop_chk:nnn {#2} {#3} {#1}
9643         }
9644       }
9645     }
9646   }

```

Since multiple redirections can only happen with increasing specificity, a loop requires that all steps are of the same specificity. The new redirection can thus only create a loop with other redirections for the exact same module, #1, and not submodules. After some initialization above, follow redirections with `\l__msg_class_tl`, and keep track in `\l__msg_class_loop_seq` of the various classes encountered. A redirection from a class to itself, or the absence of redirection both mean that there is no loop. A redirection to the initial class marks a loop. To break it, we must decide which redirection to cancel. The user most likely wants the newly added redirection to hold with no further redirection. We thus remove the redirection starting from #2, target of the new redirection. Note that no message is emitted by any of the underlying functions: otherwise we may get an infinite loop because of a message from the message system itself.

```

9647 \cs_new_protected:Npn \__msg_redirect_loop_chk:nnn #1#2#3
9648   {
9649     \seq_put_right:Nn \l__msg_class_loop_seq {#1}
9650     \prop_get:cnNT { l__msg_redirect_ #1 _prop } {#3} \l__msg_class_tl
9651     {
9652       \str_if_eq:VnF \l__msg_class_tl {#1}
9653       {
9654         \tl_if_eq:NNTF \l__msg_class_tl \l__msg_current_class_tl
9655         {
9656           \prop_put:cnn { l__msg_redirect_ #2 _prop } {#3} {#2}
9657           \msg_warning:nxxxxx
9658             { msg } { redirect-loop }

```

```

9659         { \seq_item:Nn \l__msg_class_loop_seq { 1 } }
9660         { \seq_item:Nn \l__msg_class_loop_seq { 2 } }
9661         {#3}
9662         {
9663             \seq_map_function:NN \l__msg_class_loop_seq
9664             \__msg_redirect_loop_list:n
9665             { \seq_item:Nn \l__msg_class_loop_seq { 1 } }
9666         }
9667     }
9668     { \__msg_redirect_loop_chk:onn \l__msg_class_tl {#2} {#3} }
9669 }
9670 }
9671 }
9672 \cs_generate_variant:Nn \__msg_redirect_loop_chk:nnn { o }
9673 \cs_new:Npn \__msg_redirect_loop_list:n #1 { {#1} ~ => ~ }

```

(End definition for `\msg_redirect_class:nn` and others. These functions are documented on page 84.)

48.6 Kernel-specific functions

These are all retained purely for older xparse support.

```

\__kernel_msg_new:nnnn
\__kernel_msg_new:nnn
9674 \cs_new_protected:Npn \__kernel_msg_new:nnnn #1
9675   { \msg_new:nnnn { LaTeX / #1 } }
9676 \cs_new_protected:Npn \__kernel_msg_new:nnn #1
9677   { \msg_new:nnn { LaTeX / #1 } }

```

(End definition for `__kernel_msg_new:nnnn` and `__kernel_msg_new:nnn`.)

```

\__kernel_msg_info:nnxx
\__kernel_msg_warning:nnx
\__kernel_msg_warning:nnxx
\__kernel_msg_error:nnx
\__kernel_msg_error:nnxx
\__kernel_msg_error:nnxxx
9678 \cs_new_protected:Npn \__kernel_msg_info:nnxx #1
9679   { \msg_info:nnxx { LaTeX / #1 } }
9680 \cs_new_protected:Npn \__kernel_msg_warning:nnx #1
9681   { \msg_warning:nnx { LaTeX / #1 } }
9682 \cs_new_protected:Npn \__kernel_msg_warning:nnxx #1
9683   { \msg_warning:nnxx { LaTeX / #1 } }
9684 \cs_new_protected:Npn \__kernel_msg_error:nnx #1
9685   { \msg_error:nnx { LaTeX / #1 } }
9686 \cs_new_protected:Npn \__kernel_msg_error:nnxx #1
9687   { \msg_error:nnxx { LaTeX / #1 } }
9688 \cs_new_protected:Npn \__kernel_msg_error:nnxxx #1
9689   { \msg_error:nnxxx { LaTeX / #1 } }

```

(End definition for `__kernel_msg_info:nnxx` and others.)

```

\__kernel_msg_expandable_error:nnn
\__kernel_msg_expandable_error:nnf
\__kernel_msg_expandable_error:nnff
9690 \cs_new:Npn \__kernel_msg_expandable_error:nnn #1
9691   { \msg_expandable_error:nnn { LaTeX / #1 } }
9692 \cs_new:Npn \__kernel_msg_expandable_error:nnf #1
9693   { \msg_expandable_error:nnf { LaTeX / #1 } }
9694 \cs_new:Npn \__kernel_msg_expandable_error:nnff #1
9695   { \msg_expandable_error:nnff { LaTeX / #1 } }

```

(End definition for `__kernel_msg_expandable_error:nnn` and `__kernel_msg_expandable_error:nnff`.)

48.7 Internal messages

Error messages needed to actually implement the message system itself.

```

9696 \msg_new:nnnn { msg } { already-defined }
9697 { Message~'#2'~for~module~'#1'~already-defined. }
9698 {
9699   \c_msg_coding_error_text_tl
9700   LaTeX~was~asked~to~define~a~new~message~called~'#2'\
9701   by~the~module~'#1':~this~message~already~exists.
9702   \c_msg_return_text_tl
9703 }
9704 \msg_new:nnnn { msg } { unknown }
9705 { Unknown~message~'#2'~for~module~'#1'. }
9706 {
9707   \c_msg_coding_error_text_tl
9708   LaTeX~was~asked~to~display~a~message~called~'#2'\
9709   by~the~module~'#1':~this~message~does~not~exist.
9710   \c_msg_return_text_tl
9711 }
9712 \msg_new:nnnn { msg } { class-unknown }
9713 { Unknown~message~class~'#1'. }
9714 {
9715   LaTeX~has~been~asked~to~redirect~messages~to~a~class~'#1':\
9716   this~was~never~defined.
9717   \c_msg_return_text_tl
9718 }
9719 \msg_new:nnnn { msg } { redirect-loop }
9720 {
9721   Message~redirection~loop~caused~by~ {#1} ~=>~ {#2}
9722   \tl_if_empty:nF {#3} { ~for~module~' \use_none:n #3 ' } .
9723 }
9724 {
9725   Adding~the~message~redirection~ {#1} ~=>~ {#2}
9726   \tl_if_empty:nF {#3} { ~for~the~module~' \use_none:n #3 ' } ~
9727   created~an~infinite~loop\\\
9728   \iow_indent:n { #4 \\\ }
9729 }

```

Messages for earlier kernel modules plus a few for l3keys which cover coding errors.

```

9730 \msg_new:nnnn { kernel } { bad-number-of-arguments }
9731 { Function~'#1'~cannot~be~defined~with~#2~arguments. }
9732 {
9733   \c_msg_coding_error_text_tl
9734   LaTeX~has~been~asked~to~define~a~function~'#1'~with~
9735   #2~arguments.~
9736   TeX~allows~between~0~and~9~arguments~for~a~single~function.
9737 }
9738 \msg_new:nnnn { kernel } { command-already-defined }
9739 { Control~sequence~#1~already~defined. }
9740 {
9741   \c_msg_coding_error_text_tl
9742   LaTeX~has~been~asked~to~create~a~new~control~sequence~'#1'~
9743   but~this~name~has~already~been~used~elsewhere. \ \
9744   The~current~meaning~is:\

```

```

9745     \ \ #2
9746   }
9747 \msg_new:nnnn { kernel } { command-not-defined }
9748 { Control~sequence~#1~undefined. }
9749 {
9750   \c__msg_coding_error_text_tl
9751   LaTeX~has~been~asked~to~use~a~control~sequence~'#1':\
9752   this~has~not~been~defined~yet.
9753 }
9754 \msg_new:nnnn { kernel } { empty-search-pattern }
9755 { Empty~search~pattern. }
9756 {
9757   \c__msg_coding_error_text_tl
9758   LaTeX~has~been~asked~to~replace~an~empty~pattern~by~'#1':~that~
9759   would~lead~to~an~infinite~loop!
9760 }
9761 \msg_new:nnnn { kernel } { non-base-function }
9762 { Function~'#1'~is~not~a~base~function }
9763 {
9764   \c__msg_coding_error_text_tl
9765   Functions~defined~through~\iow_char:N\cs_new:Nn~must~have~
9766   a~signature~consisting~of~only~normal~arguments~'N'~and~'n'.~
9767   The~signature~'#2'~of~'#1'~contains~other~arguments~'#3'.~
9768   To~define~variants~use~\iow_char:N\cs_generate_variant:Nn~
9769   and~to~define~other~functions~use~\iow_char:N\cs_new:Npn.
9770 }
9771 \msg_new:nnnn { kernel } { missing-colon }
9772 { Function~'#1'~contains~no~':'. }
9773 {
9774   \c__msg_coding_error_text_tl
9775   Code~level~functions~must~contain~':'~to~separate~the~
9776   argument~specification~from~the~function~name.~This~is~
9777   needed~when~defining~conditionals~or~variants,~or~when~building~a~
9778   parameter~text~from~the~number~of~arguments~of~the~function.
9779 }
9780 \msg_new:nnnn { kernel } { overflow }
9781 { Integers~larger~than~2{30}-1~cannot~be~stored~in~arrays. }
9782 {
9783   An~attempt~was~made~to~store~#3~
9784   \tl_if_empty:nF {#2} { at~position~#2~ } in~the~array~'#1'.~
9785   The~largest~allowed~value~#4~will~be~used~instead.
9786 }
9787 \msg_new:nnnn { kernel } { out-of-bounds }
9788 { Access~to~an~entry~beyond~an~array's~bounds. }
9789 {
9790   An~attempt~was~made~to~access~or~store~data~at~position~#2~of~the~
9791   array~'#1',~but~this~array~has~entries~at~positions~from~1~to~#3.
9792 }
9793 \msg_new:nnnn { kernel } { protected-predicate }
9794 { Predicate~'#1'~must~be~expandable. }
9795 {
9796   \c__msg_coding_error_text_tl
9797   LaTeX~has~been~asked~to~define~'#1'~as~a~protected~predicate.~
9798   Only~expandable~tests~can~have~a~predicate~version.

```

```

9799 }
9800 \msg_new:nnn { kernel } { randint-backward-range }
9801 { Wrong~order~of~bounds~in~\iow_char:N\int_rand:nn{#1}{#2}. }
9802 \msg_new:nnnn { kernel } { conditional-form-unknown }
9803 { Conditional~form~'#1'~for~function~'#2'~unknown. }
9804 {
9805   \c__msg_coding_error_text_tl
9806   LaTeX~has~been~asked~to~define~the~conditional~form~'#1'~of~
9807   the~function~'#2',~but~only~'TF',~'T',~'F',~and~'p'~forms~exist.
9808 }
9809 \msg_new:nnnn { kernel } { variant-too-long }
9810 { Variant~form~'#1'~longer~than~base~signature~of~'#2'. }
9811 {
9812   \c__msg_coding_error_text_tl
9813   LaTeX~has~been~asked~to~create~a~variant~of~the~function~'#2'~
9814   with~a~signature~starting~with~'#1',~but~that~is~longer~than~
9815   the~signature~(part~after~the~colon)~of~'#2'.
9816 }
9817 \msg_new:nnnn { kernel } { invalid-variant }
9818 { Variant~form~'#1'~invalid~for~base~form~'#2'. }
9819 {
9820   \c__msg_coding_error_text_tl
9821   LaTeX~has~been~asked~to~create~a~variant~of~the~function~'#2'~
9822   with~a~signature~starting~with~'#1',~but~cannot~change~an~argument~
9823   from~type~'#3'~to~type~'#4'.
9824 }
9825 \msg_new:nnnn { kernel } { invalid-exp-args }
9826 { Invalid~variant~specifier~'#1'~in~'#2'. }
9827 {
9828   \c__msg_coding_error_text_tl
9829   LaTeX~has~been~asked~to~create~an~\iow_char:N\exp_args:N...~
9830   function~with~signature~'N#2'~but~'#1'~is~not~a~valid~argument~
9831   specifier.
9832 }
9833 \msg_new:nnn { kernel } { deprecated-variant }
9834 {
9835   Variant~form~'#1'~deprecated~for~base~form~'#2'.~
9836   One~should~not~change~an~argument~from~type~'#3'~to~type~'#4'
9837   \str_case:nnF {#3}
9838   {
9839     { n } { :~use~a~'\token_if_eq_charcode:NNTF #4 c v V'~variant? }
9840     { N } { :~base~form~only~accepts~a~single~token~argument. }
9841     {#4} { :~base~form~is~already~a~variant. }
9842   } { . }
9843 }
9844 \msg_new:nnn { char } { active }
9845 { Cannot~generate~active~chars. }
9846 \msg_new:nnn { char } { invalid-catcode }
9847 { Invalid~catcode~for~char~generation. }
9848 \msg_new:nnn { char } { null-space }
9849 { Cannot~generate~null~char~as~a~space. }
9850 \msg_new:nnn { char } { out-of-range }
9851 { Charcode~requested~out~of~engine~range. }
9852 \msg_new:nnn { char } { space }

```

```

9853 { Cannot~generate~space~chars. }
9854 \msg_new:nnnn { ior } { quote-in-shell }
9855 { Quotes~in~shell~command~'#1'. }
9856 { Shell~commands~cannot~contain~quotes~("). }
9857 \msg_new:nnnn { keys } { no-property }
9858 { No~property~given~in~definition~of~key~'#1'. }
9859 {
9860   \c__msg_coding_error_text_tl
9861   Inside~\keys_define:nn each~key~name~
9862   needs~a~property: \ \ \
9863   \iow_indent:n { #1 .<property> } \ \ \
9864   LaTeX~did~not~find~a~'. '~to~indicate~the~start~of~a~property.
9865 }
9866 \msg_new:nnnn { keys } { property-boolean-values-only }
9867 { The~property~'#1'~accepts~boolean~values~only. }
9868 {
9869   \c__msg_coding_error_text_tl
9870   The~property~'#1'~only~accepts~the~values~'true'~and~'false'.
9871 }
9872 \msg_new:nnnn { keys } { property-requires-value }
9873 { The~property~'#1'~requires~a~value. }
9874 {
9875   \c__msg_coding_error_text_tl
9876   LaTeX~was~asked~to~set~property~'#1'~for~key~'#2'. \
9877   No~value~was~given~for~the~property,~and~one~is~required.
9878 }
9879 \msg_new:nnnn { keys } { property-unknown }
9880 { The~key~property~'#1'~is~unknown. }
9881 {
9882   \c__msg_coding_error_text_tl
9883   LaTeX~has~been~asked~to~set~the~property~'#1'~for~key~'#2':~
9884   this~property~is~not~defined.
9885 }
9886 \msg_new:nnnn { quark } { invalid-function }
9887 { Quark~test~function~'#1'~is~invalid. }
9888 {
9889   \c__msg_coding_error_text_tl
9890   LaTeX~has~been~asked~to~create~quark~test~function~'#1'~
9891   \tl_if_empty:nTF {#2}
9892     { but~that~name~ }
9893     { with~signature~'#2',~but~that~signature~ }
9894     is~not~valid.
9895 }
9896 \__kernel_msg_new:nnn { quark } { invalid }
9897 { Invalid~quark~variable~'#1'. }
9898 \msg_new:nnnn { scanmark } { already-defined }
9899 { Scan~mark~'#1'~already~defined. }
9900 {
9901   \c__msg_coding_error_text_tl
9902   LaTeX~has~been~asked~to~create~a~new~scan~mark~'#1'~
9903   but~this~name~has~already~been~used~for~a~scan~mark.
9904 }
9905 \msg_new:nnnn { seq } { shuffle-too-large }
9906 { The~sequence~'#1'~is~too~long~to~be~shuffled~by~TeX. }

```

```

9907 {
9908   TeX~has~ \int_eval:n { \c_max_register_int + 1 } ~
9909   toks~registers:~this~only~allows~to~shuffle~up~to~
9910   \int_use:N \c_max_register_int \ items.~
9911   The~list~will~not~be~shuffled.
9912 }
9913 \msg_new:nnnn { kernel } { variable-not-defined }
9914 { Variable~'#1~undefined. }
9915 {
9916   \c__msg_coding_error_text_tl
9917   LaTeX~has~been~asked~to~show~a~variable~'#1,~but~this~has~not~
9918   been~defined~yet.
9919 }
9920 \msg_new:nnnn { kernel } { bad-type }
9921 { Variable~'#1'~is~not~a~valid~#3. }
9922 {
9923   \c__msg_coding_error_text_tl
9924   The~variable~'#1'~with~\tl_if_empty:nTF {#4} {meaning} {value}\\\\
9925   \iow_indent:n {#2}\\\\
9926   should~be~a~#3~variable,~but~
9927   \tl_if_empty:nTF {#4}
9928   { it~is~not \str_if_eq:nnF {#3} { bool } { ~a~short~macro } . }
9929   {
9930     it~does~not~have~the~correct~
9931     \str_if_eq:nnTF {#2} {#4}
9932     { category~codes. }
9933     { internal~structure:\\\\\iow_indent:n {#4} }
9934   }
9935 }
9936 \msg_new:nnnn { clist } { non-clist }
9937 { Variable~'#1'~is~not~a~valid~clist. }
9938 {
9939   \c__msg_coding_error_text_tl
9940   The~variable~'#1'~with~value\\\\
9941   \iow_indent:n {#2}\\\\
9942   should~be~a~clist~variable,~but~it~includes~empty~or~blank~items~
9943   without~braces.
9944 }

```

Some errors are only needed in package mode if debugging is enabled by one of the options `enable-debug`, `check-declarations`, `log-functions`, or on the contrary if debugging is turned off. In format mode the error is somewhat different.

```

9945 \msg_new:nnnn { debug } { enable-debug }
9946 { To~use~'#1'~set~the~'enable-debug'~option. }
9947 {
9948   The~function~'#1'~will~be~ignored~because~it~can~only~work~if~
9949   some~internal~functions~in~expl3~have~been~appropriately~
9950   defined.~This~only~happens~if~one~of~the~options~
9951   'enable-debug',~'check-declarations'~or~'log-functions'~was~
9952   given~as~an~option:~see~the~main~expl3~documentation.
9953 }

```

Some errors only appear in expandable settings, hence don't need a “more-text” argument.

```

9954 \msg_new:nnn { kernel } { bad-exp-end-f }

```

```

9955 { Misused~\exp_end_continue_f:w or~:nw }
9956 \msg_new:nnn { kernel } { bad-variable }
9957 { Erroneous-variable~#1 used! }
9958 \msg_new:nnn { seq } { misused }
9959 { A~sequence~was~misused. }
9960 \msg_new:nnn { prop } { misused }
9961 { A~property~list~was~misused. }
9962 \msg_new:nnn { prg } { negative-replication }
9963 { Negative-argument-for~\iow_char:N\prg_replicate:nn. }
9964 \msg_new:nnn { prop } { prop-keyval }
9965 { Missing~'='~in~'#1'~(in~'..._keyval:Nn') }
9966 \msg_new:nnn { kernel } { unknown-comparison }
9967 { Relation~'#1'~not~among~<,>,<=>,<=>,<=>. }
9968 \msg_new:nnn { kernel } { zero-step }
9969 { Zero-step-size-for-function~#1. }
9970 \cs_if_exist:NF \tex_expanded:D
9971 {
9972   \msg_new:nnn { kernel } { e-type }
9973   { #1 ~ in~e-type~argument }
9974 }

```

Messages used by the “show” functions.

```

9975 \msg_new:nnn { clist } { show }
9976 {
9977   The~comma~list~ \tl_if_empty:nF {#1} { #1 ~ }
9978   \tl_if_empty:nTF {#2}
9979   { is~empty \>~ . }
9980   { contains~the~items~(without~outer~braces): #2 . }
9981 }
9982 \msg_new:nnn { intarray } { show }
9983 { The~integer~array~#1~contains~#2~items: \> #3 . }
9984 \msg_new:nnn { prop } { show }
9985 {
9986   The~property~list~#1~
9987   \tl_if_empty:nTF {#2}
9988   { is~empty \>~ . }
9989   { contains~the~pairs~(without~outer~braces): #2 . }
9990 }
9991 \msg_new:nnn { seq } { show }
9992 {
9993   The~sequence~#1~
9994   \tl_if_empty:nTF {#2}
9995   { is~empty \>~ . }
9996   { contains~the~items~(without~outer~braces): #2 . }
9997 }
9998 \msg_new:nnn { kernel } { show-streams }
9999 {
10000   \tl_if_empty:nTF {#2} { No~ } { The~following~ }
10001   \str_case:nn {#1}
10002   {
10003     { ior } { input ~ }
10004     { iow } { output ~ }
10005   }
10006   streams~are~
10007   \tl_if_empty:nTF {#2} { open } { in~use: #2 . }

```

```

10008 }
      System layer messages
10009 \msg_new:nnnn { sys } { backend-set }
10010 { Backend~configuration~already~set. }
10011 {
10012   Run-time~backend~selection~may~only~be~carried-out~once~during~a~run.~
10013   This~second~attempt~to~set~them~will~be~ignored.
10014 }
10015 \msg_new:nnnn { sys } { wrong-backend }
10016 { Backend~request~inconsistent~with~engine:~using~'~#2'~backend. }
10017 {
10018   You~have~requested~backend~'~#1'~,~but~this~is~not~suitable~for~use~with~the~
10019   active~engine.~LaTeX3~will~use~the~'~#2'~backend~instead.
10020 }

```

48.8 Expandable errors

`_msg_expandable_error:nn` In expansion only context, we cannot use the normal means of reporting errors. Instead, we rely on a low-level T_EX error caused by expanding a macro `\???` with parameter text “?” (this could be any token) which we used followed by something else (here, a space). This shows the context, which thanks to the odd-looking `\use:n` is

```

<argument> \???
                ! mypkg Error: The error message.

```

In other words, T_EX is processing the argument of `\use:n`, which is `\??? <space> ! <error type> : <error message>`.

```

10021 \cs_set_protected:Npn \_msg_tmp:w #1
10022 {
10023   \cs_new:Npn #1 ? { }
10024   \cs_new:Npn \_msg_expandable_error:nn ##1##2
10025   {
10026     \exp_after:wN \exp_after:wN
10027     \exp_after:wN \_msg_use_none_delimit_by_s_stop:w
10028     \use:n { #1 ~ ! ~ ##2 : ~ ##1 } \s_msg_stop
10029   }
10030 }
10031 \exp_args:Nc \_msg_tmp:w { ??? }

```

(End definition for `_msg_expandable_error:nn`.)

`\msg_expandable_error:nnnnnn` The command built from the csname `\c__msg_text_prefix_tl #1 / #2` takes four arguments and builds the error text, which is fed to `_msg_expandable_error:n` with appropriate expansion: just as for usual messages the arguments are first turned to strings, then the message is fully expanded. The module name also has to be determined.

```

10032 \exp_args_generate:n { oooo }
10033 \cs_new:Npn \msg_expandable_error:nnnnnn #1#2#3#4#5#6
10034 {
10035   \exp_args:Nee \_msg_expandable_error:nn
10036   {
10037     \exp_args:Nc \exp_args:Noooo
10038     { \c__msg_text_prefix_tl #1 / #2 }

```

```

10039         { \tl_to_str:n {#3} }
10040         { \tl_to_str:n {#4} }
10041         { \tl_to_str:n {#5} }
10042         { \tl_to_str:n {#6} }
10043     }
10044     { \msg_error_text:n {#1} }
10045 }
10046 \cs_new:Npn \msg_expandable_error:nnnnn #1#2#3#4#5
10047 { \msg_expandable_error:nnnnnn {#1} {#2} {#3} {#4} {#5} { } }
10048 \cs_new:Npn \msg_expandable_error:nnnn #1#2#3#4
10049 { \msg_expandable_error:nnnnnn {#1} {#2} {#3} {#4} { } { } }
10050 \cs_new:Npn \msg_expandable_error:nnn #1#2#3
10051 { \msg_expandable_error:nnnnnn {#1} {#2} {#3} { } { } { } }
10052 \cs_new:Npn \msg_expandable_error:nn #1#2
10053 { \msg_expandable_error:nnnnnn {#1} {#2} { } { } { } { } }
10054 \cs_generate_variant:Nn \msg_expandable_error:nnnnnn { nnffff }
10055 \cs_generate_variant:Nn \msg_expandable_error:nnnnnn { nnfff }
10056 \cs_generate_variant:Nn \msg_expandable_error:nnnn { nnff }
10057 \cs_generate_variant:Nn \msg_expandable_error:nnn { nnf }

```

(End definition for `\msg_expandable_error:nnnnnn` and others. These functions are documented on page 83.)

48.9 Message formatting

```

10058 \prop_gput:Nnn \g_msg_module_name_prop { kernel } { LaTeX3 }
10059 \prop_gput:Nnn \g_msg_module_type_prop { kernel } { }
10060 \clist_map_inline:nn
10061 {
10062     char , clist , coffin , debug , deprecation , msg ,
10063     quark , prg , prop , scanmark , seq , sys
10064 }
10065 {
10066     \prop_gput:Nnn \g_msg_module_name_prop {#1} { LaTeX3 }
10067     \prop_gput:Nnn \g_msg_module_type_prop {#1} { }
10068 }
10069 \prop_gput:Nnn \g_msg_module_name_prop { LaTeX / cmd } { LaTeX3 }
10070 \prop_gput:Nnn \g_msg_module_type_prop { LaTeX / cmd } { }
10071 \prop_gput:Nnn \g_msg_module_name_prop { LaTeX / ltcmd } { LaTeX3 }
10072 \prop_gput:Nnn \g_msg_module_type_prop { LaTeX / ltcmd } { }
10073 </package>

```


Chapter 49

l3file implementation

The following test files are used for this code: m3file001.

```
10074 <*>package>
```

49.1 Input operations

```
10075 <@@=ior>
```

49.1.1 Variables and constants

`\l__ior_internal_tl` Used as a short-term scratch variable.

```
10076 \tl_new:N \l__ior_internal_tl
```

(End definition for \l__ior_internal_tl.)

`\c__ior_term_ior` Reading from the terminal (with a prompt) is done using a positive but non-existent stream number. Unlike writing, there is no concept of reading from the log.

```
10077 \int_const:Nn \c__ior_term_ior { 16 }
```

(End definition for \c__ior_term_ior.)

`\g__ior_streams_seq` A list of the currently-available input streams to be used as a stack.

```
10078 \seq_new:N \g__ior_streams_seq
```

(End definition for \g__ior_streams_seq.)

`\l__ior_stream_tl` Used to recover the raw stream number from the stack.

```
10079 \tl_new:N \l__ior_stream_tl
```

(End definition for \l__ior_stream_tl.)

`\g__ior_streams_prop` The name of the file attached to each stream is tracked in a property list. To get the correct number of reserved streams in package mode the underlying mechanism needs to be queried. For L^AT_EX 2_ε and plain T_EX this data is stored in `\count16`; with the etex package loaded we need to subtract 1 as the register holds the number of the next stream to use. In ConT_EXt, we need to look at `\count38` but there is no subtraction: like the original plain T_EX/L^AT_EX 2_ε mechanism it holds the value of the *last* stream allocated.

```
10080 \prop_new:N \g__ior_streams_prop
```

```

10081 \int_step_inline:nnn
10082 { 0 }
10083 {
10084   \cs_if_exist:NTF \normalend
10085   { \tex_count:D 38 ~ }
10086   {
10087     \tex_count:D 16 ~ %
10088     \cs_if_exist:NT \loccount { - 1 }
10089   }
10090 }
10091 {
10092   \prop_gput:Nnn \g__ior_streams_prop {#1} { Reserved-by~format }
10093 }

```

(End definition for `\g__ior_streams_prop`.)

49.1.2 Stream management

`\ior_new:N` Reserving a new stream is done by defining the name as equal to using the terminal.
`\ior_new:c`

```

10094 \cs_new_protected:Npn \ior_new:N #1 { \cs_new_eq:NN #1 \c__ior_term_ior }
10095 \cs_generate_variant:Nn \ior_new:N { c }

```

(End definition for `\ior_new:N`. This function is documented on page 86.)

`\g_tmpa_ior` The usual scratch space.
`\g_tmpb_ior`

```

10096 \ior_new:N \g_tmpa_ior
10097 \ior_new:N \g_tmpb_ior

```

(End definition for `\g_tmpa_ior` and `\g_tmpb_ior`. These variables are documented on page 93.)

`\ior_open:Nn` Use the conditional version, with an error if the file is not found.
`\ior_open:cn`

```

10098 \cs_new_protected:Npn \ior_open:Nn #1#2
10099 { \ior_open:NnF #1 {#2} { \__kernel_file_missing:n {#2} } }
10100 \cs_generate_variant:Nn \ior_open:Nn { c }

```

(End definition for `\ior_open:Nn`. This function is documented on page 86.)

`\l__ior_file_name_tl` Data storage.

```

10101 \tl_new:N \l__ior_file_name_tl

```

(End definition for `\l__ior_file_name_tl`.)

`\ior_open:NnTF` An auxiliary searches for the file in the \TeX , $\text{\LaTeX} 2_{\epsilon}$ and $\text{\LaTeX} 3$ paths. Then pass the file found to the lower-level function which deals with streams. The `full_name` is empty when the file is not found.

```

10102 \prg_new_protected_conditional:Npnn \ior_open:Nn #1#2 { T , F , TF }
10103 {
10104   \file_get_full_name:nNTF {#2} \l__ior_file_name_tl
10105   {
10106     \__kernel_ior_open:No #1 \l__ior_file_name_tl
10107     \prg_return_true:
10108   }
10109   { \prg_return_false: }
10110 }
10111 \prg_generate_conditional_variant:Nnn \ior_open:Nn { c } { T , F , TF }

```

(End definition for \ior_open:NnTF. This function is documented on page 86.)

`__ior_new:N` Streams are reserved using `\newread` before they can be managed by `ior`. To prevent `ior` from being affected by redefinitions of `\newread` (such as done by the third-party package `morewrites`), this macro is saved here under a private name. The complicated code ensures that `__ior_new:N` is not `\outer` despite plain `TeX`'s `\newread` being `\outer`. For `ConTeXt`, we have to deal with the fact that `\newread` works like our own: it actually checks before altering definition.

```

10112 \exp_args:NNf \cs_new_protected:Npn \__ior_new:N
10113   { \exp_args:NNc \exp_after:wN \exp_stop_f: { newread } }
10114 \cs_if_exist:NT \normalend
10115   {
10116     \cs_new_eq:NN \__ior_new_aux:N \__ior_new:N
10117     \cs_set_protected:Npn \__ior_new:N #1
10118       {
10119         \cs_undefine:N #1
10120         \__ior_new_aux:N #1
10121       }
10122   }

```

(End definition for __ior_new:N.)

`__kernel_ior_open:Nn` The stream allocation itself uses the fact that there is a list of all of those available.
`__kernel_ior_open:No` Life gets more complex as it's important to keep things in sync. That is done using a
`__ior_open_stream:Nn` two-part approach: any streams that have already been taken up by `ior` but are now free are tracked, so we first try those. If that fails, ask plain `TeX` or `LATeX 2ε` for a new stream and use that number (after a bit of conversion).

```

10123 \cs_new_protected:Npn \__kernel_ior_open:Nn #1#2
10124   {
10125     \ior_close:N #1
10126     \seq_gpop:NNTF \g__ior_streams_seq \l__ior_stream_tl
10127     { \__ior_open_stream:Nn #1 {#2} }
10128     {
10129       \__ior_new:N #1
10130       \__kernel_tl_set:Nx \l__ior_stream_tl { \int_eval:n {#1} }
10131       \__ior_open_stream:Nn #1 {#2}
10132     }
10133   }
10134 \cs_generate_variant:Nn \__kernel_ior_open:Nn { No }

```

Here, we act defensively in case `LuaTeX` is in use with an extensionless file name.

```

10135 \cs_new_protected:Npx \__ior_open_stream:Nn #1#2
10136   {
10137     \tex_global:D \tex_chardef:D #1 = \exp_not:N \l__ior_stream_tl \scan_stop:
10138     \prop_gput:NVn \exp_not:N \g__ior_streams_prop #1 {#2}
10139     \tex_openin:D #1
10140     \sys_if_engine luatex:TF
10141       { {#2} }
10142       { \exp_not:N \__kernel_file_name_quote:n {#2} \scan_stop: }
10143   }

```

(End definition for __kernel_ior_open:Nn and __ior_open_stream:Nn.)

\ior_close:N Closing a stream means getting rid of it at the T_EX level and removing from the various data structures. Unless the name passed is an invalid stream number (outside the range [0,15]), it can be closed. On the other hand, it only gets added to the stack if it was not already there, to avoid duplicates building up.

```

10144 \cs_new_protected:Npn \ior_close:N #1
10145 {
10146   \int_compare:nT { -1 < #1 < \c__ior_term_ior }
10147   {
10148     \tex_closein:D #1
10149     \prop_gremove:NV \g__ior_streams_prop #1
10150     \seq_if_in:NVF \g__ior_streams_seq #1
10151     { \seq_gpush:NV \g__ior_streams_seq #1 }
10152     \cs_gset_eq:NN #1 \c__ior_term_ior
10153   }
10154 }
10155 \cs_generate_variant:Nn \ior_close:N { c }

```

(End definition for \ior_close:N. This function is documented on page 86.)

\ior_show:N Seek the stream in the \g__ior_streams_prop list, then show the stream as open or closed accordingly.

```

10156 \cs_new_protected:Npn \ior_show:N { \__ior_show:NN \tl_show:n }
10157 \cs_generate_variant:Nn \ior_show:N { c }
10158 \cs_new_protected:Npn \ior_log:N { \__ior_show:NN \tl_log:n }
10159 \cs_generate_variant:Nn \ior_log:N { c }
10160 \cs_new_protected:Npn \__ior_show:NN #1#2
10161 {
10162   \__kernel_chk_defined:NT #2
10163   {
10164     \prop_get:NVNTF \g__ior_streams_prop #2 \l__ior_internal_tl
10165     {
10166       \exp_args:Nx #1
10167       { \token_to_str:N #2 ~ open: ~ \l__ior_internal_tl }
10168     }
10169     { \exp_args:Nx #1 { \token_to_str:N #2 ~ closed } }
10170   }
10171 }

```

(End definition for \ior_show:N, \ior_log:N, and __ior_show:NN. These functions are documented on page 86.)

\ior_show_list: Show the property lists, but with some “pretty printing”. See the l3msg module. The first argument of the message is ior (as opposed to iow) and the second is empty if no read stream is open and non-empty (the list of streams formatted using \msg_show_item_unbraced:nn) otherwise. The code of the message show-streams takes care of translating ior/iow to English.

```

10172 \cs_new_protected:Npn \ior_show_list: { \__ior_list:N \msg_show:nnxxxx }
10173 \cs_new_protected:Npn \ior_log_list: { \__ior_list:N \msg_log:nnxxxx }
10174 \cs_new_protected:Npn \__ior_list:N #1
10175 {
10176   #1 { kernel } { show-streams }
10177   { ior }
10178   {
10179     \prop_map_function:NN \g__ior_streams_prop

```

```

10180         \msg_show_item_unbraced:nn
10181     }
10182     { } { }
10183 }

```

(End definition for `\ior_show_list:`, `\ior_log_list:`, and `_ior_list:N`. These functions are documented on page 87.)

49.1.3 Reading input

`\if_eof:w` The primitive conditional

```

10184 \cs_new_eq:NN \if_eof:w \tex_ifeof:D

```

(End definition for `\if_eof:w`. This function is documented on page 93.)

`\ior_if_eof_p:N` To test if some particular input stream is exhausted the following conditional is provided.
`\ior_if_eof:NTF` The primitive test can only deal with numbers in the range $[0, 15]$ so we catch outliers (they are exhausted).

```

10185 \prg_new_conditional:Npnn \ior_if_eof:N #1 { p , T , F , TF }
10186 {
10187     \if_int_compare:w -1 < #1
10188         \if_int_compare:w #1 < \c__ior_term_ior
10189             \if_eof:w #1
10190                 \prg_return_true:
10191             \else:
10192                 \prg_return_false:
10193             \fi:
10194         \else:
10195             \prg_return_true:
10196         \fi:
10197     \else:
10198         \prg_return_true:
10199     \fi:
10200 }

```

(End definition for `\ior_if_eof:NTF`. This function is documented on page 90.)

`\ior_get:NN` And here we read from files.

```

\__ior_get:NN
\ior_get:NNTF
10201 \cs_new_protected:Npn \ior_get:NN #1#2
10202 { \ior_get:NNF #1 #2 { \tl_set:Nn #2 { \q_no_value } } }
10203 \cs_new_protected:Npn \__ior_get:NN #1#2
10204 { \tex_read:D #1 to #2 }
10205 \prg_new_protected_conditional:Npnn \ior_get:NN #1#2 { T , F , TF }
10206 {
10207     \ior_if_eof:NTF #1
10208     { \prg_return_false: }
10209     {
10210         \__ior_get:NN #1 #2
10211         \prg_return_true:
10212     }
10213 }

```

(End definition for `\ior_get:NN`, `__ior_get:NN`, and `\ior_get:NNTF`. These functions are documented on page 87.)

`\ior_str_get:NN` Reading as strings is a more complicated wrapper, as we wish to remove the endline character and restore it afterwards.

```

\__ior_str_get:NN
\ior_str_get:NNTF
10214 \cs_new_protected:Npn \ior_str_get:NN #1#2
10215 { \ior_str_get:NNF #1 #2 { \tl_set:Nn #2 { \q_no_value } } }
10216 \cs_new_protected:Npn \__ior_str_get:NN #1#2
10217 {
10218   \exp_args:Nno \use:n
10219   {
10220     \int_set:Nn \tex_endlinechar:D { -1 }
10221     \tex_readline:D #1 to #2
10222     \int_set:Nn \tex_endlinechar:D
10223     } { \int_use:N \tex_endlinechar:D }
10224   }
10225 \prg_new_protected_conditional:Npnn \ior_str_get:NN #1#2 { T , F , TF }
10226 {
10227   \ior_if_eof:NTF #1
10228   { \prg_return_false: }
10229   {
10230     \__ior_str_get:NN #1 #2
10231     \prg_return_true:
10232   }
10233 }

```

(End definition for `\ior_str_get:NN`, `__ior_str_get:NN`, and `\ior_str_get:NNTF`. These functions are documented on page 88.)

`\c__ior_term_noprompt_ior` For reading without a prompt.

```

10234 \int_const:Nn \c__ior_term_noprompt_ior { -1 }

```

(End definition for `\c__ior_term_noprompt_ior`.)

`\ior_get_term:nN` Getting from the terminal is better with pretty-printing.

```

\ior_str_get_term:nN
\__ior_get_term:NnN
10235 \cs_new_protected:Npn \ior_get_term:nN #1#2
10236 { \__ior_get_term:NnN \__ior_get:NN {#1} #2 }
10237 \cs_new_protected:Npn \ior_str_get_term:nN #1#2
10238 { \__ior_get_term:NnN \__ior_str_get:NN {#1} #2 }
10239 \cs_new_protected:Npn \__ior_get_term:NnN #1#2#3
10240 {
10241   \group_begin:
10242   \tex_escapechar:D = -1 \scan_stop:
10243   \tl_if_blank:NTF {#2}
10244   { \exp_args:NNc #1 \c__ior_term_noprompt_ior }
10245   { \exp_args:NNc #1 \c__ior_term_ior }
10246   {#2}
10247   \exp_args:NNNv \group_end:
10248   \tl_set:Nn #3 {#2}
10249 }

```

(End definition for `\ior_get_term:nN`, `\ior_str_get_term:nN`, and `__ior_get_term:NnN`. These functions are documented on page 302.)

`\ior_map_break:` Usual map breaking functions.

```

\ior_map_break:n
10250 \cs_new:Npn \ior_map_break:
10251 { \prg_map_break:Nn \ior_map_break: { } }
10252 \cs_new:Npn \ior_map_break:n
10253 { \prg_map_break:Nn \ior_map_break: }

```

(End definition for `\ior_map_break:` and `\ior_map_break:n`. These functions are documented on page 89.)

```

\ior_map_inline:Nn Mapping over an input stream can be done on either a token or a string basis, hence the
\ior_str_map_inline:Nn set up. Within that, there is a check to avoid reading past the end of a file, hence the two
  \__ior_map_inline:NNn applications of \ior_if_eof:N and its lower-level analogue \if_eof:w. This mapping
  \__ior_map_inline:NNNn cannot be nested with twice the same stream, as the stream has only one “current line”.
\__ior_map_inline_loop:NNN
10254 \cs_new_protected:Npn \ior_map_inline:Nn
10255   { \__ior_map_inline:NNn \__ior_get:NN }
10256 \cs_new_protected:Npn \ior_str_map_inline:Nn
10257   { \__ior_map_inline:NNn \__ior_str_get:NN }
10258 \cs_new_protected:Npn \__ior_map_inline:NNn
10259   {
10260     \int_gincr:N \g__kernel_prg_map_int
10261     \exp_args:Nc \__ior_map_inline:NNNn
10262       { \__ior_map_ \int_use:N \g__kernel_prg_map_int :n }
10263   }
10264 \cs_new_protected:Npn \__ior_map_inline:NNNn #1#2#3#4
10265   {
10266     \cs_gset_protected:Npn #1 ##1 {#4}
10267     \ior_if_eof:NF #3 { \__ior_map_inline_loop:NNN #1#2#3 }
10268     \prg_break_point:Nn \ior_map_break:
10269       { \int_gdecr:N \g__kernel_prg_map_int }
10270   }
10271 \cs_new_protected:Npn \__ior_map_inline_loop:NNN #1#2#3
10272   {
10273     #2 #3 \l__ior_internal_tl
10274     \if_eof:w #3
10275       \exp_after:wN \ior_map_break:
10276     \fi:
10277     \exp_args:No #1 \l__ior_internal_tl
10278     \__ior_map_inline_loop:NNN #1#2#3
10279   }

```

(End definition for `\ior_map_inline:Nn` and others. These functions are documented on page 88.)

`\ior_map_variable:NNN` Since the TeX primitive (`\read` or `\readline`) assigns the tokens read in the same way
`\ior_str_map_variable:NNN` as a token list assignment, we simply call the appropriate primitive. The end-of-loop is
`__ior_map_variable:NNNn` checked using the primitive conditional for speed.

```

10280 \cs_new_protected:Npn \ior_map_variable:NNn
10281   { \__ior_map_variable:NNNn \ior_get:NN }
10282 \cs_new_protected:Npn \ior_str_map_variable:NNn
10283   { \__ior_map_variable:NNNn \ior_str_get:NN }
10284 \cs_new_protected:Npn \__ior_map_variable:NNNn #1#2#3#4
10285   {
10286     \ior_if_eof:NF #2 { \__ior_map_variable_loop:NNNn #1#2#3 {#4} }
10287     \prg_break_point:Nn \ior_map_break: { }
10288   }
10289 \cs_new_protected:Npn \__ior_map_variable_loop:NNNn #1#2#3#4
10290   {
10291     #1 #2 #3
10292     \if_eof:w #2
10293       \exp_after:wN \ior_map_break:
10294     \fi:

```

```

10295     #4
10296     \__ior_map_variable_loop:NNNn #1#2#3 {#4}
10297 }

```

(End definition for `\ior_map_variable:NNn` and others. These functions are documented on page 88.)

49.2 Output operations

```

10298 <@@=iow>

```

There is a lot of similarity here to the input operations, at least for many of the basics. Thus quite a bit is copied from the earlier material with minor alterations.

49.2.1 Variables and constants

`\l__iow_internal_tl` Used as a short-term scratch variable.

```

10299 \tl_new:N \l__iow_internal_tl

```

(End definition for `\l__iow_internal_tl`.)

`\c_log_iow` Here we allocate two output streams for writing to the transcript file only (`\c_log_iow`)
`\c_term_iow` and to both the terminal and transcript file (`\c_term_iow`). Recent LuaTeX provide 128 write streams; we also use `\c_term_iow` as the first non-allowed write stream so its value depends on the engine.

```

10300 \int_const:Nn \c_log_iow { -1 }
10301 \int_const:Nn \c_term_iow
10302 {
10303     \bool_lazy_and:nnTF
10304     { \sys_if_engine luatex_p: }
10305     { \int_compare_p:nNn \tex_luatexversion:D > { 80 } }
10306     { 128 }
10307     { 16 }
10308 }

```

(End definition for `\c_log_iow` and `\c_term_iow`. These variables are documented on page 93.)

`\g__iow_streams_seq` A list of the currently-available output streams to be used as a stack.

```

10309 \seq_new:N \g__iow_streams_seq

```

(End definition for `\g__iow_streams_seq`.)

`\l__iow_stream_tl` Used to recover the raw stream number from the stack.

```

10310 \tl_new:N \l__iow_stream_tl

```

(End definition for `\l__iow_stream_tl`.)

`\g__iow_streams_prop` As for reads with the appropriate adjustment of the register numbers to check on.

```

10311 \prop_new:N \g__iow_streams_prop
10312 \int_step_inline:nnn
10313 { 0 }
10314 {
10315     \cs_if_exist:NTF \normalend
10316     { \tex_count:D 39 ~ }
10317     {
10318         \tex_count:D 17 ~

```



```

10319         \cs_if_exist:NT \loccount { - 1 }
10320     }
10321 }
10322 {
10323     \prop_gput:Nnn \g__iow_streams_prop {#1} { Reserved~by~format }
10324 }

(End definition for \g__iow_streams_prop.)

```

49.2.2 Internal auxiliaries

`\s__iow_mark` Internal scan marks.

```

\s__iow_stop 10325 \scan_new:N \s__iow_mark
10326 \scan_new:N \s__iow_stop

```

(End definition for `\s__iow_mark` and `\s__iow_stop`.)

`__iow_use_i_delimit_by_s_stop:nw` Functions to gobble up to a scan mark.

```

10327 \cs_new:Npn \__iow_use_i_delimit_by_s_stop:nw #1 #2 \s__iow_stop {#1}

```

(End definition for `__iow_use_i_delimit_by_s_stop:nw`.)

`\q__iow_nil` Internal quarks.

```

10328 \quark_new:N \q__iow_nil

```

(End definition for `\q__iow_nil`.)

49.3 Stream management

`\iow_new:N` Reserving a new stream is done by defining the name as equal to writing to the terminal:
`\iow_new:c` odd but at least consistent.

```

10329 \cs_new_protected:Npn \iow_new:N #1 { \cs_new_eq:NN #1 \c_term_iow }
10330 \cs_generate_variant:Nn \iow_new:N { c }

```

(End definition for `\iow_new:N`. This function is documented on page 86.)

`\g_tmpa_iow` The usual scratch space.

```

\g_tmpb_iow 10331 \iow_new:N \g_tmpa_iow
10332 \iow_new:N \g_tmpb_iow

```

(End definition for `\g_tmpa_iow` and `\g_tmpb_iow`. These variables are documented on page 93.)

`__iow_new:N` As for read streams, copy `\newwrite`, making sure that it is not `\outer`.

```

10333 \exp_args:NNf \cs_new_protected:Npn \__iow_new:N
10334 { \exp_args:NNc \exp_after:wN \exp_stop_f: { newwrite } }

```

(End definition for `__iow_new:N`.)

`\l__iow_file_name_tl` Data storage.

```

10335 \tl_new:N \l__iow_file_name_tl

```

(End definition for `\l__iow_file_name_tl`.)

\iow_open:Nn The same idea as for reading, but without the path and without the need to allow for a conditional version.

\iow_open:cn

__iow_open_stream:Nn

__iow_open_stream:NV

```

10336 \cs_new_protected:Npn \iow_open:Nn #1#2
10337 {
10338   \__kernel_tl_set:Nx \l__iow_file_name_tl
10339   { \__kernel_file_name_sanitise:n {#2} }
10340   \iow_close:N #1
10341   \seq_gpop:NNTF \g__iow_streams_seq \l__iow_stream_tl
10342   { \__iow_open_stream:NV #1 \l__iow_file_name_tl }
10343   {
10344     \__iow_new:N #1
10345     \__kernel_tl_set:Nx \l__iow_stream_tl { \int_eval:n {#1} }
10346     \__iow_open_stream:NV #1 \l__iow_file_name_tl
10347   }
10348 }
10349 \cs_generate_variant:Nn \iow_open:Nn { c }
10350 \cs_new_protected:Npn \__iow_open_stream:Nn #1#2
10351 {
10352   \tex_global:D \tex_chardef:D #1 = \l__iow_stream_tl \scan_stop:
10353   \prop_gput:NVn \g__iow_streams_prop #1 {#2}
10354   \tex_immediate:D \tex_openout:D
10355   #1 \__kernel_file_name_quote:n {#2} \scan_stop:
10356 }
10357 \cs_generate_variant:Nn \__iow_open_stream:Nn { NV }

```

(End definition for \iow_open:Nn and __iow_open_stream:Nn. This function is documented on page 86.)

\iow_close:N Closing a stream is not quite the reverse of opening one. First, the close operation is easier than the open one, and second as the stream is actually a number we can use it directly to show that the slot has been freed up.

\iow_close:c

```

10358 \cs_new_protected:Npn \iow_close:N #1
10359 {
10360   \int_compare:nT { - \c_log_iow < #1 < \c_term_iow }
10361   {
10362     \tex_immediate:D \tex_closeout:D #1
10363     \prop_gremove:NV \g__iow_streams_prop #1
10364     \seq_if_in:NVF \g__iow_streams_seq #1
10365     { \seq_gpush:NV \g__iow_streams_seq #1 }
10366     \cs_gset_eq:NN #1 \c_term_iow
10367   }
10368 }
10369 \cs_generate_variant:Nn \iow_close:N { c }

```

(End definition for \iow_close:N. This function is documented on page 86.)

\iow_show:N Seek the stream in the \g__iow_streams_prop list, then show the stream as open or closed accordingly.

\iow_log:N

__iow_show:NN

```

10370 \cs_new_protected:Npn \iow_show:N { \__iow_show:NN \tl_show:n }
10371 \cs_generate_variant:Nn \iow_show:N { c }
10372 \cs_new_protected:Npn \iow_log:N { \__iow_show:NN \tl_log:n }
10373 \cs_generate_variant:Nn \iow_log:N { c }
10374 \cs_new_protected:Npn \__iow_show:NN #1#2
10375 {

```

```

10376 \__kernel_chk_defined:NT #2
10377 {
10378   \prop_get:NVNTF \g__iow_streams_prop #2 \l__iow_internal_tl
10379   {
10380     \exp_args:Nx #1
10381     { \token_to_str:N #2 ~ open: ~ \l__iow_internal_tl }
10382   }
10383   { \exp_args:Nx #1 { \token_to_str:N #2 ~ closed } }
10384 }
10385 }

```

(End definition for `\iow_show:N`, `\iow_log:N`, and `__iow_show:NN`. These functions are documented on page 86.)

`\iow_show_list:` Done as for input, but with a copy of the auxiliary so the name is correct.

`\iow_log_list:`

```

10386 \cs_new_protected:Npn \iow_show_list: { \__iow_list:N \msg_show:nnxxxx }
10387 \cs_new_protected:Npn \iow_log_list: { \__iow_list:N \msg_log:nnxxxx }
10388 \cs_new_protected:Npn \__iow_list:N #1
10389 {
10390   #1 { kernel } { show-streams }
10391   { iow }
10392   {
10393     \prop_map_function:NN \g__iow_streams_prop
10394     \msg_show_item_unbraced:nn
10395   }
10396   { } { }
10397 }

```

(End definition for `\iow_show_list:`, `\iow_log_list:`, and `__iow_list:N`. These functions are documented on page 87.)

49.3.1 Deferred writing

`\iow_shipout_x:Nn` First the easy part, this is the primitive, which expects its argument to be braced.

```

10398 \cs_new_protected:Npn \iow_shipout_x:Nn #1#2
10399 { \tex_write:D #1 {#2} }
10400 \cs_generate_variant:Nn \iow_shipout_x:Nn { c, Nx, cx }

```

(End definition for `\iow_shipout_x:Nn`. This function is documented on page 90.)

`\iow_shipout:Nn` With ε -TeX available deferred writing without expansion is easy.

```

10401 \cs_new_protected:Npn \iow_shipout:Nn #1#2
10402 { \tex_write:D #1 { \exp_not:n {#2} } }
10403 \cs_generate_variant:Nn \iow_shipout:Nn { c, Nx, cx }

```

(End definition for `\iow_shipout:Nn`. This function is documented on page 90.)

49.3.2 Immediate writing

`__kernel_iow_with:Nnn` If the integer #1 is equal to #2, just leave #3 in the input stream. Otherwise, pass the old value to an auxiliary, which sets the integer to the new value, runs the code, and restores the integer.

```

10404 \cs_new_protected:Npn \__kernel_iow_with:Nnn #1#2
10405 {

```

```

10406 \int_compare:nNnTF {#1} = {#2}
10407 { \use:n }
10408 { \exp_args:No \__iow_with:nNnn { \int_use:N #1 } #1 {#2} }
10409 }
10410 \cs_new_protected:Npn \__iow_with:nNnn #1#2#3#4
10411 {
10412   \int_set:Nn #2 {#3}
10413   #4
10414   \int_set:Nn #2 {#1}
10415 }

```

(End definition for `__kernel_iow_with:Nnn` and `__iow_with:nNnn`.)

`\iow_now:Nn` This routine writes the second argument onto the output stream without expansion. If
`\iow_now:Nx` this stream isn't open, the output goes to the terminal instead. If the first argument is
`\iow_now:cn` no output stream at all, we get an internal error. We don't use the expansion done by
`\iow_now:cx` `\write` to get the `Nx` variant, because it differs in subtle ways from `x`-expansion, namely,
macro parameter characters would not need to be doubled. We set the `\newlinechar`
to 10 using `__kernel_iow_with:Nnn` to support formats such as plain `TEX`: otherwise,
`\iow_newline:` would not work. We do not do this for `\iow_shipout:Nn` or `\iow_`
`shipout_x:Nn`, as `TEX` looks at the value of the `\newlinechar` at shipout time in those
cases.

```

10416 \cs_new_protected:Npn \iow_now:Nn #1#2
10417 {
10418   \__kernel_iow_with:Nnn \tex_newlinechar:D { '\^^J }
10419   { \tex_immediate:D \tex_write:D #1 { \exp_not:n {#2} } }
10420 }
10421 \cs_generate_variant:Nn \iow_now:Nn { c, Nx, cx }

```

(End definition for `\iow_now:Nn`. This function is documented on page 90.)

`\iow_log:n` Writing to the log and the terminal directly are relatively easy.
`\iow_log:x`
`\iow_term:n`
`\iow_term:x`

```

10422 \cs_set_protected:Npn \iow_log:x { \iow_now:Nx \c_log_iow }
10423 \cs_new_protected:Npn \iow_log:n { \iow_now:Nn \c_log_iow }
10424 \cs_set_protected:Npn \iow_term:x { \iow_now:Nx \c_term_iow }
10425 \cs_new_protected:Npn \iow_term:n { \iow_now:Nn \c_term_iow }

```

(End definition for `\iow_log:n` and `\iow_term:n`. These functions are documented on page 90.)

49.3.3 Special characters for writing

`\iow_newline:` Global variable holding the character that forces a new line when something is written
to an output stream.

```

10426 \cs_new:Npn \iow_newline: { ^^J }

```

(End definition for `\iow_newline:`. This function is documented on page 91.)

`\iow_char:N` Function to write any escaped char to an output stream.

```

10427 \cs_new_eq:NN \iow_char:N \cs_to_str:N

```

(End definition for `\iow_char:N`. This function is documented on page 91.)

49.3.4 Hard-wrapping lines to a character count

The code here implements a generic hard-wrapping function. This is used by the messaging system, but is designed such that it is available for other uses.

`\l_iow_line_count_int` This is the “raw” number of characters in a line which can be written to the terminal. The standard value is the line length typically used by T_EX Live and MiK_TE_X.

```
10428 \int_new:N \l_iow_line_count_int
10429 \int_set:Nn \l_iow_line_count_int { 78 }
```

(End definition for `\l_iow_line_count_int`. This variable is documented on page 92.)

`\l__iow_newline_tl` The token list inserted to produce a new line, with the *⟨run-on text⟩*.

```
10430 \tl_new:N \l__iow_newline_tl
```

(End definition for `\l__iow_newline_tl`.)

`\l__iow_line_target_int` This stores the target line count: the full number of characters in a line, minus any part for a leader at the start of each line.

```
10431 \int_new:N \l__iow_line_target_int
```

(End definition for `\l__iow_line_target_int`.)

`__iow_set_indent:n` The `one_indent` variables hold one indentation marker and its length. The `__iow_unindent:w` auxiliary removes one indentation. The function `__iow_set_indent:n` (that could possibly be public) sets the indentation in a consistent way. We set it to four spaces by default.

```
10432 \tl_new:N \l__iow_one_indent_tl
10433 \int_new:N \l__iow_one_indent_int
10434 \cs_new:Npn \__iow_unindent:w { }
10435 \cs_new_protected:Npn \__iow_set_indent:n #1
10436 {
10437   \__kernel_tl_set:Nx \l__iow_one_indent_tl
10438   { \exp_args:No \__kernel_str_to_other_fast:n { \tl_to_str:n {#1} } } }
10439   \int_set:Nn \l__iow_one_indent_int
10440   { \str_count:N \l__iow_one_indent_tl }
10441   \exp_last_unbraced:NNo
10442   \cs_set:Npn \__iow_unindent:w \l__iow_one_indent_tl { }
10443 }
10444 \exp_args:Nx \__iow_set_indent:n { \prg_replicate:nm { 4 } { ~ } }
```

(End definition for `__iow_set_indent:n` and others.)

`\l__iow_indent_tl` The current indentation (some copies of `\l__iow_one_indent_tl`) and its number of characters.

```
10445 \tl_new:N \l__iow_indent_tl
10446 \int_new:N \l__iow_indent_int
```

(End definition for `\l__iow_indent_tl` and `\l__iow_indent_int`.)

`\l__iow_line_tl` These hold the current line of text and a partial line to be added to it, respectively.

```
10447 \tl_new:N \l__iow_line_tl
10448 \tl_new:N \l__iow_line_part_tl
```

(End definition for `\l__iow_line_tl` and `\l__iow_line_part_tl`.)

`\l__iow_line_break_bool` Indicates whether the line was broken precisely at a chunk boundary.

```

10449 \bool_new:N \l__iow_line_break_bool

(End definition for \l__iow_line_break_bool.)

```

`\l__iow_wrap_tl` Used for the expansion step before detokenizing, and for the output from wrapping text: fully expanded and with lines which are not overly long.

```

10450 \tl_new:N \l__iow_wrap_tl

(End definition for \l__iow_wrap_tl.)

```

`\c__iow_wrap_marker_tl` Every special action of the wrapping code is starts with the same recognizable string, `\c__iow_wrap_marker_tl`. Upon seeing that “word”, the wrapping code reads one space-delimited argument to know what operation to perform. The setting of `\escapechar` here is not very important, but makes `\c__iow_wrap_marker_tl` look marginally nicer.

```

10451 \group_begin:
10452   \int_set:Nn \tex_escapechar:D { -1 }
10453   \tl_const:Nx \c__iow_wrap_marker_tl
10454     { \tl_to_str:n { \^^I \^^O \^^W \^^_ \^^W \^^R \^^A \^^P } }
10455 \group_end:
10456 \tl_map_inline:nn
10457   { { end } { newline } { allow_break } { indent } { unindent } }
10458   {
10459     \tl_const:cx { c__iow_wrap_ #1 _marker_tl }
10460     {
10461       \c__iow_wrap_marker_tl
10462       #1
10463       \c_catcode_other_space_tl
10464     }
10465   }

(End definition for \c__iow_wrap_marker_tl and others.)

```

`\iow_allow_break:` We set `\iow_allow_break:n` to produce an error when outside messages. Within wrapped message, it is set to `__iow_allow_break:` when valid and otherwise to `__iow_allow_break_error:`. The second produces an error expandably.

```

10466 \cs_new_protected:Npn \iow_allow_break:
10467   {
10468     \msg_error:nnnn { kernel } { iow-indent }
10469     { \iow_wrap:nnnN } { \iow_allow_break: }
10470   }
10471 \cs_new:Npx \__iow_allow_break: { \c__iow_wrap_allow_break_marker_tl }
10472 \cs_new:Npn \__iow_allow_break_error:
10473   {
10474     \msg_expandable_error:nnnn { kernel } { iow-indent }
10475     { \iow_wrap:nnnN } { \iow_allow_break: }
10476   }

(End definition for \iow_allow_break:, \__iow_allow_break:, and \__iow_allow_break_error:. This function is documented on page 302.)

```

`\iow_indent:n` We set `\iow_indent:n` to produce an error when outside messages. Within wrapped message, it is set to `__iow_indent:n` when valid and otherwise to `__iow_indent_error:n`.
`__iow_indent:n` The first places the instruction for increasing the indentation before its argument, and
`__iow_indent_error:n` the instruction for unindenting afterwards. The second produces an error expandably. Note that there are no forced line-break, so the indentation only changes when the next line is started.

```

10477 \cs_new_protected:Npn \iow_indent:n #1
10478 {
10479   \msg_error:nnnnn { kernel } { iow-indent }
10480   { \iow_wrap:nnnN } { \iow_indent:n } {#1}
10481   #1
10482 }
10483 \cs_new:Npx \__iow_indent:n #1
10484 {
10485   \c__iow_wrap_indent_marker_tl
10486   #1
10487   \c__iow_wrap_unindent_marker_tl
10488 }
10489 \cs_new:Npn \__iow_indent_error:n #1
10490 {
10491   \msg_expandable_error:nnnnn { kernel } { iow-indent }
10492   { \iow_wrap:nnnN } { \iow_indent:n } {#1}
10493   #1
10494 }
```

(End definition for `\iow_indent:n`, `__iow_indent:n`, and `__iow_indent_error:n`. This function is documented on page 92.)

`\iow_wrap:nnnN` The main wrapping function works as follows. First give `\`, `_` and other formatting commands the correct definition for messages and perform the given setup #3.
`\iow_wrap:nxnN` The definition of `_` uses an “other” space rather than a normal space, because the latter might be absorbed by \TeX to end a number or other f-type expansions. Use `\conditionally@traceoff` if defined; it is introduced by the `trace` package and suppresses uninteresting tracing of the wrapping code.

```

10495 \cs_new_protected:Npn \iow_wrap:nnnN #1#2#3#4
10496 {
10497   \group_begin:
10498     \cs_if_exist_use:N \conditionally@traceoff
10499     \int_set:Nn \tex_escapechar:D { -1 }
10500     \cs_set:Npx \{ { \token_to_str:N \{ }
10501     \cs_set:Npx \# { \token_to_str:N \# }
10502     \cs_set:Npx \} { \token_to_str:N \} }
10503     \cs_set:Npx \% { \token_to_str:N \% }
10504     \cs_set:Npx \~ { \token_to_str:N \~ }
10505     \int_set:Nn \tex_escapechar:D { 92 }
10506     \cs_set_eq:NN \ \ \iow_newline:
10507     \cs_set_eq:NN \ \ \c_catcode_other_space_tl
10508     \cs_set_eq:NN \iow_allow_break: \__iow_allow_break:
10509     \cs_set_eq:NN \iow_indent:n \__iow_indent:n
10510     #3
10511 }
```

Then fully-expand the input: in package mode, the expansion uses \LaTeX 2\epsilon ’s `\protect` mechanism in the same way as `\typeout`. In generic mode this setting is useless but

harmless. As soon as the expansion is done, reset `\iow_indent:n` to its error definition: it only works in the first argument of `\iow_wrap:nnnN`.

```

10511 \cs_set_eq:NN \protect \token_to_str:N
10512 \__kernel_tl_set:Nx \l__iow_wrap_tl {#1}
10513 \cs_set_eq:NN \iow_allow_break: \__iow_allow_break_error:
10514 \cs_set_eq:NN \iow_indent:n \__iow_indent_error:n

```

Afterwards, set the newline marker (two assignments to fully expand, then convert to a string) and initialize the target count for lines (the first line has target count `\l_iow_line_count_int` instead).

```

10515 \__kernel_tl_set:Nx \l__iow_newline_tl { \iow_newline: #2 }
10516 \__kernel_tl_set:Nx \l__iow_newline_tl { \tl_to_str:N \l__iow_newline_tl }
10517 \int_set:Nn \l__iow_line_target_int
10518 { \l_iow_line_count_int - \str_count:N \l__iow_newline_tl + 1 }

```

Sanity check.

```

10519 \int_compare:nNnT { \l__iow_line_target_int } < 0
10520 {
10521   \tl_set:Nn \l__iow_newline_tl { \iow_newline: }
10522   \int_set:Nn \l__iow_line_target_int
10523   { \l_iow_line_count_int + 1 }
10524 }

```

There is then a loop over the input, which stores the wrapped result in `\l__iow_wrap_tl`. After the loop, the resulting text is passed on to the function which has been given as a post-processor. The `\tl_to_str:N` step converts the “other” spaces back to normal spaces. The f-expansion removes a leading space from `\l__iow_wrap_tl`.

```

10525 \__iow_wrap_do:
10526 \exp_args:NNf \group_end:
10527 #4 { \tl_to_str:N \l__iow_wrap_tl }
10528 }
10529 \cs_generate_variant:Nn \iow_wrap:nnnN { nx }

```

(End definition for `\iow_wrap:nnnN`. This function is documented on page 92.)

<pre> __iow_wrap_do: __iow_wrap_fix_newline:w __iow_wrap_start:w </pre>	<p>Escape spaces and change newlines to <code>\c__iow_wrap_newline_marker_tl</code>. Set up a few variables, in particular the initial value of <code>\l__iow_wrap_tl</code>: the space stops the f-expansion of the main wrapping function and <code>\use_none:n</code> removes a newline marker inserted by later code. The main loop consists of repeatedly calling the <code>chunk</code> auxiliary to wrap chunks delimited by (newline or indentation) markers.</p>
--	---

```

10530 \cs_new_protected:Npn \__iow_wrap_do:
10531 {
10532   \__kernel_tl_set:Nx \l__iow_wrap_tl
10533   {
10534     \exp_args:No \__kernel_str_to_other_fast:n \l__iow_wrap_tl
10535     \c__iow_wrap_end_marker_tl
10536   }
10537   \__kernel_tl_set:Nx \l__iow_wrap_tl
10538   {
10539     \exp_after:wN \__iow_wrap_fix_newline:w \l__iow_wrap_tl
10540     ^^J \q__iow_nil ^^J \s__iow_stop
10541   }
10542   \exp_after:wN \__iow_wrap_start:w \l__iow_wrap_tl
10543 }

```



```

10544 \cs_new:Npn \__iow_wrap_fix_newline:w #1 ^^J #2 ^^J
10545 {
10546     #1
10547     \if_meaning:w \q__iow_nil #2
10548     \__iow_use_i_delimit_by_s_stop:nw
10549     \fi:
10550     \c__iow_wrap_newline_marker_tl
10551     \__iow_wrap_fix_newline:w #2 ^^J
10552 }
10553 \cs_new_protected:Npn \__iow_wrap_start:w
10554 {
10555     \bool_set_false:N \l__iow_line_break_bool
10556     \tl_clear:N \l__iow_line_tl
10557     \tl_clear:N \l__iow_line_part_tl
10558     \tl_set:Nn \l__iow_wrap_tl { ~ \use_none:n }
10559     \int_zero:N \l__iow_indent_int
10560     \tl_clear:N \l__iow_indent_tl
10561     \__iow_wrap_chunk:nw { \l__iow_line_count_int }
10562 }

```

(End definition for __iow_wrap_do:, __iow_wrap_fix_newline:w, and __iow_wrap_start:w.)

```

\__iow_wrap_chunk:nw
\__iow_wrap_next:nw

```

The `chunk` and `next` auxiliaries are defined indirectly to obtain the expansions of `\c_catcode_other_space_tl` and `\c__iow_wrap_marker_tl` in their definition. The `next` auxiliary calls a function corresponding to the type of marker (its ##2), which can be `newline` or `indent` or `unindent` or `end`. The first argument of the `chunk` auxiliary is a target number of characters and the second is some string to wrap. If the chunk is empty simply call `next`. Otherwise, set up a call to `__iow_wrap_line:nw`, including the indentation if the current line is empty, and including a trailing space (#1) before the `__iow_wrap_end_chunk:w` auxiliary.

```

10563 \cs_set_protected:Npn \__iow_tmp:w #1#2
10564 {
10565     \cs_new_protected:Npn \__iow_wrap_chunk:nw ##1##2 #2
10566     {
10567         \tl_if_empty:nTF {##2}
10568         {
10569             \tl_clear:N \l__iow_line_part_tl
10570             \__iow_wrap_next:nw {##1}
10571         }
10572         {
10573             \tl_if_empty:NTF \l__iow_line_tl
10574             {
10575                 \__iow_wrap_line:nw
10576                 { \l__iow_indent_tl }
10577                 ##1 - \l__iow_indent_int ;
10578             }
10579             { \__iow_wrap_line:nw { } ##1 ; }
10580             ##2 #1
10581             \__iow_wrap_end_chunk:w 7 6 5 4 3 2 1 0 \s__iow_stop
10582         }
10583     }
10584     \cs_new_protected:Npn \__iow_wrap_next:nw ##1##2 #1
10585     { \use:c { __iow_wrap_##2:n } {##1} }
10586 }

```

```
10587 \exp_args:NVV \__iow_tmp:w \c_catcode_other_space_tl \c__iow_wrap_marker_tl
```

(End definition for __iow_wrap_chunk:nw and __iow_wrap_next:nw.)

```
\__iow_wrap_line:nw
\__iow_wrap_line_loop:w
\__iow_wrap_line_aux:Nw
\__iow_wrap_line_seven:nnnnnnn
\__iow_wrap_line_end:NnnnnnnnN
\__iow_wrap_line_end:nw
\__iow_wrap_end_chunk:w
```

This is followed by $\{\langle string \rangle\} \langle intexpr \rangle$; . It stores the $\langle string \rangle$ and up to $\langle intexpr \rangle$ characters from the current chunk into $\backslash l_iow_line_part_tl$. Characters are grabbed 8 at a time and left in $\backslash l_iow_line_part_tl$ by the `line_loop` auxiliary. When $k < 8$ remain to be found, the `line_aux` auxiliary calls the `line_end` auxiliary followed by (the single digit) k , then $7 - k$ empty brace groups, then the chunk's remaining characters. The `line_end` auxiliary leaves k characters from the chunk in the line part, then ends the assignment. Ignore the `\use_none:nnnnn` line for now. If the next character is a space the line can be broken there: store what we found into the result and get the next line. Otherwise some work is needed to find a break-point. So far we have ignored what happens if the chunk is shorter than the requested number of characters: this is dealt with by the `end_chunk` auxiliary, which gets treated like a character by the rest of the code. It ends up being called either as one of the arguments #2–#9 of the `line_loop` auxiliary or as one of the arguments #2–#8 of the `line_end` auxiliary. In both cases stop the assignment and work out how many characters are still needed. Notice that when we have exactly seven arguments to clean up, a `\exp_stop_f:` has to be inserted to stop the `\exp:w`. The weird `\use_none:nnnnn` ensures that the required data is in the right place.

```
10588 \cs_new_protected:Npn \__iow_wrap_line:nw #1
10589 {
10590   \tex_edef:D \l__iow_line_part_tl { \if_false: } \fi:
10591   #1
10592   \exp_after:wN \__iow_wrap_line_loop:w
10593   \int_value:w \int_eval:w
10594 }
10595 \cs_new:Npn \__iow_wrap_line_loop:w #1 ; #2#3#4#5#6#7#8#9
10596 {
10597   \if_int_compare:w #1 < 8 \exp_stop_f:
10598     \__iow_wrap_line_aux:Nw #1
10599   \fi:
10600   #2 #3 #4 #5 #6 #7 #8 #9
10601   \exp_after:wN \__iow_wrap_line_loop:w
10602   \int_value:w \int_eval:w #1 - 8 ;
10603 }
10604 \cs_new:Npn \__iow_wrap_line_aux:Nw #1#2#3 \exp_after:wN #4 ;
10605 {
10606   #2
10607   \exp_after:wN \__iow_wrap_line_end:NnnnnnnnN
10608   \exp_after:wN #1
10609   \exp:w \exp_end_continue_f:w
10610   \exp_after:wN \exp_after:wN
10611   \if_case:w #1 \exp_stop_f:
10612     \prg_do_nothing:
10613   \or: \use_none:n
10614   \or: \use_none:nn
10615   \or: \use_none:nnn
10616   \or: \use_none:nnnn
10617   \or: \use_none:nnnnn
10618   \or: \use_none:nnnnnn
10619   \or: \__iow_wrap_line_seven:nnnnnnn
```

```

10620 \fi:
10621 { } { } { } { } { } { } { } { } #3
10622 }
10623 \cs_new:Npn \__iow_wrap_line_seven:nnnnnnn #1#2#3#4#5#6#7 { \exp_stop_f: }
10624 \cs_new:Npn \__iow_wrap_line_end:NnnnnnnnN #1#2#3#4#5#6#7#8#9
10625 {
10626     #2 #3 #4 #5 #6 #7 #8
10627     \use_none:nnnnn \int_eval:w 8 - ; #9
10628     \token_if_eq_charcode:NNTF \c_space_token #9
10629     { \__iow_wrap_line_end:nw { } }
10630     { \if_false: { \fi: } \__iow_wrap_break:w #9 }
10631 }
10632 \cs_new:Npn \__iow_wrap_line_end:nw #1
10633 {
10634     \if_false: { \fi: }
10635     \__iow_wrap_store_do:n {#1}
10636     \__iow_wrap_next_line:w
10637 }
10638 \cs_new:Npn \__iow_wrap_end_chunk:w
10639 #1 \int_eval:w #2 - #3 ; #4#5 \s__iow_stop
10640 {
10641     \if_false: { \fi: }
10642     \exp_args:Nf \__iow_wrap_next:nw { \int_eval:n { #2 - #4 } }
10643 }

```

(End definition for __iow_wrap_line:nw and others.)

<pre> __iow_wrap_break:w __iow_wrap_break_first:w __iow_wrap_break_none:w __iow_wrap_break_loop:w __iow_wrap_break_end:w </pre>	<p>Functions here are defined indirectly: __iow_tmp:w is eventually called with an “other” space as its argument. The goal is to remove from \l__iow_line_part_tl the part after the last space. In most cases this is done by repeatedly calling the <code>break_loop</code> auxiliary, which leaves “words” (delimited by spaces) until it hits the trailing space: then its argument <code>##3</code> is ? __iow_wrap_break_end:w instead of a single token, and that <code>break_end</code> auxiliary leaves in the assignment the line until the last space, then calls __iow_wrap_line_end:nw to finish up the line and move on to the next. If there is no space in \l__iow_line_part_tl then the <code>break_first</code> auxiliary calls the <code>break_none</code> auxiliary. In that case, if the current line is empty, the complete word (including <code>##4</code>, characters beyond what we had grabbed) is added to the line, making it over-long. Otherwise, the word is used for the following line (and the last space of the line so far is removed because it was inserted due to the presence of a marker).</p>
--	---

```

10644 \cs_set_protected:Npn \__iow_tmp:w #1
10645 {
10646     \cs_new:Npn \__iow_wrap_break:w
10647     {
10648         \tex_edef:D \l__iow_line_part_tl
10649         { \if_false: } \fi:
10650         \exp_after:wN \__iow_wrap_break_first:w
10651         \l__iow_line_part_tl
10652         #1
10653         { ? \__iow_wrap_break_end:w }
10654         \s__iow_mark
10655     }
10656     \cs_new:Npn \__iow_wrap_break_first:w ##1 #1 ##2
10657     {

```

```

10658     \use_none:nn ##2 \__iow_wrap_break_none:w
10659     \__iow_wrap_break_loop:w ##1 #1 ##2
10660   }
10661   \cs_new:Npn \__iow_wrap_break_none:w ##1##2 #1 ##3 \s__iow_mark ##4 #1
10662   {
10663     \tl_if_empty:NTF \l__iow_line_tl
10664     { ##2 ##4 \__iow_wrap_line_end:nw { } }
10665     { \__iow_wrap_line_end:nw { \__iow_wrap_trim:N } ##2 ##4 #1 }
10666   }
10667   \cs_new:Npn \__iow_wrap_break_loop:w ##1 #1 ##2 #1 ##3
10668   {
10669     \use_none:n ##3
10670     ##1 #1
10671     \__iow_wrap_break_loop:w ##2 #1 ##3
10672   }
10673   \cs_new:Npn \__iow_wrap_break_end:w ##1 #1 ##2 ##3 #1 ##4 \s__iow_mark
10674   { ##1 \__iow_wrap_line_end:nw { } ##3 }
10675 }
10676 \exp_args:NV \__iow_tmp:w \c_catcode_other_space_tl

```

(End definition for __iow_wrap_break:w and others.)

__iow_wrap_next_line:w The special case where the end of a line coincides with the end of a chunk is detected here, to avoid a spurious empty line. Otherwise, call __iow_wrap_line:nw to find characters for the next line (remembering to account for the indentation).

```

10677 \cs_new_protected:Npn \__iow_wrap_next_line:w #1#2 \s__iow_stop
10678 {
10679   \tl_clear:N \l__iow_line_tl
10680   \token_if_eq_meaning:NNTF #1 \__iow_wrap_end_chunk:w
10681   {
10682     \tl_clear:N \l__iow_line_part_tl
10683     \bool_set_true:N \l__iow_line_break_bool
10684     \__iow_wrap_next:nw { \l__iow_line_target_int }
10685   }
10686   {
10687     \__iow_wrap_line:nw
10688     { \l__iow_indent_tl }
10689     \l__iow_line_target_int - \l__iow_indent_int ;
10690     #1 #2 \s__iow_stop
10691   }
10692 }

```

(End definition for __iow_wrap_next_line:w.)

__iow_wrap_allow_break:n This is called after a chunk has been wrapped. The \l__iow_line_part_tl typically ends with a space (except at the beginning of a line?), which we remove since the **allow-break** marker should not insert a space. Then move on with the next chunk, making sure to adjust the target number of characters for the line in case we did remove a space.

```

10693 \cs_new_protected:Npn \__iow_wrap_allow_break:n #1
10694 {
10695   \__kernel_tl_set:Nx \l__iow_line_tl
10696   { \l__iow_line_tl \__iow_wrap_trim:N \l__iow_line_part_tl }
10697   \bool_set_false:N \l__iow_line_break_bool
10698   \tl_if_empty:NTF \l__iow_line_part_tl

```

```

10699     { \_iow_wrap_chunk:nw {#1} }
10700     { \exp_args:Nf \_iow_wrap_chunk:nw { \int_eval:n { #1 + 1 } } }
10701   }

```

(End definition for _iow_wrap_allow_break:n.)

_iow_wrap_indent:n
_iow_wrap_unindent:n

These functions are called after a chunk has been wrapped, when encountering **indent/unindent** markers. Add the line part (last line part of the previous chunk) to the line so far and reset a boolean denoting the presence of a line-break. Most importantly, add or remove one indent from the current indent (both the integer and the token list). Finally, continue wrapping.

```

10702 \cs_new_protected:Npn \_iow_wrap_indent:n #1
10703 {
10704   \tl_put_right:Nx \l__iow_line_tl { \l__iow_line_part_tl }
10705   \bool_set_false:N \l__iow_line_break_bool
10706   \int_add:Nn \l__iow_indent_int { \l__iow_one_indent_int }
10707   \tl_put_right:No \l__iow_indent_tl { \l__iow_one_indent_tl }
10708   \_iow_wrap_chunk:nw {#1}
10709 }
10710 \cs_new_protected:Npn \_iow_wrap_unindent:n #1
10711 {
10712   \tl_put_right:Nx \l__iow_line_tl { \l__iow_line_part_tl }
10713   \bool_set_false:N \l__iow_line_break_bool
10714   \int_sub:Nn \l__iow_indent_int { \l__iow_one_indent_int }
10715   \__kernel_tl_set:Nx \l__iow_indent_tl
10716     { \exp_after:wN \_iow_unindent:w \l__iow_indent_tl }
10717   \_iow_wrap_chunk:nw {#1}
10718 }

```

(End definition for _iow_wrap_indent:n and _iow_wrap_unindent:n.)

_iow_wrap_newline:n
_iow_wrap_end:n

These functions are called after a chunk has been line-wrapped, when encountering a **newline/end** marker. Unless we just took a line-break, store the line part and the line so far into the whole \l__iow_wrap_tl, trimming a trailing space. In the **newline** case look for a new line (of length \l__iow_line_target_int) in a new chunk.

```

10719 \cs_new_protected:Npn \_iow_wrap_newline:n #1
10720 {
10721   \bool_if:NF \l__iow_line_break_bool
10722     { \_iow_wrap_store_do:n { \_iow_wrap_trim:N } }
10723   \bool_set_false:N \l__iow_line_break_bool
10724   \_iow_wrap_chunk:nw { \l__iow_line_target_int }
10725 }
10726 \cs_new_protected:Npn \_iow_wrap_end:n #1
10727 {
10728   \bool_if:NF \l__iow_line_break_bool
10729     { \_iow_wrap_store_do:n { \_iow_wrap_trim:N } }
10730   \bool_set_false:N \l__iow_line_break_bool
10731 }

```

(End definition for _iow_wrap_newline:n and _iow_wrap_end:n.)

_iow_wrap_store_do:n

First add the last line part to the line, then append it to \l__iow_wrap_tl with the appropriate new line (with “run-on” text), possibly with its last space removed (#1 is empty or _iow_wrap_trim:N).

```

10732 \cs_new_protected:Npn \__iow_wrap_store_do:n #1
10733 {
10734   \__kernel_tl_set:Nx \l__iow_line_tl
10735   { \l__iow_line_tl \l__iow_line_part_tl }
10736   \__kernel_tl_set:Nx \l__iow_wrap_tl
10737   {
10738     \l__iow_wrap_tl
10739     \l__iow_newline_tl
10740     #1 \l__iow_line_tl
10741   }
10742   \tl_clear:N \l__iow_line_tl
10743 }

```

(End definition for `__iow_wrap_store_do:n`.)

```

\__iow_wrap_trim:N Remove one trailing “other” space from the argument if present.
\__iow_wrap_trim:w
\__iow_wrap_trim_aux:w
10744 \cs_set_protected:Npn \__iow_tmp:w #1
10745 {
10746   \cs_new:Npn \__iow_wrap_trim:N ##1
10747   { \exp_after:wN \__iow_wrap_trim:w ##1 \s__iow_mark #1 \s__iow_mark \s__iow_stop }
10748   \cs_new:Npn \__iow_wrap_trim:w ##1 #1 \s__iow_mark
10749   { \__iow_wrap_trim_aux:w ##1 \s__iow_mark }
10750   \cs_new:Npn \__iow_wrap_trim_aux:w ##1 \s__iow_mark ##2 \s__iow_stop {##1}
10751 }
10752 \exp_args:NV \__iow_tmp:w \c_catcode_other_space_tl

```

(End definition for `__iow_wrap_trim:N`, `__iow_wrap_trim:w`, and `__iow_wrap_trim_aux:w`.)

```

10753 <@@=file>

```

49.4 File operations

`\l__file_internal_tl` Used as a short-term scratch variable.

```

10754 \tl_new:N \l__file_internal_tl

```

(End definition for `\l__file_internal_tl`.)

`\g_file_curr_dir_str` The name of the current file should be available at all times: the name itself is set dynamically.
`\g_file_curr_ext_str`
`\g_file_curr_name_str`

```

10755 \str_new:N \g_file_curr_dir_str
10756 \str_new:N \g_file_curr_ext_str
10757 \str_new:N \g_file_curr_name_str

```

(End definition for `\g_file_curr_dir_str`, `\g_file_curr_ext_str`, and `\g_file_curr_name_str`. These variables are documented on page 93.)

`\g__file_stack_seq` The input list of files is stored as a sequence stack. In package mode we can recover the information from the details held by $\text{\LaTeX} 2_{\epsilon}$ (we must be in the preamble and loaded using `\usepackage` or `\RequirePackage`). As $\text{\LaTeX} 2_{\epsilon}$ doesn’t store directory and name separately, we stick to the same convention here. In pre-loading, `\@currnamestack` is empty so is skipped.

```

10758 \seq_new:N \g__file_stack_seq
10759 \group_begin:
10760   \cs_set_protected:Npn \__file_tmp:w #1#2#3

```

```

10761 {
10762   \tl_if_blank:nTF {#1}
10763   {
10764     \cs_set:Npn \__file_tmp:w ##1 " ##2 " ##3 \s__file_stop
10765     { { } {##2} { } }
10766     \seq_gput_right:Nx \g__file_stack_seq
10767     {
10768       \exp_after:wN \__file_tmp:w \tex_jobname:D
10769       " \tex_jobname:D " \s__file_stop
10770     }
10771   }
10772   {
10773     \seq_gput_right:Nn \g__file_stack_seq { { } {#1} {#2} }
10774     \__file_tmp:w
10775   }
10776 }
10777 \cs_if_exist:NT \@currnamestack
10778 {
10779   \tl_if_empty:NF \@currnamestack
10780   { \exp_after:wN \__file_tmp:w \@currnamestack }
10781 }
10782 \group_end:

```

(End definition for \g__file_stack_seq.)

\g__file_record_seq The total list of files used is recorded separately from the current file stack, as nothing is ever popped from this list. The current file name should be included in the file list! We will eventually copy the contents of \@filelist.

```

10783 \seq_new:N \g__file_record_seq

```

(End definition for \g__file_record_seq.)

\l__file_base_name_tl For storing the basename and full path whilst passing data internally.

```

\l__file_full_name_tl
10784 \tl_new:N \l__file_base_name_tl
10785 \tl_new:N \l__file_full_name_tl

```

(End definition for \l__file_base_name_tl and \l__file_full_name_tl.)

\l__file_dir_str Used in parsing a path into parts: in contrast to the above, these are never used outside of the current module.

```

\l__file_ext_str
\l__file_name_str
10786 \str_new:N \l__file_dir_str
10787 \str_new:N \l__file_ext_str
10788 \str_new:N \l__file_name_str

```

(End definition for \l__file_dir_str, \l__file_ext_str, and \l__file_name_str.)

\l_file_search_path_seq The current search path.

```

10789 \seq_new:N \l_file_search_path_seq

```

(End definition for \l_file_search_path_seq. This variable is documented on page 94.)

\l__file_tmp_seq Scratch space for comma list conversion.

```

10790 \seq_new:N \l__file_tmp_seq

```

(End definition for \l__file_tmp_seq.)

49.4.1 Internal auxiliaries

`\s__file_stop` Internal scan marks.

```
10791 \scan_new:N \s__file_stop
```

(End definition for `\s__file_stop`.)

`\q__file_nil` Internal quarks.

```
10792 \quark_new:N \q__file_nil
```

(End definition for `\q__file_nil`.)

`__file_quark_if_nil_p:n` Branching quark conditional.

```
\__file_quark_if_nil:nTF 10793 \__kernel_quark_new_conditional:Nn \__file_quark_if_nil:n { TF }
```

(End definition for `__file_quark_if_nil:nTF`.)

`\q__file_recursion_tail` Internal recursion quarks.

```
\q__file_recursion_stop 10794 \quark_new:N \q__file_recursion_tail
```

```
10795 \quark_new:N \q__file_recursion_stop
```

(End definition for `\q__file_recursion_tail` and `\q__file_recursion_stop`.)

`_file_if_recursion_tail_break:NN` Functions to query recursion quarks.

```
\_file_if_recursion_tail_stop_do:Nn 10796 \__kernel_quark_new_test:N \_file_if_recursion_tail_stop:N
```

```
10797 \__kernel_quark_new_test:N \_file_if_recursion_tail_stop_do:Nn
```

(End definition for `_file_if_recursion_tail_break:NN` and `_file_if_recursion_tail_stop_do:Nn`.)

`_kernel_file_name_sanitize:n`

`__file_name_expand:n`

`_file_name_expand_cleanup:Nw`

`_file_name_expand_cleanup:w`

`__file_name_expand_end:`

```
10798 \cs_new:Npn \__kernel_file_name_sanitize:n #1
```

```
10799 {
```

```
\_file_name_expand_error_aux:Nw 10800 \exp_args:Ne \_file_name_trim_spaces:n
```

`__file_name_strip_quotes:n`

```
10801 {
```

`_file_name_strip_quotes:nnnw`

```
10802 \exp_args:Ne \_file_name_strip_quotes:n
```

`_file_name_strip_quotes:nnn`

```
10803 { \_file_name_expand:n {#1} }
```

`_file_name_trim_spaces:n`

```
10804 }
```

`__file_name_trim_spaces:nw`

```
10805 }
```

`_file_name_trim_spaces_aux:n`

`_file_name_trim_spaces_aux:w`

Expanding the file name uses a `\csname`-based approach, and relies on active characters (for example from UTF-8 characters) being properly set up to expand to a expansion-safe version using `\ifcsname`. This is less conservative than the token-by-token approach used before, but it is much faster.

We'll use `\cs:w` to start expanding the file name, and to avoid creating csnames equal to `\relax` with “common” names, there's a prefix `__file_name=` to the csname. There's also a guard token at the end so we can check if there was an error during the process and (try to) clean up gracefully.

```
10806 \cs_new:Npn \__file_name_expand:n #1
```

```
10807 {
```

```
10808 \exp_after:wN \_file_name_expand_cleanup:Nw
```

```
10809 \cs:w __file_name = #1 \cs_end:
```

```
10810 \_file_name_expand_end:
```

```
10811 }
```


With the `csname` built, we grab it, and grab the remaining tokens delimited by `__file_name_expand_end:`. If there are any remaining tokens, something bad happened, so we'll call the error procedure `__file_name_expand_error:Nw`. If everything went according to plan, then use `\token_to_str:N` on the `csname` built, and call `__file_name_expand_cleanup:w` to remove the prefix we added a while back. `__file_name_expand_cleanup:w` takes a leading argument so we don't have to bother about the value of `\tex_escapechar:D`.

```

10812 \cs_new:Npn \__file_name_expand_cleanup:Nw #1 #2 \__file_name_expand_end:
10813 {
10814   \tl_if_empty:nF {#2}
10815   { \__file_name_expand_error:Nw #2 \__file_name_expand_end: }
10816   \exp_after:wN \__file_name_expand_cleanup:w \token_to_str:N #1
10817 }
10818 \exp_last_unbraced:NNNNo
10819 \cs_new:Npn \__file_name_expand_cleanup:w #1 \tl_to_str:n { __file_name = } { }

```

In non-error cases `__file_name_expand_end:` should not expand. It will only do so in case there is a `\csname` too much in the file name, so it will throw an error (while expanding), then insert the missing `\cs_end:` and yet another `__file_name_expand_end:` that will be used as a delimiter by `__file_name_expand_cleanup:Nw` (or that will expand again if yet another `\endcsname` is missing).

```

10820 \cs_new:Npn \__file_name_expand_end:
10821 {
10822   \msg_expandable_error:nn
10823   { kernel } { filename-missing-endcsname }
10824   \cs_end: \__file_name_expand_end:
10825 }

```

Now to the error case. `__file_name_expand_error:Nw` adds an extra `\cs_end:` so that in case there was an extra `\csname` in the file name, then `__file_name_expand_error_aux:Nw` throws the error.

```

10826 \cs_new:Npn \__file_name_expand_error:Nw #1 #2 \__file_name_expand_end:
10827 { \__file_name_expand_error_aux:Nw #1 #2 \cs_end: \__file_name_expand_end: }
10828 \cs_new:Npn \__file_name_expand_error_aux:Nw #1 #2 \cs_end: #3
10829   \__file_name_expand_end:
10830 {
10831   \msg_expandable_error:nnff
10832   { kernel } { filename-chars-lost }
10833   { \token_to_str:N #1 } { \exp_stop_f: #2 }
10834 }

```

Quoting file name uses basically the same approach as for `luaquotejobname:` count the " tokens and remove them.

```

10835 \cs_new:Npn \__file_name_strip_quotes:n #1
10836 {
10837   \__file_name_strip_quotes:nw { 0 }
10838   #1 " \q__file_recursion_tail " \q__file_recursion_stop {#1}
10839 }
10840 \cs_new:Npn \__file_name_strip_quotes:nw #1#2 "
10841 {
10842   \if_meaning:w \q__file_recursion_tail #2
10843     \__file_name_strip_quotes_end:wNwN
10844   \fi:
10845   #2

```

```

10846     \_file_name_strip_quotes:nw { #1 + 1 }
10847   }
10848 \cs_new:Npn \_file_name_strip_quotes_end:wnwn \fi: #1
10849   \_file_name_strip_quotes:nw #2 \q_file_recursion_stop #3
10850   {
10851     \fi:
10852     \int_if_odd:nT {#2}
10853     {
10854       \msg_expandable_error:nnn
10855         { kernel } { unbalanced-quote-in-filename } {#3}
10856     }
10857   }

```

Spaces need to be trimmed from the start of the name and from the end of any extension. However, the name we are passed might not have an extension: that means we have to look for one. If there is no extension, we still use the standard trimming function but deliberately prevent any spaces being removed at the end.

```

10858 \cs_new:Npn \_file_name_trim_spaces:n #1
10859   { \_file_name_trim_spaces:nw {#1} #1 . \q_file_nil . \s_file_stop }
10860 \cs_new:Npn \_file_name_trim_spaces:nw #1#2 . #3 . #4 \s_file_stop
10861   {
10862     \_file_quark_if_nil:nTF {#3}
10863     {
10864       \tl_trim_spaces_apply:nN { #1 \s_file_stop }
10865       \_file_name_trim_spaces_aux:n
10866     }
10867     { \tl_trim_spaces:n {#1} }
10868   }
10869 \cs_new:Npn \_file_name_trim_spaces_aux:n #1
10870   { \_file_name_trim_spaces_aux:w #1 }
10871 \cs_new:Npn \_file_name_trim_spaces_aux:w #1 \s_file_stop {#1}

```

(End definition for _kernel_file_name_sanitize:n and others.)

```

\_kernel_file_name_quote:n
\_file_name_quote:nw
10872 \cs_new:Npn \_kernel_file_name_quote:n #1
10873   { \_file_name_quote:nw {#1} #1 ~ \q_file_nil \s_file_stop }
10874 \cs_new:Npn \_file_name_quote:nw #1 #2 ~ #3 \s_file_stop
10875   {
10876     \_file_quark_if_nil:nTF {#3}
10877     { #1 }
10878     { "#1" }
10879   }

```

(End definition for _kernel_file_name_quote:n and _file_name_quote:nw.)

\c_file_marker_tl The same idea as the marker for rescanning token lists: this pair of tokens cannot appear in a file that is being input.

```

10880 \tl_const:Nx \c_file_marker_tl { : \token_to_str:N : }

```

(End definition for \c_file_marker_tl.)

\file_get:nnNTF The approach here is similar to that for \tl_set_rescan:Nnn. The file contents are grabbed as an argument delimited by \c_file_marker_tl. A few subtleties: braces in \if_false: ... \fi: to deal with possible alignment tabs, \tracingnesting to avoid

```

\_file_get_aux:nnN
\_file_get_do:Nw

```

a warning about a group being closed inside the `\scantokens`, and `\prg_return_true:` is placed after the end-of-file marker.

```

10881 \cs_new_protected:Npn \file_get:nnN #1#2#3
10882 {
10883   \file_get:nnNF {#1} {#2} #3
10884   { \tl_set:Nn #3 { \q_no_value } }
10885 }
10886 \prg_new_protected_conditional:Npnn \file_get:nnN #1#2#3 { T , F , TF }
10887 {
10888   \file_get_full_name:nNTF {#1} \l__file_full_name_tl
10889   {
10890     \exp_args:NV \__file_get_aux:nnN
10891     \l__file_full_name_tl
10892     {#2} #3
10893     \prg_return_true:
10894   }
10895   { \prg_return_false: }
10896 }
10897 \cs_new_protected:Npx \__file_get_aux:nnN #1#2#3
10898 {
10899   \exp_not:N \if_false: { \exp_not:N \fi:
10900   \group_begin:
10901     \int_set_eq:NN \tex_tracingnesting:D \c_zero_int
10902     \exp_not:N \exp_args:No \tex_everyeof:D
10903     { \exp_not:N \c__file_marker_tl }
10904     #2 \scan_stop:
10905     \exp_not:N \exp_after:wN \exp_not:N \__file_get_do:Nw
10906     \exp_not:N \exp_after:wN #3
10907     \exp_not:N \exp_after:wN \exp_not:N \prg_do_nothing:
10908     \exp_not:N \tex_input:D
10909     \sys_if_engine_luatex:TF
10910     { {#1} }
10911     { \exp_not:N \__kernel_file_name_quote:n {#1} \scan_stop: }
10912   \exp_not:N \if_false: } \exp_not:N \fi:
10913 }
10914 \exp_args:Nno \use:nn
10915 { \cs_new_protected:Npn \__file_get_do:Nw #1#2 }
10916 { \c__file_marker_tl }
10917 {
10918   \group_end:
10919   \tl_set:No #1 {#2}
10920 }

```

(End definition for `\file_get:nnNTF` and others. These functions are documented on page 94.)

`__file_size:n` A copy of the primitive where it's available.

```

10921 \cs_new_eq:NN \__file_size:n \tex_filesize:D

```

(End definition for `__file_size:n`.)

`\file_full_name:n` File searching can be carried out if the `\pdffilesize` primitive or an equivalent is available. That of course means we need to arrange for everything else to here to be done by expansion too. We start off by sanitizing the name and quoting if required: we may need to remove those quotes, so the raw name is passed too.

```

\__file_full_name:n
\__file_full_name_aux:nn
\__file_full_name_aux:Nnn
\__file_full_name_aux:nN
\__file_full_name_aux:nnN
\__file_name_cleanup:w
\__file_name_end:
\__file_name_ext_check:nn
\__file_name_ext_check:nnw
\__file_name_ext_check:nnnw
\__file_name_ext_check:nnn
\__file_name_ext_check:nnnn

```

```

10922 \cs_new:Npn \file_full_name:n #1
10923 {
10924     \exp_args:Ne \__file_full_name:n
10925     { \__kernel_file_name_sanitise:n {#1} }
10926 }

```

First, we check if the file is just here: no mapping so we do not need the break part of the broader auxiliary. We are using the fact that the primitive here returns nothing if the file is entirely absent. To avoid unnecessary filesystem lookups, the result of `\pdffilesize` is kept available as an argument. For package mode, `\input@path` is a token list not a sequence.

```

10927 \cs_new:Npn \__file_full_name:n #1
10928 {
10929     \tl_if_blank:nF {#1}
10930     { \exp_args:Nne \__file_full_name_aux:nn {#1} { \__file_size:n {#1} } }
10931 }
10932 \cs_new:Npn \__file_full_name_aux:nn #1 #2
10933 {
10934     \tl_if_blank:nTF {#2}
10935     {
10936         \seq_map_tokens:Nn \l_file_search_path_seq
10937         { \__file_full_name_aux:Nnn \seq_map_break:n {#1} }
10938         \cs_if_exist:NT \input@path
10939         {
10940             \tl_map_tokens:Nn \input@path
10941             { \__file_full_name_aux:Nnn \tl_map_break:n {#1} }
10942         }
10943         \__file_name_end:
10944     }
10945     { \__file_ext_check:nn {#1} {#2} }
10946 }

```

Two pars to the auxiliary here so we can avoid doing quoting twice in the event we find the right file.

```

10947 \cs_new:Npn \__file_full_name_aux:Nnn #1#2#3
10948 { \exp_args:Ne \__file_full_name_aux:nN { \tl_to_str:n {#3} / #2 } #1 }
10949 \cs_new:Npn \__file_full_name_aux:nN #1
10950 { \exp_args:Nne \__file_full_name_aux:nnN {#1} { \__file_size:n {#1} } }
10951 \cs_new:Npn \__file_full_name_aux:nnN #1 #2 #3
10952 {
10953     \tl_if_blank:nF {#2}
10954     {
10955         #3
10956         {
10957             \__file_ext_check:nn {#1} {#2}
10958             \__file_name_cleanup:w
10959         }
10960     }
10961 }
10962 \cs_new:Npn \__file_name_cleanup:w #1 \__file_name_end: { }
10963 \cs_new:Npn \__file_name_end: { }

```

As \TeX automatically adds `.tex` if there is no extension, there is a little clean up to do here. First, make sure we are not in the directory part, saving that. Then check for an extension.

```

10964 \cs_new:Npn \__file_ext_check:nn #1 #2
10965 { \__file_ext_check:nnw {#2} { / } #1 / \q__file_nil / \s__file_stop }
10966 \cs_new:Npn \__file_ext_check:nnw #1 #2 #3 / #4 / #5 \s__file_stop
10967 {
10968   \__file_quark_if_nil:nTF {#4}
10969   {
10970     \exp_args:No \__file_ext_check:nnnw
10971     { \use_none:n #2 } {#1} {#3} #3 . \q__file_nil . \s__file_stop
10972   }
10973   { \__file_ext_check:nnw {#1} { #2 #3 / } #4 / #5 \s__file_stop }
10974 }
10975 \cs_new:Npx \__file_ext_check:nnnw #1#2#3#4 . #5 . #6 \s__file_stop
10976 {
10977   \exp_not:N \__file_quark_if_nil:nTF {#5}
10978   {
10979     \exp_not:N \__file_ext_check:nnn
10980     { #1 #3 \tl_to_str:n { .tex } } { #1 #3 } {#2}
10981   }
10982   { #1 #3 }
10983 }
10984 \cs_new:Npn \__file_ext_check:nnn #1
10985 { \exp_args:Nne \__file_ext_check:nnnn {#1} { \__file_size:n {#1} } }
10986 \cs_new:Npn \__file_ext_check:nnnn #1#2#3#4
10987 {
10988   \tl_if_blank:nTF {#2}
10989   {#3}
10990   {
10991     \int_compare:nNnTF
10992     {#4} = {#2}
10993     {#1}
10994     {#3}
10995   }
10996 }

```

Deal with the fact that the primitive might not be available.

```

10997 \cs_if_exist:NF \tex_filesize:D
10998 {
10999   \cs_gset:Npn \file_full_name:n #1
11000   {
11001     \msg_expandable_error:nnn
11002     { kernel } { primitive-not-available }
11003     { \(\pdf)filesize }
11004   }
11005 }
11006 \msg_new:nnnn { kernel } { primitive-not-available }
11007 { Primitive~\token_to_str:N #1 not~available }
11008 {
11009   The~version-of~your~TeX~engine~does~not~provide~functionality~equivalent~to~
11010   the~#1~primitive.
11011 }

```

(End definition for `\file_full_name:n` and others. This function is documented on page 94.)

```

\file_get_full_name:nN
\file_get_full_name:VN
\file_get_full_name:nNTF
\file_get_full_name:VNTF
  \_file_get_full_name_search:nN

```

These functions pre-date using `\tex_filesize:D` for file searching, so are `get` functions with protection. To avoid having different search set ups, they are simply wrappers

around the code above.

```

11012 \cs_new_protected:Npn \file_get_full_name:nN #1#2
11013 {
11014     \file_get_full_name:nNF {#1} #2
11015     { \tl_set:Nn #2 { \q_no_value } }
11016 }
11017 \cs_generate_variant:Nn \file_get_full_name:nN { V }
11018 \prg_new_protected_conditional:Npnn \file_get_full_name:nN #1#2 { T , F , TF }
11019 {
11020     \__kernel_tl_set:Nx #2
11021     { \file_full_name:n {#1} }
11022     \tl_if_empty:NTF #2
11023     { \prg_return_false: }
11024     { \prg_return_true: }
11025 }
11026 \cs_generate_variant:Nn \file_get_full_name:nNT { V }
11027 \cs_generate_variant:Nn \file_get_full_name:nNF { V }
11028 \cs_generate_variant:Nn \file_get_full_name:nNTF { V }

```

If `\tex_filesize:D` is not available, the way to test if a file exists is to try to open it: if it does not exist then \TeX reports end-of-file. A search is made looking at each potential path in turn (starting from the current directory). The first location is of course treated as the correct one: this is done by jumping to `\prg_break_point:.` If nothing is found, `#2` is returned empty. A special case when there is no extension is that once the first location is found we test the existence of the file with `.tex` extension in that directory, and if it exists we include the `.tex` extension in the result.

```

11029 \cs_if_exist:NF \tex_filesize:D
11030 {
11031     \prg_set_protected_conditional:Npnn \file_get_full_name:nN #1#2 { T , F , TF }
11032     {
11033         \__kernel_tl_set:Nx \l__file_base_name_tl
11034         { \__kernel_file_name_sanitiz:n {#1} }
11035         \__file_get_full_name_search:nN { } \use:n
11036         \seq_map_inline:Nn \l_file_search_path_seq
11037         { \__file_get_full_name_search:nN { ##1 / } \seq_map_break:n }
11038         \cs_if_exist:NT \input@path
11039         {
11040             \tl_map_inline:Nn \input@path
11041             { \__file_get_full_name_search:nN { ##1 } \tl_map_break:n }
11042         }
11043         \tl_set:Nn \l__file_full_name_tl { \q_no_value }
11044         \prg_break_point:
11045         \quark_if_no_value:NTF \l__file_full_name_tl
11046         {
11047             \ior_close:N \g__file_internal_ior
11048             \prg_return_false:
11049         }
11050         {
11051             \file_parse_full_name:VNNN \l__file_full_name_tl
11052             \l__file_dir_str \l__file_name_str \l__file_ext_str
11053             \str_if_empty:NT \l__file_ext_str
11054             {
11055                 \__kernel_ior_open:No \g__file_internal_ior
11056                 { \l__file_full_name_tl .tex }

```

```

11057         \ior_if_eof:NF \g__file_internal_ior
11058         { \tl_put_right:Nn \l__file_full_name_tl { .tex } }
11059     }
11060     \ior_close:N \g__file_internal_ior
11061     \tl_set_eq:NN #2 \l__file_full_name_tl
11062     \prg_return_true:
11063 }
11064 }
11065 }
11066 \cs_new_protected:Npn \__file_get_full_name_search:nN #1#2
11067 {
11068     \__kernel_tl_set:Nx \l__file_full_name_tl
11069     { \tl_to_str:n {#1} \l__file_base_name_tl }
11070     \__kernel_ior_open:No \g__file_internal_ior \l__file_full_name_tl
11071     \ior_if_eof:NF \g__file_internal_ior { #2 { \prg_break: } }
11072 }

```

(End definition for `\file_get_full_name:nN`, `\file_get_full_name:nNTF`, and `__file_get_full_name_search:nN`. These functions are documented on page 94.)

`\g__file_internal_ior` A reserved stream to test for file existence (if required), and for opening a shell.

```
11073 \ior_new:N \g__file_internal_ior
```

(End definition for `\g__file_internal_ior`.)

`\file_md5five_hash:n` Getting file details by expansion is relatively easy if a bit repetitive. As the MD5 function has a slightly different syntax from the other commands, there is a little cleaning up to do.

```

\file_size:n
\file_timestamp:n
\__file_details:nn
11074 \cs_new:Npn \file_size:n #1
11075 { \__file_details:nn {#1} { size } }
11076 \cs_new:Npn \file_timestamp:n #1
11077 { \__file_details:nn {#1} { moddate } }
11078 \cs_new:Npn \__file_details:nn #1#2
11079 {
11080     \exp_args:Ne \__file_details_aux:nn
11081     { \file_full_name:n {#1} } {#2}
11082 }
11083 \cs_new:Npn \__file_details_aux:nn #1#2
11084 {
11085     \tl_if_blank:nF {#1}
11086     { \use:c { tex_file #2 :D } {#1} }
11087 }
11088 \cs_new:Npn \file_md5five_hash:n #1
11089 { \exp_args:Ne \__file_md5five_hash:n { \file_full_name:n {#1} } }
11090 \cs_new:Npn \__file_md5five_hash:n #1
11091 { \tex_md5fivesum:D file {#1} }

```

(End definition for `\file_md5five_hash:n` and others. These functions are documented on page 96.)

`\file_hex_dump:nnn` These are separate as they need multiple arguments *or* the file size. For LuaTeX, the emulation does not need the file size so we save a little on expansion.

```

\__file_hex_dump_auxi:nnn
\__file_hex_dump_auxii:nnnn
\__file_hex_dump_auxiii:nnnn
\__file_hex_dump_auxiiv:nnn
\file_hex_dump:n
11092 \cs_new:Npn \file_hex_dump:nnn #1#2#3
11093 {
11094     \exp_args:Neee \__file_hex_dump_auxi:nnn
11095     { \file_full_name:n {#1} }

```

```

11096     { \int_eval:n {#2} }
11097     { \int_eval:n {#3} }
11098   }
11099 \cs_new:Npn \__file_hex_dump_auxi:nnn #1#2#3
11100 {
11101   \bool_lazy_any:nF
11102   {
11103     { \tl_if_blank_p:n {#1} }
11104     { \int_compare_p:nNn {#2} = 0 }
11105     { \int_compare_p:nNn {#3} = 0 }
11106   }
11107   {
11108     \exp_args:Ne \__file_hex_dump_auxii:nnnn
11109     { \__file_details_aux:nn {#1} { size } }
11110     {#1} {#2} {#3}
11111   }
11112 }
11113 \cs_new:Npn \__file_hex_dump_auxii:nnnn #1#2#3#4
11114 {
11115   \int_compare:nNnTF {#3} > 0
11116   { \__file_hex_dump_auxiii:nnnn {#3} }
11117   {
11118     \exp_args:Ne \__file_hex_dump_auxiii:nnnn
11119     { \int_eval:n { #1 + #3 } }
11120   }
11121   {#1} {#2} {#4}
11122 }
11123 \cs_new:Npn \__file_hex_dump_auxiii:nnnn #1#2#3#4
11124 {
11125   \int_compare:nNnTF {#4} > 0
11126   { \__file_hex_dump_auxiv:nnn {#4} }
11127   {
11128     \exp_args:Ne \__file_hex_dump_auxiv:nnn
11129     { \int_eval:n { #2 + #4 } }
11130   }
11131   {#1} {#3}
11132 }
11133 \cs_new:Npn \__file_hex_dump_auxiv:nnn #1#2#3
11134 {
11135   \tex_filedump:D
11136   offset ~ \int_eval:n { #2 - 1 } ~
11137   length ~ \int_eval:n { #1 - #2 + 1 }
11138   {#3}
11139 }
11140 \cs_new:Npn \file_hex_dump:n #1
11141 { \exp_args:Ne \__file_hex_dump:n { \file_full_name:n {#1} } }
11142 \sys_if_engine luatex:TF
11143 {
11144   \cs_new:Npn \__file_hex_dump:n #1
11145   {
11146     \tl_if_blank:nF {#1}
11147     { \tex_filedump:D whole {#1} {#1} }
11148   }
11149 }

```



```

11150 {
11151   \cs_new:Npn \__file_hex_dump:n #1
11152   {
11153     \tl_if_blank:nF {#1}
11154     { \tex_filedump:D length \tex_filesize:D {#1} {#1} }
11155   }
11156 }

```

(End definition for `\file_hex_dump:nnn` and others. These functions are documented on page 95.)

```

\file_get_hex_dump:nN
\file_get_hex_dump:nNTF
\file_get_md5fve_hash:nN\file_get_size:nN
\file_get_md5fve_hash:nN\file_get_size:nNTF
\file_get_timestamp:nN
\file_get_timestamp:nNTF
\__file_get_details:nnN
11157 \cs_new_protected:Npn \file_get_hex_dump:nN #1#2
11158 { \file_get_hex_dump:nNF {#1} #2 { \tl_set:Nn #2 { \q_no_value } } }
11159 \cs_new_protected:Npn \file_get_md5fve_hash:nN #1#2
11160 { \file_get_md5fve_hash:nNF {#1} #2 { \tl_set:Nn #2 { \q_no_value } } }
11161 \cs_new_protected:Npn \file_get_size:nN #1#2
11162 { \file_get_size:nNF {#1} #2 { \tl_set:Nn #2 { \q_no_value } } }
11163 \cs_new_protected:Npn \file_get_timestamp:nN #1#2
11164 { \file_get_timestamp:nNF {#1} #2 { \tl_set:Nn #2 { \q_no_value } } }
11165 \prg_new_protected_conditional:Npnn \file_get_hex_dump:nN #1#2 { T , F , TF }
11166 { \__file_get_details:nnN {#1} { hex_dump } #2 }
11167 \prg_new_protected_conditional:Npnn \file_get_md5fve_hash:nN #1#2 { T , F , TF }
11168 { \__file_get_details:nnN {#1} { md5fve_hash } #2 }
11169 \prg_new_protected_conditional:Npnn \file_get_size:nN #1#2 { T , F , TF }
11170 { \__file_get_details:nnN {#1} { size } #2 }
11171 \prg_new_protected_conditional:Npnn \file_get_timestamp:nN #1#2 { T , F , TF }
11172 { \__file_get_details:nnN {#1} { timestamp } #2 }
11173 \cs_new_protected:Npn \__file_get_details:nnN #1#2#3
11174 {
11175   \__kernel_tl_set:Nx #3
11176   { \use:c { file_ #2 :n } {#1} }
11177   \tl_if_empty:NTF #3
11178   { \prg_return_false: }
11179   { \prg_return_true: }
11180 }

```

Where the primitive is not available, issue an error: this is a little more conservative than absolutely needed, but does work.

```

11181 \cs_if_exist:NF \tex_filesize:D
11182 {
11183   \cs_set_protected:Npn \__file_get_details:nnN #1#2#3
11184   {
11185     \tl_clear:N #3
11186     \msg_error:nnx
11187     { kernel } { primitive-not-available }
11188     {
11189       \token_to_str:N \(\pdf)file
11190       \str_case:nn {#2}
11191       {
11192         { hex_dump } { dump }
11193         { md5fve_hash } { md5fvesum }
11194         { timestamp } { moddate }
11195         { size } { size }
11196       }

```

```

11197     }
11198     \prg_return_false:
11199   }
11200 }

```

(End definition for `\file_get_hex_dump:nNTF` and others. These functions are documented on page 95.)

Custom code due to the additional arguments.

```

\file_get_hex_dump:nnnN
\file_get_hex_dump:nnnNTF
11201 \cs_new_protected:Npn \file_get_hex_dump:nnnN #1#2#3#4
11202 {
11203   \file_get_hex_dump:nnnNF {#1} {#2} {#3} #4
11204   { \tl_set:Nn #4 { \q_no_value } }
11205 }
11206 \prg_new_protected_conditional:Npnn \file_get_hex_dump:nnnN #1#2#3#4
11207 { T , F , TF }
11208 {
11209   \__kernel_tl_set:Nx #4
11210   { \file_hex_dump:nnn {#1} {#2} {#3} }
11211   \tl_if_empty:NTF #4
11212   { \prg_return_false: }
11213   { \prg_return_true: }
11214 }

```

(End definition for `\file_get_hex_dump:nnnNTF`. This function is documented on page 95.)

`__file_str_cmp:nn` As we are doing a fixed-length “big” integer comparison, it is easiest to use the low-level behavior of string comparisons.

```

11215 \cs_new_eq:NN \__file_str_cmp:nn \tex_strcmp:D

```

(End definition for `__file_str_cmp:nn`.)

Comparison of file date can be done by using the low-level nature of the string comparison functions.

```

\file_compare_timestamp:p:nNn
\file_compare_timestamp:nNnTF
\__file_compare_timestamp:nnN
\__file_timestamp:n
11216 \prg_new_conditional:Npnn \file_compare_timestamp:nNn #1#2#3
11217 { p , T , F , TF }
11218 {
11219   \exp_args:Nee \__file_compare_timestamp:nnN
11220   { \file_full_name:n {#1} }
11221   { \file_full_name:n {#3} }
11222   #2
11223 }
11224 \cs_new:Npn \__file_compare_timestamp:nnN #1#2#3
11225 {
11226   \tl_if_blank:nTF {#1}
11227   {
11228     \if_charcode:w #3 <
11229     \prg_return_true:
11230   \else:
11231     \prg_return_false:
11232   \fi:
11233 }
11234 {
11235   \tl_if_blank:nTF {#2}
11236   {
11237     \if_charcode:w #3 >

```

```

11238         \prg_return_true:
11239     \else:
11240         \prg_return_false:
11241     \fi:
11242 }
11243 {
11244     \if_int_compare:w
11245         \__file_str_cmp:nn
11246         { \__file_timestamp:n {#1} }
11247         { \__file_timestamp:n {#2} }
11248         #3 \c_zero_int
11249         \prg_return_true:
11250     \else:
11251         \prg_return_false:
11252     \fi:
11253 }
11254 }
11255 }
11256 \cs_new_eq:NN \__file_timestamp:n \tex_filemoddate:D
11257 \cs_if_exist:NF \__file_timestamp:n
11258 {
11259     \prg_set_conditional:Npnn \file_compare_timestamp:nNn #1#2#3
11260     { p , T , F , TF }
11261     {
11262         \msg_expandable_error:nnn
11263         { kernel } { primitive-not-available }
11264         { \pdf)filemoddate }
11265         \prg_return_false:
11266     }
11267 }

```

(End definition for `\file_compare_timestamp:nNnTF`, `__file_compare_timestamp:nnN`, and `__file_timestamp:n`. This function is documented on page 97.)

`\file_if_exist:nTF` The test for the existence of a file is a wrapper around the function to add a path to a file. If the file was found, the path contains something, whereas if the file was not located then the return value is empty.

```

11268 \prg_new_protected_conditional:Npnn \file_if_exist:n #1 { T , F , TF }
11269 {
11270     \file_get_full_name:nNTF {#1} \l__file_full_name_tl
11271     { \prg_return_true: }
11272     { \prg_return_false: }
11273 }

```

(End definition for `\file_if_exist:nTF`. This function is documented on page 94.)

`\file_if_exist_input:n` Input of a file with a test for existence. We do not define the T or TF variants because the most useful place to place the *⟨true code⟩* would be inconsistent with other conditionals.

`\file_if_exist_input:nF`

```

11274 \cs_new_protected:Npn \file_if_exist_input:n #1
11275 {
11276     \file_get_full_name:nNT {#1} \l__file_full_name_tl
11277     { \__file_input:V \l__file_full_name_tl }
11278 }
11279 \cs_new_protected:Npn \file_if_exist_input:nF #1#2

```

```

11280 {
11281   \file_get_full_name:nNTF {#1} \l__file_full_name_tl
11282   { \__file_input:V \l__file_full_name_tl }
11283   {#2}
11284 }

```

(End definition for `\file_if_exist_input:n` and `\file_if_exist_input:nF`. These functions are documented on page 97.)

`\file_input_stop:` A simple rename.

```

11285 \cs_new_protected:Npn \file_input_stop: { \tex_endinput:D }

```

(End definition for `\file_input_stop:`. This function is documented on page 97.)

`__kernel_file_missing:n` An error message for a missing file, also used in `\ior_open:Nn`.

```

11286 \cs_new_protected:Npn \__kernel_file_missing:n #1
11287 {
11288   \msg_error:nnx { kernel } { file-not-found }
11289   { \__kernel_file_name_sanitize:n {#1} }
11290 }

```

(End definition for `__kernel_file_missing:n`.)

`\file_input:n` Loading a file is done in a safe way, checking first that the file exists and loading only if it does. Push the file name on the `\g__file_stack_seq`, and add it to the file list, either `\g__file_record_seq`, or `\@filelist` in package mode.

`__file_input:n`

`__file_input:V`

`__file_input_push:n`

`__kernel_file_input_push:n`

`__file_input_pop:`

`__kernel_file_input_pop:`

`__file_input_pop:nnn`

```

11291 \cs_new_protected:Npn \file_input:n #1
11292 {
11293   \file_get_full_name:nNTF {#1} \l__file_full_name_tl
11294   { \__file_input:V \l__file_full_name_tl }
11295   { \__kernel_file_missing:n {#1} }
11296 }
11297 \cs_new_protected:Npx \__file_input:n #1
11298 {
11299   \exp_not:N \clist_if_exist:NTF \exp_not:N \@filelist
11300   { \exp_not:N \@addtofilelist {#1} }
11301   { \seq_gput_right:Nn \exp_not:N \g__file_record_seq {#1} }
11302   \exp_not:N \__file_input_push:n {#1}
11303   \exp_not:N \tex_input:D
11304   \sys_if_engine luatex:TF
11305   { {#1} }
11306   { \exp_not:N \__kernel_file_name_quote:n {#1} \scan_stop: }
11307   \exp_not:N \__file_input_pop:
11308 }
11309 \cs_generate_variant:Nn \__file_input:n { V }

```

Keeping a track of the file data is easy enough: we store the separated parts so we do not need to parse them twice.

```

11310 \cs_new_protected:Npn \__file_input_push:n #1
11311 {
11312   \seq_gpush:Nx \g__file_stack_seq
11313   {
11314     { \g_file_curr_dir_str }
11315     { \g_file_curr_name_str }
11316     { \g_file_curr_ext_str }

```

```

11317     }
11318     \file_parse_full_name:nNNN {#1}
11319     \l__file_dir_str \l__file_name_str \l__file_ext_str
11320     \str_gset_eq:NN \g_file_curr_dir_str \l__file_dir_str
11321     \str_gset_eq:NN \g_file_curr_name_str \l__file_name_str
11322     \str_gset_eq:NN \g_file_curr_ext_str \l__file_ext_str
11323   }
11324   \cs_new_eq:NN \__kernel_file_input_push:n \__file_input_push:n
11325   \cs_new_protected:Npn \__file_input_pop:
11326   {
11327     \seq_gpop:NN \g_file_stack_seq \l__file_internal_tl
11328     \exp_after:wN \__file_input_pop:nnn \l__file_internal_tl
11329   }
11330   \cs_new_eq:NN \__kernel_file_input_pop: \__file_input_pop:
11331   \cs_new_protected:Npn \__file_input_pop:nnn #1#2#3
11332   {
11333     \str_gset:Nn \g_file_curr_dir_str {#1}
11334     \str_gset:Nn \g_file_curr_name_str {#2}
11335     \str_gset:Nn \g_file_curr_ext_str {#3}
11336   }

```

(End definition for \file_input:n and others. This function is documented on page 97.)

\file_parse_full_name:n
\file_parse_full_name_apply:nN

The main parsing macro \file_parse_full_name_apply:nN passes the file name #1 through __kernel_file_name_sanitizize:n so that we have a single normalised way to treat files internally. \file_parse_full_name:n uses the former, with \prg_do_nothing: to leave each part of the name within a pair of braces.

```

11337   \cs_new:Npn \file_parse_full_name:n #1
11338   {
11339     \file_parse_full_name_apply:nN {#1}
11340     \prg_do_nothing:
11341   }
11342   \cs_new:Npn \file_parse_full_name_apply:nN #1
11343   {
11344     \exp_args:Ne \__file_parse_full_name_auxi:nN
11345     { \__kernel_file_name_sanitizize:n {#1} }
11346   }

```

__file_parse_full_name_auxi:nN
__file_parse_full_name_area:nw

__file_parse_full_name_area:nw splits the file name into chunks separated by /, until the last one is reached. The last chunk is the file name plus the extension, and everything before that is the path. When __file_parse_full_name_area:nw is done, it leaves the path within braces after the scan mark \s__file_stop and proceeds parsing the actual file name.

```

11347   \cs_new:Npn \__file_parse_full_name_auxi:nN #1
11348   {
11349     \__file_parse_full_name_area:nw { } #1
11350     / \s__file_stop
11351   }
11352   \cs_new:Npn \__file_parse_full_name_area:nw #1 #2 / #3 \s__file_stop
11353   {
11354     \tl_if_empty:nTF {#3}
11355     { \__file_parse_full_name_base:nw { } #2 . \s__file_stop {#1} }
11356     { \__file_parse_full_name_area:nw { #1 / #2 } #3 \s__file_stop }
11357   }

```

_file_parse_full_name_base:nw does roughly the same as above, but it separates the chunks at each period. However here there's some extra complications: In case #1 is empty, it is assumed that the extension is actually empty, and the file name is #2. Besides, an extra . has to be added to #2 because it is later removed in _file_parse_full_name_tidy:nnnN. In any case, if there's an extension, it is returned with a leading ..

_file_parse_full_name_base:nw

```

11358 \cs_new:Npn \_file_parse_full_name_base:nw #1 #2 . #3 \s_file_stop
11359 {
11360   \tl_if_empty:nTF {#3}
11361   {
11362     \tl_if_empty:nTF {#1}
11363     {
11364       \tl_if_empty:nTF {#2}
11365       { \_file_parse_full_name_tidy:nnnN { } { } }
11366       { \_file_parse_full_name_tidy:nnnN { .#2 } { } }
11367     }
11368     { \_file_parse_full_name_tidy:nnnN {#1} { .#2 } }
11369   }
11370   { \_file_parse_full_name_base:nw { #1 . #2 } #3 \s_file_stop }
11371 }

```

Now we just need to tidy some bits left loose before. The loop used in the two macros above start with a leading / and . in the file path an name, so here we need to remove them, except in the path, if it is a single /, in which case it's left as is. After all's done, pass to #4.

_file_parse_full_name_tidy:nnnN

```

11372 \cs_new:Npn \_file_parse_full_name_tidy:nnnN #1 #2 #3 #4
11373 {
11374   \exp_args:Nee #4
11375   {
11376     \str_if_eq:nnF {#3} { / } { \use_none:n }
11377     #3 \prg_do_nothing:
11378   }
11379   { \use_none:n #1 \prg_do_nothing: }
11380   {#2}
11381 }

```

(End definition for \file_parse_full_name:n and others. These functions are documented on page 95.)

\file_parse_full_name:nNNN

\file_parse_full_name:VNNN

```

11382 \cs_new_protected:Npn \file_parse_full_name:nNNN #1 #2 #3 #4
11383 {
11384   \file_parse_full_name_apply:nN {#1}
11385   \_file_full_name_assign:nnnNNN #2 #3 #4
11386 }
11387 \cs_new_protected:Npn \_file_full_name_assign:nnnNNN #1 #2 #3 #4 #5 #6
11388 {
11389   \str_set:Nn #4 {#1}
11390   \str_set:Nn #5 {#2}
11391   \str_set:Nn #6 {#3}
11392 }
11393 \cs_generate_variant:Nn \file_parse_full_name:nNNN { V }

```

(End definition for \file_parse_full_name:nNNN. This function is documented on page 95.)

`\file_show_list:` A function to list all files used to the log, without duplicates. In package mode, if
`\file_log_list:` `\@filelist` is still defined, we need to take this list of file names into account (we
`__file_list:N` capture it `\AtBeginDocument` into `\g__file_record_seq`), turning it to a string (this
`__file_list_aux:n` does not affect the commas of this comma list).

```

11394 \cs_new_protected:Npn \file_show_list: { \__file_list:N \msg_show:nnxxxx }
11395 \cs_new_protected:Npn \file_log_list: { \__file_list:N \msg_log:nnxxxx }
11396 \cs_new_protected:Npn \__file_list:N #1
11397 {
11398   \seq_clear:N \l__file_tmp_seq
11399   \clist_if_exist:NT \@filelist
11400   {
11401     \exp_args:NNx \seq_set_from_clist:Nn \l__file_tmp_seq
11402     { \tl_to_str:N \@filelist }
11403   }
11404   \seq_concat:NNN \l__file_tmp_seq \l__file_tmp_seq \g__file_record_seq
11405   \seq_remove_duplicates:N \l__file_tmp_seq
11406   #1 { kernel } { file-list }
11407   { \seq_map_function:NN \l__file_tmp_seq \__file_list_aux:n }
11408   { } { } { }
11409 }
11410 \cs_new:Npn \__file_list_aux:n #1 { \iow_newline: #1 }

```

(End definition for `\file_show_list:` and others. These functions are documented on page 97.)

When used as a package, there is a need to hold onto the standard file list as well as the new one here. File names recorded in `\@filelist` must be turned to strings before being added to `\g__file_record_seq`.

```

11411 \cs_if_exist:NT \@filelist
11412 {
11413   \AtBeginDocument
11414   {
11415     \exp_args:NNx \seq_set_from_clist:Nn \l__file_tmp_seq
11416     { \tl_to_str:N \@filelist }
11417     \seq_gconcat:NNN
11418     \g__file_record_seq
11419     \g__file_record_seq
11420     \l__file_tmp_seq
11421   }
11422 }

```

49.5 GetIdInfo

`\GetIdInfo` As documented in `expl3.dtx` this function extracts file name etc from an SVN Id line. This
`__file_id_info_auxi:w` used to be how we got version number and so on in all modules, so it had to be defined
`__file_id_info_auxii:w` in `l3bootstrap`. Now it's more convenient to define it after we have set up quite a lot of
`__file_id_info_auxiii:w` tools, and `l3file` seems the least unreasonable place for it.

The idea here is to extract out the information needed from a standard SVN Id line, but to avoid a line that would get changed when the file is checked in. Hence the fact that none of the lines here include both a dollar sign and the Id keyword!

```

11423 \cs_new_protected:Npn \GetIdInfo
11424 {
11425   \tl_clear_new:N \ExplFileDescription
11426   \tl_clear_new:N \ExplFileDate

```

```

11427 \tl_clear_new:N \ExplFileName
11428 \tl_clear_new:N \ExplFileExtension
11429 \tl_clear_new:N \ExplFileVersion
11430 \group_begin:
11431 \char_set_catcode_space:n { 32 }
11432 \exp_after:wN
11433 \group_end:
11434 \__file_id_info_auxi:w
11435 }

```

A first check for a completely empty SVN field. If that is not the case, there is a second case when a file created using `svn cp` but has not been checked in. That leaves a special marker `-1` version, which has no further data. Dealing correctly with that is the reason for the space in the line to use `__file_id_info_auxii:w`.

```

11436 \cs_new_protected:Npn \__file_id_info_auxi:w $ #1 $ #2
11437 {
11438   \tl_set:Nn \ExplFileDescription {#2}
11439   \str_if_eq:nnTF {#1} { Id }
11440   {
11441     \tl_set:Nn \ExplFileDate { 0000/00/00 }
11442     \tl_set:Nn \ExplFileName { [unknown] }
11443     \tl_set:Nn \ExplFileExtension { [unknown~extension] }
11444     \tl_set:Nn \ExplFileVersion {-1}
11445   }
11446   { \__file_id_info_auxii:w #1 ~ \s__file_stop }
11447 }

```

Here, `#1` is `Id`, `#2` is the file name, `#3` is the extension, `#4` is the version, `#5` is the check in date and `#6` is the check in time and user, plus some trailing spaces. If `#4` is the marker `-1` value then `#5` and `#6` are empty.

```

11448 \cs_new_protected:Npn \__file_id_info_auxii:w
11449   #1 ~ #2.#3 ~ #4 ~ #5 ~ #6 \s__file_stop
11450 {
11451   \tl_set:Nn \ExplFileName {#2}
11452   \tl_set:Nn \ExplFileExtension {#3}
11453   \tl_set:Nn \ExplFileVersion {#4}
11454   \str_if_eq:nnTF {#4} {-1}
11455   { \tl_set:Nn \ExplFileDate { 0000/00/00 } }
11456   { \__file_id_info_auxiii:w #5 - 0 - 0 - \s__file_stop }
11457 }

```

Convert an SVN-style date into a \LaTeX -style one.

```

11458 \cs_new_protected:Npn \__file_id_info_auxiii:w #1 - #2 - #3 - #4 \s__file_stop
11459 { \tl_set:Nn \ExplFileDate { #1/#2/#3 } }

```

(End definition for `\GetIdInfo` and others. This function is documented on page 10.)

49.6 Checking the version of kernel dependencies

```

\__kernel_dependency_version_check:Nn
\__kernel_dependency_version_check:nn
\__file_kernel_dependency_compare:nnn
\__file_parse_version:w

```

This function is responsible for checking if dependencies of the \LaTeX 3 kernel match the version preloaded in the \LaTeX 2 ϵ kernel. If versions don't match, the function attempts to tell why by searching for a possible stray format file.

The function starts by checking that the kernel date is defined, and if not zero is used to force the error route. The kernel date is then compared with the argument requested

date (usually the packaging date of the dependency). If the kernel date is less than the required date, it's an error and the loading should abort.

```

11460 \cs_new_protected:Npn \__kernel_dependency_version_check:Nn #1
11461 { \exp_args:NV \__kernel_dependency_version_check:nn #1 }
11462 \cs_new_protected:Npn \__kernel_dependency_version_check:nn #1
11463 {
11464   \cs_if_exist:NTF \c__kernel_expl_date_tl
11465   {
11466     \exp_args:NV \__file_kernel_dependency_compare:nnn
11467       \c__kernel_expl_date_tl {#1}
11468   }
11469   { \__file_kernel_dependency_compare:nnn { 0000-00-00 } {#1} }
11470 }
11471 \cs_new_protected:Npn \__file_kernel_dependency_compare:nnn #1 #2 #3
11472 {
11473   \int_compare:nNtT
11474     { \__file_parse_version:w #1 \s__file_stop } <
11475     { \__file_parse_version:w #2 \s__file_stop }
11476   { \__file_mismatched_dependency_error:nn {#2} {#3} }
11477 }
11478 \cs_new:Npn \__file_parse_version:w #1 - #2 - #3 \s__file_stop {#1#2#3}

```

If the versions differ, then we try to give the user some guidance. This function starts by taking the engine name `\c_sys_engine_str` and replacing `tex` by `latex`, then building a command of the form: `kpsewhich -all -engine=<engine> <format>[-dev].fmt` to query the format files available. A shell is opened and each line is read into a sequence.

```

\__file_mismatched_dependency_error:nn
11479 \cs_new_protected:Npn \__file_mismatched_dependency_error:nn #1 #2
11480 {
11481   \exp_args:NNx \ior_shell_open:Nn \g__file_internal_ior
11482   {
11483     kpsewhich ~ --all ~
11484     --engine = \c_sys_engine_exec_str
11485     \c_space_tl \c_sys_engine_format_str
11486     \bool_lazy_and:nnT
11487       { \tl_if_exist_p:N \development@branch@name }
11488       { ! \tl_if_empty_p:N \development@branch@name }
11489     { -dev } .fmt
11490   }
11491   \seq_clear:N \l__file_tmp_seq
11492   \ior_map_inline:Nn \g__file_internal_ior
11493     { \seq_put_right:Nn \l__file_tmp_seq {##1} }
11494   \ior_close:N \g__file_internal_ior
11495   \msg_error:nnnn { kernel } { mismatched-support-file }
11496     {#1} {#2}

```

And finish by ending the current file.

```

11497   \tex_endinput:D
11498 }
11499 % \begin{macrocode}
11500 %
11501 % Now define the actual error message:
11502 % \begin{macrocode}
11503 \msg_new:nnnn { kernel } { mismatched-support-file }
11504 {

```

```

11505     Mismatched~LaTeX~support~files~detected.  \\  

11506     Loading~'~#2'~aborted!

```

`\c__kernel_expl_date_tl` may not exist, due to an older format, so only print the dates when the sentinel token list exists:

```

11507     \tl_if_exist:NT \c__kernel_expl_date_tl  

11508     {  

11509         \\\ \\  

11510         The~L3~programming~layer~in~the~LaTeX~format  \\  

11511         is~dated~\c__kernel_expl_date_tl,~but~in~your~TeX~  

11512         tree~the~files~require  \\\ at~least~#1.  

11513     }  

11514 }  

11515 {

```

The sequence containing the format files should have exactly one item: the format file currently being run. If that's the case, the cause of the error is not that, so print a generic help with some possible causes. If more than one format file was found, then print the list to the user, with appropriate indications of what's in the system and what's in the user tree.

```

11516     \int_compare:nNnTF { \seq_count:N \l__file_tmp_seq } > 1  

11517     {  

11518         The~cause~seems~to~be~an~old~format~file~in~the~user~tree.  \\  

11519         LaTeX~found~these~files:  

11520         \seq_map_tokens:Nn \l__file_tmp_seq { \\\---\use:n } \\  

11521         Try~deleting~the~file~in~the~user~tree~then~run~LaTeX~again.  

11522     }  

11523     {  

11524         The~most~likely~causes~are:  

11525         \\\---A~recent~format~generation~failed;  

11526         \\\---A~stray~format~file~in~the~user~tree~which~needs~  

11527         to~be~removed~or~rebuilt;  

11528         \\\---You~are~running~a~manually~installed~version~of~#2  \\  

11529         \ \ \ which~is~incompatible~with~the~version~in~LaTeX.  \\  

11530     }  

11531     \\  

11532     LaTeX~will~abort~loading~the~incompatible~support~files~  

11533     but~this~may~lead~to  \\\ later~errors.~Please~ensure~that~  

11534     your~LaTeX~format~is~correctly~regenerated.  

11535 }

```

(End definition for `__kernel_dependency_version_check:Nn` and others.)

49.7 Messages

```

11536 \msg_new:nnnn { kernel } { file-not-found }  

11537 { File~'~#1'~not~found. }  

11538 {  

11539     The~requested~file~could~not~be~found~in~the~current~directory,~  

11540     in~the~TeX~search~path~or~in~the~LaTeX~search~path.  

11541 }  

11542 \msg_new:nnn { kernel } { file-list }  

11543 {  

11544     >~File~List~<

```

```

11545     #1 \\
11546     .....
11547 }
11548 \msg_new:nnnn { kernel } { filename-chars-lost }
11549 { #1~invalid~in~file~name.~Lost:~#2. }
11550 {
11551     There~was~an~invalid~token~in~the~file~name~that~caused~
11552     the~characters~following~it~to~be~lost.
11553 }
11554 \msg_new:nnnn { kernel } { filename-missing-endcsname }
11555 { Missing~\iow_char:N\\endcsname~inserted~in~filename. }
11556 {
11557     The~file~name~had~more~\iow_char:N\\csname~commands~than~
11558     \iow_char:N\\endcsname~ones.~LaTeX~will~add~the~missing~
11559     \iow_char:N\\endcsname~and~try~to~continue~as~best~as~it~can.
11560 }
11561 \msg_new:nnnn { kernel } { unbalanced-quote-in-filename }
11562 { Unbalanced~quotes~in~file~name~'~#1'. }
11563 {
11564     File~names~must~contain~balanced~numbers~of~quotes~(").
11565 }
11566 \msg_new:nnnn { kernel } { iow-indent }
11567 { Only~#1~allows~#2 }
11568 {
11569     The~command~#2~can~only~be~used~in~messages~
11570     which~will~be~wrapped~using~#1.
11571     \tl_if_empty:nF {#3} { ~ It~was~called~with~argument~'~#3'. }
11572 }

```

49.8 Functions delayed from earlier modules

<@@=sys>

\c_sys_platform_str Detecting the platform on LuaTeX is easy: for other engines, we use the fact that the two common cases have special null files. It is possible to probe further (see package `platform`), but that requires shell escape and seems unlikely to be useful. This is set up here as it requires file searching.

```

11573 \sys_if_engine luatex:TF
11574 {
11575     \str_const:Nx \c_sys_platform_str
11576     { \tex_directlua:D { tex.print(os.type) } }
11577 }
11578 {
11579     \file_if_exist:nTF { nul: }
11580     {
11581         \file_if_exist:nF { /dev/null }
11582         { \str_const:Nn \c_sys_platform_str { windows } }
11583     }
11584     {
11585         \file_if_exist:nT { /dev/null }
11586         { \str_const:Nn \c_sys_platform_str { unix } }
11587     }
11588 }
11589 \cs_if_exist:NF \c_sys_platform_str

```

```
11590 { \str_const:Nn \c_sys_platform_str { unknown } }
```

(End definition for `\c_sys_platform_str`. This variable is documented on page 74.)

`\sys_if_platform_unix_p:` We can now set up the tests.

`\sys_if_platform_unix:TF`

```
11591 \clist_map_inline:nn { unix , windows }
```

`\sys_if_platform_windows_p:`

```
11592 {
```

`\sys_if_platform_windows:TF`

```
11593   \__file_const:nn { sys_if_platform_ #1 }
```

```
11594     { \str_if_eq_p:Vn \c_sys_platform_str { #1 } }
```

```
11595 }
```

(End definition for `\sys_if_platform_unix:TF` and `\sys_if_platform_windows:TF`. These functions are documented on page 74.)

```
11596 \</package>
```

Chapter 50

l3luatex implementation

11597 $\langle *package \rangle$

50.1 Breaking out to Lua

11598 $\langle *tex \rangle$

11599 $\langle @@=lua \rangle$

```
\__lua_escape:n Copies of primitives.
\__lua_now:n      11600 \cs_new_eq:NN \__lua_escape:n \tex_luaescapestring:D
\__lua_shipout:n 11601 \cs_new_eq:NN \__lua_now:n      \tex_directlua:D
                  11602 \cs_new_eq:NN \__lua_shipout:n \tex_latelua:D
```

(End definition for $\backslash_lua_escape:n$, $\backslash_lua_now:n$, and $\backslash_lua_shipout:n$.)

These functions are set up in l3str for bootstrapping: we want to replace them with a “proper” version at this stage, so clean up.

11603 $\backslash cs_undefine:N \backslash lua_escape:e$

11604 $\backslash cs_undefine:N \backslash lua_now:e$

$\backslash lua_now:n$ Wrappers around the primitives. As with engines other than LuaTeX these have to be
 $\backslash lua_now:e$ macros, we give them the same status in all cases. When LuaTeX is not in use, simply
 $\backslash lua_shipout_e:n$ give an error message/
 $\backslash lua_shipout:n$

```
\lua_now:n      11605 \cs_new:Npn \lua_now:e #1 { \__lua_now:n {#1} }
\lua_now:e      11606 \cs_new:Npn \lua_now:n #1 { \lua_now:e { \exp_not:n {#1} } }
\lua_shipout_e:n 11607 \cs_new_protected:Npn \lua_shipout_e:n #1 { \__lua_shipout:n {#1} }
\lua_shipout:n   11608 \cs_new_protected:Npn \lua_shipout:n #1
                  { \lua_shipout_e:n { \exp_not:n {#1} } }
\lua_escape:n    11610 \cs_new:Npn \lua_escape:e #1 { \__lua_escape:n {#1} }
\lua_escape:e    11611 \cs_new:Npn \lua_escape:n #1 { \lua_escape:e { \exp_not:n {#1} } }
                  11612 \sys_if_engine luatex:F
                  11613 {
                  11614   \clist_map_inline:nn
                  11615   {
                  11616     \lua_escape:n , \lua_escape:e ,
                  11617     \lua_now:n , \lua_now:e
                  11618   }
                  11619   {
                  11620     \cs_set:Npn #1 ##1
```

```

11621         {
11622             \msg_expandable_error:nnn
11623             { luatex } { luatex-required } { #1 }
11624         }
11625     }
11626     \clist_map_inline:nn
11627     { \lua_shipout_e:n , \lua_shipout:n }
11628     {
11629         \cs_set_protected:Npn #1 ##1
11630         {
11631             \msg_error:nnn
11632             { luatex } { luatex-required } { #1 }
11633         }
11634     }
11635 }

```

(End definition for `\lua_now:n` and others. These functions are documented on page 98.)

50.2 Messages

```

11636 \msg_new:nnnn { luatex } { luatex-required }
11637 { LuaTeX~engine~not~in~use!~Ignoring~#1. }
11638 {
11639     The~feature~you~are~using~is~only~available~
11640     with~the~LuaTeX~engine.~LaTeX3~ignored~'~#1'.
11641 }
11642 </tex>

```

50.3 Lua functions for internal use

```

11643 <lua>

```

Most of the emulation of pdfTEX here is based heavily on Heiko Oberdiek's `pdftex-cmds` package.

ltx.utils Create a table for the kernel's own use.

```

11644 ltx = ltx or {}
11645 ltx.utils = ltx.utils or {}
11646 local ltxutils = ltx.utils

```

(End definition for `ltx.utils`. This function is documented on page 99.)

Local copies of global tables.

```

11647 local io      = io
11648 local kpse    = kpse
11649 local lfs     = lfs
11650 local math    = math
11651 local md5     = md5
11652 local os      = os
11653 local string  = string
11654 local tex     = tex
11655 local texio   = texio
11656 local tonumber = tonumber

```

Local copies of standard functions.

```

11657 local abs      = math.abs
11658 local byte      = string.byte
11659 local floor     = math.floor
11660 local format     = string.format
11661 local gsub       = string.gsub
11662 local lfs_attr   = lfs.attributes
11663 local open       = io.open
11664 local os_date    = os.date
11665 local setcatcode = tex.setcatcode
11666 local sprint     = tex.sprint
11667 local cprint     = tex.cprint
11668 local write      = tex.write
11669 local write_nl   = texio.write_nl
11670 local utf8_char  = utf8.char
11671
11672 local scan_int    = token.scan_int or token.scan_integer
11673 local scan_string = token.scan_string
11674 local scan_keyword = token.scan_keyword
11675 local put_next    = token.put_next
11676 local token_create = token.create

```

Since token.create only returns useful values after the tokens has been added to TeX's hash table, we define a variant which defines it first if necessary.

```

11677 local token_create_safe
11678 do
11679   local is_defined = token.is_defined
11680   local set_char   = token.set_char
11681   local runtoks    = tex.runtoks
11682   local let_token  = token_create'let'
11683
11684   function token_create_safe(s)
11685     local orig_token = token_create(s)
11686     if is_defined(s, true) then
11687       return orig_token
11688     end
11689     set_char(s, 0)
11690     local new_token = token_create(s)
11691     runtoks(function()
11692       put_next(let_token, new_token, orig_token)
11693     end)
11694     return new_token
11695   end
11696 end
11697
11698 local true_tok    = token_create_safe'prg_return_true:'
11699 local false_tok   = token_create_safe'prg_return_false:'

```

In ConTeXt lmtx token.command_id does not exist, but it can easily be emulated with ConTeXt's tokens.commands.

```

11700 local command_id = token.command_id
11701 if not command_id and tokens and tokens.commands then
11702   local id_map = tokens.commands
11703   function command_id(name)

```

```

11704     return id_map[name]
11705 end
11706 end

```

Deal with ConT_EXt: doesn't use kpse library.

```

11707 local kpse_find = (resolvers and resolvers.findfile) or kpse.find_file

```

escapehex An internal auxiliary to convert a string to the matching hex escape. This works on a byte basis: extension to handled UTF-8 input is covered in `pdftexcmds` but is not currently required here.

```

11708 local function escapehex(str)
11709     return (gsub(str, ".",
11710         function (ch) return format("%02X", byte(ch)) end))
11711 end

```

(End definition for escapehex.)

ltx.utils.filedump Similar comments here to the next function: read the file in binary mode to avoid any line-end weirdness.

```

11712 local function filedump(name,offset,length)
11713     local file = kpse_find(name,"tex",true)
11714     if not file then return end
11715     local f = open(file,"rb")
11716     if not f then return end
11717     if offset and offset > 0 then
11718         f:seek("set", offset)
11719     end
11720     local data = f:read(length or 'a')
11721     f:close()
11722     return escapehex(data)
11723 end
11724 ltxutils.filedump = filedump

```

(End definition for ltx.utils.filedump. This function is documented on page 99.)

md5.HEX Hash a string and return the hash in uppercase hexadecimal format. In some engines, this is build-in. For traditional LuaT_EX, the conversion to hexadecimal has to be done by us.

```

11725 local md5_HEX = md5.HEX
11726 if not md5_HEX then
11727     local md5_sum = md5.sum
11728     function md5_HEX(data)
11729         return escapehex(md5_sum(data))
11730     end
11731     md5.HEX = md5_HEX
11732 end

```

(End definition for md5.HEX. This function is documented on page ??.)

ltx.utils.filemd5sum Read an entire file and hash it: the hash function itself is a built-in. As Lua is byte-based there is no work needed here in terms of UTF-8 (see `pdftexcmds` and how it handles strings that have passed through LuaT_EX). The file is read in binary mode so that no line ending normalisation occurs.

```

11733 local function filemd5sum(name)

```



```

11734 local file = kpse_find(name, "tex", true) if not file then return end
11735 local f = open(file, "rb") if not f then return end
11736
11737 local data = f:read("*a")
11738 f:close()
11739 return md5_HEX(data)
11740 end
11741 ltxutils.filemd5sum = filemd5sum

```

(End definition for `ltx.utils.filemd5sum`. This function is documented on page 99.)

`ltx.utils.filemoddate` There are two cases: If the C standard library is C99 compliant, we can use `%z` to get the timezone in almost the right format. We only have to add primes and replace a zero or missing offset with Z.

Of course this would be boring, so Windows does things differently. There we have to manually calculate the offset. See procedure `makepdftime` in `utils.c` of pdfTeX.

```

11742 local filemoddate
11743 if os_date'%z':match'^[+-]%d%d%d$' then
11744   local pattern = lpeg.Cs(16 *
11745     (lpeg.Cg(lpeg.S'+-' * '0000' * lpeg.Cc'Z')
11746     + 3 * lpeg.Cc'"'" * 2 * lpeg.Cc'"'"
11747     + lpeg.Cc'Z'))
11748   * -1)
11749   function filemoddate(name)
11750     local file = kpse_find(name, "tex", true)
11751     if not file then return end
11752     local date = lfs_attr(file, "modification")
11753     if not date then return end
11754     return pattern:match(os_date("D:%Y%m%d%H%M%S%z", date))
11755   end
11756 else
11757   local function filemoddate(name)
11758     local file = kpse_find(name, "tex", true)
11759     if not file then return end
11760     local date = lfs_attr(file, "modification")
11761     if not date then return end
11762     local d = os_date("!*t", date)
11763     local u = os_date("!*t", date)
11764     local off = 60 * (d.hour - u.hour) + d.min - u.min
11765     if d.year ~= u.year then
11766       if d.year > u.year then
11767         off = off + 1440
11768       else
11769         off = off - 1440
11770       end
11771     elseif d.yday ~= u.yday then
11772       if d.yday > u.yday then
11773         off = off + 1440
11774       else
11775         off = off - 1440
11776       end
11777     end
11778     local timezone
11779     if off == 0 then

```

```

11780     timezone = "Z"
11781   else
11782     if off < 0 then
11783       timezone = "-"
11784       off = -off
11785     else
11786       timezone = "+"
11787     end
11788     timezone = format("%s%02d'%02d'", timezone, hours // 60, hours % 60)
11789   end
11790   return format("D:%04d%02d%02d%02d%02d%02d%s",
11791     d.year, d.month, d.day, d.hour, d.min, d.sec, timezone)
11792 end
11793 end
11794 ltxutils.filemoddate = filemoddate

```

(End definition for ltx.utils.filemoddate. This function is documented on page 99.)

ltx.utils.filesize A simple disk lookup.

```

11795 local function filesize(name)
11796   local file = kpse_find(name, "tex", true)
11797   if file then
11798     local size = lfs_attr(file, "size")
11799     if size then
11800       return size
11801     end
11802   end
11803 end
11804 ltxutils.filesize = filesize

```

(End definition for ltx.utils.filesize. This function is documented on page 99.)

luaodef An internal function for defining control sequences from Lua which behave like primitives. This acts as a wrapper around token.set_lua which accepts a function instead of an index into the functions table.

```

11805 local luacmd do
11806   local set_lua = token.set_lua
11807   local undefined_cs = command_id'undefined_cs'
11808
11809   if not context and not luatexbase then require'ltlua' end
11810   if luatexbase then
11811     local new_luafunction = luatexbase.new_luafunction
11812     local functions = lua.get_functions_table()
11813     function luacmd(name, func, ...)
11814       local id
11815       local tok = token_create(name)
11816       if tok.command == undefined_cs then
11817         id = new_luafunction(name)
11818         set_lua(name, id, ...)
11819       else
11820         id = tok.index or tok.mode
11821       end
11822       functions[id] = func
11823     end

```

```

11824 elseif context then
11825     local register = context.functions.register
11826     local functions = context.functions.known
11827     function luacmd(name, func, ...)
11828         local tok = token_create(name)
11829         if tok.command == undefined_cs then
11830             token.set_lua(name, register(func), ...)
11831         else
11832             functions[tok.index or tok.mode] = func
11833         end
11834     end
11835 end
11836 end

```

(End definition for luadef.)

50.4 Preserving iniTeX Lua data for runs

```

11837 <@@=lua>

```

The Lua state is not dumped when a forat is written, therefore any Lua variables filled doing format building need to be restored in order to be accessible during normal runs.

We provide some kernel-internal helpers for this. They will only be available if `luatexbase` is available. This is not a big restriction though, because ConT_EXt (which does not use `luatexbase`) does not load `expl3` in the format.

```

11838 local register_luadata, get_luadata
11839
11840 if luatexbase then
11841     local register = token_create'@expl@luadata@bytecode'.index
11842     if status.ini_version then

```

register_luadata `register_luadata` is only available during format generation. It accept a string which uniquely identifies the data object and has to be provided to retrieve it later. Additionally it accepts a function which is called in the `pre_dump` callback and which has to return a string that evaluates to a valid Lua object to be preserved.

```

11843     local luadata, luadata_order = {}, {}
11844
11845     function register_luadata(name, func)
11846         if luadata[name] then
11847             error(format("LaTeX error: data name %q already in use", name))
11848         end
11849         luadata[name] = func
11850         luadata_order[#luadata_order + 1] = func and name
11851     end

```

(End definition for `register_luadata`. This function is documented on page ??.)

The actual work is done in `pre_dump`. The `luadata_order` is used to ensure that the order is consistent over multiple runs.

```

11852     luatexbase.add_to_callback("pre_dump", function()
11853         if next(luadata) then
11854             local str = "return {"
11855             for i=1, #luadata_order do

```

```

11856         local name = luadata_order[i]
11857         str = format('%s[%q]=%s,', str, name, luadata[name]())
11858     end
11859     lua.bytecode[register] = assert(load(str .. "}"))
11860 end
11861 end, "ltx.luadata")
11862 else

```

get_luadata `get_luadata` is only available if data should be restored. It accept the identifier which was used when the data object was registered and returns the associated object. Every object can only be retrieved once.

```

11863     local luadata = lua.bytecode[register]
11864     if luadata then
11865         lua.bytecode[register] = nil
11866         luadata = luadata()
11867     end
11868     function get_luadata(name)
11869         if not luadata then return end
11870         local data = luadata[name]
11871         luadata[name] = nil
11872         return data
11873     end
11874 end
11875 end

```

(End definition for get_luadata. This function is documented on page ??.)

```

11876 </lua>
11877 </package>

```

Chapter 51

13legacy Implementation

```
11878 <*package>
11879 <@@=legacy>

\legacy_if_p:n A friendly wrapper.
\legacy_if:nTF 11880 \prg_new_conditional:Npnn \legacy_if:n #1 { p , T , F , TF }
11881 {
11882     \exp_args:Nc \if_meaning:w { if#1 } \iftrue
11883     \prg_return_true:
11884     \else:
11885     \prg_return_false:
11886     \fi:
11887 }
```

(End definition for `\legacy_if:nTF`. This function is documented on page 100.)

```
\legacy_if_set_true:n A friendly wrapper.
\legacy_if_set_false:n 11888 \cs_new_protected:Npn \legacy_if_set_true:n #1
\legacy_if_gset_true:n 11889 { \cs_set_eq:cN { if#1 } \if_true: }
\legacy_if_gset_false:n 11890 \cs_new_protected:Npn \legacy_if_set_false:n #1
11891 { \cs_set_eq:cN { if#1 } \if_false: }
11892 \cs_new_protected:Npn \legacy_if_gset_true:n #1
11893 { \cs_gset_eq:cN { if#1 } \if_true: }
11894 \cs_new_protected:Npn \legacy_if_gset_false:n #1
11895 { \cs_gset_eq:cN { if#1 } \if_false: }
```

(End definition for `\legacy_if_set_true:n` and others. These functions are documented on page 100.)

```
\legacy_if_set:nn A more elaborate wrapper.
\legacy_if_gset:nn 11896 \cs_new_protected:Npn \legacy_if_set:nn #1#2
11897 {
11898     \bool_if:nTF {#2} \legacy_if_set_true:n \legacy_if_set_false:n
11899     {#1}
11900 }
11901 \cs_new_protected:Npn \legacy_if_gset:nn #1#2
11902 {
11903     \bool_if:nTF {#2} \legacy_if_gset_true:n \legacy_if_gset_false:n
11904     {#1}
11905 }
```

(End definition for \legacy_if_set:nn and \legacy_if_gset:nn. These functions are documented on page 100.)

11906 </package>

Chapter 52

l3tl implementation

```
11907 <*package>
11908 <@@=tl>
```

A token list variable is a \TeX macro that holds tokens. By using the $\varepsilon\text{-TeX}$ primitive \unexpanded inside a \TeX \edef it is possible to store any tokens, including $\#$, in this way.

52.1 Functions

_kernel_tl_set:Nx These two are supplied to get better performance for macros which would otherwise use $\text{_kernel_tl_gset:Nx}$ or \tl_set:Nx or \tl_gset:Nx internally.

```
11909 \cs_new_eq:NN \_kernel\_tl\_set:Nx \cs_set_nopar:Npx
11910 \cs_new_eq:NN \_kernel\_tl\_gset:Nx \cs_gset_nopar:Npx
```

(End definition for _kernel_tl_set:Nx and $\text{_kernel_tl_gset:Nx}$.)

\tl_new:N Creating new token list variables is a case of checking for an existing definition and doing the definition.

\tl_new:c

```
11911 \cs_new_protected:Npn \tl_new:N #1
11912 {
11913   \_kernel_chk_if_free_cs:N #1
11914   \cs_gset_eq:NN #1 \c_empty_tl
11915 }
11916 \cs_generate_variant:Nn \tl_new:N { c }
```

(End definition for \tl_new:N . This function is documented on page [102](#).)

\tl_const:Nn Constants are also easy to generate. They use $\text{\cs_gset_nopar:Npx}$ instead of $\text{_kernel_tl_gset:Nx}$ so that the correct scope checking is applied if \l3debug is used.

\tl_const:Nx

\tl_const:cn

\tl_const:cx

```
11917 \cs_new_protected:Npn \tl_const:Nn #1#2
11918 {
11919   \_kernel_chk_if_free_cs:N #1
11920   \cs_gset_nopar:Npx #1 { \_kernel_exp_not:w {#2} }
11921 }
11922 \cs_new_protected:Npn \tl_const:Nx #1#2
11923 {
11924   \_kernel_chk_if_free_cs:N #1
11925   \cs_gset_nopar:Npx #1 {#2}
```

```

11926   }
11927   \cs_generate_variant:Nn \tl_const:Nn { c }
11928   \cs_generate_variant:Nn \tl_const:Nx { c }

```

(End definition for `\tl_const:Nn`. This function is documented on page 103.)

`\tl_clear:N` Clearing a token list variable means setting it to an empty value. Error checking is sorted out by the parent function.

```

\tl_clear:c
\tl_gclear:N
\tl_gclear:c
11929 \cs_new_protected:Npn \tl_clear:N #1
11930 { \tex_let:D #1 = ~ \c_empty_tl }
11931 \cs_new_protected:Npn \tl_gclear:N #1
11932 { \tex_global:D \tex_let:D #1 ~ \c_empty_tl }
11933 \cs_generate_variant:Nn \tl_clear:N { c }
11934 \cs_generate_variant:Nn \tl_gclear:N { c }

```

(End definition for `\tl_clear:N` and `\tl_gclear:N`. These functions are documented on page 103.)

`\tl_clear_new:N` Clearing a token list variable means setting it to an empty value. Error checking is sorted out by the parent function.

```

\tl_clear_new:c
\tl_gclear_new:N
\tl_gclear_new:c
11935 \cs_new_protected:Npn \tl_clear_new:N #1
11936 { \tl_if_exist:NTF #1 { \tl_clear:N #1 } { \tl_new:N #1 } }
11937 \cs_new_protected:Npn \tl_gclear_new:N #1
11938 { \tl_if_exist:NTF #1 { \tl_gclear:N #1 } { \tl_new:N #1 } }
11939 \cs_generate_variant:Nn \tl_clear_new:N { c }
11940 \cs_generate_variant:Nn \tl_gclear_new:N { c }

```

(End definition for `\tl_clear_new:N` and `\tl_gclear_new:N`. These functions are documented on page 103.)

`\tl_set_eq:NN` For setting token list variables equal to each other. To allow for patching, the arguments have to be explicit. In addition this ensures that a braced second argument will not cause problems.

```

\tl_set_eq:Nc
\tl_set_eq:cN
\tl_set_eq:cc
11941 \cs_new_protected:Npn \tl_set_eq:NN #1#2
11942 { \tex_let:D #1 = ~ #2 }
\tl_gset_eq:NN
11943 \cs_new_protected:Npn \tl_gset_eq:NN #1#2
11944 { \tex_global:D \tex_let:D #1 = ~ #2 }
\tl_gset_eq:Nc
11945 \cs_generate_variant:Nn \tl_set_eq:NN { cN, Nc, cc }
\tl_gset_eq:cN
11946 \cs_generate_variant:Nn \tl_gset_eq:NN { cN, Nc, cc }
\tl_gset_eq:cc

```

(End definition for `\tl_set_eq:NN` and `\tl_gset_eq:NN`. These functions are documented on page 103.)

`\tl_concat:NNN` Concatenating token lists is easy. When checking is turned on, all three arguments must be checked: a token list #2 or #3 equal to `\scan_stop:` would lead to problems later on.

```

\tl_concat:ccc
\tl_gconcat:NNN
\tl_gconcat:ccc
11947 \cs_new_protected:Npn \tl_concat:NNN #1#2#3
11948 {
11949   \__kernel_tl_set:Nx #1
11950   {
11951     \__kernel_exp_not:w \exp_after:wN {#2}
11952     \__kernel_exp_not:w \exp_after:wN {#3}
11953   }
11954 }
11955 \cs_new_protected:Npn \tl_gconcat:NNN #1#2#3
11956 {
11957   \__kernel_tl_gset:Nx #1
11958   {

```



```

11959         \_kernel_exp_not:w \exp_after:wN {#2}
11960         \_kernel_exp_not:w \exp_after:wN {#3}
11961     }
11962 }
11963 \cs_generate_variant:Nn \tl_concat:NNN { ccc }
11964 \cs_generate_variant:Nn \tl_gconcat:NNN { ccc }

```

(End definition for `\tl_concat:NNN` and `\tl_gconcat:NNN`. These functions are documented on page 103.)

`\tl_if_exist_p:N` Copies of the `cs` functions defined in `l3basics`.
`\tl_if_exist_p:c` 11965 `\prg_new_eq_conditional:NNn \tl_if_exist:N \cs_if_exist:N { TF , T , F , p }`
`\tl_if_exist:NTF` 11966 `\prg_new_eq_conditional:NNn \tl_if_exist:c \cs_if_exist:c { TF , T , F , p }`
`\tl_if_exist:cTF`
 (End definition for `\tl_if_exist:NTF`. This function is documented on page 103.)

52.2 Constant token lists

`\c_empty_tl` Never full. We need to define that constant before using `\tl_new:N`.

```

11967 \tl_const:Nn \c_empty_tl { }

```

(End definition for `\c_empty_tl`. This variable is documented on page 118.)

`\c_novalue_tl` A special marker: as we don't have `\char_generate:nn` yet, has to be created the old-fashioned way.

```

11968 \group_begin:
11969 \tex_catcode:D '- = 11 ~
11970 \tl_const:Nx \c_novalue_tl { - NoValue \token_to_str:N - }
11971 \group_end:

```

(End definition for `\c_novalue_tl`. This variable is documented on page 118.)

`\c_space_tl` A space as a token list (as opposed to as a character).

```

11972 \tl_const:Nn \c_space_tl { ~ }

```

(End definition for `\c_space_tl`. This variable is documented on page 118.)

52.3 Adding to token list variables

`\tl_set:Nn` By using `\exp_not:n` token list variables can contain `#` tokens, which makes the token list registers provided by TeX more or less redundant. The `\tl_set:No` version is done by hand as it is used quite a lot.

```

\tl_set:Nv
\tl_set:No
11973 \cs_new_protected:Npn \tl_set:Nn #1#2
11974 { \_kernel_tl_set:Nx #1 { \_kernel_exp_not:w {#2} } }
\tl_set:Nf
11975 \cs_new_protected:Npn \tl_set:No #1#2
11976 { \_kernel_tl_set:Nx #1 { \_kernel_exp_not:w \exp_after:wN {#2} } }
\tl_set:Nx
11977 \cs_new_protected:Npn \tl_set:Nx #1#2
11978 { \_kernel_tl_set:Nx #1 {#2} }
\tl_set:cn
11979 \cs_new_protected:Npn \tl_gset:Nn #1#2
11980 { \_kernel_tl_gset:Nx #1 { \_kernel_exp_not:w {#2} } }
\tl_set:cV
11981 \cs_new_protected:Npn \tl_gset:No #1#2
11982 { \_kernel_tl_gset:Nx #1 { \_kernel_exp_not:w \exp_after:wN {#2} } }
\tl_set:cv
11983 \cs_new_protected:Npn \tl_gset:Nx #1#2

```

```

\tl_gset:Nv
\tl_gset:Nv
\tl_gset:No
\tl_gset:Nf
\tl_gset:Nx
\tl_gset:cn
\tl_gset:cV
\tl_gset:cv
\tl_gset:co
\tl_gset:cf

```

```

11984 { \__kernel_tl_gset:Nx #1 {#2} }
11985 \cs_generate_variant:Nn \tl_set:Nn { NV , Nv , Nf }
11986 \cs_generate_variant:Nn \tl_set:Nn { c , cV , cv , cf }
11987 \cs_generate_variant:Nn \tl_set:Nx { c }
11988 \cs_generate_variant:Nn \tl_set:No { c }
11989 \cs_generate_variant:Nn \tl_gset:Nn { NV , Nv , Nf }
11990 \cs_generate_variant:Nn \tl_gset:Nn { c , cV , cv , cf }
11991 \cs_generate_variant:Nn \tl_gset:Nx { c }
11992 \cs_generate_variant:Nn \tl_gset:No { c }

```

(End definition for `\tl_set:Nn` and `\tl_gset:Nn`. These functions are documented on page 103.)

```

\tl_put_left:Nn Adding to the left is done directly to gain a little performance.
\tl_put_left:NV 11993 \cs_new_protected:Npn \tl_put_left:Nn #1#2
\tl_put_left:No 11994 {
\tl_put_left:Nx 11995   \__kernel_tl_set:Nx #1
\tl_put_left:cn 11996   { \__kernel_exp_not:w {#2} \__kernel_exp_not:w \exp_after:wN {#1} }
\tl_put_left:cV 11997 }
\tl_put_left:co 11998 \cs_new_protected:Npn \tl_put_left:NV #1#2
\tl_put_left:cx 11999 {
\tl_gput_left:Nn 12000   \__kernel_tl_set:Nx #1
\tl_gput_left:NV 12001   { \exp_not:V #2 \__kernel_exp_not:w \exp_after:wN {#1} }
\tl_gput_left:No 12002 }
\tl_gput_left:Nx 12003 \cs_new_protected:Npn \tl_gput_left:No #1#2
\tl_gput_left:cn 12004 {
\tl_gput_left:cV 12005   \__kernel_tl_set:Nx #1
\tl_gput_left:co 12006   {
\tl_gput_left:cx 12007     \__kernel_exp_not:w \exp_after:wN {#2}
12008     \__kernel_exp_not:w \exp_after:wN {#1}
12009   }
12010 }
12011 \cs_new_protected:Npn \tl_put_left:Nx #1#2
12012 { \__kernel_tl_set:Nx #1 { #2 \__kernel_exp_not:w \exp_after:wN {#1} } }
12013 \cs_new_protected:Npn \tl_gput_left:Nn #1#2
12014 {
12015   \__kernel_tl_gset:Nx #1
12016   { \__kernel_exp_not:w {#2} \__kernel_exp_not:w \exp_after:wN {#1} }
12017 }
12018 \cs_new_protected:Npn \tl_gput_left:NV #1#2
12019 {
12020   \__kernel_tl_gset:Nx #1
12021   { \exp_not:V #2 \__kernel_exp_not:w \exp_after:wN {#1} }
12022 }
12023 \cs_new_protected:Npn \tl_gput_left:No #1#2
12024 {
12025   \__kernel_tl_gset:Nx #1
12026   {
12027     \__kernel_exp_not:w \exp_after:wN {#2}
12028     \__kernel_exp_not:w \exp_after:wN {#1}
12029   }
12030 }
12031 \cs_new_protected:Npn \tl_gput_left:Nx #1#2
12032 { \__kernel_tl_gset:Nx #1 { #2 \__kernel_exp_not:w \exp_after:wN {#1} } }
12033 \cs_generate_variant:Nn \tl_put_left:Nn { c }

```

```

12034 \cs_generate_variant:Nn \tl_put_left:NV { c }
12035 \cs_generate_variant:Nn \tl_put_left:No { c }
12036 \cs_generate_variant:Nn \tl_put_left:Nx { c }
12037 \cs_generate_variant:Nn \tl_gput_left:Nn { c }
12038 \cs_generate_variant:Nn \tl_gput_left:NV { c }
12039 \cs_generate_variant:Nn \tl_gput_left:No { c }
12040 \cs_generate_variant:Nn \tl_gput_left:Nx { c }

```

(End definition for `\tl_put_left:Nn` and `\tl_gput_left:Nn`. These functions are documented on page 103.)

`\tl_put_right:Nn` The same on the right.

```

\tl_put_right:NV 12041 \cs_new_protected:Npn \tl_put_right:Nn #1#2
\tl_put_right:No 12042 { \__kernel_tl_set:Nx #1 { \__kernel_exp_not:w \exp_after:wN { #1 #2 } } }
\tl_put_right:Nx 12043 \cs_new_protected:Npn \tl_put_right:NV #1#2
\tl_put_right:cn 12044 {
\tl_put_right:cV 12045   \__kernel_tl_set:Nx #1
\tl_put_right:co 12046   { \__kernel_exp_not:w \exp_after:wN {#1} \exp_not:V #2 }
\tl_put_right:cx 12047 }
\tl_gput_right:Nn 12048 \cs_new_protected:Npn \tl_put_right:No #1#2
\tl_gput_right:NV 12049 {
\tl_gput_right:No 12050   \__kernel_tl_set:Nx #1
\tl_gput_right:Nx 12051   {
\tl_gput_right:cn 12052     \__kernel_exp_not:w \exp_after:wN {#1}
\tl_gput_right:cV 12053     \__kernel_exp_not:w \exp_after:wN {#2}
\tl_gput_right:co 12054     }
\tl_gput_right:cx 12055   }
\tl_gput_right:Nx 12056 \cs_new_protected:Npn \tl_put_right:Nx #1#2
\tl_gput_right:No 12057 { \__kernel_tl_set:Nx #1 { \__kernel_exp_not:w \exp_after:wN {#1} #2 } }
\tl_gput_right:Nn 12058 \cs_new_protected:Npn \tl_gput_right:Nn #1#2
\tl_gput_right:NV 12059 { \__kernel_tl_gset:Nx #1 { \__kernel_exp_not:w \exp_after:wN { #1 #2 } } }
\tl_gput_right:No 12060 \cs_new_protected:Npn \tl_gput_right:NV #1#2
\tl_gput_right:Nx 12061 {
\tl_gput_right:No 12062   \__kernel_tl_gset:Nx #1
\tl_gput_right:No 12063   { \__kernel_exp_not:w \exp_after:wN {#1} \exp_not:V #2 }
\tl_gput_right:Nx 12064   }
\tl_gput_right:No 12065 \cs_new_protected:Npn \tl_gput_right:No #1#2
\tl_gput_right:Nx 12066 {
\tl_gput_right:No 12067   \__kernel_tl_gset:Nx #1
\tl_gput_right:Nx 12068   {
\tl_gput_right:No 12069     \__kernel_exp_not:w \exp_after:wN {#1}
\tl_gput_right:Nx 12070     \__kernel_exp_not:w \exp_after:wN {#2}
\tl_gput_right:Nx 12071     }
\tl_gput_right:Nx 12072   }
\tl_gput_right:Nx 12073 \cs_new_protected:Npn \tl_gput_right:Nx #1#2
\tl_gput_right:Nx 12074 { \__kernel_tl_gset:Nx #1 { \__kernel_exp_not:w \exp_after:wN {#1} #2 } }
\tl_gput_right:Nx 12075 \cs_generate_variant:Nn \tl_put_right:Nn { c }
\tl_gput_right:Nx 12076 \cs_generate_variant:Nn \tl_put_right:NV { c }
\tl_gput_right:Nx 12077 \cs_generate_variant:Nn \tl_put_right:No { c }
\tl_gput_right:Nx 12078 \cs_generate_variant:Nn \tl_put_right:Nx { c }
\tl_gput_right:Nx 12079 \cs_generate_variant:Nn \tl_gput_right:Nn { c }
\tl_gput_right:Nx 12080 \cs_generate_variant:Nn \tl_gput_right:NV { c }
\tl_gput_right:Nx 12081 \cs_generate_variant:Nn \tl_gput_right:No { c }
\tl_gput_right:Nx 12082 \cs_generate_variant:Nn \tl_gput_right:Nx { c }

```

(End definition for `\tl_put_right:Nn` and `\tl_gput_right:Nn`. These functions are documented on page 104.)

52.4 Internal quarks and quark-query functions

```
\q__tl_nil Internal quarks.
\q__tl_mark 12083 \quark_new:N \q__tl_nil
\q__tl_stop 12084 \quark_new:N \q__tl_mark
12085 \quark_new:N \q__tl_stop
```

(End definition for `\q__tl_nil`, `\q__tl_mark`, and `\q__tl_stop`.)

```
\q__tl_recursion_tail Internal recursion quarks.
\q__tl_recursion_stop 12086 \quark_new:N \q__tl_recursion_tail
12087 \quark_new:N \q__tl_recursion_stop
```

(End definition for `\q__tl_recursion_tail` and `\q__tl_recursion_stop`.)

```
\_tl_if_recursion_tail_break:nN Functions to query recursion quarks.
\_tl_if_recursion_tail_stop_p:n 12088 \__kernel_quark_new_test:N \_tl_if_recursion_tail_break:nN
\_tl_if_recursion_tail_stop:nTF 12089 \__kernel_quark_new_conditional:Nn \_tl_if_recursion_tail_stop:nTF { TF }
```

(End definition for `_tl_if_recursion_tail_break:nN` and `_tl_if_recursion_tail_stop:nTF`.)

52.5 Reassigning token list category codes

`\c__tl_rescan_marker_tl` The rescanning code needs a special token list containing the same character (chosen here to be a colon) with two different category codes: it cannot appear in the tokens being rescanned since all colons have the same category code.

```
12090 \tl_const:Nx \c__tl_rescan_marker_tl { : \token_to_str:N : }
```

(End definition for `\c__tl_rescan_marker_tl`.)

```
\tl_set_rescan:Nnn In a group, after some initial setup explained below and the user setup #3 (followed by
\tl_set_rescan:Nno \scan_stop: to be safe), there is a call to \__tl_set_rescan:nNN. This shared auxiliary
\tl_set_rescan:Nnx defined later distinguishes single-line and multi-line “files”. In the simplest case of multi-
\tl_set_rescan:cnm line files, it calls (with the same arguments) \__tl_set_rescan_multi:nNN, whose code
\tl_set_rescan:cno is included here to help understand the approach. This function rescans its argument #1,
\tl_set_rescan:cnx closes the group, and performs the assignment.
```

```
\tl_gset_rescan:Nnn One difficulty when rescanning is that \scantokens treats the argument as a file,
\tl_gset_rescan:Nno and without the correct settings a TeX error occurs:
```

```
\tl_gset_rescan:Nnx ! File ended while scanning definition of ...
\tl_gset_rescan:cnm
```

```
\tl_gset_rescan:cno A related minor issue is a warning due to opening a group before the \scantokens and
\tl_gset_rescan:cnx closing it inside that temporary file; we avoid that by setting \tracingnesting. The
\tl_rescan:nx standard solution to the “File ended” error is to grab the rescanned tokens as a delimited
\_tl_rescan_aux: argument of an auxiliary, here \_tl_rescan:NNw, that performs the assignment, then let
\_tl_set_rescan:NNnn TeX “execute” the end of file marker. As usual in delimited arguments we use \prg_do_
\_tl_set_rescan_multi:nNN nothing: to avoid stripping an outer set braces: this is removed by using o-expanding
\_tl_rescan:NNw assignments. The delimiter cannot appear within the rescanned token list because it
contains twice the same character, with different catcodes.
```

For `\tl_rescan:nn` we cannot simply call `__tl_set_rescan:NNnn \prg_do_nothing: \use:n` because that would leave the end-of-file marker *after* the result of rescanning. If that rescanned result is code that looks further in the input stream for arguments, it would break.

For multi-line files the only subtlety is that `\newlinechar` should be equal to `\endlinechar` because `\newlinechar` characters become new lines and then become `\endlinechar` characters when writing to an abstract file and reading back. This equality is ensured by setting `\newlinechar` equal to `\endlinechar`. Prior to this, `\endlinechar` is set to `-1` if it was `32` (in particular true after `\ExplSyntaxOn`) to avoid unreasonable line-breaks at every space for instance in error messages triggered by the user setup. Another side effect of reading back from the file is that spaces (catcode 10) are ignored at the beginning of lines, and spaces and tabs (character code 32 and 9) are ignored at the end of lines.

The two `\if_false: ... \fi:` are there to prevent alignment tabs to cause a change of tabular cell while rescanning. We put the “opening” one after `\group_begin:` so that if one accidentally `f`-expands `\tl_set_rescan:Nnn` braces remain balanced. This is essential in `e`-type arguments when `\expanded` is not available.

```

12091 \cs_new_protected:Npn \tl_rescan:nn #1#2
12092 {
12093   \tl_set_rescan:Nnn \l__tl_internal_a_tl {#1} {#2}
12094   \exp_after:wN \__tl_rescan_aux:
12095   \l__tl_internal_a_tl
12096 }
12097 \exp_args:NNo \cs_new_protected:Npn \__tl_rescan_aux:
12098 { \tl_clear:N \l__tl_internal_a_tl }
12099 \cs_new_protected:Npn \tl_set_rescan:Nnn
12100 { \__tl_set_rescan:NNnn \tl_set:No }
12101 \cs_new_protected:Npn \tl_gset_rescan:Nnn
12102 { \__tl_set_rescan:NNnn \tl_gset:No }
12103 \cs_new_protected:Npn \__tl_set_rescan:NNnn #1#2#3#4
12104 {
12105   \group_begin:
12106   \if_false: { \fi:
12107     \int_set_eq:NN \tex_tracingnesting:D \c_zero_int
12108     \int_compare:nNnT \tex_endlinechar:D = { 32 }
12109     { \int_set:Nn \tex_endlinechar:D { -1 } }
12110     \int_set_eq:NN \tex_newlinechar:D \tex_endlinechar:D
12111     #3 \scan_stop:
12112     \exp_args:No \__tl_set_rescan:nNN { \tl_to_str:n {#4} } #1 #2
12113   \if_false: } \fi:
12114 }
12115 \cs_new_protected:Npn \__tl_set_rescan_multi:nNN #1#2#3
12116 {
12117   \tex_everyeof:D \exp_after:wN { \c__tl_rescan_marker_tl }
12118   \exp_after:wN \__tl_rescan:NNw
12119   \exp_after:wN #2
12120   \exp_after:wN #3
12121   \exp_after:wN \prg_do_nothing:
12122   \tex_scantokens:D {#1}
12123 }
12124 \exp_args:Nno \use:nn
12125 { \cs_new:Npn \__tl_rescan:NNw #1#2#3 } \c__tl_rescan_marker_tl

```

```

12126 {
12127   \group_end:
12128   #1 #2 {#3}
12129 }
12130 \cs_generate_variant:Nn \tl_set_rescan:Nnn { Nno , Nnx }
12131 \cs_generate_variant:Nn \tl_set_rescan:Nnn { c , cno , cnx }
12132 \cs_generate_variant:Nn \tl_gset_rescan:Nnn { Nno , Nnx }
12133 \cs_generate_variant:Nn \tl_gset_rescan:Nnn { c , cno }

```

(End definition for `\tl_set_rescan:Nnn` and others. These functions are documented on page 117.)

<pre> __tl_set_rescan:nNN __tl_set_rescan_single:nnNN __tl_set_rescan_single_aux:nnnNN __tl_set_rescan_single_aux:w </pre>	<p>The function <code>__tl_set_rescan:nNN</code> calls <code>__tl_set_rescan_multi:nNN</code> or <code>__tl_set_rescan_single:nnNN { ' }</code> depending on whether its argument is a single-line fragment of code/data or is made of multiple lines by testing for the presence of a <code>\newlinechar</code> character. If <code>\newlinechar</code> is out of range, the argument is assumed to be a single line.</p>
--	---

For a single line, no `\endlinechar` should be added, so it is set to `-1`, and spaces should not be removed. Trailing spaces and tabs are a difficult matter, as `TEX` removes these at a very low level. The only way to preserve them is to rescan not the argument but the argument followed by a character with a reasonable category code. Here, `11` (letter) and `12` (other) are accepted, as these are convenient, suitable for delimiting an argument, and it is very unlikely that none of the ASCII characters are in one of these categories. To avoid selecting one particular character to put at the end, whose category code may have been modified, there is a loop through characters from `'` (ASCII 39) to `~` (ASCII 127). The choice of starting point was made because this is the start of a very long range of characters whose standard category is letter or other, thus minimizing the number of steps needed by the loop (most often just a single one). If no valid character is found (very rare), fall-back on `__tl_set_rescan_multi:nNN`.

Otherwise, once a valid character is found (let us use `'` in this explanation) run some code very similar to `__tl_set_rescan_multi:nNN` but with `'` added at both ends of the input. Of course, we need to define the auxiliary `__tl_set_rescan_single:NNww` on the fly to remove the additional `'` that is just before `::` (by which we mean `\c__tl_rescan_marker_tl`). Note that the argument must be delimited by `'` with the current catcode; this is done thanks to `\char_generate:nn`. Yet another issue is that the rescanned token list may contain a comment character, in which case the `'` we expected is not there. We fix this as follows: rather than just `::` we set `\everyeof to ::{<code1>}'::{<code2>}` `\s__tl_stop`. The auxiliary `__tl_set_rescan_single:NNww` runs the `o`-expanding assignment, expanding either `<code1>` or `<code2>` before its the main argument `#3`. In the typical case without comment character, `<code1>` is expanded, removing the leading `'`. In the rarer case with comment character, `<code2>` is expanded, calling `__tl_set_rescan_single_aux:w`, which removes the trailing `::{<code1>}` and the leading `'`.

```

12134 \cs_new_protected:Npn \__tl_set_rescan:nNN #1
12135 {
12136   \int_compare:nNnTF \tex_newlinechar:D < 0
12137   { \use_ii:nn }
12138   {
12139     \exp_args:Nnf \tl_if_in:nnTF {#1}
12140     { \char_generate:nn { \tex_newlinechar:D } { 12 } }
12141   }
12142   { \__tl_set_rescan_multi:nNN }
12143   {

```

```

12144         \int_set:Nn \tex_endlinechar:D { -1 }
12145         \__tl_set_rescan_single:nnNN { ' ' }
12146     }
12147     {#1}
12148 }
12149 \cs_new_protected:Npn \__tl_set_rescan_single:nnNN #1
12150 {
12151     \int_compare:nNnTF
12152     { \char_value_catcode:n {#1} / 2 } = 6
12153     {
12154         \exp_args:Nof \__tl_set_rescan_single_aux:nnnNN
12155         \c__tl_rescan_marker_tl
12156         { \char_generate:nn {#1} { \char_value_catcode:n {#1} } }
12157     }
12158     {
12159         \int_compare:nNnTF {#1} < { '\~ }
12160         {
12161             \exp_args:Nf \__tl_set_rescan_single:nnNN
12162             { \int_eval:n { #1 + 1 } }
12163         }
12164         { \__tl_set_rescan_multi:nnN }
12165     }
12166 }
12167 \cs_new_protected:Npn \__tl_set_rescan_single_aux:nnnNN #1#2#3#4#5
12168 {
12169     \tex_everyeof:D
12170     {
12171         #1 \use_none:n
12172         #2 #1 { \exp:w \__tl_set_rescan_single_aux:w }
12173         \s__tl_stop
12174     }
12175     \cs_set:Npn \__tl_rescan:NNw ##1##2##3 #2 #1 ##4 ##5 \s__tl_stop
12176     {
12177         \group_end:
12178         ##1 ##2 { ##4 ##3 }
12179     }
12180     \exp_after:wN \__tl_rescan:NNw
12181     \exp_after:wN #4
12182     \exp_after:wN #5
12183     \tex_scantokens:D { #2 #3 #2 }
12184 }
12185 \exp_args:Nno \use:nn
12186 { \cs_new:Npn \__tl_set_rescan_single_aux:w #1 }
12187 \c__tl_rescan_marker_tl #2
12188 { \use_i:nn \exp_end: #1 }

```

(End definition for `__tl_set_rescan:nnN` and others.)

52.6 Modifying token list variables

`\tl_replace_all:Nnn` All of the `replace` functions call `__tl_replace:NnNNnn` with appropriate arguments.
`\tl_replace_all:cnn` The first two arguments are explained later. The next controls whether the replacement
`\tl_greplace_all:Nnn` function calls itself (`__tl_replace_next:w`) or stops (`__tl_replace_wrap:w`) after
`\tl_greplace_all:cnn`
`\tl_replace_once:Nnn`
`\tl_replace_once:cnn`
`\tl_greplace_once:Nnn`
`\tl_greplace_once:cnn`

the first replacement. Next comes an x-type assignment function `\tl_set:Nx` or `\tl_gset:Nx` for local or global replacements. Finally, the three arguments $\langle tl\ var \rangle \{ \langle pattern \rangle \} \{ \langle replacement \rangle \}$ provided by the user. When describing the auxiliary functions below, we denote the contents of the $\langle tl\ var \rangle$ by $\langle token\ list \rangle$.

```

12189 \cs_new_protected:Npn \tl_replace_once:Nnn
12190   { \__tl_replace:NnNNNnn \q__tl_mark ? \__tl_replace_wrap:w \__kernel_tl_set:Nx }
12191 \cs_new_protected:Npn \tl_greplace_once:Nnn
12192   { \__tl_replace:NnNNNnn \q__tl_mark ? \__tl_replace_wrap:w \__kernel_tl_gset:Nx }
12193 \cs_new_protected:Npn \tl_replace_all:Nnn
12194   { \__tl_replace:NnNNNnn \q__tl_mark ? \__tl_replace_next:w \__kernel_tl_set:Nx }
12195 \cs_new_protected:Npn \tl_greplace_all:Nnn
12196   { \__tl_replace:NnNNNnn \q__tl_mark ? \__tl_replace_next:w \__kernel_tl_gset:Nx }
12197 \cs_generate_variant:Nn \tl_replace_once:Nnn { c }
12198 \cs_generate_variant:Nn \tl_greplace_once:Nnn { c }
12199 \cs_generate_variant:Nn \tl_replace_all:Nnn { c }
12200 \cs_generate_variant:Nn \tl_greplace_all:Nnn { c }

```

(End definition for `\tl_replace_all:Nnn` and others. These functions are documented on page 116.)

```

\__tl_replace:NnNNNnn
\__tl_replace_auxi:NnnNNNnn
\__tl_replace_auxii:nNNNNnn
\__tl_replace_next:w
\__tl_replace_next_aux:w
\__tl_replace_wrap:w

```

To implement the actual replacement auxiliary `__tl_replace_auxii:nNNNNnn` we need a $\langle delimiter \rangle$ with the following properties:

- all occurrences of the $\langle pattern \rangle$ #6 in “ $\langle token\ list \rangle \langle delimiter \rangle$ ” belong to the $\langle token\ list \rangle$ and have no overlap with the $\langle delimiter \rangle$,
- the first occurrence of the $\langle delimiter \rangle$ in “ $\langle token\ list \rangle \langle delimiter \rangle$ ” is the trailing $\langle delimiter \rangle$.

We first find the building blocks for the $\langle delimiter \rangle$, namely two tokens $\langle A \rangle$ and $\langle B \rangle$ such that $\langle A \rangle$ does not appear in #6 and #6 is not $\langle B \rangle$ (this condition is trivial if #6 has more than one token). Then we consider the delimiters “ $\langle A \rangle$ ” and “ $\langle A \rangle \langle A \rangle^n \langle B \rangle \langle A \rangle^n \langle B \rangle$ ”, for $n \geq 1$, where $\langle A \rangle^n$ denotes n copies of $\langle A \rangle$, and we choose as our $\langle delimiter \rangle$ the first one which is not in the $\langle token\ list \rangle$.

Every delimiter in the set obeys the first condition: #6 does not contain $\langle A \rangle$ hence cannot be overlapping with the $\langle token\ list \rangle$ and the $\langle delimiter \rangle$, and it cannot be within the $\langle delimiter \rangle$ since it would have to be in one of the two $\langle B \rangle$ hence be equal to this single token (or empty, but this is an error case filtered separately). Given the particular form of these delimiters, for which no prefix is also a suffix, the second condition is actually a consequence of the weaker condition that the $\langle delimiter \rangle$ we choose does not appear in the $\langle token\ list \rangle$. Additionally, the set of delimiters is such that a $\langle token\ list \rangle$ of n tokens can contain at most $O(n^{1/2})$ of them, hence we find a $\langle delimiter \rangle$ with at most $O(n^{1/2})$ tokens in a time at most $O(n^{3/2})$. Bear in mind that these upper bounds are reached only in very contrived scenarios: we include the case “ $\langle A \rangle$ ” in the list of delimiters to try, so that the $\langle delimiter \rangle$ is simply `\q__tl_mark` in the most common situation where neither the $\langle token\ list \rangle$ nor the $\langle pattern \rangle$ contains `\q__tl_mark`.

Let us now ahead, optimizing for this most common case. First, two special cases: an empty $\langle pattern \rangle$ #6 is an error, and if #1 is absent from both the $\langle token\ list \rangle$ #5 and the $\langle pattern \rangle$ #6 then we can use it as the $\langle delimiter \rangle$ through `__tl_replace_auxii:nNNNNnn {#1}`. Otherwise, we end up calling `__tl_replace:NnNNNnn` repeatedly with the first two arguments `\q__tl_mark {?}`, `\? {??}`, `\?? {???}`, and so on, until #6 does not contain the control sequence #1, which we take as our $\langle A \rangle$. The argument #2 only serves to collect ? characters for #1. Note that the order of the tests means that the

first two are done every time, which is wasteful (for instance, we repeatedly test for the emptiness of #6). However, this is rare enough not to matter. Finally, choose $\langle B \rangle$ to be $\backslash\text{q_tl_nil}$ or $\backslash\text{q_tl_stop}$ such that it is not equal to #6.

The $\backslash\text{__tl_replace_auxi:NnnNNNnn}$ auxiliary receives $\{\langle A \rangle\}$ and $\{\langle A \rangle^n \langle B \rangle\}$ as its arguments, initially with $n = 1$. If “ $\langle A \rangle \langle A \rangle^n \langle B \rangle \langle A \rangle^n \langle B \rangle$ ” is in the $\langle token list \rangle$ then increase n and try again. Once it is not anymore in the $\langle token list \rangle$ we take it as our $\langle delimiter \rangle$ and pass this to the auxii auxiliary.

```

12201 \cs_new_protected:Npn \__tl_replace:NnnNNNnn #1#2#3#4#5#6#7
12202 {
12203   \tl_if_empty:nTF {#6}
12204   {
12205     \msg_error:nnx { kernel } { empty-search-pattern }
12206     { \tl_to_str:n {#7} }
12207   }
12208   {
12209     \tl_if_in:ontF { #5 #6 } {#1}
12210     {
12211       \tl_if_in:nnTF {#6} {#1}
12212       { \exp_args:Nc \__tl_replace:NnnNNNnn {#2} {#2?} }
12213       {
12214         \__tl_quark_if_nil:nTF {#6}
12215         { \__tl_replace_auxi:NnnNNNnn #5 {#1} { #1 \q_tl_stop } }
12216         { \__tl_replace_auxi:NnnNNNnn #5 {#1} { #1 \q_tl_nil } }
12217       }
12218     }
12219     { \__tl_replace_auxii:nNNNnn {#1} }
12220     #3#4#5 {#6} {#7}
12221   }
12222 }
12223 \cs_new_protected:Npn \__tl_replace_auxi:NnnNNNnn #1#2#3
12224 {
12225   \tl_if_in:NnTF #1 { #2 #3 #3 }
12226   { \__tl_replace_auxi:NnnNNNnn #1 { #2 #3 } {#2} }
12227   { \__tl_replace_auxii:nNNNnn { #2 #3 #3 } }
12228 }

```

The auxiliary $\backslash\text{__tl_replace_auxii:nNNNnn}$ receives the following arguments:

$\{\langle delimiter \rangle\}$ $\langle function \rangle$ $\langle assignment \rangle$
 $\langle tl var \rangle$ $\{\langle pattern \rangle\}$ $\{\langle replacement \rangle\}$

All of its work is done between $\backslash\text{group_align_safe_begin:}$ and $\backslash\text{group_align_safe_end:}$ to avoid issues in alignments. It does the actual replacement within #3 #4 {...}, an x-expanding $\langle assignment \rangle$ #3 to the $\langle tl var \rangle$ #4. The auxiliary $\backslash\text{__tl_replace_next:w}$ is called, followed by the $\langle token list \rangle$, some tokens including the $\langle delimiter \rangle$ #1, followed by the $\langle pattern \rangle$ #5. This auxiliary finds an argument delimited by #5 (the presence of a trailing #5 avoids runaway arguments) and calls $\backslash\text{__tl_replace_wrap:w}$ to test whether this #5 is found within the $\langle token list \rangle$ or is the trailing one.

If on the one hand it is found within the $\langle token list \rangle$, then ##1 cannot contain the $\langle delimiter \rangle$ #1 that we worked so hard to obtain, thus $\backslash\text{__tl_replace_wrap:w}$ gets ##1 as its own argument ##1, and protects it against the x-expanding assignment. It also finds $\backslash\text{exp_not:n}$ as ##2 and does nothing to it, thus letting through $\backslash\text{exp_not:n}$ $\{\langle replacement \rangle\}$ into the assignment. Note that $\backslash\text{__tl_replace_next:w}$ and $\backslash\text{__tl_replace_wrap:w}$ are always called followed by two empty brace groups. These are safe

because no delimiter can match them. They prevent losing braces when grabbing delimited arguments, but require the use of `\exp_not:o` and `\use_none:nn`, rather than simply `\exp_not:n`. Afterwards, `__tl_replace_next:w` is called to repeat the replacement, or `__tl_replace_wrap:w` if we only want a single replacement. In this second case, `##1` is the *remaining tokens* in the *token list* and `##2` is some *ending code* which ends the assignment and removes the trailing tokens `#5` using some `\if_false: { \fi: }` trickery because `#5` may contain any delimiter.

If on the other hand the argument `##1` of `__tl_replace_next:w` is delimited by the trailing *pattern* `#5`, then `##1` is “`{ } { }` *token list* *delimiter* `{ ending code }`”, hence `__tl_replace_wrap:w` finds “`{ } { }` *token list*” as `##1` and the *ending code* as `##2`. It leaves the *token list* into the assignment and unbraces the *ending code* which removes what remains (essentially the *delimiter* and *replacement*).

```

12229 \cs_new_protected:Npn \__tl_replace_auxii:nNNNnn #1#2#3#4#5#6
12230 {
12231   \group_align_safe_begin:
12232   \cs_set:Npn \__tl_replace_wrap:w ##1 #1 ##2
12233     { \__kernel_exp_not:w \exp_after:wN { \use_none:nn ##1 } ##2 }
12234   \cs_set:Npx \__tl_replace_next:w ##1 #5
12235   {
12236     \exp_not:N \__tl_replace_wrap:w ##1
12237     \exp_not:n { #1 }
12238     \exp_not:n { \exp_not:n {#6} }
12239     \exp_not:n { #2 { } { } }
12240   }
12241   #3 #4
12242   {
12243     \exp_after:wN \__tl_replace_next_aux:w
12244     #4
12245     #1
12246     {
12247       \if_false: { \fi: }
12248       \exp_after:wN \use_none:n \exp_after:wN { \if_false: } \fi:
12249     }
12250     #5
12251   }
12252   \group_align_safe_end:
12253 }
12254 \cs_new:Npn \__tl_replace_next_aux:w { \__tl_replace_next:w { } { } }
12255 \cs_new_eq:NN \__tl_replace_wrap:w ?
12256 \cs_new_eq:NN \__tl_replace_next:w ?

```

(End definition for `__tl_replace:NnNNNnn` and others.)

`\tl_remove_once:Nn`
`\tl_remove_once:cn`
`\tl_gremove_once:Nn`
`\tl_gremove_once:cn`

Removal is just a special case of replacement.

```

12257 \cs_new_protected:Npn \tl_remove_once:Nn #1#2
12258 { \tl_replace_once:Nnn #1 {#2} { } }
12259 \cs_new_protected:Npn \tl_gremove_once:Nn #1#2
12260 { \tl_greplace_once:Nnn #1 {#2} { } }
12261 \cs_generate_variant:Nn \tl_remove_once:Nn { c }
12262 \cs_generate_variant:Nn \tl_gremove_once:Nn { c }

```

(End definition for `\tl_remove_once:Nn` and `\tl_gremove_once:Nn`. These functions are documented on page 116.)

```

\__tl_remove_all:Nn Removal is just a special case of replacement.
\__tl_remove_all:cn 12263 \cs_new_protected:Npn \tl_remove_all:Nn #1#2
\__tl_gremove_all:Nn 12264 { \tl_replace_all:Nnn #1 {#2} { } }
\__tl_gremove_all:cn 12265 \cs_new_protected:Npn \tl_gremove_all:Nn #1#2
12266 { \tl_greplace_all:Nnn #1 {#2} { } }
12267 \cs_generate_variant:Nn \tl_remove_all:Nn { c }
12268 \cs_generate_variant:Nn \tl_gremove_all:Nn { c }

```

(End definition for `\tl_remove_all:Nn` and `\tl_gremove_all:Nn`. These functions are documented on page 117.)

52.7 Token list conditionals

```

\__tl_if_empty_p:N These functions check whether the token list in the argument is empty and execute the
\__tl_if_empty_p:c proper code from their argument(s).
\__tl_if_empty:NTF 12269 \prg_new_conditional:Npnn \tl_if_empty:N #1 { p , T , F , TF }
\__tl_if_empty:cTF 12270 {
12271     \if_meaning:w #1 \c_empty_tl
12272     \prg_return_true:
12273     \else:
12274     \prg_return_false:
12275     \fi:
12276 }
12277 \prg_generate_conditional_variant:Nnn \tl_if_empty:N
12278 { c } { p , T , F , TF }

```

(End definition for `\tl_if_empty:NTF`. This function is documented on page 104.)

```

\__tl_if_empty_p:n The \if:w triggers the expansion of \tl_to_str:n which converts the argument to a
\__tl_if_empty_p:V string: this is empty if and only if the argument is. Then \if:w \scan_stop: ... \scan_stop:
\__tl_if_empty:nTF is true if and only if the string ... is empty. It could be tempting to use
\__tl_if_empty:VTF \if:w \scan_stop: #1 \scan_stop: directly. But this fails on a token list expand-
ing to anything starting with \scan_stop: leaving everything that follows in the input
stream.

```

```

12279 \prg_new_conditional:Npnn \tl_if_empty:n #1 { p , TF , T , F }
12280 {
12281     \if:w \scan_stop: \tl_to_str:n {#1} \scan_stop:
12282     \prg_return_true:
12283     \else:
12284     \prg_return_false:
12285     \fi:
12286 }
12287 \prg_generate_conditional_variant:Nnn \tl_if_empty:n
12288 { V } { p , TF , T , F }

```

(End definition for `\tl_if_empty:nTF`. This function is documented on page 104.)

```

\__tl_if_empty_p:o The auxiliary function \__tl_if_empty_if:o is for use in various token list condition-
\__tl_if_empty:oTF als which reduce to testing if a given token list is empty after applying a simple func-
\__tl_if_empty_if:o tion to it. The test for emptiness is based on \tl_if_empty:nTF, but the expansion
is hard-coded for efficiency, as this auxiliary function is used in several places. We
don't put \prg_return_true: and so on in the definition of the auxiliary, because that
would prevent an optimization applied to conditionals that end with this code. Also the

```

`\@@_if_empty_if:o` is expanded once in `\tl_if_empty:oTF` for efficiency as well (and to reduce code doubling).

```

12289 \cs_new:Npn \__tl_if_empty_if:o #1
12290 {
12291   \if:w \scan_stop: \__kernel_tl_to_str:w \exp_after:wN {#1} \scan_stop:
12292 }
12293 \exp_args:Nno \use:n
12294 { \prg_new_conditional:Npnn \tl_if_empty:o #1 { p , TF , T , F } }
12295 {
12296   \__tl_if_empty_if:o {#1}
12297   \prg_return_true:
12298   \else:
12299     \prg_return_false:
12300   \fi:
12301 }

```

(End definition for `\tl_if_empty:nTF` and `__tl_if_empty_if:o`. This function is documented on page 104.)

<code>\tl_if_blank_p:n</code> <code>\tl_if_blank_p:V</code> <code>\tl_if_blank_p:o</code> <code>\tl_if_blank:nTF</code> <code>\tl_if_blank:VTF</code> <code>\tl_if_blank:oTF</code> <code>__tl_if_blank_p:NNw</code>	<p><code>\tl_if_blank_p:n</code> TeX skips spaces when reading a non-delimited arguments. Thus, a <i><token list></i> is blank if and only if <code>\use_none:n <token list> ?</code> is empty after one expansion. The auxiliary <code>__tl_if_empty_if:o</code> is a fast emptiness test, converting its argument to a string (after one expansion) and using the test <code>\if:w \scan_stop: ... \scan_stop:.</code></p>	<pre> 12302 \exp_args:Nno \use:n 12303 { \prg_new_conditional:Npnn \tl_if_blank:n #1 { p , T , F , TF } } 12304 { 12305 __tl_if_empty_if:o { \use_none:n #1 ? } 12306 \prg_return_true: 12307 \else: 12308 \prg_return_false: 12309 \fi: 12310 } 12311 \prg_generate_conditional_variant:Nnn \tl_if_blank:n 12312 { e , V , o } { p , T , F , TF } </pre>
---	---	--

(End definition for `\tl_if_blank:nTF` and `__tl_if_blank_p:NNw`. This function is documented on page 104.)

<code>\tl_if_eq_p:NN</code> <code>\tl_if_eq_p:Nc</code> <code>\tl_if_eq_p:cN</code> <code>\tl_if_eq_p:cc</code> <code>\tl_if_eq:NNTF</code> <code>\tl_if_eq:NcTF</code> <code>\tl_if_eq:cNTF</code> <code>\tl_if_eq:ccTF</code>	<p><code>\tl_if_eq_p:NN</code> Returns <code>\c_true_bool</code> if and only if the two token list variables are equal.</p>	<pre> 12313 \prg_new_conditional:Npnn \tl_if_eq:NN #1#2 { p , T , F , TF } 12314 { 12315 \if_meaning:w #1 #2 12316 \prg_return_true: 12317 \else: 12318 \prg_return_false: 12319 \fi: 12320 } 12321 \prg_generate_conditional_variant:Nnn \tl_if_eq:NN 12322 { Nc , c , cc } { p , TF , T , F } </pre>
--	---	--

(End definition for `\tl_if_eq:NNTF`. This function is documented on page 104.)

<code>\l_tl_internal_a_tl</code> <code>\l_tl_internal_b_tl</code>	<p>Temporary storage.</p>	<pre> 12323 \tl_new:N \l_tl_internal_a_tl 12324 \tl_new:N \l_tl_internal_b_tl </pre>
--	---------------------------	--

(End definition for `\l__tl_internal_a_tl` and `\l__tl_internal_b_tl`.)

`\tl_if_eq:NnTF` A simple store and compare routine.

```

12325 \prg_new_protected_conditional:Npnn \tl_if_eq:Nn #1#2 { T , F , TF }
12326 {
12327   \group_begin:
12328     \tl_set:Nn \l__tl_internal_b_tl {#2}
12329     \exp_after:wN
12330   \group_end:
12331   \if_meaning:w #1 \l__tl_internal_b_tl
12332     \prg_return_true:
12333   \else:
12334     \prg_return_false:
12335   \fi:
12336 }
12337 \prg_generate_conditional_variant:Nnn \tl_if_eq:Nn { c } { TF , T , F }

```

(End definition for `\tl_if_eq:NnTF`. This function is documented on page 104.)

`\tl_if_eq:nnTF` A simple store and compare routine.

```

12338 \prg_new_protected_conditional:Npnn \tl_if_eq:nn #1#2 { T , F , TF }
12339 {
12340   \group_begin:
12341     \tl_set:Nn \l__tl_internal_a_tl {#1}
12342     \tl_set:Nn \l__tl_internal_b_tl {#2}
12343     \exp_after:wN
12344   \group_end:
12345   \if_meaning:w \l__tl_internal_a_tl \l__tl_internal_b_tl
12346     \prg_return_true:
12347   \else:
12348     \prg_return_false:
12349   \fi:
12350 }

```

(End definition for `\tl_if_eq:nnTF`. This function is documented on page 105.)

`\tl_if_in:NnTF` See `\tl_if_in:nnTF` for further comments. Here we simply expand the token list variable
`\tl_if_in:cnTF` and pass it to `\tl_if_in:nnTF`.

```

12351 \cs_new_protected:Npn \tl_if_in:NnT { \exp_args:No \tl_if_in:nnT }
12352 \cs_new_protected:Npn \tl_if_in:NnF { \exp_args:No \tl_if_in:nnF }
12353 \cs_new_protected:Npn \tl_if_in:NnTF { \exp_args:No \tl_if_in:nnTF }
12354 \prg_generate_conditional_variant:Nnn \tl_if_in:Nn
12355 { c } { T , F , TF }

```

(End definition for `\tl_if_in:NnTF`. This function is documented on page 105.)

`\tl_if_in:nnTF` Once more, the test relies on the emptiness test for robustness. The function `__tl_`
`\tl_if_in:VnTF` `tmp:w` removes tokens until the first occurrence of `#2`. If this does not appear in `#1`, then
`\tl_if_in:onTF` the final `#2` is removed, leaving an empty token list. Otherwise some tokens remain, and
`\tl_if_in:noTF` the test is false. See `\tl_if_empty:nTF` for details on the emptiness test.

Treating correctly cases like `\tl_if_in:nnTF {a state}{states}`, where `#1#2` contains `#2` before the end, requires special care. To cater for this case, we insert `{ }{ }` between the two token lists. This marker may not appear in `#2` because of $\mathrm{T}_{\mathrm{E}}\mathrm{X}$ limitations on what can delimit a parameter, hence we are safe. Using two brace groups makes the

test work also for empty arguments. The `\if_false:` constructions are a faster way to do `\group_align_safe_begin:` and `\group_align_safe_end:`. The `\scan_stop:` ensures that f-expanding `\tl_if_in:nnTF` does not lead to unbalanced braces.

```

12356 \prg_new_protected_conditional:Npnn \tl_if_in:nn #1#2 { T , F , TF }
12357 {
12358   \scan_stop:
12359   \if_false: { \fi:
12360     \cs_set:Npn \__tl_tmp:w ##1 #2 { }
12361     \tl_if_empty:oTF { \__tl_tmp:w #1 {} {} #2 }
12362     { \prg_return_false: } { \prg_return_true: }
12363   \if_false: } \fi:
12364 }
12365 \prg_generate_conditional_variant:Nnn \tl_if_in:nn
12366 { V , o , no } { T , F , TF }

```

(End definition for `\tl_if_in:nnTF`. This function is documented on page 105.)

`\tl_if_novalue_p:n` Tests whether `##1` matches `-NoValue-` exactly (with suitable catcodes): this is similar to `\quark_if_nil:nTF`. The first argument of `__tl_if_novalue:w` is empty if and only if `##1` starts with `-NoValue-`, while the second argument is empty if `##1` is exactly `-NoValue-` or if it has a question mark just following `-NoValue-`. In this second case, however, the material after the first `?!` remains and makes the emptiness test return false.

```

12367 \cs_set_protected:Npn \__tl_tmp:w #1
12368 {
12369   \prg_new_conditional:Npnn \tl_if_novalue:n ##1
12370   { p , T , F , TF }
12371   {
12372     \__tl_if_empty_if:o { \__tl_if_novalue:w {} ##1 {} ? ! #1 ? ? ! }
12373     \prg_return_true:
12374     \else:
12375     \prg_return_false:
12376     \fi:
12377   }
12378   \cs_new:Npn \__tl_if_novalue:w ##1 #1 ##2 ? ##3 ? ! { ##1 ##2 }
12379 }
12380 \exp_args:No \__tl_tmp:w { \c_novalue_tl }

```

(End definition for `\tl_if_novalue:nTF` and `__tl_if_novalue:w`. This function is documented on page 105.)

`\tl_if_single_p:N` Expand the token list and feed it to `\tl_if_single:nTF`.
`\tl_if_single:nTF`

```

12381 \cs_new:Npn \tl_if_single_p:N { \exp_args:No \tl_if_single_p:n }
12382 \cs_new:Npn \tl_if_single:NT { \exp_args:No \tl_if_single:nT }
12383 \cs_new:Npn \tl_if_single:NF { \exp_args:No \tl_if_single:nF }
12384 \cs_new:Npn \tl_if_single:NTF { \exp_args:No \tl_if_single:nTF }

```

(End definition for `\tl_if_single:NTF`. This function is documented on page 105.)

`\tl_if_single_p:n` This test is similar to `\tl_if_empty:nTF`. Expanding `\use_none:nn #1 ??` once yields an empty result if `#1` is blank, a single `?` if `#1` has a single item, and otherwise yields some tokens ending with `??`. Then, `__kernel_tl_to_str:w` makes sure there are no odd category codes. An earlier version would compare the result to a single `?` using string comparison, but the Lua call is slow in LuaTeX. Instead, `__tl_if_single:nnw`

the second token in front of it. If #1 is empty, this token is the trailing ? and the \if:w test yields false. If #1 has a single item, the token is \scan_stop: and the \if:w test yields true. Otherwise, it is one of the characters resulting from \tl_to_str:n, and the \if:w test yields false. Note that \if:w and __kernel_tl_to_str:w are primitives that take care of expansion.

```

12385 \prg_new_conditional:Npnn \tl_if_single:n #1 { p , T , F , TF }
12386 {
12387   \if:w \scan_stop: \exp_after:wN \__tl_if_single:nnw
12388     \__kernel_tl_to_str:w
12389     \exp_after:wN { \use_none:nn #1 ?? } \scan_stop: ? \s__tl_stop
12390     \prg_return_true:
12391   \else:
12392     \prg_return_false:
12393   \fi:
12394 }
12395 \cs_new:Npn \__tl_if_single:nnw #1#2#3 \s__tl_stop {#2}

```

(End definition for \tl_if_single:nTF and __tl_if_single:nnw. This function is documented on page 105.)

\tl_if_single_token:p:n
\tl_if_single_token:nTF

There are four cases: empty token list, token list starting with a normal token, with a brace group, or with a space token. If the token list starts with a normal token, remove it and check for emptiness. For the next case, an empty token list is not a single token. Finally, we have a non-empty token list starting with a space or a brace group. Applying f-expansion yields an empty result if and only if the token list is a single space.

```

12396 \prg_new_conditional:Npnn \tl_if_single_token:n #1 { p , T , F , TF }
12397 {
12398   \tl_if_head_is_N_type:nTF {#1}
12399   { \__tl_if_empty_if:o { \use_none:n #1 } }
12400   {
12401     \tl_if_empty:nTF {#1}
12402     { \if_false: }
12403     { \__tl_if_empty_if:o { \exp:w \exp_end_continue_f:w #1 } }
12404   }
12405   \prg_return_true:
12406   \else:
12407     \prg_return_false:
12408   \fi:
12409 }

```

(End definition for \tl_if_single_token:nTF. This function is documented on page 105.)

\tl_case:Nn
\tl_case:cn
\tl_case:NnTF
\tl_case:cnTF
__tl_case:nnTF
__tl_case:Nw
__tl_case_end:nw

The aim here is to allow the case statement to be evaluated using a known number of expansion steps (two), and without needing to use an explicit “end of recursion” marker. That is achieved by using the test input as the final case, as this is always true. The trick is then to tidy up the output such that the appropriate case code plus either the true or false branch code is inserted.

```

12410 \cs_new:Npn \tl_case:Nn #1#2
12411 {
12412   \exp:w
12413   \__tl_case:NnTF #1 {#2} { } { }
12414 }
12415 \cs_new:Npn \tl_case:NnT #1#2#3

```

```

12416 {
12417   \exp:w
12418   \__tl_case:NnTF #1 {#2} {#3} { }
12419 }
12420 \cs_new:Npn \tl_case:NnF #1#2#3
12421 {
12422   \exp:w
12423   \__tl_case:NnTF #1 {#2} { } {#3}
12424 }
12425 \cs_new:Npn \tl_case:NnTF #1#2
12426 {
12427   \exp:w
12428   \__tl_case:NnTF #1 {#2}
12429 }
12430 \cs_new:Npn \__tl_case:NnTF #1#2#3#4
12431 { \__tl_case:Nw #1 #2 #1 { } \s__tl_mark {#3} \s__tl_mark {#4} \s__tl_stop }
12432 \cs_new:Npn \__tl_case:Nw #1#2#3
12433 {
12434   \tl_if_eq:NNTF #1 #2
12435   { \__tl_case_end:nw {#3} }
12436   { \__tl_case:Nw #1 }
12437 }
12438 \cs_generate_variant:Nn \tl_case:Nn { c }
12439 \prg_generate_conditional_variant:Nnn \tl_case:Nn
12440 { c } { T , F , TF }

```

To tidy up the recursion, there are two outcomes. If there was a hit to one of the cases searched for, then #1 is the code to insert, #2 is the *next* case to check on and #3 is all of the rest of the cases code. That means that #4 is the **true** branch code, and #5 tidies up the spare `\s__tl_mark` and the **false** branch. On the other hand, if none of the cases matched then we arrive here using the “termination” case of comparing the search with itself. That means that #1 is empty, #2 is the first `\s__tl_mark` and so #4 is the **false** code (the **true** code is mopped up by #3).

```

12441 \cs_new:Npn \__tl_case_end:nw #1#2#3 \s__tl_mark #4#5 \s__tl_stop
12442 { \exp_end: #1 #4 }

```

(End definition for `\tl_case:NnTF` and others. This function is documented on page 106.)

52.8 Mapping over token lists

\tl_map_function:nN Expandable loop macro for token lists. We use the internal scan mark `\s__tl_stop` (defined later), which is not allowed to show up in the token list #1 since it is internal to `\tl`. This allows us a very fast test of whether some *<item>* is the end-marker `\s__tl_stop`, namely call `__tl_use_none_delimit_by_s_stop:w <item> <function> \s__tl_stop`, which calls *<function>* if the *<item>* is the end-marker. To speed up the loop even more, only test one out of eight items, and once we hit one of the eight end-markers, go more slowly through the last few items of the list using `__tl_map_function_end:w`.

```

12443 \cs_new:Npn \tl_map_function:nN #1#2
12444 {
12445   \__tl_map_function:Nnnnnnnnn #2 #1
12446   \s__tl_stop \s__tl_stop \s__tl_stop \s__tl_stop
12447   \s__tl_stop \s__tl_stop \s__tl_stop \s__tl_stop

```



```

12448 \prg_break_point:Nn \tl_map_break: { }
12449 }
12450 \cs_new:Npn \tl_map_function:NN
12451 { \exp_args:No \tl_map_function:nN }
12452 \cs_generate_variant:Nn \tl_map_function:NN { c }
12453 \cs_new:Npn \__tl_map_function:Nnnnnnnnn #1#2#3#4#5#6#7#8#9
12454 {
12455   \__tl_use_none_delimit_by_s_stop:w
12456   #9 \__tl_map_function_end:w \s__tl_stop
12457   #1 {#2} #1 {#3} #1 {#4} #1 {#5} #1 {#6} #1 {#7} #1 {#8} #1 {#9}
12458   \__tl_map_function:Nnnnnnnnn #1
12459 }
12460 \cs_new:Npn \__tl_map_function_end:w \s__tl_stop #1#2
12461 {
12462   \__tl_use_none_delimit_by_s_stop:w #2 \tl_map_break: \s__tl_stop
12463   #1 {#2}
12464   \__tl_map_function_end:w \s__tl_stop
12465 }
12466 \cs_new:Npn \__tl_use_none_delimit_by_s_stop:w #1 \s__tl_stop { }

```

(End definition for `\tl_map_function:nN` and others. These functions are documented on page 110.)

`\tl_map_inline:nn` The inline functions are straight forward by now. We use a little trick with the counter
`\tl_map_inline:Nn` `\g__kernel_prg_map_int` to make them nestable. We can also make use of `__tl_-`
`\tl_map_inline:cn` `map_function:Nnnnnnnnn` from before.

```

12467 \cs_new_protected:Npn \tl_map_inline:nn #1#2
12468 {
12469   \int_gincr:N \g__kernel_prg_map_int
12470   \cs_gset_protected:cpn
12471   { \__tl_map_ \int_use:N \g__kernel_prg_map_int :w } ##1 {#2}
12472   \exp_args:Nc \__tl_map_function:Nnnnnnnnn
12473   { \__tl_map_ \int_use:N \g__kernel_prg_map_int :w }
12474   #1
12475   \s__tl_stop \s__tl_stop \s__tl_stop \s__tl_stop
12476   \s__tl_stop \s__tl_stop \s__tl_stop \s__tl_stop
12477   \prg_break_point:Nn \tl_map_break:
12478   { \int_gdecr:N \g__kernel_prg_map_int }
12479 }
12480 \cs_new_protected:Npn \tl_map_inline:Nn
12481 { \exp_args:No \tl_map_inline:nn }
12482 \cs_generate_variant:Nn \tl_map_inline:Nn { c }

```

(End definition for `\tl_map_inline:nn` and `\tl_map_inline:Nn`. These functions are documented on page 110.)

`\tl_map_tokens:nn` Much like the function mapping.

`\tl_map_tokens:Nn` `\cs_new:Npn \tl_map_tokens:nn #1#2`
`\tl_map_tokens:cn` `{`

```

\__tl_map_tokens:nnnnnnnnn 12485 \__tl_map_tokens:nnnnnnnnn {#2} #1
\__tl_map_tokens_end:w 12486 \s__tl_stop \s__tl_stop \s__tl_stop \s__tl_stop
12487 \s__tl_stop \s__tl_stop \s__tl_stop \s__tl_stop
12488 \prg_break_point:Nn \tl_map_break: { }
12489 }
12490 \cs_new:Npn \tl_map_tokens:Nn

```

```

12491 { \exp_args:No \tl_map_tokens:nn }
12492 \cs_generate_variant:Nn \tl_map_tokens:Nn { c }
12493 \cs_new:Npn \__tl_map_tokens:nnnnnnnn #1#2#3#4#5#6#7#8#9
12494 {
12495   \__tl_use_none_delimit_by_s_stop:w
12496   #9 \__tl_map_tokens_end:w \s__tl_stop
12497   \use:n {#1} {#2} \use:n {#1} {#3} \use:n {#1} {#4} \use:n {#1} {#5}
12498   \use:n {#1} {#6} \use:n {#1} {#7} \use:n {#1} {#8} \use:n {#1} {#9}
12499   \__tl_map_tokens:nnnnnnnn {#1}
12500 }
12501 \cs_new:Npn \__tl_map_tokens_end:w \s__tl_stop \use:n #1#2
12502 {
12503   \__tl_use_none_delimit_by_s_stop:w #2 \tl_map_break: \s__tl_stop
12504   #1 {#2}
12505   \__tl_map_tokens_end:w \s__tl_stop
12506 }

```

(End definition for `\tl_map_tokens:nn` and others. These functions are documented on page 111.)

`\tl_map_variable:nNn` `\tl_map_variable:nNn {<token list>} <tl var> {<action>}` assigns `<tl var>` to each element and executes `<action>`. The assignment to `<tl var>` is done after the quark test so that this variable does not get set to a quark.

```

\__tl_map_variable:Nnn
12507 \cs_new_protected:Npn \tl_map_variable:nNn #1#2#3
12508 { \tl_map_tokens:nn {#1} { \__tl_map_variable:Nnn #2 {#3} } }
12509 \cs_new_protected:Npn \__tl_map_variable:Nnn #1#2#3
12510 { \tl_set:Nn #1 {#3} #2 }
12511 \cs_new_protected:Npn \tl_map_variable:NNn
12512 { \exp_args:No \tl_map_variable:nNn }
12513 \cs_generate_variant:Nn \tl_map_variable:NNn { c }

```

(End definition for `\tl_map_variable:nNn`, `\tl_map_variable:NNn`, and `__tl_map_variable:Nnn`. These functions are documented on page 111.)

`\tl_map_break:` The break statements use the general `\prg_map_break:Nn`.

```

\tl_map_break:N
12514 \cs_new:Npn \tl_map_break:
12515 { \prg_map_break:Nn \tl_map_break: { } }
12516 \cs_new:Npn \tl_map_break:n
12517 { \prg_map_break:Nn \tl_map_break: }

```

(End definition for `\tl_map_break:` and `\tl_map_break:n`. These functions are documented on page 111.)

52.9 Using token lists

`\tl_to_str:n` Another name for a primitive: defined in `l3basics`.

```

\tl_to_str:V
12518 \cs_generate_variant:Nn \tl_to_str:n { V }

```

(End definition for `\tl_to_str:n`. This function is documented on page 107.)

`\tl_to_str:N` These functions return the replacement text of a token list as a string.

```

\tl_to_str:c
12519 \cs_new:Npn \tl_to_str:N #1 { \__kernel_tl_to_str:w \exp_after:wN {#1} }
12520 \cs_generate_variant:Nn \tl_to_str:N { c }

```

(End definition for `\tl_to_str:N`. This function is documented on page 107.)

`\tl_use:N` Token lists which are simply not defined give a clear TeX error here. No such luck for `\tl_use:c` ones equal to `\scan_stop:` so instead a test is made and if there is an issue an error is forced.

```

12521 \cs_new:Npn \tl_use:N #1
12522 {
12523   \tl_if_exist:NTF #1 {#1}
12524   {
12525     \msg_expandable_error:nnn
12526     { kernel } { bad-variable } {#1}
12527   }
12528 }
12529 \cs_generate_variant:Nn \tl_use:N { c }

```

(End definition for `\tl_use:N`. This function is documented on page 108.)

52.10 Working with the contents of token lists

`\tl_count:n` Count number of elements within a token list or token list variable. Brace groups within the list are read as a single element. Spaces are ignored. `__tl_count:n` grabs the element and replaces it by +1. The 0 ensures that it works on an empty list.

```

12530 \cs_new:Npn \tl_count:n #1
12531 {
12532   \int_eval:n
12533   { 0 \tl_map_function:nN {#1} \__tl_count:n }
12534 }
12535 \cs_new:Npn \tl_count:N #1
12536 {
12537   \int_eval:n
12538   { 0 \tl_map_function:NN #1 \__tl_count:n }
12539 }
12540 \cs_new:Npn \__tl_count:n #1 { + 1 }
12541 \cs_generate_variant:Nn \tl_count:n { V , o }
12542 \cs_generate_variant:Nn \tl_count:N { c }

```

(End definition for `\tl_count:n`, `\tl_count:N`, and `__tl_count:n`. These functions are documented on page 108.)

`\tl_count_tokens:n` The token count is computed through an `\int_eval:n` construction. Each 1+ is output to the *left*, into the integer expression, and the sum is ended by the `\exp_end:` inserted by `__tl_act_end:wn` (which is technically implemented as `\c_zero_int`). Somewhat a hack!

```

12543 \cs_new:Npn \tl_count_tokens:n #1
12544 {
12545   \int_eval:n
12546   {
12547     \__tl_act:NNNn
12548     \__tl_act_count_normal:N
12549     \__tl_act_count_group:n
12550     \__tl_act_count_space:
12551     {#1}
12552   }
12553 }
12554 \cs_new:Npn \__tl_act_count_normal:N #1 { 1 + }

```

```

12555 \cs_new:Npn \__tl_act_count_space: { 1 + }
12556 \cs_new:Npn \__tl_act_count_group:n #1 { 2 + \tl_count_tokens:n {#1} + }

```

(End definition for \tl_count_tokens:n and others. This function is documented on page 108.)

\tl_reverse_items:n Reversal of a token list is done by taking one item at a time and putting it after \s__tl_stop.

```

\__tl_reverse_items:nwNwn
\__tl_reverse_items:wn
12557 \cs_new:Npn \tl_reverse_items:n #1
12558 {
12559   \__tl_reverse_items:nwNwn #1 ?
12560   \s__tl_mark \__tl_reverse_items:nwNwn
12561   \s__tl_mark \__tl_reverse_items:wn
12562   \s__tl_stop { }
12563 }
12564 \cs_new:Npn \__tl_reverse_items:nwNwn #1 #2 \s__tl_mark #3 #4 \s__tl_stop #5
12565 {
12566   #3 #2
12567   \s__tl_mark \__tl_reverse_items:nwNwn
12568   \s__tl_mark \__tl_reverse_items:wn
12569   \s__tl_stop { {#1} #5 }
12570 }
12571 \cs_new:Npn \__tl_reverse_items:wn #1 \s__tl_stop #2
12572 { \__kernel_exp_not:w \exp_after:wN { \use_none:nn #2 } }

```

(End definition for \tl_reverse_items:n, __tl_reverse_items:nwNwn, and __tl_reverse_items:wn. This function is documented on page 109.)

\tl_trim_spaces:n Trimming spaces from around the input is deferred to an internal function whose first argument is the token list to trim, augmented by an initial __tl_trim_mark:, and whose second argument is a *<continuation>*, which receives as a braced argument __tl_trim_mark: *<trimmed token list>*. The control sequence __tl_trim_mark: expands to nothing in a single expansion. In the case at hand, we take __kernel_exp_not:w \exp_after:wN as our continuation, so that space trimming behaves correctly within an x-type expansion.

```

\__tl_trim_spaces:nn
\__tl_trim_spaces_auxi:w
\__tl_trim_spaces_auxii:w
\__tl_trim_spaces_auxiii:w
\__tl_trim_spaces_auxiv:w
\__tl_trim_mark:
12573 \cs_new:Npn \tl_trim_spaces:n #1
12574 {
12575   \__tl_trim_spaces:nn
12576   { \__tl_trim_mark: #1 }
12577   { \__kernel_exp_not:w \exp_after:wN }
12578 }
12579 \cs_generate_variant:Nn \tl_trim_spaces:n { o }
12580 \cs_new:Npn \tl_trim_spaces_apply:nN #1#2
12581 { \__tl_trim_spaces:nn { \__tl_trim_mark: #1 } { \exp_args:No #2 } }
12582 \cs_generate_variant:Nn \tl_trim_spaces_apply:nN { o }
12583 \cs_new_protected:Npn \tl_trim_spaces:N #1
12584 { \__kernel_tl_set:Nx #1 { \exp_args:No \tl_trim_spaces:n {#1} } }
12585 \cs_new_protected:Npn \tl_gtrim_spaces:N #1
12586 { \__kernel_tl_gset:Nx #1 { \exp_args:No \tl_trim_spaces:n {#1} } }
12587 \cs_generate_variant:Nn \tl_trim_spaces:N { c }
12588 \cs_generate_variant:Nn \tl_gtrim_spaces:N { c }

```

Trimming spaces from around the input is done using delimited arguments and quarks, and to get spaces at odd places in the definitions, we nest those in __tl_tmp:w, which then receives a single space as its argument: #1 is $_$. Removing leading spaces is done with __tl_trim_spaces_auxi:w, which loops until __tl_trim_mark: $_$ matches the

end of the token list: then ##1 is the token list and ##3 is _tl_trim_spaces_auxii:w. This hands the relevant tokens to the loop _tl_trim_spaces_auxiii:w, responsible for trimming trailing spaces. The end is reached when \s_tl_nil matches the one present in the definition of \tl_trim_spaces:n. Then _tl_trim_spaces_auxiv:w puts the token list into a group, with a lingering _tl_trim_mark: at the start (which will expand to nothing in one step of expansion), and feeds this to the *<continuation>*.

```

12589 \cs_set_protected:Npn \_tl_tmp:w #1
12590 {
12591   \cs_new:Npn \_tl_trim_spaces:nn ##1
12592   {
12593     \_tl_trim_spaces_auxi:w
12594     ##1
12595     \s\_tl_nil
12596     \_tl_trim_mark: #1 { }
12597     \_tl_trim_mark: \_tl_trim_spaces_auxii:w
12598     \_tl_trim_spaces_auxiii:w
12599     #1 \s\_tl_nil
12600     \_tl_trim_spaces_auxiv:w
12601     \s\_tl_stop
12602   }
12603   \cs_new:Npn
12604     \_tl_trim_spaces_auxi:w ##1 \_tl_trim_mark: #1 ##2 \_tl_trim_mark: ##3
12605   {
12606     ##3
12607     \_tl_trim_spaces_auxi:w
12608     \_tl_trim_mark:
12609     ##2
12610     \_tl_trim_mark: #1 {##1}
12611   }
12612   \cs_new:Npn \_tl_trim_spaces_auxii:w
12613     \_tl_trim_spaces_auxi:w \_tl_trim_mark: \_tl_trim_mark: ##1
12614   {
12615     \_tl_trim_spaces_auxiii:w
12616     ##1
12617   }
12618   \cs_new:Npn \_tl_trim_spaces_auxiii:w ##1 #1 \s\_tl_nil ##2
12619   {
12620     ##2
12621     ##1 \s\_tl_nil
12622     \_tl_trim_spaces_auxiii:w
12623   }
12624   \cs_new:Npn \_tl_trim_spaces_auxiv:w ##1 \s\_tl_nil ##2 \s\_tl_stop ##3
12625   { ##3 { ##1 } }
12626   \cs_new:Npn \_tl_trim_mark: {}
12627 }
12628 \_tl_tmp:w { ~ }

```

(End definition for \tl_trim_spaces:n and others. These functions are documented on page 109.)

\tl_sort:Nn Implemented in l3sort.

\tl_sort:cn

(End definition for \tl_sort:Nn, \tl_gsort:Nn, and \tl_sort:nN. These functions are documented on page 116.)

\tl_gsort:Nn

\tl_gsort:cn

\tl_sort:nN

52.11 The first token from a token list

`\tl_head:N` Finding the head of a token list expandably always strips braces, which is fine as this is consistent with for example mapping over a list. The empty brace groups in `\tl_head:n` ensure that a blank argument gives an empty result. The result is returned within the `\tl_head:V` `\unexpanded` primitive. The approach here is to use `\if_false:` to allow us to use `}` as the closing delimiter: this is the only safe choice, as any other token would not be able to parse it's own code. If the `\expanded` primitive is available it is used to get a fast and safe code variant in which we don't have to ensure that the left-most token is an internal to not break in an f-type expansion. If `\expanded` isn't available, using a marker, we can see if what we are grabbing is exactly the marker, or there is anything else to deal with. If there is, there is a loop. If not, tidy up and leave the item in the output stream. More detail in <http://tex.stackexchange.com/a/70168>.

```

\tl_head:f 12629 \cs_if_exist:NTF \tex_expanded:D
\tl_head:f 12630 {
\tl_head:f 12631   \cs_new:Npn \tl_head:n #1
\tl_head:f 12632   {
\tl_head:f 12633     \__kernel_exp_not:w \tex_expanded:D
\tl_head:f 12634     { { \if_false: { \fi: \__tl_head_aux:n #1 { } } } }
\tl_head:f 12635   }
\tl_head:f 12636   \cs_new:Npn \__tl_head_aux:n #1
\tl_head:f 12637   {
\tl_head:f 12638     \__kernel_exp_not:w {#1}
\tl_head:f 12639     \exp_after:wN \use_none:n \exp_after:wN { \if_false: } \fi:
\tl_head:f 12640   }
\tl_head:f 12641 }
\tl_head:f 12642 {
\tl_head:f 12643   \cs_new:Npn \tl_head:n #1
\tl_head:f 12644   {
\tl_head:f 12645     \__kernel_exp_not:w
\tl_head:f 12646     \if_false: { \fi: \__tl_head_aux:nw #1 { } \s__tl_stop }
\tl_head:f 12647   }
\tl_head:f 12648   \cs_new:Npn \__tl_head_aux:nw #1#2 \s__tl_stop
\tl_head:f 12649   {
\tl_head:f 12650     \exp_after:wN \__tl_head_auxii:n \exp_after:wN {
\tl_head:f 12651       \if_false: } \fi: {#1}
\tl_head:f 12652   }
\tl_head:f 12653   \exp_args:Nno \use:n
\tl_head:f 12654   { \cs_new:Npn \__tl_head_auxii:n #1 }
\tl_head:f 12655   {
\tl_head:f 12656     \__tl_if_empty_if:o { \use_none:n #1 }
\tl_head:f 12657     \exp_after:wN \use_ii:nnn
\tl_head:f 12658     \fi:
\tl_head:f 12659     \use_ii:nn
\tl_head:f 12660     {#1}
\tl_head:f 12661     { \if_false: { \fi: \__tl_head_aux:nw #1 } }
\tl_head:f 12662   }
\tl_head:f 12663 }
\tl_head:f 12664 \cs_generate_variant:Nn \tl_head:n { V , v , f }
\tl_head:f 12665 \cs_new:Npn \tl_head:w #1#2 \q_stop {#1}
\tl_head:f 12666 \cs_new:Npn \__tl_tl_head:w #1#2 \s__tl_stop {#1}
\tl_head:f 12667 \cs_new:Npn \tl_head:N { \exp_args:No \tl_head:n }

```

To correctly leave the tail of a token list, it's important *not* to absorb any of the tail part

as an argument. For example, the simple definition

```
\cs_new:Npn \tl_tail:n #1 { \tl_tail:w #1 \q_stop }
\cs_new:Npn \tl_tail:w #1#2 \q_stop
```

would give the wrong result for `\tl_tail:n { a { bc } }` (the braces would be stripped). Thus the only safe way to proceed is to first check that there is an item to grab (*i.e.* that the argument is not blank) and assuming there is to dispose of the first item. As with `\tl_head:n`, the result is protected from further expansion by `\unexpanded`. While we could optimise the test here, this would leave some tokens “banned” in the input, which we do not have with this definition.

```
12668 \exp_args:Nno \use:n { \cs_new:Npn \tl_tail:n #1 }
12669 {
12670     \exp_after:wN \__kernel_exp_not:w
12671     \tl_if_blank:nTF {#1}
12672     { { } }
12673     { \exp_after:wN { \use_none:n #1 } }
12674 }
12675 \cs_generate_variant:Nn \tl_tail:n { V , v , f }
12676 \cs_new:Npn \tl_tail:N { \exp_args:No \tl_tail:n }
```

(End definition for `\tl_head:N` and others. These functions are documented on page 112.)

```
\tl_if_head_eq_meaning_p:nN
\tl_if_head_eq_meaning:nNTF
\tl_if_head_eq_charcode_p:nN
\tl_if_head_eq_charcode:nNTF
\tl_if_head_eq_charcode_p:fN
\tl_if_head_eq_charcode:fNTF
\tl_if_head_eq_catcode_p:nN
\tl_if_head_eq_catcode:nNTF
\tl_if_head_eq_catcode_p:oN
\tl_if_head_eq_catcode:oNTF
__tl_head_exp_not:w
__tl_if_head_eq_empty_arg:w
```

Accessing the first token of a token list is tricky in three cases: when it has category code 1 (begin-group token), when it is an explicit space, with category code 10 and character code 32, or when the token list is empty (obviously).

Forgetting temporarily about this issue we would use the following test in `\tl_if_head_eq_charcode:nN`. Here, `\tl_head:w` yields the first token of the token list, then passed to `\exp_not:N`.

```
\if_charcode:w
    \exp_after:wN \exp_not:N \tl_head:w #1 \q_nil \q_stop
    \exp_not:N #2
```

The two first special cases are detected by testing if the token list starts with an N-type token (the extra ? sends empty token lists to the true branch of this test). In those cases, the first token is a character, and since we only care about its character code, we can use `\str_head:n` to access it (this works even if it is a space character). An empty argument results in `\tl_head:w` leaving two token: `^` and `__tl_if_head_eq_empty_arg:w` which will result in the `\if_charcode:w` test being false and remove `\exp_not:N` and `#2`.

```
12677 \prg_new_conditional:Npnn \tl_if_head_eq_charcode:nN #1#2 { p , T , F , TF }
12678 {
12679     \if_charcode:w
12680         \tl_if_head_is_N_type:nTF { #1 ? }
12681         { \__tl_head_exp_not:w #1 { ^ \__tl_if_head_eq_empty_arg:w } \s__tl_stop }
12682         { \str_head:n {#1} }
12683         \exp_not:N #2
12684         \prg_return_true:
12685     \else:
12686         \prg_return_false:
12687     \fi:
12688 }
12689 \prg_generate_conditional_variant:Nnn \tl_if_head_eq_charcode:nN
12690 { f } { p , TF , T , F }
```

For `\tl_if_head_eq_catcode:nN`, again we detect special cases with a `\tl_if_head_is_N_type:n`. Then we need to test if the first token is a begin-group token or an explicit space token, and produce the relevant token, either `\c_group_begin_token` or `\c_space_token`. Again, for an empty argument, a hack is used, removing the token given by the user and leaving two tokens in the input stream which will make the `\if_catcode:w` test return false.

```

12691 \prg_new_conditional:Npnn \tl_if_head_eq_catcode:nN #1 #2 { p , T , F , TF }
12692 {
12693   \if_catcode:w
12694     \tl_if_head_is_N_type:nTF { #1 ? }
12695     { \__tl_head_exp_not:w #1 { ^ \__tl_if_head_eq_empty_arg:w } \s__tl_stop }
12696     {
12697       \tl_if_head_is_group:nTF {#1}
12698       \c_group_begin_token
12699       \c_space_token
12700     }
12701     \exp_not:N #2
12702     \prg_return_true:
12703   \else:
12704     \prg_return_false:
12705   \fi:
12706 }
12707 \prg_generate_conditional_variant:Nnn \tl_if_head_eq_catcode:nN
12708 { o } { p , TF , T , F }

```

For `\tl_if_head_eq_meaning:nN`, again, detect special cases. In the normal case, use `\tl_head:w`, with no `\exp_not:N` this time, since `\if_meaning:w` causes no expansion. With an empty argument, the test is true, and `\use_none:nnn` removes #2 and `\prg_return_true:` and `\else:` (it is safe this way here as in this case `\prg_new_conditional:Npnn` didn't optimize these two away). In the special cases, we know that the first token is a character, hence `\if_charcode:w` and `\if_catcode:w` together are enough. We combine them in some order, hopefully faster than the reverse. Tests are not nested because the arguments may contain unmatched primitive conditionals.

```

12709 \prg_new_conditional:Npnn \tl_if_head_eq_meaning:nN #1#2 { p , T , F , TF }
12710 {
12711   \tl_if_head_is_N_type:nTF { #1 ? }
12712   \__tl_if_head_eq_meaning_normal:nN
12713   \__tl_if_head_eq_meaning_special:nN
12714   {#1} #2
12715 }
12716 \cs_new:Npn \__tl_if_head_eq_meaning_normal:nN #1 #2
12717 {
12718   \exp_after:wN \if_meaning:w
12719   \__tl_tl_head:w #1 { ?? \use_none:nnn } \s__tl_stop #2
12720   \prg_return_true:
12721   \else:
12722     \prg_return_false:
12723   \fi:
12724 }
12725 \cs_new:Npn \__tl_if_head_eq_meaning_special:nN #1 #2
12726 {
12727   \if_charcode:w \str_head:n {#1} \exp_not:N #2
12728   \exp_after:wN \use_ii:nn

```



```

12729 \else:
12730 \prg_return_false:
12731 \fi:
12732 \use_none:n
12733 {
12734 \if_catcode:w \exp_not:N #2
12735 \tl_if_head_is_group:nTF {#1}
12736 { \c_group_begin_token }
12737 { \c_space_token }
12738 \prg_return_true:
12739 \else:
12740 \prg_return_false:
12741 \fi:
12742 }
12743 }

```

Both `\tl_if_head_eq_charcode:nN` and `\tl_if_head_eq_catcode:nN` will need to get the first token of their argument and apply `\exp_not:N` to it. `__tl_head_exp_not:w` does exactly that.

```

12744 \cs_new:Npn \__tl_head_exp_not:w #1 #2 \s__tl_stop
12745 { \exp_not:N #1 }

```

If the argument of `\tl_if_head_eq_charcode:nN` and `\tl_if_head_eq_catcode:nN` was empty `__tl_if_head_eq_empty_arg:w` will be left in the input stream. This macro has to remove `\exp_not:N` and the following token from the input stream to make sure no unbalanced if-construct is created and leave tokens there which make the two tests return false.

```

12746 \cs_new:Npn \__tl_if_head_eq_empty_arg:w \exp_not:N #1
12747 { ? }

```

(End definition for `\tl_if_head_eq_meaning:nNTF` and others. These functions are documented on page 106.)

```

\tl_if_head_is_N_type_p:n
\tl_if_head_is_N_type:nTF
  \__tl_if_head_is_N_type_auxi:w
  \__tl_if_head_is_N_type_auxii:nn
  \__tl_if_head_is_N_type_auxiii:n

```

A token list can be empty, can start with an explicit space character (catcode 10 and charcode 32), can start with a begin-group token (catcode 1), or start with an N-type argument. In the first two cases, the line involving `__tl_if_head_is_N_type_auxi:w` produces `f` (and otherwise nothing). In the third case (begin-group token), the lines involving `\token_to_str:N` produce a single closing brace. The category code test is thus true exactly in the fourth case, which is what we want. One cannot optimize by moving one of the `\scan_stop:` to the beginning: if #1 contains primitive conditionals, all of its occurrences must be dealt with before the `\if:w` tries to skip the `true` branch of the conditional.

```

12748 \prg_new_conditional:Npnn \tl_if_head_is_N_type:n #1 { p , T , F , TF }
12749 {
12750 \if:w
12751 \if_false: { \fi: \__tl_if_head_is_N_type_auxi:w \prg_do_nothing: #1 ~ }
12752 { \exp_after:wN { \token_to_str:N #1 } }
12753 \scan_stop: \scan_stop:
12754 \prg_return_true:
12755 \else:
12756 \prg_return_false:
12757 \fi:
12758 }
12759 \exp_args:Nno \use:n { \cs_new:Npn \__tl_if_head_is_N_type_auxi:w #1 ~ }

```

```

12760 {
12761   \tl_if_empty:oTF { #1 }
12762     { f \exp_after:wN \use_none:nn }
12763     { \exp_after:wN \__tl_if_head_is_N_type_auxii:n }
12764   \exp_after:wN { \if_false: } \fi:
12765 }
12766 \cs_new:Npn \__tl_if_head_is_N_type_auxii:n #1
12767 { \exp_after:wN \use_none:n \exp_after:wN }

```

(End definition for `\tl_if_head_is_N_type:nTF` and others. This function is documented on page 107.)

`\tl_if_head_is_group:p:n`
`\tl_if_head_is_group:nTF`
`__tl_if_head_is_group_fi_false:w`

Pass the first token of `#1` through `\token_to_str:N`, then check for the brace balance. The extra `?` caters for an empty argument. This could be made faster, but we need all brace tricks to happen in one step of expansion, keeping the token list brace balanced at all times.

```

12768 \prg_new_conditional:Npnn \tl_if_head_is_group:n #1 { p , T , F , TF }
12769 {
12770   \if:w
12771     \exp_after:wN \use_none:n
12772     \exp_after:wN { \exp_after:wN { \token_to_str:N #1 ? } }
12773     \scan_stop: \scan_stop:
12774     \__tl_if_head_is_group_fi_false:w
12775   \fi:
12776   \if_true:
12777     \prg_return_true:
12778   \else:
12779     \prg_return_false:
12780   \fi:
12781 }
12782 \cs_new:Npn \__tl_if_head_is_group_fi_false:w \fi: \if_true: { \fi: \if_false: }

```

(End definition for `\tl_if_head_is_group:nTF` and `__tl_if_head_is_group_fi_false:w`. This function is documented on page 106.)

`\tl_if_head_is_space:p:n`
`\tl_if_head_is_space:nTF`
`__tl_if_head_is_space:w`

The auxiliary's argument is all that is before the first explicit space in `\prg_do_nothing:#1?~`. If that is a single `\prg_do_nothing:` the test yields true. Otherwise, that is more than one token, and the test yields false. The work is done within braces (with an `\if_false: { \fi: ... }` construction) both to hide potential alignment tab characters from T_EX in a table, and to allow for removing what remains of the token list after its first space. The use of `\if:w` ensures that the result of a single step of expansion directly yields a balanced token list (no trailing closing brace).

```

12783 \prg_new_conditional:Npnn \tl_if_head_is_space:n #1 { p , T , F , TF }
12784 {
12785   \if:w
12786     \if_false: { \fi: \__tl_if_head_is_space:w \prg_do_nothing: #1 ? ~ }
12787     \scan_stop: \scan_stop:
12788     \prg_return_true:
12789   \else:
12790     \prg_return_false:
12791   \fi:
12792 }
12793 \exp_args:Nno \use:n { \cs_new:Npn \__tl_if_head_is_space:w #1 ~ }
12794 {
12795   \__tl_if_empty_if:o {#1} \else: f \fi:

```

```

12796 \exp_after:wN \use_none:n \exp_after:wN { \if_false: } \fi:
12797 }

```

(End definition for `\tl_if_head_is_space:nTF` and `__tl_if_head_is_space:w`. This function is documented on page 107.)

52.12 Token by token changes

`\s__tl_act_stop` The `__tl_act_...` functions may be applied to any token list. Hence, we use a private quark, to allow any token, even quarks, in the token list. Only `\s__tl_act_stop` may not appear in the token lists manipulated by `__tl_act:NNNn` functions.

```

12798 \scan_new:N \s__tl_act_stop

```

(End definition for `\s__tl_act_stop`.)

```

\__tl_act:NNNn
\__tl_act_output:n
\__tl_act_reverse_output:n
\__tl_act_loop:w
\__tl_act_normal:NwNNN
\__tl_act_group:nwNNN
\__tl_act_space:wwNNN
\__tl_act_end:wn
\tl_act_if_head_is_space:nTF
\__tl_act_if_head_is_space:w
\tl_act_if_head_is_space_true:w
\tl_use_none_delimit_by_q_act_stop:w

```

To help control the expansion, `__tl_act:NNNn` should always be preceded by `\exp:w` and ends by producing `\exp_end:` once the result has been obtained. This way no internal token of it can be accidentally end up in the input stream. Because `\s__tl_act_stop` can't appear without braces around it in the argument #1 of `__tl_act_loop:w`, we can use this marker to set up a fast test for leading spaces.

```

12799 \cs_set_protected:Npn \__tl_tmp:w #1
12800 {
12801   \cs_new:Npn \__tl_act_if_head_is_space:nTF ##1
12802   {
12803     \__tl_act_if_head_is_space:w
12804     \s__tl_act_stop ##1 \s__tl_act_stop \__tl_act_if_head_is_space_true:w
12805     \s__tl_act_stop #1 \s__tl_act_stop \use_ii:nn
12806   }
12807   \cs_new:Npn \__tl_act_if_head_is_space:w
12808   ##1 \s__tl_act_stop #1 ##2 \s__tl_act_stop
12809   {}
12810   \cs_new:Npn \__tl_act_if_head_is_space_true:w
12811   \s__tl_act_stop #1 \s__tl_act_stop \use_ii:nn ##1 ##2
12812   {##1}
12813 }
12814 \__tl_tmp:w { ~ }

```

(We expand the definition `__tl_act_if_head_is_space:nTF` when setting up `__tl_act_loop:w`, so we can then undefine the auxiliary.) In the loop, we check how the token list begins and act accordingly. In the “group” case, we may have reached `\s__tl_act_stop`, the end of the list. Then leave `\exp_end:` and the result in the input stream, to terminate the expansion of `\exp:w`. Otherwise, apply the relevant function to the “arguments”, #3 and to the head of the token list. Then repeat the loop. The scheme is the same if the token list starts with an N-type or with a space, making sure that `__tl_act_space:wwNNN` gobbles the space.

```

12815 \exp_args:Nnx \use:n { \cs_new:Npn \__tl_act_loop:w #1 \s__tl_act_stop }
12816 {
12817   \exp_not:o { \__tl_act_if_head_is_space:nTF {#1} }
12818   \exp_not:N \__tl_act_space:wwNNN
12819   {
12820     \exp_not:o { \tl_if_head_is_group:nTF {#1} }
12821     \exp_not:N \__tl_act_group:nwNNN
12822     \exp_not:N \__tl_act_normal:NwNNN

```

```

12823     }
12824     \exp_not:n {#1} \s__tl_act_stop
12825   }
12826   \cs_undefine:N \__tl_act_if_head_is_space:nTF
12827   \cs_new:Npn \__tl_act_normal:NwNNN #1 #2 \s__tl_act_stop #3
12828   {
12829     #3 #1
12830     \__tl_act_loop:w #2 \s__tl_act_stop
12831     #3
12832   }
12833   \cs_new:Npn \__tl_use_none_delimit_by_s_act_stop:w #1 \s__tl_act_stop { }
12834   \cs_new:Npn \__tl_act_end:wn #1 \__tl_act_result:n #2
12835   { \group_align_safe_end: \exp_end: #2 }
12836   \cs_new:Npn \__tl_act_group:nwNNN #1 #2 \s__tl_act_stop #3#4#5
12837   {
12838     \__tl_use_none_delimit_by_s_act_stop:w #1 \__tl_act_end:wn \s__tl_act_stop
12839     #5 {#1}
12840     \__tl_act_loop:w #2 \s__tl_act_stop
12841     #3 #4 #5
12842   }
12843   \exp_last_unbraced:NNo
12844   \cs_new:Npn \__tl_act_space:wwNNN \c_space_tl #1 \s__tl_act_stop #2#3
12845   {
12846     #3
12847     \__tl_act_loop:w #1 \s__tl_act_stop
12848     #2 #3
12849   }

```

__tl_act:NNNn loops over tokens, groups, and spaces in #4. {\s_@@_act_stop} serves as the end of token list marker, the ? after it avoids losing outer braces. The result is stored as an argument for the dummy function __tl_act_result:n.

```

12850   \cs_new:Npn \__tl_act:NNNn #1#2#3#4
12851   {
12852     \group_align_safe_begin:
12853     \__tl_act_loop:w #4 { \s__tl_act_stop } ? \s__tl_act_stop
12854     #1 #3 #2
12855     \__tl_act_result:n { }
12856   }

```

Typically, the output is done to the right of what was already output, using __tl_act_output:n, but for the __tl_act_reverse functions, it should be done to the left.

```

12857   \cs_new:Npn \__tl_act_output:n #1 #2 \__tl_act_result:n #3
12858   { #2 \__tl_act_result:n { #3 #1 } }
12859   \cs_new:Npn \__tl_act_reverse_output:n #1 #2 \__tl_act_result:n #3
12860   { #2 \__tl_act_result:n { #1 #3 } }

```

(End definition for __tl_act:NNNn and others.)

\tl_reverse:n The goal here is to reverse without losing spaces nor braces. This is done using the
\tl_reverse:o general internal function __tl_act:NNNn. Spaces and “normal” tokens are output on
\tl_reverse:V the left of the current output. Grouped tokens are output to the left but without any
 __tl_reverse_normal:n reversal within the group.

```

\__tl_reverse_group_preserve:nn 12861 \cs_new:Npn \tl_reverse:n #1
\__tl_reverse_space:n          12862 {
                                12863   \__kernel_exp_not:w \exp_after:wN

```

```

12864     {
12865         \exp:w
12866         \__tl_act:NNNn
12867         \__tl_reverse_normal:N
12868         \__tl_reverse_group_preserve:n
12869         \__tl_reverse_space:
12870         {#1}
12871     }
12872 }
12873 \cs_generate_variant:Nn \tl_reverse:n { o , V }
12874 \cs_new:Npn \__tl_reverse_normal:N
12875 { \__tl_act_reverse_output:n }
12876 \cs_new:Npn \__tl_reverse_group_preserve:n #1
12877 { \__tl_act_reverse_output:n { {#1} } }
12878 \cs_new:Npn \__tl_reverse_space:
12879 { \__tl_act_reverse_output:n { ~ } }

```

(End definition for `\tl_reverse:n` and others. This function is documented on page 108.)

`\tl_reverse:N` This reverses the list, leaving `\exp_stop_f:` in front, which stops the `f`-expansion.

`\tl_reverse:c`

`\tl_greverse:N`

`\tl_greverse:c`

```

12880 \cs_new_protected:Npn \tl_reverse:N #1
12881 { \__kernel_tl_set:Nx #1 { \exp_args:No \tl_reverse:n { #1 } } }
12882 \cs_new_protected:Npn \tl_greverse:N #1
12883 { \__kernel_tl_gset:Nx #1 { \exp_args:No \tl_reverse:n { #1 } } }
12884 \cs_generate_variant:Nn \tl_reverse:N { c }
12885 \cs_generate_variant:Nn \tl_greverse:N { c }

```

(End definition for `\tl_reverse:N` and `\tl_greverse:N`. These functions are documented on page 108.)

52.13 Using a single item

`\tl_item:nn` The idea here is to find the offset of the item from the left, then use a loop to grab the correct item. If the resulting offset is too large, then `__tl_if_recursion_tail_break:nN` terminates the loop, and returns nothing at all.

`\tl_item:Nn`

`\tl_item:cn`

`__tl_item_aux:nn`

`__tl_item:nn`

```

12886 \cs_new:Npn \tl_item:nn #1#2
12887 {
12888     \exp_args:Nf \__tl_item:nn
12889     { \exp_args:Nf \__tl_item_aux:nn { \int_eval:n {#2} } {#1} }
12890     #1
12891     \q__tl_recursion_tail
12892     \prg_break_point:
12893 }
12894 \cs_new:Npn \__tl_item_aux:nn #1#2
12895 {
12896     \int_compare:nNnTF {#1} < 0
12897     { \int_eval:n { \tl_count:n {#2} + 1 + #1 } }
12898     {#1}
12899 }
12900 \cs_new:Npn \__tl_item:Nn #1#2
12901 {
12902     \__tl_if_recursion_tail_break:nN {#2} \prg_break:
12903     \int_compare:nNnTF {#1} = 1
12904     { \prg_break:n { \exp_not:n {#2} } } }

```

```

12905     { \exp_args:Nf \__tl_item:nn { \int_eval:n { #1 - 1 } } }
12906   }
12907 \cs_new:Npn \tl_item:Nn { \exp_args:No \tl_item:nn }
12908 \cs_generate_variant:Nn \tl_item:Nn { c }

```

(End definition for `\tl_item:nn` and others. These functions are documented on page 113.)

`\tl_rand_item:n` Importantly `\tl_item:nn` only evaluates its argument once.

```

12909 \cs_new:Npn \tl_rand_item:n #1
12910 {
12911   \tl_if_blank:nF {#1}
12912   { \tl_item:nn {#1} { \int_rand:nn { 1 } { \tl_count:n {#1} } } }
12913 }
12914 \cs_new:Npn \tl_rand_item:N { \exp_args:No \tl_rand_item:n }
12915 \cs_generate_variant:Nn \tl_rand_item:N { c }

```

(End definition for `\tl_rand_item:n` and `\tl_rand_item:N`. These functions are documented on page 114.)

`\tl_range:Nnn` To avoid checking for the end of the token list at every step, start by counting the number l of items and “normalizing” the bounds, namely clamping them to the interval $[0, l]$ and dealing with negative indices. More precisely, `__tl_range_items:nnNn` receives the number of items to skip at the beginning of the token list, the index of the last item to keep, a function which is either `__tl_range:w` or the token list itself. If nothing should be kept, leave `{}`: this stops the `f`-expansion of `\tl_head:f` and that function produces an empty result. Otherwise, repeatedly call `__tl_range_skip:w` to delete `#1` items from the input stream (the extra brace group avoids an off-by-one shift). For the braced version `__tl_range_braced:w` sets up `__tl_range_collect_braced:w` which stores items one by one in an argument after the semicolon. Depending on the first token of the tail, either just move it (if it is a space) or also decrement the number of items left to find. Eventually, the result is a brace group followed by the rest of the token list, and `\tl_head:f` cleans up and gives the result in `\exp_not:n`.

```

12916 \cs_new:Npn \tl_range:Nnn { \exp_args:No \tl_range:nnn }
12917 \cs_generate_variant:Nn \tl_range:Nnn { c }
12918 \cs_new:Npn \tl_range:nnn { \__tl_range:Nnnn \__tl_range:w }
12919 \cs_new:Npn \__tl_range:Nnnn #1#2#3#4
12920 {
12921   \tl_head:f
12922   {
12923     \exp_args:Nf \__tl_range:nnnNn
12924     { \tl_count:n {#2} } {#3} {#4} #1 {#2}
12925   }
12926 }
12927 \cs_new:Npn \__tl_range:nnnNn #1#2#3
12928 {
12929   \exp_args:Nff \__tl_range:nnNn
12930   {
12931     \exp_args:Nf \__tl_range_normalize:nn
12932     { \int_eval:n { #2 - 1 } } {#1}
12933   }
12934   {
12935     \exp_args:Nf \__tl_range_normalize:nn
12936     { \int_eval:n {#3} } {#1}
12937   }

```

```

12938 }
12939 \cs_new:Npn \__tl_range:nnNn #1#2#3#4
12940 {
12941   \if_int_compare:w #2 > #1 \exp_stop_f: \else:
12942     \exp_after:wN { \exp_after:wN }
12943   \fi:
12944   \exp_after:wN #3
12945   \int_value:w \int_eval:n { #2 - #1 } \exp_after:wN ;
12946   \exp_after:wN { \exp:w \__tl_range_skip:w #1 ; { } #4 }
12947 }
12948 \cs_new:Npn \__tl_range_skip:w #1 ; #2
12949 {
12950   \if_int_compare:w #1 > \c_zero_int
12951     \exp_after:wN \__tl_range_skip:w
12952     \int_value:w \int_eval:n { #1 - 1 } \exp_after:wN ;
12953   \else:
12954     \exp_after:wN \exp_end:
12955   \fi:
12956 }
12957 \cs_new:Npn \__tl_range:w #1 ; #2
12958 {
12959   \exp_args:Nf \__tl_range_collect:nn
12960   { \__tl_range_skip_spaces:n {#2} } {#1}
12961 }
12962 \cs_new:Npn \__tl_range_skip_spaces:n #1
12963 {
12964   \tl_if_head_is_space:nTF {#1}
12965   { \exp_args:Nf \__tl_range_skip_spaces:n {#1} }
12966   { { } #1 }
12967 }
12968 \cs_new:Npn \__tl_range_collect:nn #1#2
12969 {
12970   \int_compare:nNnTF {#2} = 0
12971   {#1}
12972   {
12973     \exp_args:No \tl_if_head_is_space:nTF { \use_none:n #1 }
12974     {
12975       \exp_args:Nf \__tl_range_collect:nn
12976       { \__tl_range_collect_space:nw #1 }
12977       {#2}
12978     }
12979     {
12980       \__tl_range_collect:ff
12981       {
12982         \exp_args:No \tl_if_head_is_N_type:nTF { \use_none:n #1 }
12983         { \__tl_range_collect_N:nN }
12984         { \__tl_range_collect_group:nn }
12985         #1
12986       }
12987       { \int_eval:n { #2 - 1 } }
12988     }
12989   }
12990 }
12991 \cs_new:Npn \__tl_range_collect_space:nw #1 ~ { { #1 ~ } }

```

```

12992 \cs_new:Npn \__tl_range_collect:N:nN #1#2 { { #1 #2 } }
12993 \cs_new:Npn \__tl_range_collect_group:nn #1#2 { { #1 {#2} } }
12994 \cs_generate_variant:Nn \__tl_range_collect:nn { ff }

```

(End definition for `\tl_range:Nnn` and others. These functions are documented on page 115.)

`__tl_range_normalize:nn` This function converts an $\langle index \rangle$ argument into an explicit position in the token list (a result of 0 denoting “out of bounds”). Expects two explicit integer arguments: the $\langle index \rangle$ #1 and the string count #2. If #1 is negative, replace it by #1 + #2 + 1, then limit to the range [0, #2].

```

12995 \cs_new:Npn \__tl_range_normalize:nn #1#2
12996 {
12997   \int_eval:n
12998   {
12999     \if_int_compare:w #1 < \c_zero_int
13000     \if_int_compare:w #1 < -#2 \exp_stop_f:
13001       0
13002     \else:
13003       #1 + #2 + 1
13004     \fi:
13005   \else:
13006     \if_int_compare:w #1 < #2 \exp_stop_f:
13007       #1
13008     \else:
13009       #2
13010     \fi:
13011   \fi:
13012 }
13013 }

```

(End definition for `__tl_range_normalize:nn`.)

52.14 Viewing token lists

`\tl_show:N` Showing token list variables is done after checking that the variable is defined (see `\tl_show:c` `__kernel_register_show:N`).

`\tl_log:N` `\tl_log:c` `__tl_show:NN`

```

13014 \cs_new_protected:Npn \tl_show:N { \__tl_show:NN \tl_show:n }
13015 \cs_generate_variant:Nn \tl_show:N { c }
13016 \cs_new_protected:Npn \tl_log:N { \__tl_show:NN \tl_log:n }
13017 \cs_generate_variant:Nn \tl_log:N { c }
13018 \cs_new_protected:Npn \__tl_show:NN #1#2
13019 {
13020   \__kernel_chk_defined:NT #2
13021   {
13022     \exp_args:Nf \tl_if_empty:nTF
13023     { \cs_prefix_spec:N #2 \cs_argument_spec:N #2 }
13024     {
13025       \exp_args:Ne #1
13026       { \token_to_str:N #2 = \__kernel_exp_not:w \exp_after:wN {#2} }
13027     }
13028     {
13029       \msg_error:nnxxx { kernel } { bad-type }
13030       { \token_to_str:N #2 } { \token_to_meaning:N #2 } { tl }

```



```

13031     }
13032   }
13033 }

```

(End definition for `\tl_show:N`, `\tl_log:N`, and `__tl_show:NN`. These functions are documented on page 109.)

`\tl_show:n` Many `show` functions are based on `\tl_show:n`. The argument of `\tl_show:n` is line-wrapped using `\iow_wrap:nnnN` but with a leading `>~` and trailing period, both removed before passing the wrapped text to the `\showtokens` primitive. This primitive shows the result with a leading `>~` and trailing period.

The token list `\l__tl_internal_a_tl` containing the result of all these manipulations is displayed to the terminal using `\tex_showtokens:D` and an odd `\exp_after:wN` which expand the closing brace to improve the output slightly. The calls to `__kernel_iow_with:Nnn` ensure that the `\newlinechar` is set to 10 so that the `\iow_newline:` inserted by the line-wrapping code are correctly recognized by \TeX , and that `\errorcontextlines` is `-1` to avoid printing irrelevant context.

```

13034 \cs_new_protected:Npn \tl_show:n #1
13035 { \iow_wrap:nnnN { >~ \tl_to_str:n {#1} . } { } { } \__tl_show:n }
13036 \cs_new_protected:Npn \__tl_show:n #1
13037 {
13038   \tl_set:Nf \l__tl_internal_a_tl { \__tl_show:w #1 \s__tl_stop }
13039   \__kernel_iow_with:Nnn \tex_newlinechar:D { 10 }
13040   {
13041     \__kernel_iow_with:Nnn \tex_errorcontextlines:D { -1 }
13042     {
13043       \tex_showtokens:D \exp_after:wN \exp_after:wN \exp_after:wN
13044       { \exp_after:wN \l__tl_internal_a_tl }
13045     }
13046   }
13047 }
13048 \cs_new:Npn \__tl_show:w #1 > #2 . \s__tl_stop {#2}

```

(End definition for `\tl_show:n`, `__tl_show:n`, and `__tl_show:w`. This function is documented on page 109.)

`\tl_log:n` Logging is much easier, simply line-wrap. The `>~` and trailing period is there to match the output of `\tl_show:n`.

```

13049 \cs_new_protected:Npn \tl_log:n #1
13050 { \iow_wrap:nnnN { > ~ \tl_to_str:n {#1} . } { } { } \iow_log:n }

```

(End definition for `\tl_log:n`. This function is documented on page 110.)

`__kernel_chk_tl_type:NnnT` Helper for checking that `#1` has the correct internal structure to be of a certain type. Make sure that it is defined and that it is a token list, namely a macro with no `\long` nor `\protected` prefix. Then compare `#1` to an attempt at reconstructing a valid structure of the given type using `#2` (see implementation of `\seq_show:N` for instance). If that is successful run the requested code `#4`.

```

13051 \cs_new_protected:Npn \__kernel_chk_tl_type:NnnT #1#2#3#4
13052 {
13053   \__kernel_chk_defined:NT #1
13054   {
13055     \exp_args:Nf \tl_if_empty:nTF
13056     { \cs_prefix_spec:N #1 \cs_argument_spec:N #1 }

```

```

13057     {
13058         \tl_set:Nx \l__tl_internal_a_tl {#3}
13059         \tl_if_eq:NNTF #1 \l__tl_internal_a_tl
13060             {#4}
13061         {
13062             \msg_error:nnxxxx { kernel } { bad-type }
13063             { \token_to_str:N #1 } { \tl_to_str:N #1 }
13064             {#2} { \tl_to_str:N \l__tl_internal_a_tl }
13065         }
13066     }
13067     {
13068         \msg_error:nnxxx { kernel } { bad-type }
13069         { \token_to_str:N #1 } { \token_to_meaning:N #1 } {#2}
13070     }
13071 }
13072 }

```

(End definition for `__kernel_chk_tl_type:NnnT.`)

52.15 Internal scan marks

`\s__tl_nil` Internal scan marks. These are defined here at the end because the code for `\scan_new:N` depends on some `!3tl` functions.

```

\s__tl_mark
\s__tl_stop
13073 \scan_new:N \s__tl_nil
13074 \scan_new:N \s__tl_mark
13075 \scan_new:N \s__tl_stop

```

(End definition for `\s__tl_nil`, `\s__tl_mark`, and `\s__tl_stop`.)

52.16 Scratch token lists

`\g_tmpa_tl` Global temporary token list variables. They are supposed to be set and used immediately, with no delay between the definition and the use because you can't count on other macros not to redefine them from under you.

```

13076 \tl_new:N \g_tmpa_tl
13077 \tl_new:N \g_tmpb_tl

```

(End definition for `\g_tmpa_tl` and `\g_tmpb_tl`. These variables are documented on page 119.)

`\l_tmpa_tl` These are local temporary token list variables. Be sure not to assume that the value you put into them will survive for long—see discussion above.

```

13078 \tl_new:N \l_tmpa_tl
13079 \tl_new:N \l_tmpb_tl

```

(End definition for `\l_tmpa_tl` and `\l_tmpb_tl`. These variables are documented on page 118.)

We finally clean up a temporary control sequence that we have used at various points to set up some definitions.

```

13080 \cs_undefine:N \__tl_tmp:w
13081 \</package>

```

Chapter 53

l3str implementation

13082 $\langle *package \rangle$

13083 $\langle @@=str \rangle$

53.1 Internal auxiliaries

$\backslash s_str_mark$ Internal scan marks.

$\backslash s_str_stop$ 13084 $\backslash scan_new:N \backslash s_str_mark$

13085 $\backslash scan_new:N \backslash s_str_stop$

(End definition for $\backslash s_str_mark$ and $\backslash s_str_stop$.)

$\backslash_str_use_none_delimit_by_s_stop:w$ Functions to gobble up to a scan mark.

$\backslash_str_use_i_delimit_by_s_stop:nw$ 13086 $\backslash cs_new:Npn \backslash_str_use_none_delimit_by_s_stop:w \#1 \backslash s_str_stop \{ \}$

13087 $\backslash cs_new:Npn \backslash_str_use_i_delimit_by_s_stop:nw \#1 \#2 \backslash s_str_stop \{ \#1 \}$

(End definition for $\backslash_str_use_none_delimit_by_s_stop:w$ and $\backslash_str_use_i_delimit_by_s_stop:nw$.)

$\backslash q_str_recursion_tail$ Internal recursion quarks.

$\backslash q_str_recursion_stop$ 13088 $\backslash quark_new:N \backslash q_str_recursion_tail$

13089 $\backslash quark_new:N \backslash q_str_recursion_stop$

(End definition for $\backslash q_str_recursion_tail$ and $\backslash q_str_recursion_stop$.)

$\backslash_str_if_recursion_tail_break:NN$ Functions to query recursion quarks.

$\backslash_str_if_recursion_tail_stop_do:Nn$ 13090 $\backslash_kernel_quark_new_test:N \backslash_str_if_recursion_tail_break:NN$

13091 $\backslash_kernel_quark_new_test:N \backslash_str_if_recursion_tail_stop_do:Nn$

(End definition for $\backslash_str_if_recursion_tail_break:NN$ and $\backslash_str_if_recursion_tail_stop_do:Nn$.)

53.2 Creating and setting string variables

`\str_new:N` A string is simply a token list. The full mapping system isn't set up yet so do things by hand.

```

\str_new:c
\str_use:N
\str_use:c
\str_clear:N
\str_clear:c
\str_gclear:N
\str_gclear:c
\str_clear_new:N
\str_clear_new:c
\str_gclear_new:N
\str_gclear_new:c
\str_set_eq:NN
\str_set_eq:cN
\str_set_eq:Nc
\str_set_eq:cc
\str_gset_eq:NN
\str_gset_eq:cN
\str_gset_eq:Nc
\str_gset_eq:cc
\str_concat:NNN
\str_concat:ccc
\str_gconcat:NNN
\str_gconcat:ccc
13092 \group_begin:
13093   \cs_set_protected:Npn \__str_tmp:n #1
13094   {
13095     \tl_if_blank:nF {#1}
13096     {
13097       \cs_new_eq:cc { str_ #1 :N } { tl_ #1 :N }
13098       \exp_args:Nc \cs_generate_variant:Nn { str_ #1 :N } { c }
13099       \__str_tmp:n
13100     }
13101   }
13102   \__str_tmp:n
13103   { new }
13104   { use }
13105   { clear }
13106   { gclear }
13107   { clear_new }
13108   { gclear_new }
13109   { }
13110 \group_end:
13111 \cs_new_eq:NN \str_set_eq:NN \tl_set_eq:NN
13112 \cs_new_eq:NN \str_gset_eq:NN \tl_gset_eq:NN
13113 \cs_generate_variant:Nn \str_set_eq:NN { c , Nc , cc }
13114 \cs_generate_variant:Nn \str_gset_eq:NN { c , Nc , cc }
13115 \cs_new_eq:NN \str_concat:NNN \tl_concat:NNN
13116 \cs_new_eq:NN \str_gconcat:NNN \tl_gconcat:NNN
13117 \cs_generate_variant:Nn \str_concat:NNN { ccc }
13118 \cs_generate_variant:Nn \str_gconcat:NNN { ccc }

```

(End definition for `\str_new:N` and others. These functions are documented on page 121.)

`\str_set:Nn` Simply convert the token list inputs to $\langle strings \rangle$.

```

\str_set:NV
\str_set:Nx
\str_set:cn
\str_set:cV
\str_set:cx
\str_gset:Nn
\str_gset:NV
\str_gset:Nx
\str_gset:cn
\str_gset:cV
\str_gset:cx
\str_const:Nn
\str_const:NV
\str_const:Nx
\str_const:cn
\str_const:cV
\str_const:cx
\str_put_left:Nn
\str_put_left:NV
\str_put_left:Nx
\str_put_left:cn
\str_put_left:cV
\str_put_left:cx
\str_gput_left:Nn
\str_gput_left:NV
\str_gput_left:Nx
\str_gput_left:cn
\str_gput_left:cV
13119 \group_begin:
13120   \cs_set_protected:Npn \__str_tmp:n #1
13121   {
13122     \tl_if_blank:nF {#1}
13123     {
13124       \cs_new_protected:cpx { str_ #1 :Nn } ##1##2
13125       {
13126         \exp_not:c { tl_ #1 :Nx } ##1
13127         { \exp_not:N \tl_to_str:n {##2} }
13128       }
13129       \cs_generate_variant:cn { str_ #1 :Nn } { NV , Nx , cn , cV , cx }
13130       \__str_tmp:n
13131     }
13132   }
13133   \__str_tmp:n
13134   { set }
13135   { gset }
13136   { const }
13137   { put_left }

```

```

13138 { gput_left }
13139 { put_right }
13140 { gput_right }
13141 { }
13142 \group_end:

```

(End definition for `\str_set:Nn` and others. These functions are documented on page 122.)

53.3 Modifying string variables

```

\str_replace_all:Nnn Start by applying \tl_to_str:n to convert the old and new token lists to strings, and
\str_replace_all:cnn also apply \tl_to_str:N to avoid any issues if we are fed a token list variable. Then
\str_greplace_all:Nnn the code is a much simplified version of the token list code because neither the delimiter
\str_greplace_all:cnn nor the replacement can contain macro parameters or braces. The delimiter \s__str_-
\str_replace_once:Nnn mark cannot appear in the string to edit so it is used in all cases. Some x-expansion is
\str_replace_once:cnn unnecessary. There is no need to avoid losing braces nor to protect against expansion.
\str_greplace_once:Nnn The ending code is much simplified and does not need to hide in braces.
\str_greplace_once:cnn
  \__str_replace:NNNnn
\__str_replace_aux:NNNnnn
  \__str_replace_next:w
13143 \cs_new_protected:Npn \str_replace_once:Nnn
13144 { \__str_replace:NNNnn \prg_do_nothing: \__kernel_tl_set:Nx }
13145 \cs_new_protected:Npn \str_greplace_once:Nnn
13146 { \__str_replace:NNNnn \prg_do_nothing: \__kernel_tl_gset:Nx }
13147 \cs_new_protected:Npn \str_replace_all:Nnn
13148 { \__str_replace:NNNnn \__str_replace_next:w \__kernel_tl_set:Nx }
13149 \cs_new_protected:Npn \str_greplace_all:Nnn
13150 { \__str_replace:NNNnn \__str_replace_next:w \__kernel_tl_gset:Nx }
13151 \cs_generate_variant:Nn \str_replace_once:Nnn { c }
13152 \cs_generate_variant:Nn \str_greplace_once:Nnn { c }
13153 \cs_generate_variant:Nn \str_replace_all:Nnn { c }
13154 \cs_generate_variant:Nn \str_greplace_all:Nnn { c }
13155 \cs_new_protected:Npn \__str_replace:NNNnn #1#2#3#4#5
13156 {
13157   \tl_if_empty:nTF {#4}
13158   {
13159     \msg_error:nnx { kernel } { empty-search-pattern } {#5}
13160   }
13161   {
13162     \use:x
13163     {
13164       \exp_not:n { \__str_replace_aux:NNNnnn #1 #2 #3 }
13165       { \tl_to_str:N #3 }
13166       { \tl_to_str:n {#4} } { \tl_to_str:n {#5} }
13167     }
13168   }
13169 }
13170 \cs_new_protected:Npn \__str_replace_aux:NNNnnn #1#2#3#4#5#6
13171 {
13172   \cs_set:Npn \__str_replace_next:w ##1 #5 { ##1 #6 #1 }
13173   #2 #3
13174   {
13175     \__str_replace_next:w
13176     #4
13177     \__str_use_none_delimit_by_s_stop:w
13178     #5

```

```

13179         \s__str_stop
13180     }
13181 }
13182 \cs_new_eq:NN \__str_replace_next:w ?

```

(End definition for `\str_replace_all:Nnn` and others. These functions are documented on page 129.)

```

\str_remove_once:Nn Removal is just a special case of replacement.
\str_remove_once:cn 13183 \cs_new_protected:Npn \str_remove_once:Nn #1#2
\str_gremove_once:Nn 13184 { \str_replace_once:Nnn #1 {#2} { } }
\str_gremove_once:cn 13185 \cs_new_protected:Npn \str_gremove_once:Nn #1#2
13186 { \str_greplace_once:Nnn #1 {#2} { } }
13187 \cs_generate_variant:Nn \str_remove_once:Nn { c }
13188 \cs_generate_variant:Nn \str_gremove_once:Nn { c }

```

(End definition for `\str_remove_once:Nn` and `\str_gremove_once:Nn`. These functions are documented on page 129.)

```

\str_remove_all:Nn Removal is just a special case of replacement.
\str_remove_all:cn 13189 \cs_new_protected:Npn \str_remove_all:Nn #1#2
\str_gremove_all:Nn 13190 { \str_replace_all:Nnn #1 {#2} { } }
\str_gremove_all:cn 13191 \cs_new_protected:Npn \str_gremove_all:Nn #1#2
13192 { \str_greplace_all:Nnn #1 {#2} { } }
13193 \cs_generate_variant:Nn \str_remove_all:Nn { c }
13194 \cs_generate_variant:Nn \str_gremove_all:Nn { c }

```

(End definition for `\str_remove_all:Nn` and `\str_gremove_all:Nn`. These functions are documented on page 129.)

53.4 String comparisons

```

\str_if_empty_p:N More copy-paste!
\str_if_empty_p:c 13195 \prg_new_eq_conditional:NNn \str_if_exist:N \tl_if_exist:N
\str_if_empty:NTF 13196 { p , T , F , TF }
\str_if_empty:cTF 13197 \prg_new_eq_conditional:NNn \str_if_exist:c \tl_if_exist:c
\str_if_exist_p:N 13198 { p , T , F , TF }
\str_if_exist_p:c 13199 \prg_new_eq_conditional:NNn \str_if_empty:N \tl_if_empty:N
\str_if_exist:NTF 13200 { p , T , F , TF }
\str_if_exist:cTF 13201 \prg_new_eq_conditional:NNn \str_if_empty:c \tl_if_empty:c
13202 { p , T , F , TF }

```

(End definition for `\str_if_empty:NTF` and `\str_if_exist:NTF`. These functions are documented on page 122.)

```

\__str_if_eq:nn String comparisons rely on the primitive \pdfstrcmp, so we define a new name for it.
13203 \cs_new_eq:NN \__str_if_eq:nn \tex_strcmp:D

```

(End definition for `__str_if_eq:nn`.)

```

\str_compare_p:nNn Simply rely on \__str_if_eq:nn, which expands to -1, 0 or 1. The ee version is created
\str_compare_p:eNe directly because it is more efficient.
\str_compare:nNnTF 13204 \prg_new_conditional:Npnn \str_compare:nNn #1#2#3 { p , T , F , TF }
\str_compare:eNeTF 13205 {
13206     \if_int_compare:w

```

```

13207     \_str_if_eq:nn { \exp_not:n {#1} } { \exp_not:n {#3} }
13208     #2 \c_zero_int
13209     \prg_return_true: \else: \prg_return_false: \fi:
13210 }
13211 \prg_new_conditional:Npnn \str_compare:eNe #1#2#3 { p , T , F , TF }
13212 {
13213     \if_int_compare:w \_str_if_eq:nn {#1} {#3} #2 \c_zero_int
13214     \prg_return_true: \else: \prg_return_false: \fi:
13215 }

```

(End definition for `\str_compare:nNnTF`. This function is documented on page 124.)

`\str_if_eq_p:nn` Modern engines provide a direct way of comparing two token lists, but returning a number. This set of conditionals therefore makes life a bit clearer. The `nn` and `ee` versions are created directly as this is most efficient. Since `_str_if_eq:nn` will expand to 0 as an explicit character with category 12 if the two lists match (and either -1 or 1 if they don't) we can use `\if:w` here which is faster than using `\if_int_compare:w`.

```

13216 \prg_new_conditional:Npnn \str_if_eq:nn #1#2 { p , T , F , TF }
13217 {
13218     \if:w 0 \_str_if_eq:nn { \exp_not:n {#1} } { \exp_not:n {#2} }
13219     \prg_return_true: \else: \prg_return_false: \fi:
13220 }
13221 \prg_generate_conditional_variant:Nnn \str_if_eq:nn
13222 { V , v , o , nV , no , VV , nv } { p , T , F , TF }
13223 \prg_new_conditional:Npnn \str_if_eq:ee #1#2 { p , T , F , TF }
13224 {
13225     \if:w 0 \_str_if_eq:nn {#1} {#2}
13226     \prg_return_true: \else: \prg_return_false: \fi:
13227 }

```

(End definition for `\str_if_eq:nnTF`. This function is documented on page 123.)

`\str_if_eq_p:NN` Note that `\str_if_eq:NNTF` is different from `\tl_if_eq:NNTF` because it needs to ignore category codes.

```

13228 \prg_new_conditional:Npnn \str_if_eq:NN #1#2 { p , TF , T , F }
13229 {
13230     \if:w 0 \_str_if_eq:nn { \tl_to_str:N #1 } { \tl_to_str:N #2 }
13231     \prg_return_true: \else: \prg_return_false: \fi:
13232 }
13233 \prg_generate_conditional_variant:Nnn \str_if_eq:NN
13234 { c , Nc , cc } { T , F , TF , p }

```

(End definition for `\str_if_eq:NNTF`. This function is documented on page 122.)

`\str_if_in:NnTF` Everything here needs to be detokenized but beyond that it is a simple token list test.
`\str_if_in:cnTF` It would be faster to fine-tune the T, F, TF variants by calling the appropriate variant of
`\str_if_in:nnTF` `\tl_if_in:nnTF` directly but that takes more code.

```

13235 \prg_new_protected_conditional:Npnn \str_if_in:Nn #1#2 { T , F , TF }
13236 {
13237     \use:x
13238     { \tl_if_in:nnTF { \tl_to_str:N #1 } { \tl_to_str:n {#2} } }
13239     { \prg_return_true: } { \prg_return_false: }
13240 }
13241 \prg_generate_conditional_variant:Nnn \str_if_in:Nn

```

```

13242 { c } { T , F , TF }
13243 \prg_new_protected_conditional:Npnn \str_if_in:nn #1#2 { T , F , TF }
13244 {
13245   \use:x
13246   { \tl_if_in:nnTF { \tl_to_str:n {#1} } { \tl_to_str:n {#2} } }
13247   { \prg_return_true: } { \prg_return_false: }
13248 }

```

(End definition for `\str_if_in:NnTF` and `\str_if_in:nnTF`. These functions are documented on page 123.)

```

\str_case:nn Much the same as \tl_case:nnTF here: just a change in the internal comparison.
\str_case:Vn 13249 \cs_new:Npn \str_case:nn #1#2
\str_case:on 13250 {
\str_case:nV 13251   \exp:w
\str_case:nv 13252   \__str_case:nnTF {#1} {#2} { } { }
13253 }
\str_case:nnTF 13254 \cs_new:Npn \str_case:nnT #1#2#3
\str_case:VnTF 13255 {
\str_case:onTF 13256   \exp:w
\str_case:nVTF 13257   \__str_case:nnTF {#1} {#2} {#3} { }
\str_case:nvTF 13258 }
\str_case_e:nn 13259 \cs_new:Npn \str_case:nnF #1#2
\str_case_e:nnTF 13260 {
\__str_case:nnTF 13261   \exp:w
\__str_case_e:nnTF 13262   \__str_case:nnTF {#1} {#2} { }
\__str_case:nw 13263 }
\__str_case_e:nw 13264 \cs_new:Npn \str_case:nnTF #1#2
\__str_case_end:nw 13265 {
13266   \exp:w
13267   \__str_case:nnTF {#1} {#2}
13268 }
13269 \cs_new:Npn \__str_case:nnTF #1#2#3#4
13270 { \__str_case:nw {#1} #2 {#1} { } \s_str_mark {#3} \s_str_mark {#4} \s_str_stop }
13271 \cs_generate_variant:Nn \str_case:nn { V , o , nV , nv }
13272 \prg_generate_conditional_variant:Nnn \str_case:nn
13273 { V , o , nV , nv } { T , F , TF }
13274 \cs_new:Npn \__str_case:nw #1#2#3
13275 {
13276   \str_if_eq:nnTF {#1} {#2}
13277   { \__str_case_end:nw {#3} }
13278   { \__str_case:nw {#1} }
13279 }
13280 \cs_new:Npn \str_case_e:nn #1#2
13281 {
13282   \exp:w
13283   \__str_case_e:nnTF {#1} {#2} { } { }
13284 }
13285 \cs_new:Npn \str_case_e:nnT #1#2#3
13286 {
13287   \exp:w
13288   \__str_case_e:nnTF {#1} {#2} {#3} { }
13289 }
13290 \cs_new:Npn \str_case_e:nnF #1#2

```



```

13291 {
13292   \exp:w
13293   \__str_case_e:nnTF {#1} {#2} { }
13294 }
13295 \cs_new:Npn \str_case_e:nnTF #1#2
13296 {
13297   \exp:w
13298   \__str_case_e:nnTF {#1} {#2}
13299 }
13300 \cs_new:Npn \__str_case_e:nnTF #1#2#3#4
13301 { \__str_case_e:nw {#1} #2 {#1} { } \s__str_mark {#3} \s__str_mark {#4} \s__str_stop }
13302 \cs_new:Npn \__str_case_e:nw #1#2#3
13303 {
13304   \str_if_eq:eeTF {#1} {#2}
13305   { \__str_case_end:nw {#3} }
13306   { \__str_case_e:nw {#1} }
13307 }
13308 \cs_new:Npn \__str_case_end:nw #1#2#3 \s__str_mark #4#5 \s__str_stop
13309 { \exp_end: #1 #4 }

```

(End definition for `\str_case:nnTF` and others. These functions are documented on page 123.)

53.5 Mapping over strings

`\str_map_function:NN` The inline and variable mappings are similar to the usual token list mappings but start out by turning the argument to an “other string”. Doing the same for the expandable function mapping would require `__kernel_str_to_other:n`, quadratic in the string length. To deal with spaces in that case, `__str_map_function:w` replaces the following space by a braced space and a further call to itself. These are received by `__str_map_function:nn`, which passes the space to `#1` and calls `__str_map_function:w` to deal with the next space. The space before the braced space allows to optimize the `\q__str_recursion_tail` test. Of course we need to include a trailing space (the question mark is needed to avoid losing the space when TeX tokenizes the line). At the cost of about three more auxiliaries this code could get a 9 times speed up by testing only every 9-th character for whether it is `\q__str_recursion_tail` (also by converting 9 spaces at a time in the `\str_map_function:nN` case).

`\str_map_function:cN` For the `map_variable` functions we use a string assignment to store each character because spaces are made catcode 12 before the loop.

`\str_map_function:nN`

`\str_map_inline:Nn`

`\str_map_inline:cn`

`\str_map_inline:nn`

`\str_map_variable:NNn`

`\str_map_variable:cNn`

`\str_map_variable:nNn`

`\str_map_break:n`

`\str_map_break:n`

`__str_map_function:w`

`__str_map_function:nn`

`__str_map_inline:NN`

`__str_map_variable:NnN`

```

13310 \cs_new:Npn \str_map_function:nN #1#2
13311 {
13312   \exp_after:wN \__str_map_function:w
13313   \exp_after:wN \__str_map_function:nn \exp_after:wN #2
13314   \__kernel_tl_to_str:w {#1}
13315   \q__str_recursion_tail ? ~
13316   \prg_break_point:Nn \str_map_break: { }
13317 }
13318 \cs_new:Npn \str_map_function:NN
13319 { \exp_args:No \str_map_function:nN }
13320 \cs_new:Npn \__str_map_function:w #1 ~
13321 { #1 { ~ { ~ } \__str_map_function:w } }
13322 \cs_new:Npn \__str_map_function:nn #1#2
13323 {

```

```

13324     \if_meaning:w \q__str_recursion_tail #2
13325     \exp_after:wN \str_map_break:
13326     \fi:
13327     #1 #2 \__str_map_function:nn {#1}
13328 }
13329 \cs_generate_variant:Nn \str_map_function:NN { c }
13330 \cs_new_protected:Npn \str_map_inline:nn #1#2
13331 {
13332     \int_gincr:N \g__kernel_prg_map_int
13333     \cs_gset_protected:cpn
13334     { \__str_map_ \int_use:N \g__kernel_prg_map_int :w } ##1 {#2}
13335     \use:x
13336     {
13337         \exp_not:N \__str_map_inline:NN
13338         \exp_not:c { \__str_map_ \int_use:N \g__kernel_prg_map_int :w }
13339         \__kernel_str_to_other_fast:n {#1}
13340     }
13341     \q__str_recursion_tail
13342     \prg_break_point:Nn \str_map_break:
13343     { \int_gdecr:N \g__kernel_prg_map_int }
13344 }
13345 \cs_new_protected:Npn \str_map_inline:Nn
13346 { \exp_args:No \str_map_inline:nn }
13347 \cs_generate_variant:Nn \str_map_inline:Nn { c }
13348 \cs_new:Npn \__str_map_inline:NN #1#2
13349 {
13350     \__str_if_recursion_tail_break:NN #2 \str_map_break:
13351     \exp_args:No #1 { \token_to_str:N #2 }
13352     \__str_map_inline:NN #1
13353 }
13354 \cs_new_protected:Npn \str_map_variable:nNn #1#2#3
13355 {
13356     \use:x
13357     {
13358         \exp_not:n { \__str_map_variable:NnN #2 {#3} }
13359         \__kernel_str_to_other_fast:n {#1}
13360     }
13361     \q__str_recursion_tail
13362     \prg_break_point:Nn \str_map_break: { }
13363 }
13364 \cs_new_protected:Npn \str_map_variable:NNn
13365 { \exp_args:No \str_map_variable:nNn }
13366 \cs_new_protected:Npn \__str_map_variable:NnN #1#2#3
13367 {
13368     \__str_if_recursion_tail_break:NN #3 \str_map_break:
13369     \str_set:Nn #1 {#3}
13370     \use:n {#2}
13371     \__str_map_variable:NnN #1 {#2}
13372 }
13373 \cs_generate_variant:Nn \str_map_variable:NNn { c }
13374 \cs_new:Npn \str_map_break:
13375 { \prg_map_break:Nn \str_map_break: { } }
13376 \cs_new:Npn \str_map_break:n
13377 { \prg_map_break:Nn \str_map_break: }

```

(End definition for `\str_map_function:Nn` and others. These functions are documented on page 124.)

```

\str_map_tokens:Nn Uses an auxiliary of \str_map_function:Nn.
\str_map_tokens:cn 13378 \cs_new:Npn \str_map_tokens:nn #1#2
\str_map_tokens:nn 13379 {
13380   \exp_args:Nno \use:nn
13381   { \__str_map_function:w \__str_map_function:nn {#2} }
13382   { \__kernel_tl_to_str:w {#1} }
13383   \q__str_recursion_tail ? ~
13384   \prg_break_point:Nn \str_map_break: { }
13385 }
13386 \cs_new:Npn \str_map_tokens:Nn { \exp_args:No \str_map_tokens:nn }
13387 \cs_generate_variant:Nn \str_map_tokens:Nn { c }

```

(End definition for `\str_map_tokens:Nn` and `\str_map_tokens:nn`. These functions are documented on page 125.)

53.6 Accessing specific characters in a string

`__kernel_str_to_other:n` First apply `\tl_to_str:n`, then replace all spaces by “other” spaces, 8 at a time, storing the converted part of the string between the `\s__str_mark` and `\s__str_stop` markers. `__str_to_other_loop:w` The end is detected when `__str_to_other_loop:w` finds one of the trailing A, distinguished from any contents of the initial token list by their category. Then `__str_to_other_end:w` is called, and finds the result between `\s__str_mark` and the first A (well, there is also the need to remove a space).

```

13388 \cs_new:Npn \__kernel_str_to_other:n #1
13389 {
13390   \exp_after:wN \__str_to_other_loop:w
13391   \tl_to_str:n {#1} ~ A ~ A ~ A ~ A ~ A ~ A ~ A ~ \s__str_mark \s__str_stop
13392 }
13393 \group_begin:
13394 \tex_lccode:D '\* = '\ %
13395 \tex_lccode:D '\A = '\A %
13396 \tex_lowercase:D
13397 {
13398   \group_end:
13399   \cs_new:Npn \__str_to_other_loop:w
13400   #1 ~ #2 ~ #3 ~ #4 ~ #5 ~ #6 ~ #7 ~ #8 ~ #9 \s__str_stop
13401   {
13402     \if_meaning:w A #8
13403     \__str_to_other_end:w
13404     \fi:
13405     \__str_to_other_loop:w
13406     #9 #1 * #2 * #3 * #4 * #5 * #6 * #7 * #8 * \s__str_stop
13407   }
13408   \cs_new:Npn \__str_to_other_end:w \fi: #1 \s__str_mark #2 * A #3 \s__str_stop
13409   { \fi: #2 }
13410 }

```

(End definition for `__kernel_str_to_other:n`, `__str_to_other_loop:w`, and `__str_to_other_end:w`.)

```

\__kernel_str_to_other_fast:n
\__kernel_str_to_other_fast_loop:w
\__str_to_other_fast_end:w

```

The difference with `__kernel_str_to_other:n` is that the converted part is left in the input stream, making these commands only restricted-expandable.

```

13411 \cs_new:Npn \__kernel_str_to_other_fast:n #1
13412 {
13413   \exp_after:wN \__str_to_other_fast_loop:w \tl_to_str:n {#1} ~
13414   A ~ A ~ A ~ A ~ A ~ A ~ A ~ A ~ A ~ \s__str_stop
13415 }
13416 \group_begin:
13417 \tex_lccode:D '\* = '\ %
13418 \tex_lccode:D '\A = '\A %
13419 \tex_lowercase:D
13420 {
13421   \group_end:
13422   \cs_new:Npn \__str_to_other_fast_loop:w
13423     #1 ~ #2 ~ #3 ~ #4 ~ #5 ~ #6 ~ #7 ~ #8 ~ #9 ~
13424     {
13425       \if_meaning:w A #9
13426       \__str_to_other_fast_end:w
13427       \fi:
13428       #1 * #2 * #3 * #4 * #5 * #6 * #7 * #8 * #9
13429       \__str_to_other_fast_loop:w *
13430     }
13431   \cs_new:Npn \__str_to_other_fast_end:w #1 * A #2 \s__str_stop {#1}
13432 }

```

(End definition for `__kernel_str_to_other_fast:n`, `__kernel_str_to_other_fast_loop:w`, and `__str_to_other_fast_end:w`.)

```

\str_item:Nn
\str_item:cn
\str_item:nn
\str_item_ignore_spaces:nn
\__str_item:nn
\__str_item:w

```

The `\str_item:nn` hands its argument with spaces escaped to `__str_item:nn`, and makes sure to turn the result back into a proper string (with category code 10 spaces) eventually. The `\str_item_ignore_spaces:nn` function does not escape spaces, which are thus ignored by `__str_item:nn` since everything else is done with undelimited arguments. Evaluate the $\langle index \rangle$ argument `#2` and count characters in the string, passing those two numbers to `__str_item:w` for further analysis. If the $\langle index \rangle$ is negative, shift it by the $\langle count \rangle$ to know the how many character to discard, and if that is still negative give an empty result. If the $\langle index \rangle$ is larger than the $\langle count \rangle$, give an empty result, and otherwise discard $\langle index \rangle - 1$ characters before returning the following one. The shift by -1 is obtained by inserting an empty brace group before the string in that case: that brace group also covers the case where the $\langle index \rangle$ is zero.

```

13433 \cs_new:Npn \str_item:Nn { \exp_args:No \str_item:nn }
13434 \cs_generate_variant:Nn \str_item:Nn { c }
13435 \cs_new:Npn \str_item:nn #1#2
13436 {
13437   \exp_args:Nf \tl_to_str:n
13438   {
13439     \exp_args:Nf \__str_item:nn
13440     { \__kernel_str_to_other:n {#1} } {#2}
13441   }
13442 }
13443 \cs_new:Npn \str_item_ignore_spaces:nn #1
13444 { \exp_args:No \__str_item:nn { \tl_to_str:n {#1} } }
13445 \cs_new:Npn \__str_item:nn #1#2
13446 {

```

```

13447     \exp_after:wN \__str_item:w
13448     \int_value:w \int_eval:n {#2} \exp_after:wN ;
13449     \int_value:w \__str_count:n {#1} ;
13450     #1 \s__str_stop
13451 }
13452 \cs_new:Npn \__str_item:w #1; #2;
13453 {
13454     \int_compare:nNnTF {#1} < 0
13455     {
13456         \int_compare:nNnTF {#1} < {-#2}
13457         { \__str_use_none_delimit_by_s_stop:w }
13458         {
13459             \exp_after:wN \__str_use_i_delimit_by_s_stop:nw
13460             \exp:w \exp_after:wN \__str_skip_exp_end:w
13461             \int_value:w \int_eval:n { #1 + #2 } ;
13462         }
13463     }
13464     {
13465         \int_compare:nNnTF {#1} > {#2}
13466         { \__str_use_none_delimit_by_s_stop:w }
13467         {
13468             \exp_after:wN \__str_use_i_delimit_by_s_stop:nw
13469             \exp:w \__str_skip_exp_end:w #1 ; { }
13470         }
13471     }
13472 }

```

(End definition for `\str_item:Nn` and others. These functions are documented on page 127.)

```

\__str_skip_exp_end:w
\__str_skip_loop:wNNNNNNNN
\__str_skip_end:w
\__str_skip_end:NNNNNNNN

```

Removes $\max(\#1, 0)$ characters from the input stream, and then leaves `\exp_end:.` This should be expanded using `\exp:w`. We remove characters 8 at a time until there are at most 8 to remove. Then we do a dirty trick: the `\if_case:w` construction leaves between 0 and 8 times the `\or:` control sequence, and those `\or:` become arguments of `__str_skip_end:NNNNNNNN`. If the number of characters to remove is 6, say, then there are two `\or:` left, and the 8 arguments of `__str_skip_end:NNNNNNNN` are the two `\or:`, and 6 characters from the input stream, exactly what we wanted to remove. Then close the `\if_case:w` conditional with `\fi:`, and stop the initial expansion with `\exp_end:` (see places where `__str_skip_exp_end:w` is called).

```

13473 \cs_new:Npn \__str_skip_exp_end:w #1;
13474 {
13475     \if_int_compare:w #1 > 8 \exp_stop_f:
13476     \exp_after:wN \__str_skip_loop:wNNNNNNNN
13477     \else:
13478         \exp_after:wN \__str_skip_end:w
13479         \int_value:w \int_eval:w
13480     \fi:
13481     #1 ;
13482 }
13483 \cs_new:Npn \__str_skip_loop:wNNNNNNNN #1; #2#3#4#5#6#7#8#9
13484 {
13485     \exp_after:wN \__str_skip_exp_end:w
13486     \int_value:w \int_eval:n { #1 - 8 } ;
13487 }
13488 \cs_new:Npn \__str_skip_end:w #1 ;

```

```

13489 {
13490     \exp_after:wN \__str_skip_end:NNNNNNNN
13491     \if_case:w #1 \exp_stop_f: \or: \or: \or: \or: \or: \or: \or: \or:
13492 }
13493 \cs_new:Npn \__str_skip_end:NNNNNNNN #1#2#3#4#5#6#7#8 { \fi: \exp_end: }

```

(End definition for __str_skip_exp_end:w and others.)

\str_range:Nnn Sanitize the string. Then evaluate the arguments. At this stage we also decrement the $\langle start\ index \rangle$, since our goal is to know how many characters should be removed. Then

\str_range:nnn limit the range to be non-negative and at most the length of the string (this avoids

\str_range_ignore_spaces:nnn needing to check for the end of the string when grabbing characters), shifting negative

__str_range:nnn numbers by the appropriate amount. Afterwards, skip characters, then keep some more,

__str_range:w and finally drop the end of the string.

__str_range:nw

```

13494 \cs_new:Npn \str_range:Nnn { \exp_args:No \str_range:nnn }
13495 \cs_generate_variant:Nn \str_range:Nnn { c }
13496 \cs_new:Npn \str_range:nnn #1#2#3
13497 {
13498     \exp_args:Nf \tl_to_str:n
13499     {
13500         \exp_args:Nf \__str_range:nnn
13501         { \__kernel_str_to_other:n {#1} } {#2} {#3}
13502     }
13503 }
13504 \cs_new:Npn \str_range_ignore_spaces:nnn #1
13505 { \exp_args:No \__str_range:nnn { \tl_to_str:n {#1} } }
13506 \cs_new:Npn \__str_range:nnn #1#2#3
13507 {
13508     \exp_after:wN \__str_range:w
13509     \int_value:w \__str_count:n {#1} \exp_after:wN ;
13510     \int_value:w \int_eval:n { (#2) - 1 } \exp_after:wN ;
13511     \int_value:w \int_eval:n {#3} ;
13512     #1 \s__str_stop
13513 }
13514 \cs_new:Npn \__str_range:w #1; #2; #3;
13515 {
13516     \exp_args:Nf \__str_range:nnw
13517     { \__str_range_normalize:nn {#2} {#1} }
13518     { \__str_range_normalize:nn {#3} {#1} }
13519 }
13520 \cs_new:Npn \__str_range:nnw #1#2
13521 {
13522     \exp_after:wN \__str_collect_delimit_by_q_stop:w
13523     \int_value:w \int_eval:n { #2 - #1 } \exp_after:wN ;
13524     \exp:w \__str_skip_exp_end:w #1 ;
13525 }

```

(End definition for \str_range:Nnn and others. These functions are documented on page 128.)

__str_range_normalize:nn This function converts an $\langle index \rangle$ argument into an explicit position in the string (a result of 0 denoting “out of bounds”). Expects two explicit integer arguments: the $\langle index \rangle$ #1 and the string count #2. If #1 is negative, replace it by #1 + #2 + 1, then limit to the range [0, #2].

```

13526 \cs_new:Npn \__str_range_normalize:nn #1#2

```

```

13527 {
13528   \int_eval:n
13529   {
13530     \if_int_compare:w #1 < \c_zero_int
13531     \if_int_compare:w #1 < -#2 \exp_stop_f:
13532       0
13533     \else:
13534       #1 + #2 + 1
13535     \fi:
13536   \else:
13537     \if_int_compare:w #1 < #2 \exp_stop_f:
13538       #1
13539     \else:
13540       #2
13541     \fi:
13542   \fi:
13543 }
13544 }

```

(End definition for _str_range_normalize:nn.)

```

\_str_collect_delimit_by_q_stop:w
\_str_collect_loop:wn
  \_str_collect_loop:wnNNNNNNN
  \_str_collect_end:wn
\_str_collect_end:nnnnnnnnnw

```

Collects $\max(\#1, 0)$ characters, and removes everything else until `\s__str_stop`. This is somewhat similar to `_str_skip_exp_end:w`, but accepts integer expression arguments. This time we can only grab 7 characters at a time. At the end, we use an `\if_case:w` trick again, so that the 8 first arguments of `_str_collect_end:nnnnnnnnnw` are some `\or:`, followed by an `\fi:`, followed by `#1` characters from the input stream. Simply leaving this in the input stream closes the conditional properly and the `\or:` disappear.

```

13545 \cs_new:Npn \_str_collect_delimit_by_q_stop:w #1;
13546 { \_str_collect_loop:wn #1 ; { } }
13547 \cs_new:Npn \_str_collect_loop:wn #1 ;
13548 {
13549   \if_int_compare:w #1 > 7 \exp_stop_f:
13550   \exp_after:wN \_str_collect_loop:wnNNNNNNN
13551   \else:
13552     \exp_after:wN \_str_collect_end:wn
13553   \fi:
13554   #1 ;
13555 }
13556 \cs_new:Npn \_str_collect_loop:wnNNNNNNN #1; #2 #3#4#5#6#7#8#9
13557 {
13558   \exp_after:wN \_str_collect_loop:wn
13559   \int_value:w \int_eval:n { #1 - 7 } ;
13560   { #2 #3#4#5#6#7#8#9 }
13561 }
13562 \cs_new:Npn \_str_collect_end:wn #1 ;
13563 {
13564   \exp_after:wN \_str_collect_end:nnnnnnnnnw
13565   \if_case:w \if_int_compare:w #1 > \c_zero_int
13566     #1 \else: 0 \fi: \exp_stop_f:
13567   \or: \or: \or: \or: \or: \or: \fi:
13568 }
13569 \cs_new:Npn \_str_collect_end:nnnnnnnnnw #1#2#3#4#5#6#7#8 #9 \s__str_stop
13570 { #1#2#3#4#5#6#7#8 }

```

(End definition for _str_collect_delimit_by_q_stop:w and others.)

53.7 Counting characters

`\str_count_spaces:N` To speed up this function, we grab and discard 9 space-delimited arguments in each iteration of the loop. The loop stops when the last argument is one of the trailing `X⟨number⟩`, and that `⟨number⟩` is added to the sum of 9 that precedes, to adjust the result.

```

13571 \cs_new:Npn \str_count_spaces:N
13572   { \exp_args:No \str_count_spaces:n }
13573 \cs_generate_variant:Nn \str_count_spaces:N { c }
13574 \cs_new:Npn \str_count_spaces:n #1
13575   {
13576     \int_eval:n
13577     {
13578       \exp_after:wN \__str_count_spaces_loop:w
13579       \tl_to_str:n {#1} ~
13580       X 7 ~ X 6 ~ X 5 ~ X 4 ~ X 3 ~ X 2 ~ X 1 ~ X 0 ~ X -1 ~
13581       \s__str_stop
13582     }
13583   }
13584 \cs_new:Npn \__str_count_spaces_loop:w #1~#2~#3~#4~#5~#6~#7~#8~#9~
13585   {
13586     \if_meaning:w X #9
13587     \__str_use_i_delimit_by_s_stop:nw
13588     \fi:
13589     9 + \__str_count_spaces_loop:w
13590   }

```

(End definition for `\str_count_spaces:N`, `\str_count_spaces:n`, and `__str_count_spaces_loop:w`. These functions are documented on page 126.)

`\str_count:N` To count characters in a string we could first escape all spaces using `__kernel_str_to_other:n`, then pass the result to `\tl_count:n`. However, the escaping step would be quadratic in the number of characters in the string, and we can do better. Namely, sum the number of spaces (`\str_count_spaces:n`) and the result of `\tl_count:n`, which ignores spaces. Since strings tend to be longer than token lists, we use specialized functions to count characters ignoring spaces. Namely, loop, grabbing 9 non-space characters at each step, and end as soon as we reach one of the 9 trailing items. The internal function `__str_count:n`, used in `\str_item:nn` and `\str_range:nnn`, is similar to `\str_count_ignore_spaces:n` but expects its argument to already be a string or a string with spaces escaped.

```

13591 \cs_new:Npn \str_count:N { \exp_args:No \str_count:n }
13592 \cs_generate_variant:Nn \str_count:N { c }
13593 \cs_new:Npn \str_count:n #1
13594   {
13595     \__str_count_aux:n
13596     {
13597       \str_count_spaces:n {#1}
13598       + \exp_after:wN \__str_count_loop:NNNNNNNNN \tl_to_str:n {#1}
13599     }
13600   }
13601 \cs_new:Npn \__str_count:n #1
13602   {
13603     \__str_count_aux:n

```



```

13604     { \_str_count_loop:NNNNNNNN #1 }
13605   }
13606 \cs_new:Npn \str_count_ignore_spaces:n #1
13607 {
13608   \_str_count_aux:n
13609   { \exp_after:wN \_str_count_loop:NNNNNNNN \tl_to_str:n {#1} }
13610 }
13611 \cs_new:Npn \_str_count_aux:n #1
13612 {
13613   \int_eval:n
13614   {
13615     #1
13616     { X 8 } { X 7 } { X 6 }
13617     { X 5 } { X 4 } { X 3 }
13618     { X 2 } { X 1 } { X 0 }
13619     \s__str_stop
13620   }
13621 }
13622 \cs_new:Npn \_str_count_loop:NNNNNNNN #1#2#3#4#5#6#7#8#9
13623 {
13624   \if_meaning:w X #9
13625     \exp_after:wN \_str_use_none_delimit_by_s_stop:w
13626   \fi:
13627   9 + \_str_count_loop:NNNNNNNN
13628 }

```

(End definition for `\str_count:N` and others. These functions are documented on page 126.)

53.8 The first character in a string

`\str_head:N` The `_ignore_spaces` variant applies `\tl_to_str:n` then grabs the first item, thus skipping spaces. As usual, `\str_head:N` expands its argument and hands it to `\str_head:n`.
`\str_head:c` To circumvent the fact that \TeX skips spaces when grabbing undelimited macro parameters, `_str_head:w` takes an argument delimited by a space. If `#1` starts with a non-space character, `_str_use_i_delimit_by_s_stop:nw` leaves that in the input stream. On the other hand, if `#1` starts with a space, the `_str_head:w` takes an empty argument, and the single (initially braced) space in the definition of `_str_head:w` makes its way to the output. Finally, for an empty argument, the (braced) empty brace group in the definition of `\str_head:n` gives an empty result after passing through `_str_use_i_delimit_by_s_stop:nw`.

```

13629 \cs_new:Npn \str_head:N { \exp_args:No \str_head:n }
13630 \cs_generate_variant:Nn \str_head:N { c }
13631 \cs_new:Npn \str_head:n #1
13632 {
13633   \exp_after:wN \_str_head:w
13634   \tl_to_str:n {#1}
13635   { { } } ~ \s__str_stop
13636 }
13637 \cs_new:Npn \_str_head:w #1 ~ %
13638 { \_str_use_i_delimit_by_s_stop:nw #1 { ~ } }
13639 \cs_new:Npn \str_head_ignore_spaces:n #1
13640 {

```

```

13641     \exp_after:wN \__str_use_i_delimit_by_s_stop:nw
13642     \tl_to_str:n {#1} { } \s__str_stop
13643 }

```

(End definition for `\str_head:N` and others. These functions are documented on page 127.)

`\str_tail:N` Getting the tail is a little bit more convoluted than the head of a string. We hit the front of the string with `\reverse_if:N \if_charcode:w \scan_stop:.` This removes the first character, and necessarily makes the test true, since the character cannot match `\scan_stop:.` The auxiliary function then inserts the required `\fi:` to close the conditional, and leaves the tail of the string in the input stream. The details are such that an empty string has an empty tail (this requires in particular that the end-marker `X` be unexpandable and not a control sequence). The `_ignore_spaces` is rather simpler: after converting the input to a string, `__str_tail_auxii:w` removes one undelimited argument and leaves everything else until an end-marker `\s__str_mark`. One can check that an empty (or blank) string yields an empty tail.

```

13644 \cs_new:Npn \str_tail:N { \exp_args:No \str_tail:n }
13645 \cs_generate_variant:Nn \str_tail:N { c }
13646 \cs_new:Npn \str_tail:n #1
13647 {
13648     \exp_after:wN \__str_tail_auxi:w
13649     \reverse_if:N \if_charcode:w
13650         \scan_stop: \tl_to_str:n {#1} X X \s__str_stop
13651 }
13652 \cs_new:Npn \__str_tail_auxi:w #1 X #2 \s__str_stop { \fi: #1 }
13653 \cs_new:Npn \str_tail_ignore_spaces:n #1
13654 {
13655     \exp_after:wN \__str_tail_auxii:w
13656     \tl_to_str:n {#1} \s__str_mark \s__str_mark \s__str_stop
13657 }
13658 \cs_new:Npn \__str_tail_auxii:w #1 #2 \s__str_mark #3 \s__str_stop { #2 }

```

(End definition for `\str_tail:N` and others. These functions are documented on page 127.)

53.9 String manipulation

`\str_foldcase:n` Case changing for programmatic reasons is done by first detokenizing input then doing a simple loop that only has to worry about spaces and everything else. The output is detokenized to allow data sharing with text-based case changing.

```

\str_foldcase:V
\str_lowercase:n
\str_lowercase:f
\str_uppercase:n
\str_uppercase:f
\__str_change_case:nn
\__str_change_case_aux:nn
\__str_change_case_result:n
\__str_change_case_output:nw
\__str_change_case_output:fw
\__str_change_case_end:nw
\__str_change_case_loop:nw
\__str_change_case_space:n
\__str_change_case_char:nN
13659 \cs_new:Npn \str_foldcase:n #1 { \__str_change_case:nn {#1} { fold } }
13660 \cs_new:Npn \str_lowercase:n #1 { \__str_change_case:nn {#1} { lower } }
13661 \cs_new:Npn \str_uppercase:n #1 { \__str_change_case:nn {#1} { upper } }
13662 \cs_generate_variant:Nn \str_foldcase:n { V }
13663 \cs_generate_variant:Nn \str_lowercase:n { f }
13664 \cs_generate_variant:Nn \str_uppercase:n { f }
13665 \cs_new:Npn \__str_change_case:nn #1
13666 {
13667     \exp_after:wN \__str_change_case_aux:nn \exp_after:wN
13668         { \tl_to_str:n {#1} }
13669 }
13670 \cs_new:Npn \__str_change_case_aux:nn #1#2
13671 {

```

```

13672     \__str_change_case_loop:nw {#2} #1 \q__str_recursion_tail \q__str_recursion_stop
13673     \__str_change_case_result:n { }
13674 }
13675 \cs_new:Npn \__str_change_case_output:nw #1#2 \__str_change_case_result:n #3
13676 { #2 \__str_change_case_result:n { #3 #1 } }
13677 \cs_generate_variant:Nn \__str_change_case_output:nw { f }
13678 \cs_new:Npn \__str_change_case_end:wn #1 \__str_change_case_result:n #2
13679 { \tl_to_str:n {#2} }
13680 \cs_new:Npn \__str_change_case_loop:nw #1#2 \q__str_recursion_stop
13681 {
13682     \tl_if_head_is_space:nTF {#2}
13683     { \__str_change_case_space:n }
13684     { \__str_change_case_char:nN }
13685     {#1} #2 \q__str_recursion_stop
13686 }
13687 \exp_last_unbraced:NNNN
13688 \cs_new:Npn \__str_change_case_space:n #1 \c_space_tl
13689 {
13690     \__str_change_case_output:nw { ~ }
13691     \__str_change_case_loop:nw {#1}
13692 }
13693 \cs_new:Npn \__str_change_case_char:nN #1#2
13694 {
13695     \__str_if_recursion_tail_stop_do:Nn #2
13696     { \__str_change_case_end:wn }
13697     \__str_change_case_output:fw
13698     { \use:c { char_str_ #1 case:N } #2 }
13699     \__str_change_case_loop:nw {#1}
13700 }

```

(End definition for `\str_foldcase:n` and others. These functions are documented on page 131.)

<code>\c_ampersand_str</code> <code>\c_atsign_str</code> <code>\c_backslash_str</code> <code>\c_left_brace_str</code> <code>\c_right_brace_str</code> <code>\c_circumflex_str</code> <code>\c_colon_str</code> <code>\c_dollar_str</code> <code>\c_hash_str</code> <code>\c_percent_str</code> <code>\c_tilde_str</code> <code>\c_underscore_str</code> <code>\c_zero_str</code>	<p>For all of those strings, use <code>\cs_to_str:N</code> to get characters with the correct category code without worries</p> <pre> 13701 \str_const:Nx \c_ampersand_str { \cs_to_str:N \& } 13702 \str_const:Nx \c_atsign_str { \cs_to_str:N \@ } 13703 \str_const:Nx \c_backslash_str { \cs_to_str:N \ } 13704 \str_const:Nx \c_left_brace_str { \cs_to_str:N \{ } 13705 \str_const:Nx \c_right_brace_str { \cs_to_str:N \} } 13706 \str_const:Nx \c_circumflex_str { \cs_to_str:N \^ } 13707 \str_const:Nx \c_colon_str { \cs_to_str:N \: } 13708 \str_const:Nx \c_dollar_str { \cs_to_str:N \\$ } 13709 \str_const:Nx \c_hash_str { \cs_to_str:N \# } 13710 \str_const:Nx \c_percent_str { \cs_to_str:N \% } 13711 \str_const:Nx \c_tilde_str { \cs_to_str:N \~ } 13712 \str_const:Nx \c_underscore_str { \cs_to_str:N _ } 13713 \str_const:Nx \c_zero_str { 0 } </pre>
--	--

(End definition for `\c_ampersand_str` and others. These variables are documented on page 132.)

<code>\l_tmpa_str</code> <code>\l_tmpb_str</code> <code>\g_tmpa_str</code> <code>\g_tmpb_str</code>	<p>Scratch strings.</p> <pre> 13714 \str_new:N \l_tmpa_str 13715 \str_new:N \l_tmpb_str 13716 \str_new:N \g_tmpa_str 13717 \str_new:N \g_tmpb_str </pre>
--	--

(End definition for `\l_tmpa_str` and others. These variables are documented on page 132.)

53.10 Viewing strings

```

\str_show:n Displays a string on the terminal.
\str_show:N 13718 \cs_new_eq:NN \str_show:n \tl_show:n
\str_show:c 13719 \cs_new_protected:Npn \str_show:N #1
\str_log:n 13720 {
\str_log:N 13721 \__kernel_chk_tl_type:NnnT #1 { str } { \tl_to_str:N #1 }
\str_log:c 13722 { \tl_show:N #1 }
13723 }
13724 \cs_generate_variant:Nn \str_show:N { c }
13725 \cs_new_eq:NN \str_log:n \tl_log:n
13726 \cs_new_protected:Npn \str_log:N #1
13727 {
13728 \__kernel_chk_tl_type:NnnT #1 { str } { \tl_to_str:N #1 }
13729 { \tl_log:N #1 }
13730 }
13731 \cs_generate_variant:Nn \str_log:N { c }

```

(End definition for `\str_show:n` and others. These functions are documented on page 131.)

```
13732 </package>
```

Chapter 54

l3str-convert implementation

```
13733 <*package>
```

```
13734 <@@=str>
```

54.1 Helpers

54.1.1 Variables and constants

```
    \__str_tmp:w Internal scratch space for some functions.  
\l__str_internal_tl 13735 \cs_new_protected:Npn \__str_tmp:w { }  
13736 \tl_new:N \l__str_internal_tl
```

(End definition for __str_tmp:w and \l__str_internal_tl.)

```
\g__str_result_tl The \g__str_result_tl variable is used to hold the result of various internal string  
operations (mostly conversions) which are typically performed in a group. The variable  
is global so that it remains defined outside the group, to be assigned to a user-provided  
variable.
```

```
13737 \tl_new:N \g__str_result_tl
```

(End definition for \g__str_result_tl.)

```
\c__str_replacement_char_int When converting, invalid bytes are replaced by the Unicode replacement character  
"FFFD.
```

```
13738 \int_const:Nn \c__str_replacement_char_int { "FFFD }
```

(End definition for \c__str_replacement_char_int.)

```
\c__str_max_byte_int The maximal byte number.  
13739 \int_const:Nn \c__str_max_byte_int { 255 }
```

(End definition for \c__str_max_byte_int.)

```
\s__str Internal scan marks.
```

```
13740 \scan_new:N \s__str
```

(End definition for \s__str.)

`\q__str_nil` Internal quarks.

```
13741 \quark_new:N \q__str_nil
```

(End definition for `\q__str_nil`.)

`\g__str_alias_prop` To avoid needing one file per encoding/escaping alias, we keep track of those in a property list.

```
13742 \prop_new:N \g__str_alias_prop
13743 \prop_gput:Nnn \g__str_alias_prop { latin1 } { iso88591 }
13744 \prop_gput:Nnn \g__str_alias_prop { latin2 } { iso88592 }
13745 \prop_gput:Nnn \g__str_alias_prop { latin3 } { iso88593 }
13746 \prop_gput:Nnn \g__str_alias_prop { latin4 } { iso88594 }
13747 \prop_gput:Nnn \g__str_alias_prop { latin5 } { iso88599 }
13748 \prop_gput:Nnn \g__str_alias_prop { latin6 } { iso885910 }
13749 \prop_gput:Nnn \g__str_alias_prop { latin7 } { iso885913 }
13750 \prop_gput:Nnn \g__str_alias_prop { latin8 } { iso885914 }
13751 \prop_gput:Nnn \g__str_alias_prop { latin9 } { iso885915 }
13752 \prop_gput:Nnn \g__str_alias_prop { latin10 } { iso885916 }
13753 \prop_gput:Nnn \g__str_alias_prop { utf16le } { utf16 }
13754 \prop_gput:Nnn \g__str_alias_prop { utf16be } { utf16 }
13755 \prop_gput:Nnn \g__str_alias_prop { utf32le } { utf32 }
13756 \prop_gput:Nnn \g__str_alias_prop { utf32be } { utf32 }
13757 \prop_gput:Nnn \g__str_alias_prop { hexadecimal } { hex }
13758 \bool_lazy_any:nTF
13759 {
13760   \sys_if_engine luatex_p:
13761   \sys_if_engine xetex_p:
13762 }
13763 {
13764   \prop_gput:Nnn \g__str_alias_prop { default } { }
13765 }
13766 {
13767   \prop_gput:Nnn \g__str_alias_prop { default } { utf8 }
13768 }
```

(End definition for `\g__str_alias_prop`.)

`\g__str_error_bool` In conversion functions with a built-in conditional, errors are not reported directly to the user, but the information is collected in this boolean, used at the end to decide on which branch of the conditional to take.

```
13769 \bool_new:N \g__str_error_bool
```

(End definition for `\g__str_error_bool`.)

str_byte Conversions from one *<encoding>*/*<escaping>* pair to another are done within x-expanding assignments. Errors are signalled by raising the relevant flag.

```
13770 \flag_new:n { str_byte }
13771 \flag_new:n { str_error }
```

(End definition for `str_byte` and `str_error`. These variables are documented on page ??.)

54.2 String conditionals

`_str_if_contains_char:NnT` `_str_if_contains_char:nnTF {<token list>} <char>`
`_str_if_contains_char:NnTF` Expects the *<token list>* to be an *<other string>*: the caller is responsible for ensuring
`_str_if_contains_char:nnTF` that no (too-)special catcodes remain. Loop over the characters of the string, comparing
 `_str_if_contains_char_aux:nn` character codes. The loop is broken if character codes match. Otherwise we return
 `_str_if_contains_char_aux:nN` “false”.
 `_str_if_contains_char_true:`

```

13772 \prg_new_conditional:Npnn \_str_if_contains_char:Nn #1#2 { T , TF }
13773 {
13774   \exp_after:wN \_str_if_contains_char_aux:nn \exp_after:wN {#1} {#2}
13775   { \prg_break:n { ? \fi: } }
13776   \prg_break_point:
13777   \prg_return_false:
13778 }
13779 \cs_new:Npn \_str_if_contains_char_aux:nn #1#2
13780 { \_str_if_contains_char_aux:nN {#2} #1 }
13781 \prg_new_conditional:Npnn \_str_if_contains_char:nn #1#2 { TF }
13782 {
13783   \_str_if_contains_char_aux:nN {#2} #1 { \prg_break:n { ? \fi: } }
13784   \prg_break_point:
13785   \prg_return_false:
13786 }
13787 \cs_new:Npn \_str_if_contains_char_aux:nN #1#2
13788 {
13789   \if_charcode:w #1 #2
13790     \exp_after:wN \_str_if_contains_char_true:
13791     \fi:
13792     \_str_if_contains_char_aux:nN {#1}
13793 }
13794 \cs_new:Npn \_str_if_contains_char_true:
13795 { \prg_break:n { \prg_return_true: \use_none:n } }

```

(End definition for `_str_if_contains_char:NnT` and others.)

`_str_octal_use:NnTF` `_str_octal_use:NnTF <token> {<true code>} {<false code>}`
 If the *<token>* is an octal digit, it is left in the input stream, *followed* by the *<true code>*. Otherwise, the *<false code>* is left in the input stream.

T_EXhackers note: This function will fail if the escape character is an octal digit. We are thus careful to set the escape character to a known value before using it. T_EX dutifully detects

octal digits for us: if #1 is an octal digit, then the right-hand side of the comparison is '1#1, greater than 1. Otherwise, the right-hand side stops as '1, and the conditional takes the false branch.

```

13796 \prg_new_conditional:Npnn \_str_octal_use:Nn #1 { TF }
13797 {
13798   \if_int_compare:w 1 < '1 \token_to_str:N #1 \exp_stop_f:
13799   #1 \prg_return_true:
13800   \else:
13801     \prg_return_false:
13802   \fi:
13803 }

```

(End definition for `_str_octal_use:NnTF`.)

`__str_hexadecimal_use:N`TF TeX detects uppercase hexadecimal digits for us (see `__str_octal_use:N`TF), but not the lowercase letters, which we need to detect and replace by their uppercase counterpart.

```

13804 \prg_new_conditional:Npnn \__str_hexadecimal_use:N #1 { TF }
13805 {
13806   \if_int_compare:w 1 < "1 \token_to_str:N #1 \exp_stop_f:
13807     #1 \prg_return_true:
13808   \else:
13809     \if_case:w \int_eval:n { \exp_after:wN ' \token_to_str:N #1 - 'a }
13810       A
13811     \or: B
13812     \or: C
13813     \or: D
13814     \or: E
13815     \or: F
13816   \else:
13817     \prg_return_false:
13818     \exp_after:wN \use_none:n
13819   \fi:
13820   \prg_return_true:
13821 \fi:
13822 }
```

(End definition for `__str_hexadecimal_use:N`TF.)

54.3 Conversions

54.3.1 Producing one byte or character

`\c__str_byte_0_tl` For each integer N in the range $[0, 255]$, we create a constant token list which holds three character tokens with category code other: the character with character code N , followed by the representation of N as two hexadecimal digits. The value -1 is given a default token list which ensures that later functions give an empty result for the input -1 .

```

13823 \group_begin:
13824   \__kernel_tl_set:Nx \l__str_internal_tl { \tl_to_str:n { 0123456789ABCDEF } }
13825   \tl_map_inline:Nn \l__str_internal_tl
13826   {
13827     \tl_map_inline:Nn \l__str_internal_tl
13828     {
13829       \tl_const:cx { c__str_byte_ \int_eval:n {"#1##1} _tl }
13830       { \char_generate:nn { "#1##1 } { 12 } #1 ##1 }
13831     }
13832   }
13833 \group_end:
13834 \tl_const:cn { c__str_byte_-1_tl } { { } \use_none:n { } }
```

(End definition for `\c__str_byte_0_tl` and others.)

`__str_output_byte:n` Those functions must be used carefully: feeding them a value outside the range $[-1, 255]$ will attempt to use the undefined token list variable `\c__str_byte_⟨number⟩_tl`. Assuming that the argument is in the right range, we expand the corresponding token list, and pick either the byte (first token) or the hexadecimal representations (second and third tokens). The value -1 produces an empty result in both cases.


```

13835 \cs_new:Npn \__str_output_byte:n #1
13836 { \__str_output_byte:w #1 \__str_output_end: }
13837 \cs_new:Npn \__str_output_byte:w
13838 {
13839   \exp_after:wN \exp_after:wN
13840   \exp_after:wN \use_i:nnn
13841   \cs:w c__str_byte_ \int_eval:w
13842 }
13843 \cs_new:Npn \__str_output_hexadecimal:n #1
13844 {
13845   \exp_after:wN \exp_after:wN
13846   \exp_after:wN \use_none:n
13847   \cs:w c__str_byte_ \int_eval:n {#1} _tl \cs_end:
13848 }
13849 \cs_new:Npn \__str_output_end:
13850 { \scan_stop: _tl \cs_end: }

```

(End definition for __str_output_byte:n and others.)

__str_output_byte_pair_be:n Convert a number in the range [0,65535] to a pair of bytes, either big-endian or little-endian.
 __str_output_byte_pair_le:n
 __str_output_byte_pair:nnN

```

13851 \cs_new:Npn \__str_output_byte_pair_be:n #1
13852 {
13853   \exp_args:Nf \__str_output_byte_pair:nnN
13854   { \int_div_truncate:nn { #1 } { "100 } } {#1} \use:nn
13855 }
13856 \cs_new:Npn \__str_output_byte_pair_le:n #1
13857 {
13858   \exp_args:Nf \__str_output_byte_pair:nnN
13859   { \int_div_truncate:nn { #1 } { "100 } } {#1} \use_ii_i:nn
13860 }
13861 \cs_new:Npn \__str_output_byte_pair:nnN #1#2#3
13862 {
13863   #3
13864   { \__str_output_byte:n { #1 } }
13865   { \__str_output_byte:n { #2 - #1 * "100 } }
13866 }

```

(End definition for __str_output_byte_pair_be:n, __str_output_byte_pair_le:n, and __str_output_byte_pair:nnN.)

54.3.2 Mapping functions for conversions

__str_convert_gmap:N This maps the function #1 over all characters in \g__str_result_tl, which should be a byte string in most cases, sometimes a native string.
 __str_convert_gmap_loop:NN

```

13867 \cs_new_protected:Npn \__str_convert_gmap:N #1
13868 {
13869   \__kernel_tl_gset:Nx \g__str_result_tl
13870   {
13871     \exp_after:wN \__str_convert_gmap_loop:NN
13872     \exp_after:wN #1
13873     \g__str_result_tl { ? \prg_break: }
13874     \prg_break_point:
13875   }

```

```

13876 }
13877 \cs_new:Npn \__str_convert_gmap_loop:NN #1#2
13878 {
13879     \use_none:n #2
13880     #1#2
13881     \__str_convert_gmap_loop:NN #1
13882 }

```

(End definition for `__str_convert_gmap:N` and `__str_convert_gmap_loop:NN`.)

```

\__str_convert_gmap_internal:N
\__str_convert_gmap_internal_loop:Nw

```

This maps the function #1 over all character codes in `\g__str_result_tl`, which must be in the internal representation.

```

13883 \cs_new_protected:Npn \__str_convert_gmap_internal:N #1
13884 {
13885     \__kernel_tl_gset:Nx \g__str_result_tl
13886     {
13887         \exp_after:wN \__str_convert_gmap_internal_loop:Nww
13888         \exp_after:wN #1
13889         \g__str_result_tl \s__str \s__str_stop \prg_break: \s__str
13890         \prg_break_point:
13891     }
13892 }
13893 \cs_new:Npn \__str_convert_gmap_internal_loop:Nww #1 #2 \s__str #3 \s__str
13894 {
13895     \__str_use_none_delimit_by_s_stop:w #3 \s__str_stop
13896     #1 {#3}
13897     \__str_convert_gmap_internal_loop:Nww #1
13898 }

```

(End definition for `__str_convert_gmap_internal:N` and `__str_convert_gmap_internal_loop:Nw`.)

54.3.3 Error-reporting during conversion

```

\__str_if_flag_error:nnx
\__str_if_flag_no_error:nnx

```

When converting using the function `\str_set_convert:Nnnn`, errors should be reported to the user after each step in the conversion. Errors are signalled by raising some flag (typically `@@_error`), so here we test that flag: if it is raised, give the user an error, otherwise remove the arguments. On the other hand, in the conditional functions `\str_set_convert:NnnnTF`, errors should be suppressed. This is done by changing `__str_if_flag_error:nnx` into `__str_if_flag_no_error:nnx` locally.

```

13899 \cs_new_protected:Npn \__str_if_flag_error:nnx #1
13900 {
13901     \flag_if_raised:nTF {#1}
13902     { \msg_error:nnx { str } }
13903     { \use_none:nn }
13904 }
13905 \cs_new_protected:Npn \__str_if_flag_no_error:nnx #1#2#3
13906 { \flag_if_raised:nT {#1} { \bool_gset_true:N \g__str_error_bool } }

```

(End definition for `__str_if_flag_error:nnx` and `__str_if_flag_no_error:nnx`.)

```

\__str_if_flag_times:nT

```

At the end of each conversion step, we raise all relevant errors as one error message, built on the fly. The height of each flag indicates how many times a given error was encountered. This function prints #2 followed by the number of occurrences of an error if it occurred, nothing otherwise.

```

13907 \cs_new:Npn \__str_if_flag_times:nT #1#2
13908 { \flag_if_raised:nT {#1} { #2~(x \flag_height:n {#1} ) } }

```

(End definition for `__str_if_flag_times:nT`.)

54.3.4 Framework for conversions

Most functions in this module expect to be working with “native” strings. Strings can also be stored as bytes, in one of many encodings, for instance UTF8. The bytes themselves can be expressed in various ways in terms of T_EX tokens, for instance as pairs of hexadecimal digits. The questions of going from arbitrary Unicode code points to bytes, and from bytes to tokens are mostly independent.

Conversions are done in four steps:

- “unescape” produces a string of bytes;
- “decode” takes in a string of bytes, and converts it to a list of Unicode characters in an internal representation, with items of the form

$\langle bytes \rangle \backslash s_str \langle Unicode\ code\ point \rangle \backslash s_str$

where we have collected the $\langle bytes \rangle$ which combined to form this particular Unicode character, and the $\langle Unicode\ code\ point \rangle$ is in the range [0, "10FFFF].

- “encode” encodes the internal list of code points as a byte string in the new encoding;
- “escape” escapes bytes as requested.

The process is modified in case one of the encoding is empty (or the conversion function has been set equal to the empty encoding because it was not found): then the unescape or escape step is ignored, and the decode or encode steps work on tokens instead of bytes. Otherwise, each step must ensure that it passes a correct byte string or internal string to the next step.

```

\str_set_convert:Nnnn
\str_gset_convert:Nnnn
\str_set_convert:NnnnTF
\str_gset_convert:NnnnTF
\__str_convert:nNNnnn

```

The input string is stored in `\g__str_result_tl`, then we: unescape and decode; encode and escape; exit the group and store the result in the user’s variable. The various conversion functions all act on `\g__str_result_tl`. Errors are silenced for the conditional functions by redefining `__str_if_flag_error:nxx` locally.

```

13909 \cs_new_protected:Npn \str_set_convert:Nnnn
13910 { \__str_convert:nNNnnn { } \tl_set_eq:NN }
13911 \cs_new_protected:Npn \str_gset_convert:Nnnn
13912 { \__str_convert:nNNnnn { } \tl_gset_eq:NN }
13913 \prg_new_protected_conditional:Npnn
13914 \str_set_convert:Nnnn #1#2#3#4 { T , F , TF }
13915 {
13916   \bool_gset_false:N \g__str_error_bool
13917   \__str_convert:nNNnnn
13918   { \cs_set_eq:NN \__str_if_flag_error:nxx \__str_if_flag_no_error:nxx }
13919   \tl_set_eq:NN #1 {#2} {#3} {#4}
13920   \bool_if:NTF \g__str_error_bool \prg_return_false: \prg_return_true:
13921 }
13922 \prg_new_protected_conditional:Npnn
13923 \str_gset_convert:Nnnn #1#2#3#4 { T , F , TF }

```

```

13924 {
13925     \bool_gset_false:N \g__str_error_bool
13926     \__str_convert:nNNnnn
13927     { \cs_set_eq:NN \__str_if_flag_error:nmx \__str_if_flag_no_error:nmx }
13928     \tl_gset_eq:NN #1 {#2} {#3} {#4}
13929     \bool_if:NTF \g__str_error_bool \prg_return_false: \prg_return_true:
13930 }
13931 \cs_new_protected:Npn \__str_convert:nNNnnn #1#2#3#4#5#6
13932 {
13933     \group_begin:
13934     #1
13935     \__kernel_tl_gset:Nx \g__str_result_tl { \__kernel_str_to_other_fast:n {#4} }
13936     \exp_after:wN \__str_convert:wwwnn
13937     \tl_to_str:n {#5} /// \s__str_stop
13938     { decode } { unescape }
13939     \prg_do_nothing:
13940     \__str_convert_decode_:
13941     \exp_after:wN \__str_convert:wwwnn
13942     \tl_to_str:n {#6} /// \s__str_stop
13943     { encode } { escape }
13944     \use_ii_i:nn
13945     \__str_convert_encode_:
13946     \group_end:
13947     #2 #3 \g__str_result_tl
13948 }

```

(End definition for `\str_set_convert:Nnnn` and others. These functions are documented on page 135.)

`__str_convert:wwwnn`
`__str_convert:NNnNN`

The task of `__str_convert:wwwnn` is to split $\langle encoding \rangle / \langle escaping \rangle$ pairs into their components, #1 and #2. Calls to `__str_convert:nnn` ensure that the corresponding conversion functions are defined. The third auxiliary does the main work.

- #1 is the encoding conversion function;
- #2 is the escaping function;
- #3 is the escaping name for use in an error message;
- #4 is `\prg_do_nothing:` for unescaping/decoding, and `\use_ii_i:nn` for encoding/escaping;
- #5 is the default encoding function (either “decode” or “encode”), for which there should be no escaping.

Let us ignore the native encoding for a second. In the unescaping/decoding phase, we want to do #2#1 in this order, and in the encoding/escaping phase, the order should be reversed: #4#2#1 does exactly that. If one of the encodings is the default (native), then the escaping should be ignored, with an error if any was given, and only the encoding, #1, should be performed.

```

13949 \cs_new_protected:Npn \__str_convert:wwwnn
13950     #1 / #2 // #3 \s__str_stop #4#5
13951 {
13952     \__str_convert:nnn {enc} {#4} {#1}
13953     \__str_convert:nnn {esc} {#5} {#2}
13954     \exp_args:Ncc \__str_convert:NNnNN

```

```

13955         { __str_convert_#4_#1: } { __str_convert_#5_#2: } {#2}
13956     }
13957 \cs_new_protected:Npn \__str_convert:NNnNN #1#2#3#4#5
13958 {
13959     \if_meaning:w #1 #5
13960         \tl_if_empty:nF {#3}
13961         { \msg_error:nxx { str } { native-escaping } {#3} }
13962         #1
13963     \else:
13964         #4 #2 #1
13965     \fi:
13966 }

```

(End definition for `__str_convert:wwwnn` and `__str_convert:NNnNN`.)

`__str_convert:nnn` The arguments of `__str_convert:nnn` are: `enc` or `esc`, used to build filenames, the type of the conversion (unescape, decode, encode, escape), and the encoding or escaping name. If the function is already defined, no need to do anything. Otherwise, filter out all non-alphanumerics in the name, and lowercase it. Feed that, and the same three arguments, to `__str_convert:nnnn`. The task is then to make sure that the conversion function `#3_#1` corresponding to the type `#3` and filtered name `#1` is defined, then set our initial conversion function `#3_#4` equal to that.

How do we get the `#3_#1` conversion to be defined if it isn't? Two main cases.

First, if `#1` is a key in `\g__str_alias_prop`, then the value `\l__str_internal_tl` tells us what file to load. Loading is skipped if the file was already read, *i.e.*, if the conversion command based on `\l__str_internal_tl` already exists. Otherwise, try to load the file; if that fails, there is an error, use the default empty name instead.

Second, `#1` may be absent from the property list. The `\cs_if_exist:cF` test is automatically false, and we search for a file defining the encoding or escaping `#1` (this should allow third-party `.def` files). If the file is not found, there is an error, use the default empty name instead.

In all cases, the conversion based on `\l__str_internal_tl` is defined, so we can set the `#3_#1` function equal to that. In some cases (*e.g.*, `utf16be`), the `#3_#1` function is actually defined within the file we just loaded, and it is different from the `\l__str_internal_tl`-based function: we mustn't clobber that different definition.

```

13967 \cs_new_protected:Npn \__str_convert:nnn #1#2#3
13968 {
13969     \cs_if_exist:cF { __str_convert_#2_#3: }
13970     {
13971         \exp_args:Nx \__str_convert:nnnn
13972         { \__str_convert_lowercase_alphanum:n {#3} }
13973         {#1} {#2} {#3}
13974     }
13975 }
13976 \cs_new_protected:Npn \__str_convert:nnnn #1#2#3#4
13977 {
13978     \cs_if_exist:cF { __str_convert_#3_#1: }
13979     {
13980         \prop_get:NnNF \g__str_alias_prop {#1} \l__str_internal_tl
13981         { \tl_set:Nn \l__str_internal_tl {#1} }
13982         \cs_if_exist:cF { __str_convert_#3_ \l__str_internal_tl : }
13983         {
13984             \file_if_exist:nTF { l3str-#2- \l__str_internal_tl .def }

```

```

13985         {
13986         \group_begin:
13987         \cctab_select:N \c_code_cctab
13988         \file_input:n { l3str-#2- \l__str_internal_tl .def }
13989         \group_end:
13990         }
13991         {
13992         \tl_clear:N \l__str_internal_tl
13993         \msg_error:nxxx { str } { unknown-#2 } {#4} {#1}
13994         }
13995     }
13996     \cs_if_exist:cF { __str_convert_#3_#1: }
13997     {
13998         \cs_gset_eq:cc { __str_convert_#3_#1: }
13999         { __str_convert_#3_ \l__str_internal_tl : }
14000     }
14001 }
14002 \cs_gset_eq:cc { __str_convert_#3_#4: } { __str_convert_#3_#1: }
14003 }

```

(End definition for `__str_convert:nnn` and `__str_convert:nnnn`.)

`__str_convert_lowercase_alphanum:n`
`__str_convert_lowercase_alphanum_loop:N`

This function keeps only letters and digits, with upper case letters converted to lower case.

```

14004 \cs_new:Npn \__str_convert_lowercase_alphanum:n #1
14005 {
14006     \exp_after:wN \__str_convert_lowercase_alphanum_loop:N
14007     \tl_to_str:n {#1} { ? \prg_break: }
14008     \prg_break_point:
14009 }
14010 \cs_new:Npn \__str_convert_lowercase_alphanum_loop:N #1
14011 {
14012     \use_none:n #1
14013     \if_int_compare:w '#1 > 'Z \exp_stop_f:
14014     \if_int_compare:w '#1 > 'z \exp_stop_f: \else:
14015         \if_int_compare:w '#1 < 'a \exp_stop_f: \else:
14016             #1
14017         \fi:
14018     \fi:
14019     \else:
14020         \if_int_compare:w '#1 < 'A \exp_stop_f:
14021         \if_int_compare:w 1 < 1#1 \exp_stop_f:
14022             #1
14023         \fi:
14024     \else:
14025         \__str_output_byte:n { '#1 + 'a - 'A }
14026     \fi:
14027     \fi:
14028     \__str_convert_lowercase_alphanum_loop:N
14029 }

```

(End definition for `__str_convert_lowercase_alphanum:n` and `__str_convert_lowercase_alphanum_loop:N`.)

54.3.5 Byte unescape and escape

Strings of bytes may need to be stored in auxiliary files in safe “escaping” formats. Each such escaping is only loaded as needed. By default, on input any non-byte is filtered out, while the output simply consists in letting bytes through.

In the case of 8-bit engines, every character is a byte. For Unicode-aware engines, test the character code; non-bytes cause us to raise the flag `str_byte`. Spaces have already been given the correct category code when this function is called.

```

14030 \bool_lazy_any:nTF
14031 {
14032   \sys_if_engine_luatex_p:
14033   \sys_if_engine_xetex_p:
14034 }
14035 {
14036   \cs_new:Npn \__str_filter_bytes:n #1
14037   {
14038     \__str_filter_bytes_aux:N #1
14039     { ? \prg_break: }
14040     \prg_break_point:
14041   }
14042   \cs_new:Npn \__str_filter_bytes_aux:N #1
14043   {
14044     \use_none:n #1
14045     \if_int_compare:w '#1 < 256 \exp_stop_f:
14046       #1
14047     \else:
14048       \flag_raise:n { str_byte }
14049     \fi:
14050     \__str_filter_bytes_aux:N
14051   }
14052 }
14053 { \cs_new_eq:NN \__str_filter_bytes:n \use:n }
```

(End definition for `__str_filter_bytes:n` and `__str_filter_bytes_aux:N`.)

The simplest unescaping method removes non-bytes from `\g__str_result_tl`.

```

14054 \bool_lazy_any:nTF
14055 {
14056   \sys_if_engine_luatex_p:
14057   \sys_if_engine_xetex_p:
14058 }
14059 {
14060   \cs_new_protected:Npn \__str_convert_unescape_:
14061   {
14062     \flag_clear:n { str_byte }
14063     \__kernel_tl_gset:Nx \g__str_result_tl
14064     { \exp_args:No \__str_filter_bytes:n \g__str_result_tl }
14065     \__str_if_flag_error:nx { str_byte } { non-byte } { bytes }
14066   }
14067 }
14068 { \cs_new_protected:Npn \__str_convert_unescape_: { } }
14069 \cs_new_eq:NN \__str_convert_unescape_bytes: \__str_convert_unescape_:
```

(End definition for `__str_convert_unescape_:` and `__str_convert_unescape_bytes:.`)

`__str_convert_escape_:` The simplest form of escape leaves the bytes from the previous step of the conversion unchanged.
`__str_convert_escape_bytes:`

```
14070 \cs_new_protected:Npn \__str_convert_escape_: { }
14071 \cs_new_eq:NN \__str_convert_escape_bytes: \__str_convert_escape_:
```

(End definition for `__str_convert_escape_:` and `__str_convert_escape_bytes:`.)

54.3.6 Native strings

`__str_convert_decode_:` Convert each character to its character code, one at a time.
`__str_decode_native_char:N`

```
14072 \cs_new_protected:Npn \__str_convert_decode_:
14073 { \__str_convert_gmap:N \__str_decode_native_char:N }
14074 \cs_new:Npn \__str_decode_native_char:N #1
14075 { #1 \s__str \int_value:w '#1 \s__str }
```

(End definition for `__str_convert_decode_:` and `__str_decode_native_char:N`.)

`__str_convert_encode_:` The conversion from an internal string to native character tokens basically maps `\char_generate:nn` through the code-points, but in non-Unicode-aware engines we use a fall-back character `?` rather than nothing when given a character code outside `[0,255]`. We detect the presence of bad characters using a flag and only produce a single error after the `x`-expanding assignment.
`__str_encode_native_char:n`

```
14076 \bool_lazy_any:nTF
14077 {
14078   \sys_if_engine luatex_p:
14079   \sys_if_engine xetex_p:
14080 }
14081 {
14082   \cs_new_protected:Npn \__str_convert_encode_:
14083   { \__str_convert_gmap_internal:N \__str_encode_native_char:n }
14084   \cs_new:Npn \__str_encode_native_char:n #1
14085   { \char_generate:nn {#1} {12} }
14086 }
14087 {
14088   \cs_new_protected:Npn \__str_convert_encode_:
14089   {
14090     \flag_clear:n { str_error }
14091     \__str_convert_gmap_internal:N \__str_encode_native_char:n
14092     \__str_if_flag_error:nmx { str_error }
14093     { native-overflow } { }
14094   }
14095   \cs_new:Npn \__str_encode_native_char:n #1
14096   {
14097     \if_int_compare:w #1 > \c__str_max_byte_int
14098     \flag_raise:n { str_error }
14099     ?
14100   \else:
14101     \char_generate:nn {#1} {12}
14102   \fi:
14103   }
14104   \msg_new:nnnn { str } { native-overflow }
14105   { Character-code-too-large-for-this-engine. }
14106   {
```



```

14107         This~engine~only~support~8-bit~characters:~
14108         valid~character~codes~are~in~the~range~[0,255].~
14109         To~manipulate~arbitrary~Unicode,~use~LuaTeX~or~XeTeX.
14110     }
14111 }

```

(End definition for `__str_convert_encode_:` and `__str_encode_native_char:n`.)

54.3.7 `clist`

`__str_convert_decode_clist:` Convert each integer to the internal form. We first turn `\g__str_result_tl` into a `clist` variable, as this avoids problems with leading or trailing commas.

```

14112 \cs_new_protected:Npn \__str_convert_decode_clist:
14113 {
14114     \clist_gset:No \g__str_result_tl \g__str_result_tl
14115     \__kernel_tl_gset:Nx \g__str_result_tl
14116     {
14117         \exp_args:No \clist_map_function:nN
14118         \g__str_result_tl \__str_decode_clist_char:n
14119     }
14120 }
14121 \cs_new:Npn \__str_decode_clist_char:n #1
14122 { #1 \s__str \int_eval:n {#1} \s__str }

```

(End definition for `__str_convert_decode_clist:` and `__str_decode_clist_char:n`.)

`__str_convert_encode_clist:` Convert the internal list of character codes to a comma-list of character codes. The first line produces a comma-list with a leading comma, removed in the next step (this also works in the empty case, since `\tl_tail:N` does not trigger an error in this case).

```

14123 \cs_new_protected:Npn \__str_convert_encode_clist:
14124 {
14125     \__str_convert_gmap_internal:N \__str_encode_clist_char:n
14126     \__kernel_tl_gset:Nx \g__str_result_tl { \tl_tail:N \g__str_result_tl }
14127 }
14128 \cs_new:Npn \__str_encode_clist_char:n #1 { , #1 }

```

(End definition for `__str_convert_encode_clist:` and `__str_encode_clist_char:n`.)

54.3.8 8-bit encodings

It is not clear in what situations 8-bit encodings are used, hence it is not clear what should be optimized. The current approach is reasonably efficient to convert long strings, and it scales well when using many different encodings.

The data needed to support a given 8-bit encoding is stored in a file that consists of a single function call

```

\__str_declare_eight_bit_encoding:nnnn {⟨name⟩} {⟨modulo⟩} {⟨mapping⟩}
{⟨missing⟩}

```

This declares the encoding `⟨name⟩` to map bytes to Unicode characters according to the `⟨mapping⟩`, and map those bytes which are not mentioned in the `⟨mapping⟩` either to the replacement character (if they appear in `⟨missing⟩`), or to themselves. The `⟨mapping⟩` argument is a token list of pairs `{⟨byte⟩} {⟨Unicode⟩}` expressed in uppercase hexadecimal notation. The `⟨missing⟩` argument is a token list of `{⟨byte⟩}`. Every `⟨byte⟩` which does

not appear in the *<mapping>* nor the *<missing>* lists maps to itself in Unicode, so for instance the `latin1` encoding has empty *<mapping>* and *<missing>* lists. The *<modulo>* is a (decimal) integer between 256 and 558 inclusive, modulo which all Unicode code points supported by the encodings must be different.

We use two integer arrays per encoding. When decoding we only use the `decode` integer array, with entry $n + 1$ (offset needed because integer array indices start at 1) equal to the Unicode code point that corresponds to the n -th byte in the encoding under consideration, or -1 if the given byte is invalid in this encoding. When encoding we use both arrays: upon seeing a code point n , we look up the entry $(1 \text{ plus } n) \text{ modulo } M$ in the `encode` array, which tells us the byte that might encode the given Unicode code point, then we check in the `decode` array that indeed this byte encodes the Unicode code point we want. Here, M is an encoding-dependent integer between 256 and 558 (it turns out), chosen so that among the Unicode code points that can be validly represented in the given encoding, no pair of code points have the same value modulo M .

Loop through both lists of bytes to fill in the `decode` integer array, then fill the `encode` array accordingly. For bytes that are invalid in the given encoding, store -1 in the `decode` array.

```

14129 \_str_declare_eight_bit_encoding:nnnn
14130 \_str_declare_eight_bit_aux:NNnnn
14131 \_str_declare_eight_bit_loop:Nnn
14132 \_str_declare_eight_bit_loop:Nn
14129 \cs_new_protected:Npn \_str_declare_eight_bit_encoding:nnnn #1
14130 {
14131   \tl_set:Nn \l__str_internal_tl {#1}
14132   \cs_new_protected:cpn { __str_convert_decode_#1: }
14133     { \_str_convert_decode_eight_bit:n {#1} }
14134   \cs_new_protected:cpn { __str_convert_encode_#1: }
14135     { \_str_convert_encode_eight_bit:n {#1} }
14136   \exp_args:Ncc \_str_declare_eight_bit_aux:NNnnn
14137     { g__str_decode_#1_intarray } { g__str_encode_#1_intarray }
14138 }
14139 \cs_new_protected:Npn \_str_declare_eight_bit_aux:NNnnn #1#2#3#4#5
14140 {
14141   \intarray_new:Nn #1 { 256 }
14142   \int_step_inline:nnn { 0 } { 255 }
14143     { \intarray_gset:Nnn #1 { 1 + ##1 } {##1} }
14144   \_str_declare_eight_bit_loop:Nnn #1
14145     #4 { \s__str_stop \prg_break: } { }
14146   \prg_break_point:
14147   \_str_declare_eight_bit_loop:Nn #1
14148     #5 { \s__str_stop \prg_break: }
14149   \prg_break_point:
14150   \intarray_new:Nn #2 {#3}
14151   \int_step_inline:nnn { 0 } { 255 }
14152     {
14153       \int_compare:nNf { \intarray_item:Nn #1 { 1 + ##1 } } = { -1 }
14154       {
14155         \intarray_gset:Nnn #2
14156           {
14157             1 +
14158             \int_mod:nn { \intarray_item:Nn #1 { 1 + ##1 } }
14159               { \intarray_count:N #2 }
14160           }
14161         {##1}
14162       }
14163     }

```

```

14164     }
14165 \cs_new_protected:Npn \__str_declare_eight_bit_loop:Nnn #1#2#3
14166 {
14167     \__str_use_none_delimit_by_s_stop:w #2 \s__str_stop
14168     \intarray_gset:Nnn #1 { 1 + "#2 } { "#3 }
14169     \__str_declare_eight_bit_loop:Nnn #1
14170 }
14171 \cs_new_protected:Npn \__str_declare_eight_bit_loop:Nn #1#2
14172 {
14173     \__str_use_none_delimit_by_s_stop:w #2 \s__str_stop
14174     \intarray_gset:Nnn #1 { 1 + "#2 } { -1 }
14175     \__str_declare_eight_bit_loop:Nn #1
14176 }

```

(End definition for __str_declare_eight_bit_encoding:nnnn and others.)

__str_convert_decode_eight_bit:n
 __str_decode_eight_bit_aux:n
 __str_decode_eight_bit_aux:Nn

The map from bytes to Unicode code points is in the `decode` array corresponding to the given encoding. Define `__str_tmp:w` and pass it successively all bytes in the string. It produces an internal representation with suitable `\s__str` inserted, and the corresponding code point is obtained by looking it up in the integer array. If the entry is `-1` then issue a replacement character and raise the flag indicating that there was an error.

```

14177 \cs_new_protected:Npn \__str_convert_decode_eight_bit:n #1
14178 {
14179     \cs_set:Npx \__str_tmp:w
14180     {
14181         \exp_not:N \__str_decode_eight_bit_aux:Nn
14182         \exp_not:c { g__str_decode_#1_intarray }
14183     }
14184     \flag_clear:n { str_error }
14185     \__str_convert_gmap:N \__str_tmp:w
14186     \__str_if_flag_error:nxx { str_error } { decode-8-bit } {#1}
14187 }
14188 \cs_new:Npn \__str_decode_eight_bit_aux:Nn #1#2
14189 {
14190     #2 \s__str
14191     \exp_args:Nf \__str_decode_eight_bit_aux:n
14192     { \intarray_item:Nn #1 { 1 + '#2 } }
14193     \s__str
14194 }
14195 \cs_new:Npn \__str_decode_eight_bit_aux:n #1
14196 {
14197     \if_int_compare:w #1 < \c_zero_int
14198         \flag_raise:n { str_error }
14199         \int_value:w \c__str_replacement_char_int
14200     \else:
14201         #1
14202     \fi:
14203 }

```

(End definition for __str_convert_decode_eight_bit:n, __str_decode_eight_bit_aux:n, and __str_decode_eight_bit_aux:Nn.)

__str_convert_encode_eight_bit:n
 __str_encode_eight_bit_aux:nnN
 __str_encode_eight_bit_aux:NNn

It is not practical to make an integer array with indices in the full Unicode range, so we work modulo some number, which is simply the size of the `encode` integer array for the

given encoding. This gives us a candidate byte for representing a given Unicode code point. Of course taking the modulo leads to collisions so we check in the `decode` array that the byte we got is indeed correct. Otherwise the Unicode code point we started from is simply not representable in the given encoding.

```

14204 \int_new:N \l__str_modulo_int
14205 \cs_new_protected:Npn \__str_convert_encode_eight_bit:n #1
14206 {
14207   \cs_set:Npx \__str_tmp:w
14208   {
14209     \exp_not:N \__str_encode_eight_bit_aux:NNn
14210     \exp_not:c { g__str_encode_#1_intarray }
14211     \exp_not:c { g__str_decode_#1_intarray }
14212   }
14213   \flag_clear:n { str_error }
14214   \__str_convert_gmap_internal:N \__str_tmp:w
14215   \__str_if_flag_error:nnx { str_error } { encode-8-bit } {#1}
14216 }
14217 \cs_new:Npn \__str_encode_eight_bit_aux:NNn #1#2#3
14218 {
14219   \exp_args:Nf \__str_encode_eight_bit_aux:nnN
14220   {
14221     \intarray_item:Nn #1
14222     { 1 + \int_mod:nn {#3} { \intarray_count:N #1 } }
14223   }
14224   {#3}
14225   #2
14226 }
14227 \cs_new:Npn \__str_encode_eight_bit_aux:nnN #1#2#3
14228 {
14229   \int_compare:nNnTF { \intarray_item:Nn #3 { 1 + #1 } } = {#2}
14230   { \__str_output_byte:n {#1} }
14231   { \flag_raise:n { str_error } }
14232 }

```

(End definition for `__str_convert_encode_eight_bit:n`, `__str_encode_eight_bit_aux:nnN`, and `__str_encode_eight_bit_aux:NNn`.)

54.4 Messages

General messages, and messages for the encodings and escapings loaded by default (“native”, and “bytes”).

```

14233 \msg_new:nnn { str } { unknown-esc }
14234 { Escaping-scheme~'#1'~(filtered:~'#2')~unknown. }
14235 \msg_new:nnn { str } { unknown-enc }
14236 { Encoding-scheme~'#1'~(filtered:~'#2')~unknown. }
14237 \msg_new:nnnn { str } { native-escaping }
14238 { The~'native'~encoding-scheme~does~not~support~any~escaping. }
14239 {
14240   Since~native~strings~do~not~consist~in~bytes,~
14241   none~of~the~escaping~methods~make~sense.~
14242   The~specified~escaping,~'~'#1',~will be ignored.
14243 }
14244 \msg_new:nnn { str } { file-not-found }

```

```
14245 { File~'l3str-#1.def'~not~found. }
```

Message used when the “bytes” unescaping fails because the string given to `\str_set_convert:Nnnn` contains a non-byte. This cannot happen for the -8-bit engines. Messages used for other escapings and encodings are defined in each definition file.

```
14246 \bool_lazy_any:nT
14247 {
14248   \sys_if_engine_luatex_p:
14249   \sys_if_engine_xetex_p:
14250 }
14251 {
14252   \msg_new:nnnn { str } { non-byte }
14253   { String~invalid~in~escaping~'#1':~it~may~only~contain~bytes. }
14254   {
14255     Some~characters~in~the~string~you~asked~to~convert~are~not~
14256     8-bit~characters.~Perhaps~the~string~is~a~'native'~Unicode~string?~
14257     If~it~is,~try~using\\
14258     \\
14259     \iow_indent:n
14260     {
14261       \iow_char:N\\str_set_convert:Nnnn \\
14262       \ \ <str-var>~\{~<string>~\}~\{~native~\}~\{~<target-encoding>~\}
14263     }
14264   }
14265 }
```

Those messages are used when converting to and from 8-bit encodings.

```
14266 \msg_new:nnnn { str } { decode-8-bit }
14267 { Invalid~string~in~encoding~'#1'. }
14268 {
14269   LaTeX~came~across~a~byte~which~is~not~defined~to~represent~
14270   any~character~in~the~encoding~'#1'.
14271 }
14272 \msg_new:nnnn { str } { encode-8-bit }
14273 { Unicode~string~cannot~be~converted~to~encoding~'#1'. }
14274 {
14275   The~encoding~'#1'~only~contains~a~subset~of~all~Unicode~characters.~
14276   LaTeX~was~asked~to~convert~a~string~to~that~encoding,~but~that~
14277   string~contains~a~character~that~'#1'~does~not~support.
14278 }
```

54.5 Escaping definitions

Several of those encodings are defined by the pdf file format. The following byte storage methods are defined:

- **bytes** (default), non-bytes are filtered out, and bytes are left untouched (this is defined by default);
- **hex** or **hexadecimal**, as per the pdfTeX primitive `\pdfescapehex`
- **name**, as per the pdfTeX primitive `\pdfescapename`
- **string**, as per the pdfTeX primitive `\pdfescapestring`
- **url**, as per the percent encoding of urls.

54.5.1 Unescape methods

`__str_convert_unescape_hex:` Take chars two by two, and interpret each pair as the hexadecimal code for a byte.
`__str_unescape_hex_auxi:N` Anything else than hexadecimal digits is ignored, raising the flag. A string which contains
`__str_unescape_hex_auxii:N` an odd number of hexadecimal digits gets 0 appended to it: this is equivalent to appending a 0 in all cases, and dropping it if it is alone.

```

14279 \cs_new_protected:Npn \__str_convert_unescape_hex:
14280 {
14281   \group_begin:
14282     \flag_clear:n { str_error }
14283     \int_set:Nn \tex_escapechar:D { 92 }
14284     \__kernel_tl_gset:Nx \g__str_result_tl
14285     {
14286       \__str_output_byte:w "
14287       \exp_last_unbraced:Nf \__str_unescape_hex_auxi:N
14288       { \tl_to_str:N \g__str_result_tl }
14289       0 { ? 0 - 1 \prg_break: }
14290       \prg_break_point:
14291       \__str_output_end:
14292     }
14293     \__str_if_flag_error:nmx { str_error } { unescape-hex } { }
14294   \group_end:
14295 }
14296 \cs_new:Npn \__str_unescape_hex_auxi:N #1
14297 {
14298   \use_none:n #1
14299   \__str_hexadecimal_use:NTF #1
14300   { \__str_unescape_hex_auxii:N }
14301   {
14302     \flag_raise:n { str_error }
14303     \__str_unescape_hex_auxi:N
14304   }
14305 }
14306 \cs_new:Npn \__str_unescape_hex_auxii:N #1
14307 {
14308   \use_none:n #1
14309   \__str_hexadecimal_use:NTF #1
14310   {
14311     \__str_output_end:
14312     \__str_output_byte:w " \__str_unescape_hex_auxi:N
14313   }
14314   {
14315     \flag_raise:n { str_error }
14316     \__str_unescape_hex_auxii:N
14317   }
14318 }
14319 \msg_new:nnnn { str } { unescape-hex }
14320 { String~invalid~in~escaping~'hex':~only~hexadecimal~digits~allowed. }
14321 {
14322   Some~characters~in~the~string~you~asked~to~convert~are~not~
14323   hexadecimal~digits~(0-9,~A-F,~a-f)~nor~spaces.
14324 }

```

(End definition for `__str_convert_unescape_hex:`, `__str_unescape_hex_auxi:N`, and `__str_unescape_hex_auxii:N`.)

`__str_convert_unescape_name:` The `__str_convert_unescape_name:` function replaces each occurrence of # followed by two hexadecimal digits in `\g__str_result_tl` by the corresponding byte. The `url` function is identical, with escape character % instead of #. Thus we define the two together. The arguments of `__str_tmp:w` are the character code of # or % in hexadecimal, the name of the main function to define, and the name of the auxiliary which performs the loop.

The looping auxiliary #3 finds the next escape character, reads the following two characters, and tests them. The test `__str_hexadecimal_use:NTF` leaves the upper-case digit in the input stream, hence we surround the test with `__str_output_byte:w "` and `__str_output_end:.` If both characters are hexadecimal digits, they should be removed before looping: this is done by `\use_i:nnn`. If one of the characters is not a hexadecimal digit, then feed "#1 to `__str_output_byte:w` to produce the escape character, raise the flag, and call the looping function followed by the two characters (remove `\use_i:nnn`).

```

14325 \cs_set_protected:Npn \__str_tmp:w #1#2#3
14326 {
14327   \cs_new_protected:cpn { __str_convert_unescape_#2: }
14328   {
14329     \group_begin:
14330     \flag_clear:n { str_byte }
14331     \flag_clear:n { str_error }
14332     \int_set:Nn \tex_escapechar:D { 92 }
14333     \__kernel_tl_gset:Nx \g__str_result_tl
14334     {
14335       \exp_after:wN #3 \g__str_result_tl
14336       #1 ? { ? \prg_break: }
14337       \prg_break_point:
14338     }
14339     \__str_if_flag_error:nnx { str_byte } { non-byte } { #2 }
14340     \__str_if_flag_error:nnx { str_error } { unescape-#2 } { }
14341     \group_end:
14342   }
14343   \cs_new:Npn #3 ##1#1##2##3
14344   {
14345     \__str_filter_bytes:n {##1}
14346     \use_none:n ##3
14347     \__str_output_byte:w "
14348     \__str_hexadecimal_use:NTF ##2
14349     {
14350       \__str_hexadecimal_use:NTF ##3
14351       { }
14352       {
14353         \flag_raise:n { str_error }
14354         * 0 + ‘#1 \use_i:nn
14355       }
14356     }
14357     {
14358       \flag_raise:n { str_error }
14359       0 + ‘#1 \use_i:nn
14360     }
14361     \__str_output_end:
14362     \use_i:nnn #3 ##2##3
14363   }

```

```

14364 \msg_new:nnnn { str } { unescape-#2 }
14365 { String~invalid~in~escaping~'#2'. }
14366 {
14367   LaTeX~came~across~the~escape~character~'#1'~not~followed~by~
14368   two~hexadecimal~digits.~This~is~invalid~in~the~escaping~'#2'.
14369 }
14370 }
14371 \exp_after:wN \__str_tmp:w \c_hash_str { name }
14372 \__str_unescape_name_loop:wNN
14373 \exp_after:wN \__str_tmp:w \c_percent_str { url }
14374 \__str_unescape_url_loop:wNN

```

(End definition for __str_convert_unescape_name: and others.)

```

\__str_convert_unescape_string:
\__str_unescape_string_newlines:wN
\__str_unescape_string_loop:wNNN
\__str_unescape_string_repeat:NNNNNN

```

The **string** escaping is somewhat similar to the **name** and **url** escapings, with escape character \. The first step is to convert all three line endings, ^^J, ^^M, and ^^M^^J to the common ^^J, as per the PDF specification. This step cannot raise the flag.

Then the following escape sequences are decoded.

```

\n Line feed (10)
\r Carriage return (13)
\t Horizontal tab (9)
\b Backspace (8)
\f Form feed (12)
\ ( Left parenthesis
\) Right parenthesis
\\ Backslash

```

\ddd (backslash followed by 1 to 3 octal digits) Byte ddd (octal), subtracting 256 in case of overflow.

If followed by an end-of-line character, the backslash and the end-of-line are ignored. If followed by anything else, the backslash is ignored, raising the error flag.

```

14375 \group_begin:
14376 \char_set_catcode_other:N ^^J
14377 \char_set_catcode_other:N ^^M
14378 \cs_set_protected:Npn \__str_tmp:w #1
14379 {
14380   \cs_new_protected:Npn \__str_convert_unescape_string:
14381   {
14382     \group_begin:
14383     \flag_clear:n { str_byte }
14384     \flag_clear:n { str_error }
14385     \int_set:Nn \tex_escapechar:D { 92 }
14386     \__kernel_tl_gset:Nx \g__str_result_tl
14387     {
14388       \exp_after:wN \__str_unescape_string_newlines:wN
14389       \g__str_result_tl \prg_break: ^^M ?
14390       \prg_break_point:

```



```

14391     }
14392     \__kernel_tl_gset:Nx \g__str_result_tl
14393     {
14394         \exp_after:wN \__str_unescape_string_loop:wNNN
14395         \g__str_result_tl #1 ?? { ? \prg_break: }
14396         \prg_break_point:
14397     }
14398     \__str_if_flag_error:nmx { str_byte } { non-byte } { string }
14399     \__str_if_flag_error:nmx { str_error } { unescape-string } { }
14400 \group_end:
14401 }
14402 }
14403 \exp_args:No \__str_tmp:w { \c_backslash_str }
14404 \exp_last_unbraced:NNNNo
14405 \cs_new:Npn \__str_unescape_string_loop:wNNN #1 \c_backslash_str #2#3#4
14406 {
14407     \__str_filter_bytes:n {#1}
14408     \use_none:n #4
14409     \__str_output_byte:w '
14410     \__str_octal_use:NTF #2
14411     {
14412         \__str_octal_use:NTF #3
14413         {
14414             \__str_octal_use:NTF #4
14415             {
14416                 \if_int_compare:w #2 > 3 \exp_stop_f:
14417                 - 256
14418                 \fi:
14419                 \__str_unescape_string_repeat:NNNNNN
14420             }
14421             { \__str_unescape_string_repeat:NNNNNN ? }
14422         }
14423         { \__str_unescape_string_repeat:NNNNNN ?? }
14424     }
14425     {
14426         \str_case_e:nnF {#2}
14427         {
14428             { \c_backslash_str } { 134 }
14429             { ( } { 50 }
14430             { ) } { 51 }
14431             { r } { 15 }
14432             { f } { 14 }
14433             { n } { 12 }
14434             { t } { 11 }
14435             { b } { 10 }
14436             { ^^J } { 0 - 1 }
14437         }
14438         {
14439             \flag_raise:n { str_error }
14440             0 - 1 \use_i:nn
14441         }
14442     }
14443 \__str_output_end:
14444 \use_i:nn \__str_unescape_string_loop:wNNN #2#3#4

```

```

14445     }
14446     \cs_new:Npn \__str_unescape_string_repeat:NNNNNN #1#2#3#4#5#6
14447     { \__str_output_end: \__str_unescape_string_loop:wNNN }
14448     \cs_new:Npn \__str_unescape_string_newlines:wN #1 ^M #2
14449     {
14450       #1
14451       \if_charcode:w ^^J #2 \else: ^^J \fi:
14452       \__str_unescape_string_newlines:wN #2
14453     }
14454     \msg_new:nnnn { str } { unescape-string }
14455     { String~invalid~in~escaping~'string'. }
14456     {
14457       LaTeX~came~across~an~escape~character~'\c_backslash_str'~
14458       not~followed~by~any~of:~'n',~'r',~'t',~'b',~'f',~'(',~')',~
14459       '\c_backslash_str',~one~to~three~octal~digits,~or~the~end~
14460       of~a~line.
14461     }
14462     \group_end:

```

(End definition for `__str_convert_unescape_string:` and others.)

54.5.2 Escape methods

Currently, none of the escape methods can lead to errors, assuming that their input is made out of bytes.

`__str_convert_escape_hex:` Loop and convert each byte to hexadecimal.

```

\__str_escape_hex_char:N
14463 \cs_new_protected:Npn \__str_convert_escape_hex:
14464 { \__str_convert_gmap:N \__str_escape_hex_char:N }
14465 \cs_new:Npn \__str_escape_hex_char:N #1
14466 { \__str_output_hexadecimal:n { '#1' } }

```

(End definition for `__str_convert_escape_hex:` and `__str_escape_hex_char:N`.)

`__str_convert_escape_name:` For each byte, test whether it should be output as is, or be “hash-encoded”. Roughly, bytes outside the range [“2A”, “7E”] are hash-encoded. We keep two lists of exceptions: `__str_escape_name_char:n` characters in `\c__str_escape_name_not_str` are not hash-encoded, and characters in `\c__str_escape_name_str` are encoded.

```

\__str_escape_name_char:n
\__str_if_escape_name:nTF
\c__str_escape_name_str
\c__str_escape_name_not_str
14467 \str_const:Nn \c__str_escape_name_not_str { ! " $ & ' } %$
14468 \str_const:Nn \c__str_escape_name_str { { } / < > [ ] }
14469 \cs_new_protected:Npn \__str_convert_escape_name:
14470 { \__str_convert_gmap:N \__str_escape_name_char:n }
14471 \cs_new:Npn \__str_escape_name_char:n #1
14472 {
14473   \__str_if_escape_name:nTF {#1} {#1}
14474   { \c_hash_str \__str_output_hexadecimal:n { '#1' } }
14475 }
14476 \prg_new_conditional:Npnn \__str_if_escape_name:n #1 { TF }
14477 {
14478   \if_int_compare:w '#1 < "2A \exp_stop_f:
14479   \__str_if_contains_char:NnTF \c__str_escape_name_not_str {#1}
14480   \prg_return_true: \prg_return_false:
14481   \else:
14482     \if_int_compare:w '#1 > "7E \exp_stop_f:

```

```

14483         \prg_return_false:
14484     \else:
14485         \__str_if_contains_char:NnTF \c__str_escape_name_str {#1}
14486         \prg_return_false: \prg_return_true:
14487     \fi:
14488 \fi:
14489 }

```

(End definition for __str_convert_escape_name: and others.)

__str_convert_escape_string: Any character below (and including) space, and any character above (and including) del, are converted to octal. One backslash is added before each parenthesis and backslash.

```

\__str_escape_string_char:N
\__str_if_escape_string:N
\c__str_escape_string_str
14490 \str_const:Nx \c__str_escape_string_str
14491 { \c_backslash_str ( ) }
14492 \cs_new_protected:Npn \__str_convert_escape_string:
14493 { \__str_convert_gmap:N \__str_escape_string_char:N }
14494 \cs_new:Npn \__str_escape_string_char:N #1
14495 {
14496     \__str_if_escape_string:NTF #1
14497     {
14498         \__str_if_contains_char:NnT
14499         \c__str_escape_string_str {#1}
14500         { \c_backslash_str }
14501         #1
14502     }
14503     {
14504         \c_backslash_str
14505         \int_div_truncate:nn {'#1} {64}
14506         \int_mod:nn { \int_div_truncate:nn {'#1} { 8 } } { 8 }
14507         \int_mod:nn {'#1} { 8 }
14508     }
14509 }
14510 \prg_new_conditional:Npnn \__str_if_escape_string:N #1 { TF }
14511 {
14512     \if_int_compare:w '#1 < "21 \exp_stop_f:
14513     \prg_return_false:
14514     \else:
14515         \if_int_compare:w '#1 > "7E \exp_stop_f:
14516         \prg_return_false:
14517         \else:
14518             \prg_return_true:
14519         \fi:
14520     \fi:
14521 }

```

(End definition for __str_convert_escape_string: and others.)

__str_convert_escape_url: This function is similar to __str_convert_escape_name:, escaping different characters.

```

\__str_escape_url_char:n
\__str_if_escape_url:nTF
14522 \cs_new_protected:Npn \__str_convert_escape_url:
14523 { \__str_convert_gmap:N \__str_escape_url_char:n }
14524 \cs_new:Npn \__str_escape_url_char:n #1
14525 {
14526     \__str_if_escape_url:nTF {#1} {#1}
14527     { \c_percent_str \__str_output_hexadecimal:n { '#1 } }

```

```

14528     }
14529 \prg_new_conditional:Npnn \__str_if_escape_url:n #1 { TF }
14530 {
14531     \if_int_compare:w '#1 < "41 \exp_stop_f:
14532     \__str_if_contains_char:nnTF { "-.<> } {#1}
14533     \prg_return_true: \prg_return_false:
14534     \else:
14535     \if_int_compare:w '#1 > "7E \exp_stop_f:
14536     \prg_return_false:
14537     \else:
14538     \__str_if_contains_char:nnTF { [ ] } {#1}
14539     \prg_return_false: \prg_return_true:
14540     \fi:
14541 \fi:
14542 }

```

(End definition for `__str_convert_escape_url:`, `__str_escape_url_char:n`, and `__str_if_escape_url:nTF`.)

54.6 Encoding definitions

The `native` encoding is automatically defined. Other encodings are loaded as needed. The following encodings are supported:

- UTF-8;
- UTF-16, big-, little-endian, or with byte order mark;
- UTF-32, big-, little-endian, or with byte order mark;
- the ISO 8859 code pages, numbered from 1 to 16, skipping the inexistent ISO 8859-12.

54.6.1 utf-8 support

`__str_convert_encode_utf8:` Loop through the internal string, and convert each character to its UTF-8 representation. The representation is built from the right-most (least significant) byte to the left-most (most significant) byte. Continuation bytes are in the range [128, 191], taking 64 different values, hence we roughly want to express the character code in base 64, shifting the first digit in the representation by some number depending on how many continuation bytes there are. In the range [0, 127], output the corresponding byte directly. In the range [128, 2047], output the remainder modulo 64, plus 128 as a continuation byte, then output the quotient (which is in the range [0, 31]), shifted by 192. In the next range, [2048, 65535], split the character code into residue and quotient modulo 64, output the residue as a first continuation byte, then repeat; this leaves us with a quotient in the range [0, 15], which we output shifted by 224. The last range, [65536, 1114111], follows the same pattern: once we realize that dividing twice by 64 leaves us with a number larger than 15, we repeat, producing a last continuation byte, and offset the quotient by 240 for the leading byte.

How is that implemented? `__str_encode_utf_vii_loop:wwnnw` takes successive quotients as its first argument, the quotient from the previous step as its second argument (except in step 1), the bound for quotients that trigger one more step or not, and finally the offset used if this step should produce the leading byte. Leading bytes can be in

the ranges [0, 127], [192, 223], [224, 239], and [240, 247] (really, that last limit should be 244 because Unicode stops at the code point 1114111). At each step, if the quotient #1 is less than the limit #3 for that range, output the leading byte (#1 shifted by #4) and stop. Otherwise, we need one more step: use the quotient of #1 by 64, and #1 as arguments for the looping auxiliary, and output the continuation byte corresponding to the remainder #2 - 64#1 + 128. The bizarre construction - 1 + 0 * removes the spurious initial continuation byte (better methods welcome).

```

14543 \cs_new_protected:cpn { __str_convert_encode_utf8: }
14544 { __str_convert_gmap_internal:N __str_encode_utf_viii_char:n }
14545 \cs_new:Npn __str_encode_utf_viii_char:n #1
14546 {
14547   __str_encode_utf_viii_loop:wnnnw #1 ; - 1 + 0 * ;
14548   { 128 } { 0 }
14549   { 32 } { 192 }
14550   { 16 } { 224 }
14551   { 8 } { 240 }
14552   \s__str_stop
14553 }
14554 \cs_new:Npn __str_encode_utf_viii_loop:wnnnw #1; #2; #3#4 #5 \s__str_stop
14555 {
14556   \if_int_compare:w #1 < #3 \exp_stop_f:
14557   __str_output_byte:n { #1 + #4 }
14558   \exp_after:wN __str_use_none_delimit_by_s_stop:w
14559   \fi:
14560   \exp_after:wN __str_encode_utf_viii_loop:wnnnw
14561   \int_value:w \int_div_truncate:nn {#1} {64} ; #1 ;
14562   #5 \s__str_stop
14563   __str_output_byte:n { #2 - 64 * ( #1 - 2 ) }
14564 }

```

(End definition for __str_convert_encode_utf8:, __str_encode_utf_viii_char:n, and __str_encode_utf_viii_loop:wnnnw.)

\l__str_missing_flag
 \l__str_extra_flag
 \l__str_overlong_flag
 \l__str_overflow_flag

When decoding a string that is purportedly in the UTF-8 encoding, four different errors can occur, signalled by a specific flag for each (we define those flags using \flag_clear_new:n rather than \flag_new:n, because they are shared with other encoding definition files).

- “Missing continuation byte”: a leading byte is not followed by the right number of continuation bytes.
- “Extra continuation byte”: a continuation byte appears where it was not expected, *i.e.*, not after an appropriate leading byte.
- “Overlong”: a Unicode character is expressed using more bytes than necessary, for instance, "C0"80 for the code point 0, instead of a single null byte.
- “Overflow”: this occurs when decoding produces Unicode code points greater than 1114111.

We only raise one L^AT_EX3 error message, combining all the errors which occurred. In the short message, the leading comma must be removed to get a grammatically correct sentence. In the long text, first remind the user what a correct UTF-8 string should look like, then add error-specific information.

```

14565 \flag_clear_new:n { str_missing }
14566 \flag_clear_new:n { str_extra }
14567 \flag_clear_new:n { str_overlong }
14568 \flag_clear_new:n { str_overflow }
14569 \msg_new:nnnn { str } { utf8-decode }
14570 {
14571   Invalid-UTF-8-string:
14572   \exp_last_unbraced:Nf \use_none:n
14573   {
14574     \__str_if_flag_times:nT { str_missing } { ,~missing-continuation-byte }
14575     \__str_if_flag_times:nT { str_extra } { ,~extra-continuation-byte }
14576     \__str_if_flag_times:nT { str_overlong } { ,~overlong-form }
14577     \__str_if_flag_times:nT { str_overflow } { ,~code-point-too-large }
14578   }
14579   .
14580 }
14581 {
14582   In-the-UTF-8-encoding,~each-Unicode-character-consists-in-
14583   1-to-4-bytes,~with-the-following-bit-pattern: \\
14584   \iow_indent:n
14585   {
14586     Code-point~\ \ \ <~128:~0xxxxxxx \\
14587     Code-point~\ \ \ <~2048:~110xxxxx~10xxxxxx \\
14588     Code-point~\ \ \ <~65536:~1110xxxx~10xxxxxx~10xxxxxx \\
14589     Code-point~ \ <~1114112:~11110xxx~10xxxxxx~10xxxxxx~10xxxxxx \\
14590   }
14591   Bytes-of-the-form-10xxxxxx-are-called-continuation-bytes.
14592   \flag_if_raised:nT { str_missing }
14593   {
14594     \\\
14595     A-leading-byte-(in-the-range-[192,255])~was-not-followed-by-
14596     the-appropriate-number-of-continuation-bytes.
14597   }
14598   \flag_if_raised:nT { str_extra }
14599   {
14600     \\\
14601     LaTeX-came-across-a-continuation-byte-when-it-was-not-expected.
14602   }
14603   \flag_if_raised:nT { str_overlong }
14604   {
14605     \\\
14606     Every-Unicode-code-point-must-be-expressed-in-the-shortest-
14607     possible-form.~For-instance,~'0xC0'~'0x83'~is-not-a-valid-
14608     representation-for-the-code-point~3.
14609   }
14610   \flag_if_raised:nT { str_overflow }
14611   {
14612     \\\
14613     Unicode-limits-code-points-to-the-range-[0,1114111].
14614   }
14615 }
14616 \prop_gput:Nnn \g_msg_module_name_prop { str } { LaTeX3 }
14617 \prop_gput:Nnn \g_msg_module_type_prop { str } { }

```

(End definition for \l__str_missing_flag and others.)

```

\__str_convert_decode_utf8:
  \_str_decode_utf_viii_start:N
  \_str_decode_utf_viii_continuation:wwN
  \_str_decode_utf_viii_aux:wNnnwN
  \_str_decode_utf_viii_overflow:w
\__str_decode_utf_viii_end:

```

Decoding is significantly harder than encoding. As before, lower some flags, which are tested at the end (in bulk, to trigger at most one L^AT_EX3 error, as explained above). We expect successive multi-byte sequences of the form *⟨start byte⟩ ⟨continuation bytes⟩*. The `_start` auxiliary tests the first byte:

- [0, "7F]: the byte stands alone, and is converted to its own character code;
- ["80, "BF]: unexpected continuation byte, raise the appropriate flag, and convert that byte to the replacement character "FFFD;
- ["C0, "FF]: this byte should be followed by some continuation byte(s).

In the first two cases, `\use_none_delimit_by_q_stop:w` removes data that only the third case requires, namely the limits of ranges of Unicode characters which can be expressed with 1, 2, 3, or 4 bytes.

We can now concentrate on the multi-byte case and the `_continuation` auxiliary. We expect `#3` to be in the range ["80, "BF]. The test for this goes as follows: if the character code is less than "80, we compare it to −"C0, yielding `false`; otherwise to "C0, yielding `true` in the range ["80, "BF] and `false` otherwise. If we find that the byte is not a continuation range, stop the current slew of bytes, output the replacement character, and continue parsing with the `_start` auxiliary, starting at the byte we just tested. Once we know that the byte is a continuation byte, leave it behind us in the input stream, compute what code point the bytes read so far would produce, and feed that number to the `_aux` function.

The `_aux` function tests whether we should look for more continuation bytes or not. If the number it receives as `#1` is less than the maximum `#4` for the current range, then we are done: check for an overlong representation by comparing `#1` with the maximum `#3` for the previous range. Otherwise, we call the `_continuation` auxiliary again, after shifting the “current code point” by `#4` (maximum from the range we just checked).

Two additional tests are needed: if we reach the end of the list of range maxima and we are still not done, then we are faced with an overflow. Clean up, and again insert the code point "FFFD for the replacement character. Also, every time we read a byte, we need to check whether we reached the end of the string. In a correct UTF-8 string, this happens automatically when the `_start` auxiliary leaves its first argument in the input stream: the end-marker begins with `\prg_break:`, which ends the loop. On the other hand, if the end is reached when looking for a continuation byte, the `\use_none:n #3` construction removes the first token from the end-marker, and leaves the `_end` auxiliary, which raises the appropriate error flag before ending the mapping.

```

14618 \cs_new_protected:cpn { __str_convert_decode_utf8: }
14619 {
14620   \flag_clear:n { str_error }
14621   \flag_clear:n { str_missing }
14622   \flag_clear:n { str_extra }
14623   \flag_clear:n { str_overlong }
14624   \flag_clear:n { str_overflow }
14625   \__kernel_tl_gset:Nx \g__str_result_tl
14626   {
14627     \exp_after:wN \_str_decode_utf_viii_start:N \g__str_result_tl
14628     { \prg_break: \_str_decode_utf_viii_end: }
14629     \prg_break_point:
14630   }
14631   \__str_if_flag_error:nxn { str_error } { utf8-decode } { }

```

```

14632 }
14633 \cs_new:Npn \__str_decode_utf_viii_start:N #1
14634 {
14635     #1
14636     \if_int_compare:w '#1 < "C0 \exp_stop_f:
14637         \s__str
14638         \if_int_compare:w '#1 < "80 \exp_stop_f:
14639             \int_value:w '#1
14640         \else:
14641             \flag_raise:n { str_extra }
14642             \flag_raise:n { str_error }
14643             \int_use:N \c__str_replacement_char_int
14644         \fi:
14645     \else:
14646         \exp_after:wN \__str_decode_utf_viii_continuation:wwN
14647         \int_value:w \int_eval:n { '#1 - "C0 } \exp_after:wN
14648     \fi:
14649     \s__str
14650     \__str_use_none_delimit_by_s_stop:w {"80} {"800} {"10000} {"110000} \s__str_stop
14651     \__str_decode_utf_viii_start:N
14652 }
14653 \cs_new:Npn \__str_decode_utf_viii_continuation:wwN
14654     #1 \s__str #2 \__str_decode_utf_viii_start:N #3
14655 {
14656     \use_none:n #3
14657     \if_int_compare:w '#3 <
14658         \if_int_compare:w '#3 < "80 \exp_stop_f: - \fi:
14659         "C0 \exp_stop_f:
14660     #3
14661     \exp_after:wN \__str_decode_utf_viii_aux:wNnnwN
14662     \int_value:w \int_eval:n { #1 * "40 + '#3 - "80 } \exp_after:wN
14663     \else:
14664         \s__str
14665         \flag_raise:n { str_missing }
14666         \flag_raise:n { str_error }
14667         \int_use:N \c__str_replacement_char_int
14668     \fi:
14669     \s__str
14670     #2
14671     \__str_decode_utf_viii_start:N #3
14672 }
14673 \cs_new:Npn \__str_decode_utf_viii_aux:wNnnwN
14674     #1 \s__str #2#3#4 #5 \__str_decode_utf_viii_start:N #6
14675 {
14676     \if_int_compare:w #1 < #4 \exp_stop_f:
14677         \s__str
14678         \if_int_compare:w #1 < #3 \exp_stop_f:
14679             \flag_raise:n { str_overlong }
14680             \flag_raise:n { str_error }
14681             \int_use:N \c__str_replacement_char_int
14682         \else:
14683             #1
14684         \fi:
14685     \else:

```



```

14686     \if_meaning:w \s__str_stop #5
14687     \__str_decode_utf_viii_overflow:w #1
14688     \fi:
14689     \exp_after:wN \__str_decode_utf_viii_continuation:wwN
14690     \int_value:w \int_eval:n { #1 - #4 } \exp_after:wN
14691     \fi:
14692     \s__str
14693     #2 {#4} #5
14694     \__str_decode_utf_viii_start:N
14695   }
14696 \cs_new:Npn \__str_decode_utf_viii_overflow:w #1 \fi: #2 \fi:
14697 {
14698   \fi: \fi:
14699   \flag_raise:n { str_overflow }
14700   \flag_raise:n { str_error }
14701   \int_use:N \c__str_replacement_char_int
14702 }
14703 \cs_new:Npn \__str_decode_utf_viii_end:
14704 {
14705   \s__str
14706   \flag_raise:n { str_missing }
14707   \flag_raise:n { str_error }
14708   \int_use:N \c__str_replacement_char_int \s__str
14709   \prg_break:
14710 }

```

(End definition for `__str_convert_decode_utf8:` and others.)

54.6.2 utf-16 support

The definitions are done in a category code regime where the bytes 254 and 255 used by the byte order mark have catcode 12.

```

14711 \group_begin:
14712   \char_set_catcode_other:N ^^fe
14713   \char_set_catcode_other:N ^^ff

```

`__str_convert_encode_utf16:` When the endianness is not specified, it is big-endian by default, and we add a byte-order mark. Convert characters one by one in a loop, with different behaviours depending on the character code.

- `[0, "D7FF]`: converted to two bytes;
- `["D800, "DFFF]` are used as surrogates: they cannot be converted and are replaced by the replacement character;
- `["E000, "FFFF]`: converted to two bytes;
- `["10000, "10FFFF]`: converted to a pair of surrogates, each two bytes. The magic "D7C0 is "D800 – "10000/"400.

For the duration of this operation, `__str_tmp:w` is defined as a function to convert a number in the range `[0, "FFFF]` to a pair of bytes (either big endian or little endian), by feeding the quotient of the division of `#1` by "100, followed by `#1` to `__str_encode_utf_xvi_be:nn` or its `le` analog: those compute the remainder, and output two bytes for the quotient and remainder.

```

14714 \cs_new_protected:cpn { __str_convert_encode_utf16: }
14715 {
14716   \__str_encode_utf_xvi_aux:N \__str_output_byte_pair_be:n
14717   \tl_gput_left:Nx \g__str_result_tl { ^^fe ^^ff }
14718 }
14719 \cs_new_protected:cpn { __str_convert_encode_utf16be: }
14720 { \__str_encode_utf_xvi_aux:N \__str_output_byte_pair_be:n }
14721 \cs_new_protected:cpn { __str_convert_encode_utf16le: }
14722 { \__str_encode_utf_xvi_aux:N \__str_output_byte_pair_le:n }
14723 \cs_new_protected:Npn \__str_encode_utf_xvi_aux:N #1
14724 {
14725   \flag_clear:n { str_error }
14726   \cs_set_eq:NN \__str_tmp:w #1
14727   \__str_convert_gmap_internal:N \__str_encode_utf_xvi_char:n
14728   \__str_if_flag_error:nxx { str_error } { utf16-encode } { }
14729 }
14730 \cs_new:Npn \__str_encode_utf_xvi_char:n #1
14731 {
14732   \if_int_compare:w #1 < "D800 \exp_stop_f:
14733     \__str_tmp:w {#1}
14734   \else:
14735     \if_int_compare:w #1 < "10000 \exp_stop_f:
14736       \if_int_compare:w #1 < "E000 \exp_stop_f:
14737         \flag_raise:n { str_error }
14738         \__str_tmp:w { \c__str_replacement_char_int }
14739       \else:
14740         \__str_tmp:w {#1}
14741       \fi:
14742     \else:
14743       \exp_args:Nf \__str_tmp:w { \int_div_truncate:nn {#1} {"400} + "D7C0 }
14744       \exp_args:Nf \__str_tmp:w { \int_mod:nn {#1} {"400} + "DC00 }
14745     \fi:
14746   \fi:
14747 }

```

(End definition for __str_convert_encode_utf16: and others.)

\l__str_missing_flag When encoding a Unicode string to UTF-16, only one error can occur: code points in the range ["D800,"DFFF], corresponding to surrogates, cannot be encoded. We use the
 \l__str_extra_flag all-purpose flag @@_error to signal that error.
 \l__str_end_flag

When decoding a Unicode string which is purportedly in UTF-16, three errors can occur: a missing trail surrogate, an unexpected trail surrogate, and a string containing an odd number of bytes.

```

14748 \flag_clear_new:n { str_missing }
14749 \flag_clear_new:n { str_extra }
14750 \flag_clear_new:n { str_end }
14751 \msg_new:nnnn { str } { utf16-encode }
14752 { Unicode~string~cannot~be~expressed~in~UTF-16:~surrogate. }
14753 {
14754   Surrogate~code~points~(in~the~range~[U+D800,~U+DFFF])~
14755   can~be~expressed~in~the~UTF-8~and~UTF-32~encodings,~
14756   but~not~in~the~UTF-16~encoding.
14757 }
14758 \msg_new:nnnn { str } { utf16-decode }

```

```

14759 {
14760   Invalid-UTF-16-string:
14761   \exp_last_unbraced:Nf \use_none:n
14762   {
14763     \__str_if_flag_times:nT { str_missing } { ,~missing~trail~surrogate }
14764     \__str_if_flag_times:nT { str_extra } { ,~extra~trail~surrogate }
14765     \__str_if_flag_times:nT { str_end } { ,~odd~number~of~bytes }
14766   }
14767   .
14768 }
14769 {
14770   In~the~UTF-16~encoding,~each~Unicode~character~is~encoded~as~
14771   2~or~4~bytes: \\
14772   \iow_indent:n
14773   {
14774     Code~point~in~[U+0000,~U+D7FF]:~two~bytes \\
14775     Code~point~in~[U+D800,~U+DFFF]:~illegal \\
14776     Code~point~in~[U+E000,~U+FFFF]:~two~bytes \\
14777     Code~point~in~[U+10000,~U+10FFFF]:~
14778     a~lead~surrogate~and~a~trail~surrogate \\
14779   }
14780   Lead~surrogates~are~pairs~of~bytes~in~the~range~[0xD800,~0xDBFF],~
14781   and~trail~surrogates~are~in~the~range~[0xDC00,~0xDFFF].
14782   \flag_if_raised:nT { str_missing }
14783   {
14784     \\\
14785     A~lead~surrogate~was~not~followed~by~a~trail~surrogate.
14786   }
14787   \flag_if_raised:nT { str_extra }
14788   {
14789     \\\
14790     LaTeX~came~across~a~trail~surrogate~when~it~was~not~expected.
14791   }
14792   \flag_if_raised:nT { str_end }
14793   {
14794     \\\
14795     The~string~contained~an~odd~number~of~bytes.~This~is~invalid:~
14796     the~basic~code~unit~for~UTF-16~is~16~bits~(2~bytes).
14797   }
14798 }

```

(End definition for \l__str_missing_flag, \l__str_extra_flag, and \l__str_end_flag.)

__str_convert_decode_utf16: As for UTF-8, decoding UTF-16 is harder than encoding it. If the endianness is unknown, check the first two bytes: if those are "FE and "FF in either order, remove them and use the corresponding endianness, otherwise assume big-endianness. The three endianness cases are based on a common auxiliary whose first argument is 1 for big-endian and 2 for little-endian, and whose second argument, delimited by the scan mark \s__str_stop, is expanded once (the string may be long; passing \g__str_result_tl as an argument before expansion is cheaper).

The __str_decode_utf_xvi:Nw function defines __str_tmp:w to take two arguments and return the character code of the first one if the string is big-endian, and the second one if the string is little-endian, then loops over the string using __str_decode_utf_xvi_pair:NN described below.

```

14799 \cs_new_protected:cpn { __str_convert_decode_utf16be: }
14800 { \__str_decode_utf_xvi:Nw 1 \g__str_result_tl \s__str_stop }
14801 \cs_new_protected:cpn { __str_convert_decode_utf16le: }
14802 { \__str_decode_utf_xvi:Nw 2 \g__str_result_tl \s__str_stop }
14803 \cs_new_protected:cpn { __str_convert_decode_utf16: }
14804 {
14805   \exp_after:wN \__str_decode_utf_xvi_bom:NN
14806   \g__str_result_tl \s__str_stop \s__str_stop \s__str_stop
14807 }
14808 \cs_new_protected:Npn \__str_decode_utf_xvi_bom:NN #1#2
14809 {
14810   \str_if_eq:nnTF { #1#2 } { ^^ff ^^fe }
14811   { \__str_decode_utf_xvi:Nw 2 }
14812   {
14813     \str_if_eq:nnTF { #1#2 } { ^^fe ^^ff }
14814     { \__str_decode_utf_xvi:Nw 1 }
14815     { \__str_decode_utf_xvi:Nw 1 #1#2 }
14816   }
14817 }
14818 \cs_new_protected:Npn \__str_decode_utf_xvi:Nw #1#2 \s__str_stop
14819 {
14820   \flag_clear:n { str_error }
14821   \flag_clear:n { str_missing }
14822   \flag_clear:n { str_extra }
14823   \flag_clear:n { str_end }
14824   \cs_set:Npn \__str_tmp:w ##1 ##2 { ' ## #1 }
14825   \__kernel_tl_gset:Nx \g__str_result_tl
14826   {
14827     \exp_after:wN \__str_decode_utf_xvi_pair:NN
14828     #2 \q__str_nil \q__str_nil
14829     \prg_break_point:
14830   }
14831   \__str_if_flag_error:nnx { str_error } { utf16-decode } { }
14832 }

```

(End definition for __str_convert_decode_utf16: and others.)

```

\__str_decode_utf_xvi_pair:NN
\__str_decode_utf_xvi_quad:NNwNN
\__str_decode_utf_xvi_pair_end:Nw
\__str_decode_utf_xvi_error:nNN
\__str_decode_utf_xvi_extra:NNw

```

Bytes are read two at a time. At this stage, \@@_tmp:w #1#2 expands to the character code of the most significant byte, and we distinguish cases depending on which range it lies in:

- ["D8, "DB] signals a lead surrogate, and the integer expression yields 1 (ε -TeX rounds ties away from zero);
- ["DC, "DF] signals a trail surrogate, unexpected here, and the integer expression yields 2;
- any other value signals a code point in the Basic Multilingual Plane, which stands for itself, and the \if_case:w construction expands to nothing (cases other than 1 or 2), leaving the relevant material in the input stream, followed by another call to the _pair auxiliary.

The case of a lead surrogate is treated by the _quad auxiliary, whose arguments #1, #2, #4 and #5 are the four bytes. We expect the most significant byte of #4#5 to be in the range ["DC, "DF] (trail surrogate). The test is similar to the test used for continuation bytes

in the UTF-8 decoding functions. In the case where #4#5 is indeed a trail surrogate, leave #1#2#4#5 \s__str <code point> \s__str, and remove the pair #4#5 before looping with __str_decode_utf_xvi_pair:NN. Otherwise, of course, complain about the missing surrogate.

The magic number "D7F7 is such that "D7F7*"400 = "D800*"400+"DC00-"10000.

Every time we read a pair of bytes, we test for the end-marker \q__str_nil. When reaching the end, we additionally check that the string had an even length. Also, if the end is reached when expecting a trail surrogate, we treat that as a missing surrogate.

```

14833 \cs_new:Npn \__str_decode_utf_xvi_pair:NN #1#2
14834 {
14835   \if_meaning:w \q__str_nil #2
14836     \__str_decode_utf_xvi_pair_end:Nw #1
14837   \fi:
14838   \if_case:w
14839     \int_eval:n { ( \__str_tmp:w #1#2 - "D6 ) / 4 } \scan_stop:
14840   \or: \exp_after:wN \__str_decode_utf_xvi_quad:NNwNN
14841   \or: \exp_after:wN \__str_decode_utf_xvi_extra:NNw
14842   \fi:
14843   #1#2 \s__str
14844   \int_eval:n { "100 * \__str_tmp:w #1#2 + \__str_tmp:w #2#1 } \s__str
14845   \__str_decode_utf_xvi_pair:NN
14846 }
14847 \cs_new:Npn \__str_decode_utf_xvi_quad:NNwNN
14848   #1#2 #3 \__str_decode_utf_xvi_pair:NN #4#5
14849 {
14850   \if_meaning:w \q__str_nil #5
14851     \__str_decode_utf_xvi_error:nNN { missing } #1#2
14852     \__str_decode_utf_xvi_pair_end:Nw #4
14853   \fi:
14854   \if_int_compare:w
14855     \if_int_compare:w \__str_tmp:w #4#5 < "DC \exp_stop_f:
14856       0 = 1
14857     \else:
14858       \__str_tmp:w #4#5 < "E0
14859     \fi:
14860     \exp_stop_f:
14861     #1 #2 #4 #5 \s__str
14862     \int_eval:n
14863     {
14864       ( "100 * \__str_tmp:w #1#2 + \__str_tmp:w #2#1 - "D7F7 ) * "400
14865       + "100 * \__str_tmp:w #4#5 + \__str_tmp:w #5#4
14866     }
14867     \s__str
14868     \exp_after:wN \use_i:nnn
14869   \else:
14870     \__str_decode_utf_xvi_error:nNN { missing } #1#2
14871   \fi:
14872   \__str_decode_utf_xvi_pair:NN #4#5
14873 }
14874 \cs_new:Npn \__str_decode_utf_xvi_pair_end:Nw #1 \fi:
14875 {
14876   \fi:
14877   \if_meaning:w \q__str_nil #1

```

```

14878     \else:
14879         \__str_decode_utf_xvi_error:nNN { end } #1 \prg_do_nothing:
14880     \fi:
14881     \prg_break:
14882 }
14883 \cs_new:Npn \__str_decode_utf_xvi_extra:NNw #1#2 \s__str #3 \s__str
14884 { \__str_decode_utf_xvi_error:nNN { extra } #1#2 }
14885 \cs_new:Npn \__str_decode_utf_xvi_error:nNN #1#2#3
14886 {
14887     \flag_raise:n { str_error }
14888     \flag_raise:n { str_#1 }
14889     #2 #3 \s__str
14890     \int_use:N \c__str_replacement_char_int \s__str
14891 }

```

(End definition for __str_decode_utf_xvi_pair:NN and others.)

Restore the original catcodes of bytes 254 and 255.

```

14892 \group_end:

```

54.6.3 utf-32 support

The definitions are done in a category code regime where the bytes 0, 254 and 255 used by the byte order mark have catcode “other”.

```

14893 \group_begin:
14894     \char_set_catcode_other:N ^^00
14895     \char_set_catcode_other:N ^^fe
14896     \char_set_catcode_other:N ^^ff

```

`__str_convert_encode_utf32:` Convert each integer in the comma-list `\g__str_result_tl` to a sequence of four bytes. The functions for big-endian and little-endian encodings are very similar, but the `__str_output_byte:n` instructions are reversed.

```

\__str_convert_encode_utf32be:
    \__str_convert_encode_utf32le:
\__str_encode_utf_xxxii_be:n
    \__str_encode_utf_xxxii_be_aux:nn
\__str_encode_utf_xxxii_le:n
    \__str_encode_utf_xxxii_le_aux:nn
14897     \cs_new_protected:cpn { __str_convert_encode_utf32: }
14898     {
14899         \__str_convert_gmap_internal:N \__str_encode_utf_xxxii_be:n
14900         \tl_gput_left:Nx \g__str_result_tl { ^^00 ^^00 ^^fe ^^ff }
14901     }
14902     \cs_new_protected:cpn { __str_convert_encode_utf32be: }
14903     { \__str_convert_gmap_internal:N \__str_encode_utf_xxxii_be:n }
14904     \cs_new_protected:cpn { __str_convert_encode_utf32le: }
14905     { \__str_convert_gmap_internal:N \__str_encode_utf_xxxii_le:n }
14906     \cs_new:Npn \__str_encode_utf_xxxii_be:n #1
14907     {
14908         \exp_args:Nf \__str_encode_utf_xxxii_be_aux:nn
14909         { \int_div_truncate:nn {#1} { "100 } } {#1}
14910     }
14911     \cs_new:Npn \__str_encode_utf_xxxii_be_aux:nn #1#2
14912     {
14913         ^^00
14914         \__str_output_byte_pair_be:n {#1}
14915         \__str_output_byte:n { #2 - #1 * "100 }
14916     }
14917     \cs_new:Npn \__str_encode_utf_xxxii_le:n #1
14918     {

```

```

14919     \exp_args:Nf \__str_encode_utf_xxxii_le_aux:nn
14920     { \int_div_truncate:nn {#1} { "100 } } {#1}
14921   }
14922   \cs_new:Npn \__str_encode_utf_xxxii_le_aux:nn #1#2
14923   {
14924     \__str_output_byte:n { #2 - #1 * "100 }
14925     \__str_output_byte_pair_le:n {#1}
14926     ^^00
14927   }

```

(End definition for `__str_convert_encode_utf32:` and others.)

str_overflow There can be no error when encoding in UTF-32. When decoding, the string may not
str_end have length $4n$, or it may contain code points larger than "10FFFF. The latter case often happens if the encoding was in fact not UTF-32, because most arbitrary strings are not valid in UTF-32.

```

14928   \flag_clear_new:n { str_overflow }
14929   \flag_clear_new:n { str_end }
14930   \msg_new:nnnn { str } { utf32-decode }
14931   {
14932     Invalid~UTF-32~string:
14933     \exp_last_unbraced:Nf \use_none:n
14934     {
14935       \__str_if_flag_times:nT { str_overflow } { ,~code-point~too-large }
14936       \__str_if_flag_times:nT { str_end } { ,~truncated-string }
14937     }
14938     .
14939   }
14940   {
14941     In~the~UTF-32~encoding,~every~Unicode~character~
14942     (in~the~range~[U+0000,~U+10FFFF])~is~encoded~as~4~bytes.
14943     \flag_if_raised:nT { str_overflow }
14944     {
14945       \\\
14946       LaTeX~came~across~a~code~point~larger~than~1114111,~
14947       the~maximum~code~point~defined~by~Unicode.~
14948       Perhaps~the~string~was~not~encoded~in~the~UTF-32~encoding?
14949     }
14950     \flag_if_raised:nT { str_end }
14951     {
14952       \\\
14953       The~length~of~the~string~is~not~a~multiple~of~4.~
14954       Perhaps~the~string~was~truncated?
14955     }
14956   }

```

(End definition for `str_overflow` and `str_end`. These variables are documented on page ??.)

`__str_convert_decode_utf32:` The structure is similar to UTF-16 decoding functions. If the endianness is not given, test
`__str_convert_decode_utf32be:` the first 4 bytes of the string (possibly `\s__str_stop` if the string is too short) for the
`__str_convert_decode_utf32le:` presence of a byte-order mark. If there is a byte-order mark, use that endianness, and
`__str_decode_utf_xxxii_bom:NNNN` remove the 4 bytes, otherwise default to big-endian, and leave the 4 bytes in place. The
`__str_decode_utf_xxxii:Nw` `__str_decode_utf_xxxii:Nw` auxiliary receives 1 or 2 as its first argument indicating
`__str_decode_utf_xxxii_loop:NNNN` endianness, and the string to convert as its second argument (expanded or not). It sets
`__str_decode_utf_xxxii_end:w`

_str_tmp:w to expand to the character code of either of its two arguments depending on endianness, then triggers the _loop auxiliary inside an x-expanding assignment to \g__str_result_tl.

The _loop auxiliary first checks for the end-of-string marker \s__str_stop, calling the _end auxiliary if appropriate. Otherwise, leave the *<4 bytes>* \s__str behind, then check that the code point is not overflowing: the leading byte must be 0, and the following byte at most 16.

In the ending code, we check that there remains no byte: there should be nothing left until the first \s__str_stop. Break the map.

```

14957 \cs_new_protected:cpn { __str_convert_decode_utf32be: }
14958 { \__str_decode_utf_xxxii:Nw 1 \g__str_result_tl \s__str_stop }
14959 \cs_new_protected:cpn { __str_convert_decode_utf32le: }
14960 { \__str_decode_utf_xxxii:Nw 2 \g__str_result_tl \s__str_stop }
14961 \cs_new_protected:cpn { __str_convert_decode_utf32: }
14962 {
14963   \exp_after:wN \__str_decode_utf_xxxii_bom:NNNN \g__str_result_tl
14964   \s__str_stop \s__str_stop \s__str_stop \s__str_stop \s__str_stop
14965 }
14966 \cs_new_protected:Npn \__str_decode_utf_xxxii_bom:NNNN #1#2#3#4
14967 {
14968   \str_if_eq:nnTF { #1#2#3#4 } { ^ff ^fe ^00 ^00 }
14969   { \__str_decode_utf_xxxii:Nw 2 }
14970   {
14971     \str_if_eq:nnTF { #1#2#3#4 } { ^00 ^00 ^fe ^ff }
14972     { \__str_decode_utf_xxxii:Nw 1 }
14973     { \__str_decode_utf_xxxii:Nw 1 #1#2#3#4 }
14974   }
14975 }
14976 \cs_new_protected:Npn \__str_decode_utf_xxxii:Nw #1#2 \s__str_stop
14977 {
14978   \flag_clear:n { str_overflow }
14979   \flag_clear:n { str_end }
14980   \flag_clear:n { str_error }
14981   \cs_set:Npn \__str_tmp:w ##1 ##2 { ' ## #1 }
14982   \__kernel_tl_gset:Nx \g__str_result_tl
14983   {
14984     \exp_after:wN \__str_decode_utf_xxxii_loop:NNNN
14985     #2 \s__str_stop \s__str_stop \s__str_stop \s__str_stop
14986     \prg_break_point:
14987   }
14988   \__str_if_flag_error:nnx { str_error } { utf32-decode } { }
14989 }
14990 \cs_new:Npn \__str_decode_utf_xxxii_loop:NNNN #1#2#3#4
14991 {
14992   \if_meaning:w \s__str_stop #4
14993   \exp_after:wN \__str_decode_utf_xxxii_end:w
14994   \fi:
14995   #1#2#3#4 \s__str
14996   \if_int_compare:w \__str_tmp:w #1#4 > \c_zero_int
14997   \flag_raise:n { str_overflow }
14998   \flag_raise:n { str_error }
14999   \int_use:N \c__str_replacement_char_int
15000 \else:

```



```

15001         \if_int_compare:w \__str_tmp:w #2#3 > 16 \exp_stop_f:
15002         \flag_raise:n { str_overflow }
15003         \flag_raise:n { str_error }
15004         \int_use:N \c__str_replacement_char_int
15005     \else:
15006         \int_eval:n
15007         { \__str_tmp:w #2#3*"10000 + \__str_tmp:w #3#2*"100 + \__str_tmp:w #4#1 }
15008     \fi:
15009 \fi:
15010 \s__str
15011 \__str_decode_utf_xxxii_loop:NNNN
15012 }
15013 \cs_new:Npn \__str_decode_utf_xxxii_end:w #1 \s__str_stop
15014 {
15015     \tl_if_empty:nF {#1}
15016     {
15017         \flag_raise:n { str_end }
15018         \flag_raise:n { str_error }
15019         #1 \s__str
15020         \int_use:N \c__str_replacement_char_int \s__str
15021     }
15022 \prg_break:
15023 }

```

(End definition for `__str_convert_decode_utf32:` and others.)

Restore the original catcodes of bytes 0, 254 and 255.

```

15024 \group_end:

```

54.7 PDF names and strings by expansion

```

\str_convert_pdfname:n
\__str_convert_pdfname:n
  \__str_convert_pdfname_bytes:n
  \__str_convert_pdfname_bytes_aux:n
\__str_convert_pdfname_bytes_aux:nnn

```

To convert to PDF names by expansion, we work purely on UTF-8 input. The first step is to make a string with “other” spaces, after which we use a simple token-by-token approach. In Unicode engines, we break down everything before one-byte codepoints, but for 8-bit engines there is no need to worry. Actual escaping is covered by the same code as used in the non-expandable route.

```

15025 \cs_new:Npn \str_convert_pdfname:n #1
15026 {
15027     \exp_args:Ne \tl_to_str:n
15028     { \str_map_function:nN {#1} \__str_convert_pdfname:n }
15029 }
15030 \bool_lazy_or:nnTF
15031 { \sys_if_engine luatex_p: }
15032 { \sys_if_engine xetex_p: }
15033 {
15034     \cs_new:Npn \__str_convert_pdfname:n #1
15035     {
15036         \int_compare:nNnTF { '#1 } > { "7F }
15037         { \__str_convert_pdfname_bytes:n {#1} }
15038         { \__str_escape_name_char:n {#1} }
15039     }
15040 \cs_new:Npn \__str_convert_pdfname_bytes:n #1
15041 {

```

```

15042     \exp_args:Ne \__str_convert_pdfname_bytes_aux:n
15043     { \char_to_utfviii_bytes:n {'#1} }
15044   }
15045   \cs_new:Npn \__str_convert_pdfname_bytes_aux:n #1
15046   { \__str_convert_pdfname_bytes_aux:nnnn #1 }
15047   \cs_new:Npx \__str_convert_pdfname_bytes_aux:nnnn #1#2#3#4
15048   {
15049     \c_hash_str \exp_not:N \__str_output_hexadecimal:n {#1}
15050     \c_hash_str \exp_not:N \__str_output_hexadecimal:n {#2}
15051     \exp_not:N \tl_if_blank:nF {#3}
15052     {
15053       \c_hash_str \exp_not:N \__str_output_hexadecimal:n {#3}
15054       \exp_not:N \tl_if_blank:nF {#4}
15055       {
15056         \c_hash_str \exp_not:N \__str_output_hexadecimal:n {#4}
15057       }
15058     }
15059   }
15060 }
15061 { \cs_new_eq:NN \__str_convert_pdfname:n \__str_escape_name_char:n }

```

(End definition for `\str_convert_pdfname:n` and others. This function is documented on page [135](#).)

```

15062 </package>

```

54.7.1 iso 8859 support

The ISO-8859-1 encoding exactly matches with the 256 first Unicode characters. For other 8-bit encodings of the ISO-8859 family, we keep track only of differences, and of unassigned bytes.

```

15063 <*iso88591>
15064 \__str_declare_eight_bit_encoding:nnnn { iso88591 } { 256 }
15065 {
15066 }
15067 {
15068 }
15069 </iso88591>
15070 <*iso88592>
15071 \__str_declare_eight_bit_encoding:nnnn { iso88592 } { 399 }
15072 {
15073   { A1 } { 0104 }
15074   { A2 } { 02D8 }
15075   { A3 } { 0141 }
15076   { A5 } { 013D }
15077   { A6 } { 015A }
15078   { A9 } { 0160 }
15079   { AA } { 015E }
15080   { AB } { 0164 }
15081   { AC } { 0179 }
15082   { AE } { 017D }
15083   { AF } { 017B }
15084   { B1 } { 0105 }
15085   { B2 } { 02DB }
15086   { B3 } { 0142 }

```

```

15087     { B5 } { 013E }
15088     { B6 } { 015B }
15089     { B7 } { 02C7 }
15090     { B9 } { 0161 }
15091     { BA } { 015F }
15092     { BB } { 0165 }
15093     { BC } { 017A }
15094     { BD } { 02DD }
15095     { BE } { 017E }
15096     { BF } { 017C }
15097     { C0 } { 0154 }
15098     { C3 } { 0102 }
15099     { C5 } { 0139 }
15100     { C6 } { 0106 }
15101     { C8 } { 010C }
15102     { CA } { 0118 }
15103     { CC } { 011A }
15104     { CF } { 010E }
15105     { D0 } { 0110 }
15106     { D1 } { 0143 }
15107     { D2 } { 0147 }
15108     { D5 } { 0150 }
15109     { D8 } { 0158 }
15110     { D9 } { 016E }
15111     { DB } { 0170 }
15112     { DE } { 0162 }
15113     { E0 } { 0155 }
15114     { E3 } { 0103 }
15115     { E5 } { 013A }
15116     { E6 } { 0107 }
15117     { E8 } { 010D }
15118     { EA } { 0119 }
15119     { EC } { 011B }
15120     { EF } { 010F }
15121     { F0 } { 0111 }
15122     { F1 } { 0144 }
15123     { F2 } { 0148 }
15124     { F5 } { 0151 }
15125     { F8 } { 0159 }
15126     { F9 } { 016F }
15127     { FB } { 0171 }
15128     { FE } { 0163 }
15129     { FF } { 02D9 }
15130 }
15131 {
15132 }
15133 </iso88592>
15134 <*iso88593>
15135 \_str_declare_eight_bit_encoding:nnnn { iso88593 } { 384 }
15136 {
15137     { A1 } { 0126 }
15138     { A2 } { 02D8 }
15139     { A6 } { 0124 }
15140     { A9 } { 0130 }

```

```

15141     { AA } { 015E }
15142     { AB } { 011E }
15143     { AC } { 0134 }
15144     { AF } { 017B }
15145     { B1 } { 0127 }
15146     { B6 } { 0125 }
15147     { B9 } { 0131 }
15148     { BA } { 015F }
15149     { BB } { 011F }
15150     { BC } { 0135 }
15151     { BF } { 017C }
15152     { C5 } { 010A }
15153     { C6 } { 0108 }
15154     { D5 } { 0120 }
15155     { D8 } { 011C }
15156     { DD } { 016C }
15157     { DE } { 015C }
15158     { E5 } { 010B }
15159     { E6 } { 0109 }
15160     { F5 } { 0121 }
15161     { F8 } { 011D }
15162     { FD } { 016D }
15163     { FE } { 015D }
15164     { FF } { 02D9 }
15165 }
15166 {
15167     { A5 }
15168     { AE }
15169     { BE }
15170     { C3 }
15171     { D0 }
15172     { E3 }
15173     { F0 }
15174 }
15175 </iso88593>
15176 <*iso88594>
15177 \_str_declare\_eight\_bit\_encoding:nmn { iso88594 } { 383 }
15178 {
15179     { A1 } { 0104 }
15180     { A2 } { 0138 }
15181     { A3 } { 0156 }
15182     { A5 } { 0128 }
15183     { A6 } { 013B }
15184     { A9 } { 0160 }
15185     { AA } { 0112 }
15186     { AB } { 0122 }
15187     { AC } { 0166 }
15188     { AE } { 017D }
15189     { B1 } { 0105 }
15190     { B2 } { 02DB }
15191     { B3 } { 0157 }
15192     { B5 } { 0129 }
15193     { B6 } { 013C }
15194     { B7 } { 02C7 }

```

```

15195     { B9 } { 0161 }
15196     { BA } { 0113 }
15197     { BB } { 0123 }
15198     { BC } { 0167 }
15199     { BD } { 014A }
15200     { BE } { 017E }
15201     { BF } { 014B }
15202     { C0 } { 0100 }
15203     { C7 } { 012E }
15204     { C8 } { 010C }
15205     { CA } { 0118 }
15206     { CC } { 0116 }
15207     { CF } { 012A }
15208     { D0 } { 0110 }
15209     { D1 } { 0145 }
15210     { D2 } { 014C }
15211     { D3 } { 0136 }
15212     { D9 } { 0172 }
15213     { DD } { 0168 }
15214     { DE } { 016A }
15215     { E0 } { 0101 }
15216     { E7 } { 012F }
15217     { E8 } { 010D }
15218     { EA } { 0119 }
15219     { EC } { 0117 }
15220     { EF } { 012B }
15221     { F0 } { 0111 }
15222     { F1 } { 0146 }
15223     { F2 } { 014D }
15224     { F3 } { 0137 }
15225     { F9 } { 0173 }
15226     { FD } { 0169 }
15227     { FE } { 016B }
15228     { FF } { 02D9 }
15229     }
15230     {
15231     }
15232 </iso88594>
15233 <*iso88595>
15234 \_str_declare_eight_bit_encoding:nnnn { iso88595 } { 374 }
15235     {
15236         { A1 } { 0401 }
15237         { A2 } { 0402 }
15238         { A3 } { 0403 }
15239         { A4 } { 0404 }
15240         { A5 } { 0405 }
15241         { A6 } { 0406 }
15242         { A7 } { 0407 }
15243         { A8 } { 0408 }
15244         { A9 } { 0409 }
15245         { AA } { 040A }
15246         { AB } { 040B }
15247         { AC } { 040C }
15248         { AE } { 040E }

```

15249	{ AF }	{ 040F }
15250	{ B0 }	{ 0410 }
15251	{ B1 }	{ 0411 }
15252	{ B2 }	{ 0412 }
15253	{ B3 }	{ 0413 }
15254	{ B4 }	{ 0414 }
15255	{ B5 }	{ 0415 }
15256	{ B6 }	{ 0416 }
15257	{ B7 }	{ 0417 }
15258	{ B8 }	{ 0418 }
15259	{ B9 }	{ 0419 }
15260	{ BA }	{ 041A }
15261	{ BB }	{ 041B }
15262	{ BC }	{ 041C }
15263	{ BD }	{ 041D }
15264	{ BE }	{ 041E }
15265	{ BF }	{ 041F }
15266	{ C0 }	{ 0420 }
15267	{ C1 }	{ 0421 }
15268	{ C2 }	{ 0422 }
15269	{ C3 }	{ 0423 }
15270	{ C4 }	{ 0424 }
15271	{ C5 }	{ 0425 }
15272	{ C6 }	{ 0426 }
15273	{ C7 }	{ 0427 }
15274	{ C8 }	{ 0428 }
15275	{ C9 }	{ 0429 }
15276	{ CA }	{ 042A }
15277	{ CB }	{ 042B }
15278	{ CC }	{ 042C }
15279	{ CD }	{ 042D }
15280	{ CE }	{ 042E }
15281	{ CF }	{ 042F }
15282	{ D0 }	{ 0430 }
15283	{ D1 }	{ 0431 }
15284	{ D2 }	{ 0432 }
15285	{ D3 }	{ 0433 }
15286	{ D4 }	{ 0434 }
15287	{ D5 }	{ 0435 }
15288	{ D6 }	{ 0436 }
15289	{ D7 }	{ 0437 }
15290	{ D8 }	{ 0438 }
15291	{ D9 }	{ 0439 }
15292	{ DA }	{ 043A }
15293	{ DB }	{ 043B }
15294	{ DC }	{ 043C }
15295	{ DD }	{ 043D }
15296	{ DE }	{ 043E }
15297	{ DF }	{ 043F }
15298	{ E0 }	{ 0440 }
15299	{ E1 }	{ 0441 }
15300	{ E2 }	{ 0442 }
15301	{ E3 }	{ 0443 }
15302	{ E4 }	{ 0444 }

```

15303     { E5 } { 0445 }
15304     { E6 } { 0446 }
15305     { E7 } { 0447 }
15306     { E8 } { 0448 }
15307     { E9 } { 0449 }
15308     { EA } { 044A }
15309     { EB } { 044B }
15310     { EC } { 044C }
15311     { ED } { 044D }
15312     { EE } { 044E }
15313     { EF } { 044F }
15314     { FO } { 2116 }
15315     { F1 } { 0451 }
15316     { F2 } { 0452 }
15317     { F3 } { 0453 }
15318     { F4 } { 0454 }
15319     { F5 } { 0455 }
15320     { F6 } { 0456 }
15321     { F7 } { 0457 }
15322     { F8 } { 0458 }
15323     { F9 } { 0459 }
15324     { FA } { 045A }
15325     { FB } { 045B }
15326     { FC } { 045C }
15327     { FD } { 00A7 }
15328     { FE } { 045E }
15329     { FF } { 045F }
15330 }
15331 {
15332 }
15333 </iso88595>
15334 <*iso88596>
15335 \__str_declare_eight_bit_encoding:nnnn { iso88596 } { 344 }
15336 {
15337     { AC } { 060C }
15338     { BB } { 061B }
15339     { BF } { 061F }
15340     { C1 } { 0621 }
15341     { C2 } { 0622 }
15342     { C3 } { 0623 }
15343     { C4 } { 0624 }
15344     { C5 } { 0625 }
15345     { C6 } { 0626 }
15346     { C7 } { 0627 }
15347     { C8 } { 0628 }
15348     { C9 } { 0629 }
15349     { CA } { 062A }
15350     { CB } { 062B }
15351     { CC } { 062C }
15352     { CD } { 062D }
15353     { CE } { 062E }
15354     { CF } { 062F }
15355     { D0 } { 0630 }
15356     { D1 } { 0631 }

```

```

15357      { D2 } { 0632 }
15358      { D3 } { 0633 }
15359      { D4 } { 0634 }
15360      { D5 } { 0635 }
15361      { D6 } { 0636 }
15362      { D7 } { 0637 }
15363      { D8 } { 0638 }
15364      { D9 } { 0639 }
15365      { DA } { 063A }
15366      { E0 } { 0640 }
15367      { E1 } { 0641 }
15368      { E2 } { 0642 }
15369      { E3 } { 0643 }
15370      { E4 } { 0644 }
15371      { E5 } { 0645 }
15372      { E6 } { 0646 }
15373      { E7 } { 0647 }
15374      { E8 } { 0648 }
15375      { E9 } { 0649 }
15376      { EA } { 064A }
15377      { EB } { 064B }
15378      { EC } { 064C }
15379      { ED } { 064D }
15380      { EE } { 064E }
15381      { EF } { 064F }
15382      { FO } { 0650 }
15383      { F1 } { 0651 }
15384      { F2 } { 0652 }
15385      }
15386      {
15387          { A1 }
15388          { A2 }
15389          { A3 }
15390          { A5 }
15391          { A6 }
15392          { A7 }
15393          { A8 }
15394          { A9 }
15395          { AA }
15396          { AB }
15397          { AE }
15398          { AF }
15399          { B0 }
15400          { B1 }
15401          { B2 }
15402          { B3 }
15403          { B4 }
15404          { B5 }
15405          { B6 }
15406          { B7 }
15407          { B8 }
15408          { B9 }
15409          { BA }
15410          { BC }

```



```

15411     { BD }
15412     { BE }
15413     { C0 }
15414     { DB }
15415     { DC }
15416     { DD }
15417     { DE }
15418     { DF }
15419     }
15420 </iso88596>

15421 <*iso88597>
15422 \__str_declare_eight_bit_encoding:nnnn { iso88597 } { 498 }
15423 {
15424     { A1 } { 2018 }
15425     { A2 } { 2019 }
15426     { A4 } { 20AC }
15427     { A5 } { 20AF }
15428     { AA } { 037A }
15429     { AF } { 2015 }
15430     { B4 } { 0384 }
15431     { B5 } { 0385 }
15432     { B6 } { 0386 }
15433     { B8 } { 0388 }
15434     { B9 } { 0389 }
15435     { BA } { 038A }
15436     { BC } { 038C }
15437     { BE } { 038E }
15438     { BF } { 038F }
15439     { C0 } { 0390 }
15440     { C1 } { 0391 }
15441     { C2 } { 0392 }
15442     { C3 } { 0393 }
15443     { C4 } { 0394 }
15444     { C5 } { 0395 }
15445     { C6 } { 0396 }
15446     { C7 } { 0397 }
15447     { C8 } { 0398 }
15448     { C9 } { 0399 }
15449     { CA } { 039A }
15450     { CB } { 039B }
15451     { CC } { 039C }
15452     { CD } { 039D }
15453     { CE } { 039E }
15454     { CF } { 039F }
15455     { D0 } { 03A0 }
15456     { D1 } { 03A1 }
15457     { D3 } { 03A3 }
15458     { D4 } { 03A4 }
15459     { D5 } { 03A5 }
15460     { D6 } { 03A6 }
15461     { D7 } { 03A7 }
15462     { D8 } { 03A8 }
15463     { D9 } { 03A9 }
15464     { DA } { 03AA }

```

```

15465     { DB } { 03AB }
15466     { DC } { 03AC }
15467     { DD } { 03AD }
15468     { DE } { 03AE }
15469     { DF } { 03AF }
15470     { E0 } { 03B0 }
15471     { E1 } { 03B1 }
15472     { E2 } { 03B2 }
15473     { E3 } { 03B3 }
15474     { E4 } { 03B4 }
15475     { E5 } { 03B5 }
15476     { E6 } { 03B6 }
15477     { E7 } { 03B7 }
15478     { E8 } { 03B8 }
15479     { E9 } { 03B9 }
15480     { EA } { 03BA }
15481     { EB } { 03BB }
15482     { EC } { 03BC }
15483     { ED } { 03BD }
15484     { EE } { 03BE }
15485     { EF } { 03BF }
15486     { F0 } { 03C0 }
15487     { F1 } { 03C1 }
15488     { F2 } { 03C2 }
15489     { F3 } { 03C3 }
15490     { F4 } { 03C4 }
15491     { F5 } { 03C5 }
15492     { F6 } { 03C6 }
15493     { F7 } { 03C7 }
15494     { F8 } { 03C8 }
15495     { F9 } { 03C9 }
15496     { FA } { 03CA }
15497     { FB } { 03CB }
15498     { FC } { 03CC }
15499     { FD } { 03CD }
15500     { FE } { 03CE }
15501   }
15502   {
15503     { AE }
15504     { D2 }
15505   }
15506   </iso88597>
15507   <*iso88598>
15508   \_str_declare_eight_bit_encoding:nnnn { iso88598 } { 308 }
15509   {
15510     { AA } { 00D7 }
15511     { BA } { 00F7 }
15512     { DF } { 2017 }
15513     { E0 } { 05D0 }
15514     { E1 } { 05D1 }
15515     { E2 } { 05D2 }
15516     { E3 } { 05D3 }
15517     { E4 } { 05D4 }
15518     { E5 } { 05D5 }

```

```

15519      { E6 } { 05D6 }
15520      { E7 } { 05D7 }
15521      { E8 } { 05D8 }
15522      { E9 } { 05D9 }
15523      { EA } { 05DA }
15524      { EB } { 05DB }
15525      { EC } { 05DC }
15526      { ED } { 05DD }
15527      { EE } { 05DE }
15528      { EF } { 05DF }
15529      { F0 } { 05E0 }
15530      { F1 } { 05E1 }
15531      { F2 } { 05E2 }
15532      { F3 } { 05E3 }
15533      { F4 } { 05E4 }
15534      { F5 } { 05E5 }
15535      { F6 } { 05E6 }
15536      { F7 } { 05E7 }
15537      { F8 } { 05E8 }
15538      { F9 } { 05E9 }
15539      { FA } { 05EA }
15540      { FD } { 200E }
15541      { FE } { 200F }
15542      }
15543      {
15544          { A1 }
15545          { BF }
15546          { C0 }
15547          { C1 }
15548          { C2 }
15549          { C3 }
15550          { C4 }
15551          { C5 }
15552          { C6 }
15553          { C7 }
15554          { C8 }
15555          { C9 }
15556          { CA }
15557          { CB }
15558          { CC }
15559          { CD }
15560          { CE }
15561          { CF }
15562          { D0 }
15563          { D1 }
15564          { D2 }
15565          { D3 }
15566          { D4 }
15567          { D5 }
15568          { D6 }
15569          { D7 }
15570          { D8 }
15571          { D9 }
15572          { DA }

```

```

15573     { DB }
15574     { DC }
15575     { DD }
15576     { DE }
15577     { FB }
15578     { FC }
15579 }
15580 </iso88598>
15581 (*iso88599)
15582 \_str_declare\_eight\_bit\_encoding:nnnn { iso88599 } { 352 }
15583 {
15584     { D0 } { 011E }
15585     { DD } { 0130 }
15586     { DE } { 015E }
15587     { FO } { 011F }
15588     { FD } { 0131 }
15589     { FE } { 015F }
15590 }
15591 {
15592 }
15593 </iso88599>
15594 (*iso885910)
15595 \_str_declare\_eight\_bit\_encoding:nnnn { iso885910 } { 383 }
15596 {
15597     { A1 } { 0104 }
15598     { A2 } { 0112 }
15599     { A3 } { 0122 }
15600     { A4 } { 012A }
15601     { A5 } { 0128 }
15602     { A6 } { 0136 }
15603     { A8 } { 013B }
15604     { A9 } { 0110 }
15605     { AA } { 0160 }
15606     { AB } { 0166 }
15607     { AC } { 017D }
15608     { AE } { 016A }
15609     { AF } { 014A }
15610     { B1 } { 0105 }
15611     { B2 } { 0113 }
15612     { B3 } { 0123 }
15613     { B4 } { 012B }
15614     { B5 } { 0129 }
15615     { B6 } { 0137 }
15616     { B8 } { 013C }
15617     { B9 } { 0111 }
15618     { BA } { 0161 }
15619     { BB } { 0167 }
15620     { BC } { 017E }
15621     { BD } { 2015 }
15622     { BE } { 016B }
15623     { BF } { 014B }
15624     { C0 } { 0100 }
15625     { C7 } { 012E }

```

```

15626     { C8 } { 010C }
15627     { CA } { 0118 }
15628     { CC } { 0116 }
15629     { D1 } { 0145 }
15630     { D2 } { 014C }
15631     { D7 } { 0168 }
15632     { D9 } { 0172 }
15633     { E0 } { 0101 }
15634     { E7 } { 012F }
15635     { E8 } { 010D }
15636     { EA } { 0119 }
15637     { EC } { 0117 }
15638     { F1 } { 0146 }
15639     { F2 } { 014D }
15640     { F7 } { 0169 }
15641     { F9 } { 0173 }
15642     { FF } { 0138 }
15643 }
15644 {
15645 }
15646 </iso885910>
15647 <*iso885911>
15648 \_str_declare_eight_bit_encoding:nnnn { iso885911 } { 369 }
15649 {
15650     { A1 } { 0E01 }
15651     { A2 } { 0E02 }
15652     { A3 } { 0E03 }
15653     { A4 } { 0E04 }
15654     { A5 } { 0E05 }
15655     { A6 } { 0E06 }
15656     { A7 } { 0E07 }
15657     { A8 } { 0E08 }
15658     { A9 } { 0E09 }
15659     { AA } { 0EOA }
15660     { AB } { 0EOB }
15661     { AC } { 0EOC }
15662     { AD } { 0EOD }
15663     { AE } { 0EOE }
15664     { AF } { 0EOF }
15665     { B0 } { 0E10 }
15666     { B1 } { 0E11 }
15667     { B2 } { 0E12 }
15668     { B3 } { 0E13 }
15669     { B4 } { 0E14 }
15670     { B5 } { 0E15 }
15671     { B6 } { 0E16 }
15672     { B7 } { 0E17 }
15673     { B8 } { 0E18 }
15674     { B9 } { 0E19 }
15675     { BA } { 0E1A }
15676     { BB } { 0E1B }
15677     { BC } { 0E1C }
15678     { BD } { 0E1D }
15679     { BE } { 0E1E }

```

15680	{ BF }	{ 0E1F }
15681	{ C0 }	{ 0E20 }
15682	{ C1 }	{ 0E21 }
15683	{ C2 }	{ 0E22 }
15684	{ C3 }	{ 0E23 }
15685	{ C4 }	{ 0E24 }
15686	{ C5 }	{ 0E25 }
15687	{ C6 }	{ 0E26 }
15688	{ C7 }	{ 0E27 }
15689	{ C8 }	{ 0E28 }
15690	{ C9 }	{ 0E29 }
15691	{ CA }	{ 0E2A }
15692	{ CB }	{ 0E2B }
15693	{ CC }	{ 0E2C }
15694	{ CD }	{ 0E2D }
15695	{ CE }	{ 0E2E }
15696	{ CF }	{ 0E2F }
15697	{ D0 }	{ 0E30 }
15698	{ D1 }	{ 0E31 }
15699	{ D2 }	{ 0E32 }
15700	{ D3 }	{ 0E33 }
15701	{ D4 }	{ 0E34 }
15702	{ D5 }	{ 0E35 }
15703	{ D6 }	{ 0E36 }
15704	{ D7 }	{ 0E37 }
15705	{ D8 }	{ 0E38 }
15706	{ D9 }	{ 0E39 }
15707	{ DA }	{ 0E3A }
15708	{ DF }	{ 0E3F }
15709	{ E0 }	{ 0E40 }
15710	{ E1 }	{ 0E41 }
15711	{ E2 }	{ 0E42 }
15712	{ E3 }	{ 0E43 }
15713	{ E4 }	{ 0E44 }
15714	{ E5 }	{ 0E45 }
15715	{ E6 }	{ 0E46 }
15716	{ E7 }	{ 0E47 }
15717	{ E8 }	{ 0E48 }
15718	{ E9 }	{ 0E49 }
15719	{ EA }	{ 0E4A }
15720	{ EB }	{ 0E4B }
15721	{ EC }	{ 0E4C }
15722	{ ED }	{ 0E4D }
15723	{ EE }	{ 0E4E }
15724	{ EF }	{ 0E4F }
15725	{ F0 }	{ 0E50 }
15726	{ F1 }	{ 0E51 }
15727	{ F2 }	{ 0E52 }
15728	{ F3 }	{ 0E53 }
15729	{ F4 }	{ 0E54 }
15730	{ F5 }	{ 0E55 }
15731	{ F6 }	{ 0E56 }
15732	{ F7 }	{ 0E57 }
15733	{ F8 }	{ 0E58 }

```

15734     { F9 } { OE59 }
15735     { FA } { OE5A }
15736     { FB } { OE5B }
15737   }
15738   {
15739     { DB }
15740     { DC }
15741     { DD }
15742     { DE }
15743   }
15744   </iso885911>
15745   (*iso885913)
15746   \_str_declare_eight_bit_encoding:nnnn { iso885913 } { 399 }
15747   {
15748     { A1 } { 201D }
15749     { A5 } { 201E }
15750     { A8 } { 00D8 }
15751     { AA } { 0156 }
15752     { AF } { 00C6 }
15753     { B4 } { 201C }
15754     { B8 } { 00F8 }
15755     { BA } { 0157 }
15756     { BF } { 00E6 }
15757     { C0 } { 0104 }
15758     { C1 } { 012E }
15759     { C2 } { 0100 }
15760     { C3 } { 0106 }
15761     { C6 } { 0118 }
15762     { C7 } { 0112 }
15763     { C8 } { 010C }
15764     { CA } { 0179 }
15765     { CB } { 0116 }
15766     { CC } { 0122 }
15767     { CD } { 0136 }
15768     { CE } { 012A }
15769     { CF } { 013B }
15770     { D0 } { 0160 }
15771     { D1 } { 0143 }
15772     { D2 } { 0145 }
15773     { D4 } { 014C }
15774     { D8 } { 0172 }
15775     { D9 } { 0141 }
15776     { DA } { 015A }
15777     { DB } { 016A }
15778     { DD } { 017B }
15779     { DE } { 017D }
15780     { E0 } { 0105 }
15781     { E1 } { 012F }
15782     { E2 } { 0101 }
15783     { E3 } { 0107 }
15784     { E6 } { 0119 }
15785     { E7 } { 0113 }
15786     { E8 } { 010D }
15787     { EA } { 017A }

```

```

15788     { EB } { 0117 }
15789     { EC } { 0123 }
15790     { ED } { 0137 }
15791     { EE } { 012B }
15792     { EF } { 013C }
15793     { F0 } { 0161 }
15794     { F1 } { 0144 }
15795     { F2 } { 0146 }
15796     { F4 } { 014D }
15797     { F8 } { 0173 }
15798     { F9 } { 0142 }
15799     { FA } { 015B }
15800     { FB } { 016B }
15801     { FD } { 017C }
15802     { FE } { 017E }
15803     { FF } { 2019 }
15804 }
15805 {
15806 }
15807 </iso885913>
15808 <*iso885914>
15809 \__str_declare_eight_bit_encoding:nnnn { iso885914 } { 529 }
15810 {
15811     { A1 } { 1E02 }
15812     { A2 } { 1E03 }
15813     { A4 } { 010A }
15814     { A5 } { 010B }
15815     { A6 } { 1E0A }
15816     { A8 } { 1E80 }
15817     { AA } { 1E82 }
15818     { AB } { 1E0B }
15819     { AC } { 1EF2 }
15820     { AF } { 0178 }
15821     { B0 } { 1E1E }
15822     { B1 } { 1E1F }
15823     { B2 } { 0120 }
15824     { B3 } { 0121 }
15825     { B4 } { 1E40 }
15826     { B5 } { 1E41 }
15827     { B7 } { 1E56 }
15828     { B8 } { 1E81 }
15829     { B9 } { 1E57 }
15830     { BA } { 1E83 }
15831     { BB } { 1E60 }
15832     { BC } { 1EF3 }
15833     { BD } { 1E84 }
15834     { BE } { 1E85 }
15835     { BF } { 1E61 }
15836     { D0 } { 0174 }
15837     { D7 } { 1E6A }
15838     { DE } { 0176 }
15839     { F0 } { 0175 }
15840     { F7 } { 1E6B }
15841     { FE } { 0177 }

```



```

15842     }
15843     {
15844     }
15845     </iso885914>
15846     <*iso885915>
15847     \_str_declare_eight_bit_encoding:nnnn { iso885915 } { 383 }
15848     {
15849         { A4 } { 20AC }
15850         { A6 } { 0160 }
15851         { A8 } { 0161 }
15852         { B4 } { 017D }
15853         { B8 } { 017E }
15854         { BC } { 0152 }
15855         { BD } { 0153 }
15856         { BE } { 0178 }
15857     }
15858     {
15859     }
15860     </iso885915>
15861     <*iso885916>
15862     \_str_declare_eight_bit_encoding:nnnn { iso885916 } { 558 }
15863     {
15864         { A1 } { 0104 }
15865         { A2 } { 0105 }
15866         { A3 } { 0141 }
15867         { A4 } { 20AC }
15868         { A5 } { 201E }
15869         { A6 } { 0160 }
15870         { A8 } { 0161 }
15871         { AA } { 0218 }
15872         { AC } { 0179 }
15873         { AE } { 017A }
15874         { AF } { 017B }
15875         { B2 } { 010C }
15876         { B3 } { 0142 }
15877         { B4 } { 017D }
15878         { B5 } { 201D }
15879         { B8 } { 017E }
15880         { B9 } { 010D }
15881         { BA } { 0219 }
15882         { BC } { 0152 }
15883         { BD } { 0153 }
15884         { BE } { 0178 }
15885         { BF } { 017C }
15886         { C3 } { 0102 }
15887         { C5 } { 0106 }
15888         { D0 } { 0110 }
15889         { D1 } { 0143 }
15890         { D5 } { 0150 }
15891         { D7 } { 015A }
15892         { D8 } { 0170 }
15893         { DD } { 0118 }
15894         { DE } { 021A }

```

```
15895      { E3 } { 0103 }
15896      { E5 } { 0107 }
15897      { F0 } { 0111 }
15898      { F1 } { 0144 }
15899      { F5 } { 0151 }
15900      { F7 } { 015B }
15901      { F8 } { 0171 }
15902      { FD } { 0119 }
15903      { FE } { 021B }
15904      }
15905      {
15906      }
15907 </iso885916>
```

Chapter 55

l3quark implementation

The following test files are used for this code: *m3quark001.lvt*.

```
15908 <*package>
```

55.1 Quarks

```
15909 <@@=quark>
```

\quark_new:N Allocate a new quark.

```
15910 \cs_new_protected:Npn \quark_new:N #1
15911 {
15912   \__kernel_chk_if_free_cs:N #1
15913   \cs_gset_nopar:Npn #1 {#1}
15914 }
```

(End definition for `\quark_new:N`. This function is documented on page 138.)

\q_nil Some “public” quarks. `\q_stop` is an “end of argument” marker, `\q_nil` is a empty value and `\q_no_value` marks an empty argument.

```
15915 \quark_new:N \q_nil
15916 \quark_new:N \q_mark
15917 \quark_new:N \q_no_value
15918 \quark_new:N \q_stop
```

(End definition for `\q_nil` and others. These variables are documented on page 138.)

\q_recursion_tail Quarks for ending recursions. Only ever used there! `\q_recursion_tail` is appended to whatever list structure we are doing recursion on, meaning it is added as a proper list item with whatever list separator is in use. `\q_recursion_stop` is placed directly after the list.

```
15919 \quark_new:N \q_recursion_tail
15920 \quark_new:N \q_recursion_stop
```

(End definition for `\q_recursion_tail` and `\q_recursion_stop`. These variables are documented on page 139.)

\s__quark Private scan mark used in `l3quark`. We don’t have `l3scan` yet, so we declare the scan mark here and add it to the scan mark pool later.

```
15921 \cs_new_eq:NN \s__quark \scan_stop:
```

(End definition for \s__quark.)

\q__quark_nil Private quark use for some tests.

15922 \quark_new:N \q__quark_nil

(End definition for \q__quark_nil.)

\quark_if_recursion_tail_stop:N
\quark_if_recursion_tail_stop_do:Nn

When doing recursions, it is easy to spend a lot of time testing if the end marker has been found. To avoid this, a dedicated end marker is used each time a recursion is set up. Thus if the marker is found everything can be wrapper up and finished off. The simple case is when the test can guarantee that only a single token is being tested. In this case, there is just a dedicated copy of the standard quark test. Both a gobbling version and one inserting end code are provided.

```
15923 \cs_new:Npn \quark_if_recursion_tail_stop:N #1
15924 {
15925   \if_meaning:w \q_recursion_tail #1
15926   \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
15927   \fi:
15928 }
15929 \cs_new:Npn \quark_if_recursion_tail_stop_do:Nn #1
15930 {
15931   \if_meaning:w \q_recursion_tail #1
15932   \exp_after:wN \use_i_delimit_by_q_recursion_stop:nw
15933   \else:
15934   \exp_after:wN \use_none:n
15935   \fi:
15936 }
```

(End definition for \quark_if_recursion_tail_stop:N and \quark_if_recursion_tail_stop_do:Nn. These functions are documented on page 139.)

\quark_if_recursion_tail_stop:n
\quark_if_recursion_tail_stop:o
\quark_if_recursion_tail_stop_do:nn
\quark_if_recursion_tail_stop_do:nn
_quark_if_recursion_tail:w

See \quark_if_nil:nTF for the details. Expanding _quark_if_recursion_tail:w once in front of the tokens chosen here gives an empty result if and only if #1 is exactly \q_recursion_tail.

```
15937 \cs_new:Npn \quark_if_recursion_tail_stop:n #1
15938 {
15939   \tl_if_empty:oTF
15940   { \_quark_if_recursion_tail:w {} #1 {} ?! \q_recursion_tail ??? }
15941   { \use_none_delimit_by_q_recursion_stop:w }
15942   { }
15943 }
15944 \cs_new:Npn \quark_if_recursion_tail_stop_do:nn #1
15945 {
15946   \tl_if_empty:oTF
15947   { \_quark_if_recursion_tail:w {} #1 {} ?! \q_recursion_tail ??? }
15948   { \use_i_delimit_by_q_recursion_stop:nw }
15949   { \use_none:n }
15950 }
15951 \cs_new:Npn \_quark_if_recursion_tail:w
15952   #1 \q_recursion_tail #2 ? #3 ?! { #1 #2 }
15953 \cs_generate_variant:Nn \quark_if_recursion_tail_stop:n { o }
15954 \cs_generate_variant:Nn \quark_if_recursion_tail_stop_do:nn { o }
```

(End definition for \quark_if_recursion_tail_stop:n, \quark_if_recursion_tail_stop_do:nn, and _quark_if_recursion_tail:w. These functions are documented on page 139.)

`\quark_if_recursion_tail_break:NN` Analogues of the `\quark_if_recursion_tail_stop...` functions. Break the mapping
`\quark_if_recursion_tail_break:nN` using #2.

```

15955 \cs_new:Npn \quark_if_recursion_tail_break:NN #1#2
15956 {
15957   \if_meaning:w \q_recursion_tail #1
15958   \exp_after:wN #2
15959   \fi:
15960 }
15961 \cs_new:Npn \quark_if_recursion_tail_break:nN #1#2
15962 {
15963   \tl_if_empty:oT
15964   { \__quark_if_recursion_tail:w {} #1 {} ?! \q_recursion_tail ??? }
15965   {#2}
15966 }

```

(End definition for `\quark_if_recursion_tail_break:NN` and `\quark_if_recursion_tail_break:nN`. These functions are documented on page 140.)

`\quark_if_nil_p:N` Here we test if we found a special quark as the first argument. We better start with
`\quark_if_nil:NTF` `\q_no_value` as the first argument since the whole thing may otherwise loop if #1 is
`\quark_if_no_value_p:N` wrongly given a string like aabc instead of a single token.⁹

```

15967 \prg_new_conditional:Npnn \quark_if_nil:N #1 { p, T, F, TF }
15968 {
15969   \if_meaning:w \q_nil #1
15970   \prg_return_true:
15971   \else:
15972   \prg_return_false:
15973   \fi:
15974 }
15975 \prg_new_conditional:Npnn \quark_if_no_value:N #1 { p, T, F, TF }
15976 {
15977   \if_meaning:w \q_no_value #1
15978   \prg_return_true:
15979   \else:
15980   \prg_return_false:
15981   \fi:
15982 }
15983 \prg_generate_conditional_variant:Nnn \quark_if_no_value:N
15984 { c } { p, T, F, TF }

```

(End definition for `\quark_if_nil:N` and `\quark_if_no_value:N`. These functions are documented on page 138.)

`\quark_if_nil_p:n` Let us explain `\quark_if_nil:nTF`. Expanding `__quark_if_nil:w` once is safe thanks
`\quark_if_nil_p:V` to the trailing `\q_nil ???`. The result of expanding once is empty if and only if both
`\quark_if_nil_p:o` delimited arguments #1 and #2 are empty and #3 is delimited by the last tokens ?!.
`\quark_if_nil:nTF` Thanks to the leading {}, the argument #1 is empty if and only if the argument of
`\quark_if_nil:VTF` `\quark_if_nil:n` starts with `\q_nil`. The argument #2 is empty if and only if this `\q_`
`\quark_if_nil:oTF` `nil` is followed immediately by ? or by {}?, coming either from the trailing tokens in the
`\quark_if_no_value_p:n` definition of `\quark_if_nil:n`, or from its argument. In the first case, `__quark_if_`
`\quark_if_no_value:nTF` `nil:w` is followed by `\q_nil {}? !\q_nil ???`, hence #3 is delimited by the final ?!,
`__quark_if_nil:w` and the test returns true as wanted. In the second case, the result is not empty since
`__quark_if_no_value:w`
`__quark_if_empty_if:o`

⁹It may still loop in special circumstances however!

the first ?! in the definition of `\quark_if_nil:n stop #3`. The auxiliary here is the same as `__tl_if_empty_if:o`, with the same comments applying.

```

15985 \prg_new_conditional:Npnn \quark_if_nil:n #1 { p, T , F , TF }
15986 {
15987   \__quark_if_empty_if:o
15988   { \__quark_if_nil:w {} #1 {} ? ! \q_nil ? ? ! }
15989   \prg_return_true:
15990   \else:
15991     \prg_return_false:
15992   \fi:
15993 }
15994 \cs_new:Npn \__quark_if_nil:w #1 \q_nil #2 ? #3 ? ! { #1 #2 }
15995 \prg_new_conditional:Npnn \quark_if_no_value:n #1 { p, T , F , TF }
15996 {
15997   \__quark_if_empty_if:o
15998   { \__quark_if_no_value:w {} #1 {} ? ! \q_no_value ? ? ! }
15999   \prg_return_true:
16000   \else:
16001     \prg_return_false:
16002   \fi:
16003 }
16004 \cs_new:Npn \__quark_if_no_value:w #1 \q_no_value #2 ? #3 ? ! { #1 #2 }
16005 \prg_generate_conditional_variant:Nnn \quark_if_nil:n
16006 { V , o } { p , TF , T , F }
16007 \cs_new:Npn \__quark_if_empty_if:o #1
16008 {
16009   \exp_after:wN \if_meaning:w \exp_after:wN \q_nil
16010   \__kernel_tl_to_str:w \exp_after:wN {#1} \q_nil
16011 }

```

(End definition for `\quark_if_nil:nTF` and others. These functions are documented on page 138.)

`__kernel_quark_new_test:N` The function `__kernel_quark_new_test:N` defines #1 in a similar way as `\quark_if_recursion_tail...` functions (as described below), using `\q_<namespace>_recursion_tail` as the test quark and `\q_<namespace>_recursion_stop` as the delimiter quark, where the `<namespace>` is determined as the first `_`-delimited part in #1.

There are six possible function types which this function can define, and which is defined depends on the signature of the function being defined:

`:n` gives an analogue of `\quark_if_recursion_tail_stop:n`
`:nn` gives an analogue of `\quark_if_recursion_tail_stop_do:nn`
`:nN` gives an analogue of `\quark_if_recursion_tail_break:nN`
`:N` gives an analogue of `\quark_if_recursion_tail_stop:N`
`:Nn` gives an analogue of `\quark_if_recursion_tail_stop_do:Nn`
`:NN` gives an analogue of `\quark_if_recursion_tail_break:NN`

Any other signature causes an error, as does a function without signature.

_kernel_quark_new_conditional:Nn

Similar to _kernel_quark_new_test:N, but defines quark branching conditionals like \quark_if_nil:nTF that test for the quark \q_<namespace>_<name>. The <namespace> and <name> are determined from the conditional #1, which must take the rather rigid form _<namespace>_quark_if_<name>:<arg spec>. There are only two cases for the <arg spec> here:

:n gives an analogue of \quark_if_nil:nTF

:N gives an analogue of \quark_if_nil:NTF

Any other signature causes an error, as does a function without signature. We use low-level emptiness tests as l3tl is not available yet when these functions are used; thankfully we only care about whether strings are empty so a simple \if_meaning:w \q_nil <string> \q_nil suffices.

```

16012 \cs_new_protected:Npn \_kernel\_quark\_new\_test:N #1
16013 { \_quark\_new\_test\_aux:Nx #1 { \_quark\_module\_name:N #1 } }
\_quark\_new\_test:NNNn
\_quark\_new\_test:Nccn
  \_quark\_new\_test\_aux:nnNNnnnn
  \_quark\_new\_conditional:Nnnn
  \_quark\_new\_conditional:Nxxn
16014 \cs_new_protected:Npn \_quark\_new\_test\_aux:Nn #1 #2
16015 {
16016   \if\_meaning:w \q\_nil #2 \q\_nil
16017     \msg\_error:nnx { quark } { invalid-function }
16018     { \token\_to\_str:N #1 }
16019   \else:
16020     \_quark\_new\_test:Nccn #1
16021     { q\_#2\_recursion\_tail } { q\_#2\_recursion\_stop } { \_#2 }
16022   \fi:
16023 }
16024 \cs\_generate\_variant:Nn \_quark\_new\_test\_aux:Nn { Nx }
16025 \cs\_new\_protected:Npn \_quark\_new\_test:NNNn #1
16026 {
16027   \exp\_last\_unbraced:Nf \_quark\_new\_test\_aux:nnNNnnnn
16028   { \cs\_split\_function:N #1 }
16029   #1 { test }
16030 }
16031 \cs\_generate\_variant:Nn \_quark\_new\_test:NNNn { Ncc }
16032 \cs\_new\_protected:Npn \_kernel\_quark\_new\_conditional:Nn #1
16033 {
16034   \_quark\_new\_conditional:Nxxn #1
16035   { \_quark\_quark\_conditional\_name:N #1 }
16036   { \_quark\_module\_name:N #1 }
16037 }
16038 \cs\_new\_protected:Npn \_quark\_new\_conditional:Nnnn #1#2#3#4
16039 {
16040   \if\_meaning:w \q\_nil #2 \q\_nil
16041     \msg\_error:nnx { quark } { invalid-function }
16042     { \token\_to\_str:N #1 }
16043   \else:
16044     \if\_meaning:w \q\_nil #3 \q\_nil
16045       \msg\_error:nnx { quark } { invalid-function }
16046       { \token\_to\_str:N #1 }
16047     \else:
16048       \exp\_last\_unbraced:Nf \_quark\_new\_test\_aux:nnNNnnnn
16049       { \cs\_split\_function:N #1 }

```

```

16050         #1 { conditional }
16051         {#2} {#3} {#4}
16052     \fi:
16053 \fi:
16054 }
16055 \cs_generate_variant:Nn \__quark_new_conditional:Nnnn { Nxx }
16056 \cs_new_protected:Npn \__quark_new_test_aux:nnNNnnnn #1 #2 #3 #4 #5
16057 {
16058     \cs_if_exist_use:cTF { __quark_new_#5_#2:Nnnn } { #4 }
16059     {
16060         \msg_error:nnxx { quark } { invalid-function }
16061         { \token_to_str:N #4 } {#2}
16062         \use_none:nnn
16063     }
16064 }

```

(End definition for __kernel_quark_new_test:N and others.)

__quark_new_test_n:Nnnn These macros implement the six possibilities mentioned above, passing the right arguments to __quark_new_test_aux_do:nNNnnnnNNn, which defines some auxiliaries, and then to __quark_new_test_define_tl:nNnNNn (:n(n) variants) or to __quark_new_test_define_ifx:nNnNNn (:N(n)) which define the main conditionals.

```

16065 \cs_new_protected:Npn \__quark_new_test_n:Nnnn #1 #2 #3 #4
16066 {
16067     \__quark_new_test_aux_do:nNNnnnnNNn {#4} #2 #3 { none } { } { } { }
16068     \__quark_new_test_define_tl:nNnNNn #1 { }
16069 }
16070 \cs_new_protected:Npn \__quark_new_test_nn:Nnnn #1 #2 #3 #4
16071 {
16072     \__quark_new_test_aux_do:nNNnnnnNNn {#4} #2 #3 { i } { n } {##1} {##2}
16073     \__quark_new_test_define_tl:nNnNNn #1 { \use_none:n }
16074 }
16075 \cs_new_protected:Npn \__quark_new_test_nN:Nnnn #1 #2 #3 #4
16076 {
16077     \__quark_new_test_aux_do:nNNnnnnNNn {#4} #2 #3 { i } { n } {##1} {##2}
16078     \__quark_new_test_define_break_tl:nNNNNn #1 { }
16079 }
16080 \cs_new_protected:Npn \__quark_new_test_N:Nnnn #1 #2 #3 #4
16081 {
16082     \__quark_new_test_aux_do:nNNnnnnNNn {#4} #2 #3 { none } { } { } { }
16083     \__quark_new_test_define_ifx:nNnNNn #1 { }
16084 }
16085 \cs_new_protected:Npn \__quark_new_test_Nn:Nnnn #1 #2 #3 #4
16086 {
16087     \__quark_new_test_aux_do:nNNnnnnNNn {#4} #2 #3 { i } { n } {##1} {##2}
16088     \__quark_new_test_define_ifx:nNnNNn #1
16089     { \else: \exp_after:wN \use_none:n }
16090 }
16091 \cs_new_protected:Npn \__quark_new_test_NN:Nnnn #1 #2 #3 #4
16092 {
16093     \__quark_new_test_aux_do:nNNnnnnNNn {#4} #2 #3 { i } { n } {##1} {##2}
16094     \__quark_new_test_define_break_ifx:nNNNNn #1 { }
16095 }

```

(End definition for __quark_new_test_n:Nnnn and others.)

_quark_new_test_aux_do:nNNnnnnNNn
_quark_test_define_aux:NNNNnnNNn

_quark_new_test_aux_do:nNNnnnnNNn makes the control sequence names which will be used by _quark_test_define_aux:NNNNnnNNn, and then later by _quark_new_test_define_tl:nNnNNn or _quark_new_test_define_ifx:nNnNNn. The control sequences defined here are analogous to _quark_if_recursion_tail:w and to \use_-(none|i)_delimit_by_q_recursion_stop:(|n)w.

The name is composed by the name-space and the name of the quarks. Suppose _kernel_quark_new_test:N was used with:

_kernel_quark_new_test:N _test_quark_tail:n

then the first auxiliary will be _test_quark_recursion_tail:w, and the second one will be _test_use_none_delimit_by_q_recursion_stop:w.

Note that the actual quarks are *not* defined here. They should be defined separately using \quark_new:N.

```
16096 \cs_new_protected:Npn \_quark_new_test_aux_do:nNNnnnnNNn #1 #2 #3 #4 #5
16097 {
16098   \exp_args:Ncc \_quark_test_define_aux:NNNNnnNNn
16099   { #1 \_quark_recursion_tail:w }
16100   { #1 \_use_ #4 \_delimit_by_q_recursion_stop: #5 w }
16101   #2 #3
16102 }
16103 \cs_new_protected:Npn \_quark_test_define_aux:NNNNnnNNn #1 #2 #3 #4 #5 #6 #7
16104 {
16105   \cs_gset:Npn #1 ##1 #3 ##2 ? ##3 ?! { ##1 ##2 }
16106   \cs_gset:Npn #2 ##1 #6 #4 {#5}
16107   #7 {##1} #1 #2 #3
16108 }
```

(End definition for _quark_new_test_aux_do:nNNnnnnNNn and _quark_test_define_aux:NNNNnnNNn.)

_quark_new_test_define_tl:nNnNNn
_quark_new_test_define_ifx:nNnNNn
_quark_new_test_define_break_tl:nNNNNn
_quark_new_test_define_break_ifx:nNNNNn

Finally, these two macros define the main conditional function using what's been set up before.

```
16109 \cs_new_protected:Npn \_quark_new_test_define_tl:nNnNNn #1 #2 #3 #4 #5 #6
16110 {
16111   \cs_new:Npn #5 #1
16112   {
16113     \tl_if_empty:oTF
16114     { #2 {} ##1 {} ?! #4 ??! }
16115     {#3} {#6}
16116   }
16117 }
16118 \cs_new_protected:Npn \_quark_new_test_define_ifx:nNnNNn #1 #2 #3 #4 #5 #6
16119 {
16120   \cs_new:Npn #5 #1
16121   {
16122     \if_meaning:w #4 ##1
16123     \exp_after:wN #3
16124     #6
16125     \fi:
16126   }
16127 }
16128 \cs_new_protected:Npn \_quark_new_test_define_break_tl:nNNNNn #1 #2 #3
16129 { \_quark_new_test_define_tl:nNnNNn {##1##2} #2 {##2} }
16130 \cs_new_protected:Npn \_quark_new_test_define_break_ifx:nNNNNn #1 #2 #3
```

```
16131 { \_quark_new_test_define_ifx:nNnNn {##1##2} #2 {##2} }
```

(End definition for _quark_new_test_define_tl:nNnNn and others.)

```
\_quark_new_conditional_n:Nnnn
\_quark_new_conditional_N:Nnnn
```

These macros implement the two possibilities for branching quark conditionals, passing the right arguments to _quark_new_conditional_aux_do:NNnnn, which defines some auxiliaries and defines the main conditionals.

```
16132 \cs_new_protected:Npn \_quark_new_conditional_n:Nnnn
16133 { \_quark_new_conditional_aux_do:NNnnn \use_i:nn }
16134 \cs_new_protected:Npn \_quark_new_conditional_N:Nnnn
16135 { \_quark_new_conditional_aux_do:NNnnn \use_ii:nn }
```

(End definition for _quark_new_conditional_n:Nnnn and _quark_new_conditional_N:Nnnn.)

```
\_quark_new_conditional_aux_do:NNnnn
\_quark_new_conditional_define:NNNNn
```

Similar to the previous macros, but branching conditionals only require one auxiliary, so we take a shortcut. In _quark_new_conditional_define:NNNNn, #4 is \use_i:nn to define the n-type function (which needs an auxiliary) and is \use_ii:nn to define the N-type function.

```
16136 \cs_new_protected:Npn \_quark_new_conditional_aux_do:NNnnn #1 #2 #3 #4
16137 {
16138   \exp_args:Ncc \_quark_new_conditional_define:NNNNn
16139   { __ #4 _if_quark_ #3 :w } { q__ #4 _ #3 } #2 #1
16140 }
16141 \cs_new_protected:Npn \_quark_new_conditional_define:NNNNn #1 #2 #3 #4 #5
16142 {
16143   #4 { \cs_gset:Npn #1 ##1 #2 ##2 ? ##3 ?! { ##1 ##2 } } { }
16144   \exp_args:Nno \use:n { \prg_new_conditional:Npnn #3 ##1 {#5} }
16145   {
16146     #4 { \_quark_if_empty_if:o { #1 {} ##1 {} ?! #2 ?! } }
16147     { \if_meaning:w #2 ##1 }
16148     \prg_return_true: \else: \prg_return_false: \fi:
16149   }
16150 }
```

(End definition for _quark_new_conditional_aux_do:NNnnn and _quark_new_conditional_define:NNNNn.)

```
\_quark_module_name:N
\_quark_module_name:w
\_quark_module_name_loop:w
\_quark_module_name_end:w
```

_quark_module_name:N takes a control sequence and returns its $\langle module \rangle$ name, determined as the first non-empty non-single-character word, separated by _ or :. These rules give the correct result for public functions $\langle module \rangle \dots$, private functions $_ \langle module \rangle \dots$, and variables such as $_l \langle module \rangle \dots$. If no valid module is found the result is an empty string. The approach is to first cut off everything after the (first) : if any is present, then repeatedly grab _-delimited words until finding one of length at least 2 (we use low-level tests as l3tl is not fully available when _kernel_quark_new_test:N is first used. If no $\langle module \rangle$ is found (such as in \: :n) we get the trailing marker \use_none:n {}, which expands to nothing.

```
16151 \cs_set:Npn \_quark_tmp:w #1#2
16152 {
16153   \cs_new:Npn \_quark_module_name:N ##1
16154   {
16155     \exp_last_unbraced:Nf \_quark_module_name:w
16156     { \cs_to_str:N ##1 } #1 \s__quark
16157   }
16158   \cs_new:Npn \_quark_module_name:w ##1 #1 ##2 \s__quark
16159   { \_quark_module_name_loop:w ##1 #2 \use_none:n { } #2 \s__quark }
```

```

16160 \cs_new:Npn \__quark_module_name_loop:w ##1 #2
16161 {
16162     \use_i_ii:nnn \if_meaning:w \prg_do_nothing:
16163     ##1 \prg_do_nothing: \prg_do_nothing:
16164     \exp_after:wN \__quark_module_name_loop:w
16165     \else:
16166     \__quark_module_name_end:w ##1
16167     \fi:
16168 }
16169 \cs_new:Npn \__quark_module_name_end:w
16170 ##1 \fi: ##2 \s__quark { \fi: ##1 }
16171 }
16172 \exp_after:wN \__quark_tmp:w \tl_to_str:n { : _ }

```

(End definition for __quark_module_name:N and others.)

__quark_quark_conditional_name:N determines the quark name that the quark conditional function ##1 queries, as the part of the function name between `_quark_if_` and the trailing `:`. Again we define it through `__quark_tmp:w`, which receives `:` as #1 and `_quark_if_` as #2. The auxiliary `__quark_quark_conditional_name:w` returns the part between the first `_quark_if_` and the next `:`, and we apply this auxiliary to the function name followed by `:` (in case the function name is lacking a signature), and `_quark_if_`: so that `__quark_quark_conditional_name:N` returns an empty string if `_quark_if_` is not present.

```

16173 \cs_set:Npn \__quark_tmp:w #1 #2 \s__quark
16174 {
16175     \cs_new:Npn \__quark_quark_conditional_name:N ##1
16176     {
16177         \exp_last_unbraced:Nf \__quark_quark_conditional_name:w
16178         { \cs_to_str:N ##1 } #1 #2 #1 \s__quark
16179     }
16180     \cs_new:Npn \__quark_quark_conditional_name:w
16181     ##1 #2 ##2 #1 ##3 \s__quark {##2}
16182 }
16183 \exp_after:wN \__quark_tmp:w \tl_to_str:n { : _quark_if_ } \s__quark

```

(End definition for __quark_quark_conditional_name:N and __quark_quark_conditional_name:w.)

55.2 Scan marks

```

16184 <@@=scan>

```

\g__scan_marks_tl The list of all scan marks currently declared. No `l3tl` yet, so define this by hand.

```

16185 \cs_gset:Npn \g__scan_marks_tl { }

```

(End definition for \g__scan_marks_tl.)

\scan_new:N Check whether the variable is already a scan mark, then declare it to be equal to `\scan_stop:` globally.

```

16186 \cs_new_protected:Npn \scan_new:N #1
16187 {
16188     \tl_if_in:NnTF \g__scan_marks_tl { #1 }
16189     {

```

```

16190     \msg_error:nxx { scanmark } { already-defined }
16191     { \token_to_str:N #1 }
16192   }
16193   {
16194     \tl_gput_right:Nn \g__scan_marks_tl {#1}
16195     \cs_new_eq:NN #1 \scan_stop:
16196   }
16197 }

```

(End definition for `\scan_new:N`. This function is documented on page 141.)

\s_stop We only declare one scan mark here, more can be defined by specific modules. Can't use `\scan_new:N` yet because `l3tl` isn't loaded, so define `\s_stop` by hand and add it to `\g__scan_marks_tl`. We also add `\s__quark` (declared earlier) to the pool here. Since it lives in a different namespace, a little `l3docstrip` cheating is necessary.

```

16198 \cs_new_eq:NN \s_stop \scan_stop:
16199 \cs_gset_nopar:Npx \g__scan_marks_tl
16200 {
16201   \exp_not:o \g__scan_marks_tl
16202   \s_stop
16203   <@@=quark>
16204   \s__quark
16205   <@@=scan>
16206 }

```

(End definition for `\s_stop`. This variable is documented on page 141.)

\use_none_delimit_by_s_stop:w Similar to `\use_none_delimit_by_q_stop:w`.

```

16207 \cs_new:Npn \use_none_delimit_by_s_stop:w #1 \s_stop { }

```

(End definition for `\use_none_delimit_by_s_stop:w`. This function is documented on page 141.)

```

16208 </package>

```

Chapter 56

l3seq implementation

The following test files are used for this code: *m3seq002*, *m3seq003*.

```
16209 <*package>
16210 <@@=seq>
```

A sequence is a control sequence whose top-level expansion is of the form “\s__seq __seq_item:n {<item₁>} ... __seq_item:n {<item_n>}”, with a leading scan mark followed by *n* items of the same form. An earlier implementation used the structure “\seq_elt:w <item₁> \seq_elt_end: ... \seq_elt:w <item_n> \seq_elt_end:”. This allowed rapid searching using a delimited function, but was not suitable for items containing {, } and # tokens, and also lead to the loss of surrounding braces around items

```
\_\_seq_item:n ★ \_\_seq_item:n {<item>}
```

The internal token used to begin each sequence entry. If expanded outside of a mapping or manipulation function, an error is raised. The definition should always be set globally.

```
\_\_seq_push_item_def:n \_\_seq_push_item_def:n {<code>}
\_\_seq_push_item_def:x
```

Saves the definition of __seq_item:n and redefines it to accept one parameter and expand to <code>. This function should always be balanced by use of __seq_pop_item_def:.

```
\_\_seq_pop_item_def: \_\_seq_pop_item_def:
```

Restores the definition of __seq_item:n most recently saved by __seq_push_item_def:n. This function should always be used in a balanced pair with __seq_push_item_def:n.

```
\s__seq This private scan mark.
16211 \scan_new:N \s__seq
(End definition for \s__seq.)
```

```
\s__seq_mark Private scan marks.
\s__seq_stop 16212 \scan_new:N \s__seq_mark
16213 \scan_new:N \s__seq_stop
```

(End definition for \s__seq_mark and \s__seq_stop.)

`__seq_item:n` The delimiter is always defined, but when used incorrectly simply removes its argument and hits an undefined control sequence to raise an error.

```
16214 \cs_new:Npn \__seq_item:n
16215 {
16216     \msg_expandable_error:nn { seq } { misused }
16217     \use_none:n
16218 }
```

(End definition for __seq_item:n.)

`\l__seq_internal_a_tl` Scratch space for various internal uses.

```
\l__seq_internal_b_tl
16219 \tl_new:N \l__seq_internal_a_tl
16220 \tl_new:N \l__seq_internal_b_tl
```

(End definition for \l__seq_internal_a_tl and \l__seq_internal_b_tl.)

`__seq_tmp:w` Scratch function for internal use.

```
16221 \cs_new_eq:NN \__seq_tmp:w ?
```

(End definition for __seq_tmp:w.)

`\c_empty_seq` A sequence with no item, following the structure mentioned above.

```
16222 \tl_const:Nn \c_empty_seq { \s_seq }
```

(End definition for \c_empty_seq. This variable is documented on page 153.)

56.1 Allocation and initialisation

`\seq_new:N` Sequences are initialized to `\c_empty_seq`.

```
\seq_new:c
16223 \cs_new_protected:Npn \seq_new:N #1
16224 {
16225     \__kernel_chk_if_free_cs:N #1
16226     \cs_gset_eq:NN #1 \c_empty_seq
16227 }
16228 \cs_generate_variant:Nn \seq_new:N { c }
```

(End definition for \seq_new:N. This function is documented on page 142.)

`\seq_clear:N` Clearing a sequence is similar to setting it equal to the empty one.

```
\seq_clear:c
16229 \cs_new_protected:Npn \seq_clear:N #1
\seq_gclear:N
16230 { \seq_set_eq:NN #1 \c_empty_seq }
\seq_gclear:c
16231 \cs_generate_variant:Nn \seq_clear:N { c }
16232 \cs_new_protected:Npn \seq_gclear:N #1
16233 { \seq_gset_eq:NN #1 \c_empty_seq }
16234 \cs_generate_variant:Nn \seq_gclear:N { c }
```

(End definition for \seq_clear:N and \seq_gclear:N. These functions are documented on page 142.)

`\seq_clear_new:N` Once again we copy code from the token list functions.

```
\seq_clear_new:c
16235 \cs_new_protected:Npn \seq_clear_new:N #1
\seq_gclear_new:N
16236 { \seq_if_exist:NTF #1 { \seq_clear:N #1 } { \seq_new:N #1 } }
\seq_gclear_new:c
16237 \cs_generate_variant:Nn \seq_clear_new:N { c }
16238 \cs_new_protected:Npn \seq_gclear_new:N #1
16239 { \seq_if_exist:NTF #1 { \seq_gclear:N #1 } { \seq_new:N #1 } }
16240 \cs_generate_variant:Nn \seq_gclear_new:N { c }
```

(End definition for `\seq_clear_new:N` and `\seq_gclear_new:N`. These functions are documented on page 142.)

`\seq_set_eq:NN` Copying a sequence is the same as copying the underlying token list.
`\seq_set_eq:cN`
`\seq_set_eq:Nc`
`\seq_set_eq:cc`
`\seq_gset_eq:NN`
`\seq_gset_eq:cN`
`\seq_gset_eq:Nc`
`\seq_gset_eq:cc`

(End definition for `\seq_set_eq:NN` and `\seq_gset_eq:NN`. These functions are documented on page 142.)

`\seq_set_from_clist:NN` Setting a sequence from a comma-separated list is done using a simple mapping.
`\seq_set_from_clist:cN`
`\seq_set_from_clist:Nc`
`\seq_set_from_clist:cc`
`\seq_set_from_clist:Nn`
`\seq_set_from_clist:cn`
`\seq_gset_from_clist:NN`
`\seq_gset_from_clist:cN`
`\seq_gset_from_clist:Nc`
`\seq_gset_from_clist:cc`
`\seq_gset_from_clist:Nn`
`\seq_gset_from_clist:cn`

(End definition for `\seq_set_from_clist:NN` and others. These functions are documented on page 143.)

`\seq_const_from_clist:Nn` Almost identical to `\seq_set_from_clist:Nn`.
`\seq_const_from_clist:cn`

(End definition for `\seq_const_from_clist:Nn`. This function is documented on page 143.)

```

\seq_set_split:Nnn
\seq_set_split:NnV
\seq_gset_split:Nnn
\seq_gset_split:NnV
\seq_set_split_keep_spaces:Nnn
\seq_set_split_keep_spaces:NnV
\seq_gset_split_keep_spaces:Nnn
\seq_gset_split_keep_spaces:NnV
\__seq_set_split:NNnn
\__seq_set_split:Nw
\__seq_set_split:w
\__seq_set_split_end:

```

When the separator is empty, everything is very simple, just map `__seq_wrap_item:n` through the items of the last argument. For non-trivial separators, the goal is to split a given token list at the marker, strip spaces from each item, and remove one set of outer braces if after removing leading and trailing spaces the item is enclosed within braces. After `\tl_replace_all:Nnn`, the token list `\l__seq_internal_a_tl` is a repetition of the pattern `__seq_set_split_auxi:w \prg_do_nothing: <item with spaces> __seq_set_split_end:.` Then, x-expansion causes `__seq_set_split_auxi:w` to trim spaces, and leaves its result as `__seq_set_split_auxii:w <trimmed item> __seq_set_split_end:.` This is then converted to the `l3seq` internal structure by another x-expansion. In the first step, we insert `\prg_do_nothing:` to avoid losing braces too early: that would cause space trimming to act within those lost braces. The second step is solely there to strip braces which are outermost after space trimming.

```

16281 \cs_new_protected:Npn \seq_set_split:Nnn
16282 { \__seq_set_split:NNNnn \__kernel_tl_set:Nx \tl_trim_spaces:n }
16283 \cs_new_protected:Npn \seq_gset_split:Nnn
16284 { \__seq_set_split:NNNnn \__kernel_tl_gset:Nx \tl_trim_spaces:n }
16285 \cs_new_protected:Npn \seq_set_split_keep_spaces:Nnn
16286 { \__seq_set_split:NNNnn \__kernel_tl_set:Nx \exp_not:n }
16287 \cs_new_protected:Npn \seq_gset_split_keep_spaces:Nnn
16288 { \__seq_set_split:NNNnn \__kernel_tl_gset:Nx \exp_not:n }
16289 \cs_new_protected:Npn \__seq_set_split:NNNnn #1#2#3#4#5
16290 {
16291   \tl_if_empty:nTF {#4}
16292   {
16293     \tl_set:Nn \l__seq_internal_a_tl
16294       { \tl_map_function:nN {#5} \__seq_wrap_item:n }
16295   }
16296   {
16297     \tl_set:Nn \l__seq_internal_a_tl
16298     {
16299       \__seq_set_split:Nw #2 \prg_do_nothing:
16300       #5
16301       \__seq_set_split_end:
16302     }
16303     \tl_replace_all:Nnn \l__seq_internal_a_tl {#4}
16304     {
16305       \__seq_set_split_end:
16306       \__seq_set_split:Nw #2 \prg_do_nothing:
16307     }
16308     \__kernel_tl_set:Nx \l__seq_internal_a_tl { \l__seq_internal_a_tl }
16309   }
16310   #1 #3 { \s_seq \l__seq_internal_a_tl }
16311 }
16312 \cs_new:Npn \__seq_set_split:Nw #1#2 \__seq_set_split_end:
16313 {
16314   \exp_not:N \__seq_set_split:w
16315   \exp_args:No #1 {#2}
16316   \exp_not:N \__seq_set_split_end:
16317 }
16318 \cs_new:Npn \__seq_set_split:w #1 \__seq_set_split_end:
16319 { \__seq_wrap_item:n {#1} }
16320 \cs_generate_variant:Nn \seq_set_split:Nnn { NnV }
16321 \cs_generate_variant:Nn \seq_gset_split:Nnn { NnV }

```



```

16322 \cs_generate_variant:Nn \seq_set_split_keep_spaces:Nnn { NnV }
16323 \cs_generate_variant:Nn \seq_gset_split_keep_spaces:Nnn { NnV }

```

(End definition for `\seq_set_split:Nnn` and others. These functions are documented on page 143.)

```

\seq_concat:NNN When concatenating sequences, one must remove the leading \s__seq of the second
\seq_concat:ccc sequence. The result starts with \s__seq (of the first sequence), which stops f-expansion.
\seq_gconcat:NNN
\seq_gconcat:ccc
16324 \cs_new_protected:Npn \seq_concat:NNN #1#2#3
16325 { \tl_set:Nf #1 { \exp_after:wN \use_i:nn \exp_after:wN #2 #3 } }
16326 \cs_new_protected:Npn \seq_gconcat:NNN #1#2#3
16327 { \tl_gset:Nf #1 { \exp_after:wN \use_i:nn \exp_after:wN #2 #3 } }
16328 \cs_generate_variant:Nn \seq_concat:NNN { ccc }
16329 \cs_generate_variant:Nn \seq_gconcat:NNN { ccc }

```

(End definition for `\seq_concat:NNN` and `\seq_gconcat:NNN`. These functions are documented on page 143.)

```

\seq_if_exist_p:N Copies of the cs functions defined in l3basics.
\seq_if_exist_p:c
\seq_if_exist:NTF
\seq_if_exist:cTF
16330 \prg_new_eq_conditional:Nnn \seq_if_exist:N \cs_if_exist:N
16331 { TF , T , F , p }
16332 \prg_new_eq_conditional:Nnn \seq_if_exist:c \cs_if_exist:c
16333 { TF , T , F , p }

```

(End definition for `\seq_if_exist:N`. This function is documented on page 144.)

56.2 Appending data to either end

```

\seq_put_left:Nn When adding to the left of a sequence, remove \s__seq. This is done by \__seq_put_
\seq_put_left:NV left_aux:w, which also stops f-expansion.
\seq_put_left:Nv
\seq_put_left:No
\seq_put_left:Nx
\seq_put_left:cn
\seq_put_left:cV
\seq_put_left:cv
\seq_put_left:co
\seq_put_left:cx
16334 \cs_new_protected:Npn \seq_put_left:Nn #1#2
16335 {
16336   \__kernel_tl_set:Nx #1
16337   {
16338     \exp_not:n { \s__seq \__seq_item:n {#2} }
16339     \exp_not:f { \exp_after:wN \__seq_put_left_aux:w #1 }
16340   }
16341 }
16342 \cs_new_protected:Npn \seq_gput_left:Nn #1#2
16343 {
16344   \__kernel_tl_gset:Nx #1
16345   {
16346     \exp_not:n { \s__seq \__seq_item:n {#2} }
16347     \exp_not:f { \exp_after:wN \__seq_put_left_aux:w #1 }
16348   }
16349 }
16350 \cs_new:Npn \__seq_put_left_aux:w \s__seq { \exp_stop_f: }
16351 \cs_generate_variant:Nn \seq_put_left:Nn { NV , Nv , No , Nx }
16352 \cs_generate_variant:Nn \seq_put_left:Nn { c , cV , cv , co , cx }
16353 \cs_generate_variant:Nn \seq_gput_left:Nn { NV , Nv , No , Nx }
16354 \cs_generate_variant:Nn \seq_gput_left:Nn { c , cV , cv , co , cx }
\__seq_put_left_aux:w

```

(End definition for `\seq_put_left:Nn`, `\seq_gput_left:Nn`, and `__seq_put_left_aux:w`. These functions are documented on page 144.)

`\seq_put_right:Nn` Since there is no trailing marker, adding an item to the right of a sequence simply means wrapping it in `__seq_item:n`.

```

\seq_put_right:NV
\seq_put_right:Nv
\seq_put_right:No
\seq_put_right:Nx
\seq_put_right:cn
\seq_put_right:cV
\seq_put_right:cv
\seq_put_right:co
\seq_put_right:cx
16355 \cs_new_protected:Npn \seq_put_right:Nn #1#2
16356 { \tl_put_right:Nn #1 { \__seq_item:n {#2} } }
\seq_put_right:Nx
16357 \cs_new_protected:Npn \seq_gput_right:Nn #1#2
16358 { \tl_gput_right:Nn #1 { \__seq_item:n {#2} } }
\seq_put_right:cV
16359 \cs_generate_variant:Nn \seq_gput_right:Nn { NV , Nv , No , Nx }
\seq_put_right:cv
16360 \cs_generate_variant:Nn \seq_gput_right:Nn { c , cV , cv , co , cx }
\seq_put_right:co
16361 \cs_generate_variant:Nn \seq_put_right:Nn { NV , Nv , No , Nx }
\seq_put_right:cx
16362 \cs_generate_variant:Nn \seq_put_right:Nn { c , cV , cv , co , cx }

```

(End definition for `\seq_put_right:Nn` and `\seq_gput_right:Nn`. These functions are documented on page 144.)

56.3 Modifying sequences

This function converts its argument to a proper sequence item in an x-expansion context.

```

\seq_wrap_item:n
16363 \cs_new:Npn \__seq_wrap_item:n #1 { \exp_not:n { \__seq_item:n {#1} } }
\seq_gput_right:cV
\seq_gput_right:cv
\seq_gput_right:co
\seq_gput_right:cx
\l__seq_remove_seq

```

(End definition for `__seq_wrap_item:n`.)

An internal sequence for the removal routines.

```
16364 \seq_new:N \l__seq_remove_seq
```

(End definition for `\l__seq_remove_seq`.)

`\seq_remove_duplicates:N` Removing duplicates means making a new list then copying it.

```

\seq_remove_duplicates:c
\seq_gremove_duplicates:N
\seq_gremove_duplicates:c
\__seq_remove_duplicates:NN
16365 \cs_new_protected:Npn \seq_remove_duplicates:N
16366 { \__seq_remove_duplicates:NN \seq_set_eq:NN }
16367 \cs_new_protected:Npn \seq_gremove_duplicates:N
16368 { \__seq_remove_duplicates:NN \seq_gset_eq:NN }
16369 \cs_new_protected:Npn \__seq_remove_duplicates:NN #1#2
16370 {
16371   \seq_clear:N \l__seq_remove_seq
16372   \seq_map_inline:Nn #2
16373   {
16374     \seq_if_in:NnF \l__seq_remove_seq {##1}
16375     { \seq_put_right:Nn \l__seq_remove_seq {##1} }
16376   }
16377   #1 #2 \l__seq_remove_seq
16378 }
16379 \cs_generate_variant:Nn \seq_remove_duplicates:N { c }
16380 \cs_generate_variant:Nn \seq_gremove_duplicates:N { c }

```

(End definition for `\seq_remove_duplicates:N`, `\seq_gremove_duplicates:N`, and `__seq_remove_duplicates:NN`. These functions are documented on page 147.)

`\seq_remove_all:Nn` The idea of the code here is to avoid a relatively expensive addition of items one at a time to an intermediate sequence. The approach taken is therefore similar to that in `__seq_pop_right:NNN`, using a “flexible” x-type expansion to do most of the work. As `\tl_if_eq:nnT` is not expandable, a two-part strategy is needed. First, the x-type expansion uses `\str_if_eq:nnT` to find potential matches. If one is found, the expansion is halted and the necessary set up takes place to use the `\tl_if_eq:NNT` test. The x-type is started

again, including all of the items copied already. This happens repeatedly until the entire sequence has been scanned. The code is set up to avoid needing and intermediate scratch list: the lead-off x-type expansion (#1 #2 {#2}) ensures that nothing is lost.

```

16381 \cs_new_protected:Npn \seq_remove_all:Nn
16382 { \__seq_remove_all_aux:NNn \__kernel_tl_set:Nx }
16383 \cs_new_protected:Npn \seq_gremove_all:Nn
16384 { \__seq_remove_all_aux:NNn \__kernel_tl_gset:Nx }
16385 \cs_new_protected:Npn \__seq_remove_all_aux:NNn #1#2#3
16386 {
16387   \__seq_push_item_def:n
16388   {
16389     \str_if_eq:nnT {##1} {#3}
16390     {
16391       \if_false: { \fi: }
16392       \tl_set:Nn \l__seq_internal_b_tl {##1}
16393       #1 #2
16394       { \if_false: } \fi:
16395       \exp_not:o {#2}
16396       \tl_if_eq:NNT \l__seq_internal_a_tl \l__seq_internal_b_tl
16397       { \use_none:nn }
16398     }
16399     \__seq_wrap_item:n {##1}
16400   }
16401   \tl_set:Nn \l__seq_internal_a_tl {#3}
16402   #1 #2 {#2}
16403   \__seq_pop_item_def:
16404 }
16405 \cs_generate_variant:Nn \seq_remove_all:Nn { c }
16406 \cs_generate_variant:Nn \seq_gremove_all:Nn { c }

```

(End definition for `\seq_remove_all:Nn`, `\seq_gremove_all:Nn`, and `__seq_remove_all_aux:NNn`. These functions are documented on page 147.)

<pre> \seq_reverse:N \seq_reverse:c \seq_greverse:N \seq_greverse:c __seq_reverse:NN __seq_reverse_item:nwn </pre>	<p>Previously, <code>\seq_reverse:N</code> was coded by collecting the items in reverse order after an <code>\exp_stop_f:</code> marker.</p> <pre> \cs_new_protected:Npn \seq_reverse:N #1 { \cs_set_eq:NN \@@_item:n \@@_reverse_item:nw \tl_set:Nf #2 { #2 \exp_stop_f: } } \cs_new:Npn \@@_reverse_item:nw #1 #2 \exp_stop_f: { #2 \exp_stop_f: \@@_item:n {#1} } </pre>
--	---

At first, this seems optimal, since we can forget about each item as soon as it is placed after `\exp_stop_f:`. Unfortunately, \TeX 's usual tail recursion does not take place in this case: since the following `__seq_reverse_item:nw` only reads tokens until `\exp_stop_f:`, and never reads the `\@@_item:n {#1}` left by the previous call, \TeX cannot remove that previous call from the stack, and in particular must retain the various macro parameters in memory, until the end of the replacement text is reached. The stack is thus

only flushed after all the `__seq_reverse_item:nw` are expanded. Keeping track of the arguments of all those calls uses up a memory quadratic in the length of the sequence. \TeX can then not cope with more than a few thousand items.

Instead, we collect the items in the argument of `\exp_not:n`. The previous calls are cleanly removed from the stack, and the memory consumption becomes linear.

```

16407 \cs_new_protected:Npn \seq_reverse:N
16408 { \__seq_reverse:NN \__kernel_tl_set:Nx }
16409 \cs_new_protected:Npn \seq_greverse:N
16410 { \__seq_reverse:NN \__kernel_tl_gset:Nx }
16411 \cs_new_protected:Npn \__seq_reverse:NN #1 #2
16412 {
16413   \cs_set_eq:NN \__seq_tmp:w \__seq_item:n
16414   \cs_set_eq:NN \__seq_item:n \__seq_reverse_item:nwn
16415   #1 #2 { #2 \exp_not:n { } }
16416   \cs_set_eq:NN \__seq_item:n \__seq_tmp:w
16417 }
16418 \cs_new:Npn \__seq_reverse_item:nwn #1 #2 \exp_not:n #3
16419 {
16420   #2
16421   \exp_not:n { \__seq_item:n {#1} #3 }
16422 }
16423 \cs_generate_variant:Nn \seq_reverse:N { c }
16424 \cs_generate_variant:Nn \seq_greverse:N { c }

```

(End definition for `\seq_reverse:N` and others. These functions are documented on page 147.)

`\seq_sort:Nn` Implemented in `l3sort`.

`\seq_sort:cn`

(End definition for `\seq_sort:Nn` and `\seq_gsort:Nn`. These functions are documented on page 147.)

`\seq_gsort:Nn`

`\seq_gsort:cn`

56.4 Sequence conditionals

`\seq_if_empty_p:N` Similar to token lists, we compare with the empty sequence.

`\seq_if_empty_p:c`

`\seq_if_empty:NTF`

`\seq_if_empty:cTF`

```

16425 \prg_new_conditional:Npnn \seq_if_empty:N #1 { p , T , F , TF }
16426 {
16427   \if_meaning:w #1 \c_empty_seq
16428   \prg_return_true:
16429   \else:
16430     \prg_return_false:
16431   \fi:
16432 }
16433 \prg_generate_conditional_variant:Nnn \seq_if_empty:N
16434 { c } { p , T , F , TF }

```

(End definition for `\seq_if_empty:N`. This function is documented on page 147.)

`\seq_shuffle:N`

`\seq_shuffle:c`

`\seq_gshuffle:N`

`\seq_gshuffle:c`

`__seq_shuffle:NN`

`__seq_shuffle_item:n`

`\g__seq_internal_seq`

We apply the Fisher–Yates shuffle, storing items in `\toks` registers. We use the primitive `\tex_uniformdeviate:D` for speed reasons. Its non-uniformity is of order its argument divided by 2^{28} , not too bad for small lists. For sequences with more than 13 elements there are more possible permutations than possible seeds ($13! > 2^{28}$) so the question of uniformity is somewhat moot. The integer variables are declared in `l3int`: load-order issues.

```

16435 \cs_if_exist:NTF \tex_uniformdeviate:D

```

```

16436 {
16437   \seq_new:N \g__seq_internal_seq
16438   \cs_new_protected:Npn \seq_shuffle:N { \__seq_shuffle:NN \seq_set_eq:NN }
16439   \cs_new_protected:Npn \seq_gshuffle:N { \__seq_shuffle:NN \seq_gset_eq:NN }
16440   \cs_new_protected:Npn \__seq_shuffle:NN #1#2
16441   {
16442     \int_compare:nNnTF { \seq_count:N #2 } > \c_max_register_int
16443     {
16444       \msg_error:nnx { seq } { shuffle-too-large }
16445       { \token_to_str:N #2 }
16446     }
16447     {
16448       \group_begin:
16449       \int_zero:N \l__seq_internal_a_int
16450       \__seq_push_item_def:
16451       \cs_gset_eq:NN \__seq_item:n \__seq_shuffle_item:n
16452       #2
16453       \__seq_pop_item_def:
16454       \seq_gset_from_inline_x:Nnn \g__seq_internal_seq
16455       { \int_step_function:nN { \l__seq_internal_a_int } }
16456       { \tex_the:D \tex_toks:D ##1 }
16457       \group_end:
16458       #1 #2 \g__seq_internal_seq
16459       \seq_gclear:N \g__seq_internal_seq
16460     }
16461   }
16462   \cs_new_protected:Npn \__seq_shuffle_item:n
16463   {
16464     \int_incr:N \l__seq_internal_a_int
16465     \int_set:Nn \l__seq_internal_b_int
16466     { 1 + \tex_uniformdeviate:D \l__seq_internal_a_int }
16467     \tex_toks:D \l__seq_internal_a_int
16468     = \tex_toks:D \l__seq_internal_b_int
16469     \tex_toks:D \l__seq_internal_b_int
16470   }
16471 }
16472 {
16473   \cs_new_protected:Npn \seq_shuffle:N #1
16474   {
16475     \msg_error:nnn { kernel } { fp-no-random }
16476     { \seq_shuffle:N #1 }
16477   }
16478   \cs_new_eq:NN \seq_gshuffle:N \seq_shuffle:N
16479 }
16480 \cs_generate_variant:Nn \seq_shuffle:N { c }
16481 \cs_generate_variant:Nn \seq_gshuffle:N { c }

```

(End definition for `\seq_shuffle:N` and others. These functions are documented on page 147.)

`\seq_if_in:NnTF` The approach here is to define `__seq_item:n` to compare its argument with the test sequence. If the two items are equal, the mapping is terminated and `\group_end: \prg_return_true:` is inserted after skipping over the rest of the recursion. On the other hand, if there is no match then the loop breaks, returning `\prg_return_false:.` Everything is inside a group so that `__seq_item:n` is preserved in nested situations.

`\seq_if_in:NvTF`

`\seq_if_in:NvTF`

`\seq_if_in:NoTF`

`\seq_if_in:NxTF`

`\seq_if_in:cnTF`

`\seq_if_in:cVTF`

`\seq_if_in:cvTF`

`\seq_if_in:coTF`

`\seq_if_in:cxTF`

`__seq_if_in:`

```

16482 \prg_new_protected_conditional:Npnn \seq_if_in:Nn #1#2
16483 { T , F , TF }
16484 {
16485   \group_begin:
16486     \tl_set:Nn \l__seq_internal_a_tl {#2}
16487     \cs_set_protected:Npn \__seq_item:n ##1
16488     {
16489       \tl_set:Nn \l__seq_internal_b_tl {##1}
16490       \if_meaning:w \l__seq_internal_a_tl \l__seq_internal_b_tl
16491       \exp_after:wN \__seq_if_in:
16492       \fi:
16493     }
16494     #1
16495   \group_end:
16496   \prg_return_false:
16497   \prg_break_point:
16498 }
16499 \cs_new:Npn \__seq_if_in:
16500 { \prg_break:n { \group_end: \prg_return_true: } }
16501 \prg_generate_conditional_variant:Nnn \seq_if_in:Nn
16502 { NV , Nv , No , Nx , c , cV , cv , co , cx } { T , F , TF }

```

(End definition for `\seq_if_in:NnTF` and `__seq_if_in:..` This function is documented on page 148.)

56.5 Recovering data from sequences

`__seq_pop:NNNN`
`__seq_pop_TF:NNNN`

The two `pop` functions share their emptiness tests. We also use a common emptiness test for all branching `get` and `pop` functions.

```

16503 \cs_new_protected:Npn \__seq_pop:NNNN #1#2#3#4
16504 {
16505   \if_meaning:w #3 \c_empty_seq
16506   \tl_set:Nn #4 { \q_no_value }
16507   \else:
16508     #1#2#3#4
16509   \fi:
16510 }
16511 \cs_new_protected:Npn \__seq_pop_TF:NNNN #1#2#3#4
16512 {
16513   \if_meaning:w #3 \c_empty_seq
16514   % \tl_set:Nn #4 { \q_no_value }
16515   \prg_return_false:
16516   \else:
16517     #1#2#3#4
16518   \prg_return_true:
16519   \fi:
16520 }

```

(End definition for `__seq_pop:NNNN` and `__seq_pop_TF:NNNN`.)

`\seq_get_left:NN`
`\seq_get_left:cN`
`__seq_get_left:wnw`

Getting an item from the left of a sequence is pretty easy: just trim off the first item after `__seq_item:n` at the start. We append a `\q_no_value` item to cover the case of an empty sequence

```

16521 \cs_new_protected:Npn \seq_get_left:NN #1#2

```

```

16522 {
16523   \__kernel_tl_set:Nx #2
16524   {
16525     \exp_after:wN \__seq_get_left:wnw
16526     #1 \__seq_item:n { \q_no_value } \s__seq_stop
16527   }
16528 }
16529 \cs_new:Npn \__seq_get_left:wnw #1 \__seq_item:n #2#3 \s__seq_stop
16530 { \exp_not:n {#2} }
16531 \cs_generate_variant:Nn \seq_get_left:NN { c }

```

(End definition for \seq_get_left:NN and __seq_get_left:wnw. This function is documented on page 144.)

```

\seq_pop_left:NN
\seq_pop_left:cN
\seq_gpop_left:NN
\seq_gpop_left:cN
\__seq_pop_left:NNN
\__seq_pop_left:wnwNNN

```

The approach to popping an item is pretty similar to that to get an item, with the only difference being that the sequence itself has to be redefined. This makes it more sensible to use an auxiliary function for the local and global cases.

```

16532 \cs_new_protected:Npn \seq_pop_left:NN
16533 { \__seq_pop:NNNN \__seq_pop_left:NNN \tl_set:Nn }
16534 \cs_new_protected:Npn \seq_gpop_left:NN
16535 { \__seq_pop:NNNN \__seq_pop_left:NNN \tl_gset:Nn }
16536 \cs_new_protected:Npn \__seq_pop_left:NNN #1#2#3
16537 { \exp_after:wN \__seq_pop_left:wnwNNN #2 \s__seq_stop #1#2#3 }
16538 \cs_new_protected:Npn \__seq_pop_left:wnwNNN
16539 #1 \__seq_item:n #2#3 \s__seq_stop #4#5#6
16540 {
16541   #4 #5 { #1 #3 }
16542   \tl_set:Nn #6 {#2}
16543 }
16544 \cs_generate_variant:Nn \seq_pop_left:NN { c }
16545 \cs_generate_variant:Nn \seq_gpop_left:NN { c }

```

(End definition for \seq_pop_left:NN and others. These functions are documented on page 144.)

```

\seq_get_right:NN
\seq_get_right:cN
\__seq_get_right_loop:nw
\__seq_get_right_end:NnN

```

First remove \s__seq and prepend \q_no_value. The first argument of __seq_get_right_loop:nw is the last item found, and the second argument is empty until the end of the loop, where it is code that applies \exp_not:n to the last item and ends the loop.

```

16546 \cs_new_protected:Npn \seq_get_right:NN #1#2
16547 {
16548   \__kernel_tl_set:Nx #2
16549   {
16550     \exp_after:wN \use_i_ii:nnn
16551     \exp_after:wN \__seq_get_right_loop:nw
16552     \exp_after:wN \q_no_value
16553     #1
16554     \__seq_get_right_end:NnN \__seq_item:n
16555   }
16556 }
16557 \cs_new:Npn \__seq_get_right_loop:nw #1#2 \__seq_item:n
16558 {
16559   #2 \use_none:n {#1}
16560   \__seq_get_right_loop:nw
16561 }
16562 \cs_new:Npn \__seq_get_right_end:NnN #1#2#3 { \exp_not:n {#2} }
16563 \cs_generate_variant:Nn \seq_get_right:NN { c }

```

(End definition for `\seq_get_right:NN`, `__seq_get_right_loop:nw`, and `__seq_get_right_end:NnN`. This function is documented on page 144.)

`\seq_pop_right:NN` The approach to popping from the right is a bit more involved, but does use some of the same ideas as getting from the right. What is needed is a “flexible length” way to set a token list variable. This is supplied by the `{\if_false:}\fi:` `\seq_gpop_right:NN` `...\if_false:{\fi:}` construct. Using an x-type expansion and a “non-expanding” `\seq_gpop_right:cN` definition for `__seq_item:n`, the left-most $n - 1$ entries in a sequence of n items are stored back in the sequence. That needs a loop of unknown length, hence using the strange `\if_false:` way of including braces. When the last item of the sequence is reached, the closing brace for the assignment is inserted, and `\tl_set:Nn #3` is inserted in front of the final entry. This therefore does the pop assignment. One more iteration is performed, with an empty argument and `\use_none:nn`, which finally stops the loop.

```

16564 \cs_new_protected:Npn \seq_pop_right:NN
16565   { \__seq_pop:NNNN \__seq_pop_right:NNN \__kernel_tl_set:Nx }
16566 \cs_new_protected:Npn \seq_gpop_right:NN
16567   { \__seq_pop:NNNN \__seq_pop_right:NNN \__kernel_tl_gset:Nx }
16568 \cs_new_protected:Npn \__seq_pop_right:NNN #1#2#3
16569   {
16570     \cs_set_eq:NN \__seq_tmp:w \__seq_item:n
16571     \cs_set_eq:NN \__seq_item:n \scan_stop:
16572     #1 #2
16573     { \if_false: } \fi: \s__seq
16574     \exp_after:wN \use_i:nnn
16575     \exp_after:wN \__seq_pop_right_loop:nn
16576     #2
16577     {
16578       \if_false: { \fi: }
16579       \__kernel_tl_set:Nx #3
16580     }
16581     { } \use_none:nn
16582     \cs_set_eq:NN \__seq_item:n \__seq_tmp:w
16583   }
16584 \cs_new:Npn \__seq_pop_right_loop:nn #1#2
16585   {
16586     #2 { \exp_not:n {#1} }
16587     \__seq_pop_right_loop:nn
16588   }
16589 \cs_generate_variant:Nn \seq_pop_right:NN { c }
16590 \cs_generate_variant:Nn \seq_gpop_right:NN { c }

```

(End definition for `\seq_pop_right:NN` and others. These functions are documented on page 145.)

`\seq_get_left:NNTF` Getting from the left or right with a check on the results. The first argument to `__seq_pop_TF:NNNN` is left unused.

```

\seq_get_left:cNTF
\seq_get_right:NNTF
\seq_get_right:cNTF
16591 \prg_new_protected_conditional:Npnn \seq_get_left:NN #1#2 { T , F , TF }
16592   { \__seq_pop_TF:NNNN \prg_do_nothing: \seq_get_left:NN #1#2 }
16593 \prg_new_protected_conditional:Npnn \seq_get_right:NN #1#2 { T , F , TF }
16594   { \__seq_pop_TF:NNNN \prg_do_nothing: \seq_get_right:NN #1#2 }
16595 \prg_generate_conditional_variant:Nnn \seq_get_left:NN
16596   { c } { T , F , TF }
16597 \prg_generate_conditional_variant:Nnn \seq_get_right:NN
16598   { c } { T , F , TF }

```


(End definition for `\seq_get_left:NNTF` and `\seq_get_right:NNTF`. These functions are documented on page 146.)

```

\seq_pop_left:NNTF More or less the same for popping.
\seq_pop_left:cNTF 16599 \prg_new_protected_conditional:Npnn \seq_pop_left:NN #1#2
\seq_gpop_left:NNTF 16600 { T , F , TF }
\seq_gpop_left:cNTF 16601 { \__seq_pop_TF:NNNN \__seq_pop_left:NNN \tl_set:Nn #1 #2 }
\seq_pop_right:NNTF 16602 \prg_new_protected_conditional:Npnn \seq_gpop_left:NN #1#2
\seq_pop_right:cNTF 16603 { T , F , TF }
\seq_gpop_right:NNTF 16604 { \__seq_pop_TF:NNNN \__seq_pop_left:NNN \tl_gset:Nn #1 #2 }
\seq_gpop_right:cNTF 16605 \prg_new_protected_conditional:Npnn \seq_pop_right:NN #1#2
16606 { T , F , TF }
16607 { \__seq_pop_TF:NNNN \__seq_pop_right:NNN \__kernel_tl_set:Nx #1 #2 }
16608 \prg_new_protected_conditional:Npnn \seq_gpop_right:NN #1#2
16609 { T , F , TF }
16610 { \__seq_pop_TF:NNNN \__seq_pop_right:NNN \__kernel_tl_gset:Nx #1 #2 }
16611 \prg_generate_conditional_variant:Nnn \seq_pop_left:NN { c }
16612 { T , F , TF }
16613 \prg_generate_conditional_variant:Nnn \seq_gpop_left:NN { c }
16614 { T , F , TF }
16615 \prg_generate_conditional_variant:Nnn \seq_pop_right:NN { c }
16616 { T , F , TF }
16617 \prg_generate_conditional_variant:Nnn \seq_gpop_right:NN { c }
16618 { T , F , TF }

```

(End definition for `\seq_pop_left:NNTF` and others. These functions are documented on page 146.)

```

\seq_item:Nn The idea here is to find the offset of the item from the left, then use a loop to grab
\seq_item:cn the correct item. If the resulting offset is too large, then the argument delimited by
\__seq_item:wNn \__seq_item:n is \prg_break: instead of being empty, terminating the loop and re-
\__seq_item:nN turning nothing at all.
\__seq_item:nwn 16619 \cs_new:Npn \seq_item:Nn #1
16620 { \exp_after:wN \__seq_item:wNn #1 \s__seq_stop #1 }
16621 \cs_new:Npn \__seq_item:wNn \s__seq #1 \s__seq_stop #2#3
16622 {
16623   \exp_args:Nf \__seq_item:nwn
16624   { \exp_args:Nf \__seq_item:nN { \int_eval:n {#3} } #2 }
16625   #1
16626   \prg_break: \__seq_item:n { }
16627   \prg_break_point:
16628 }
16629 \cs_new:Npn \__seq_item:nN #1#2
16630 {
16631   \int_compare:nNnTF {#1} < 0
16632   { \int_eval:n { \seq_count:N #2 + 1 + #1 } }
16633   {#1}
16634 }
16635 \cs_new:Npn \__seq_item:nwn #1#2 \__seq_item:n #3
16636 {
16637   #2
16638   \int_compare:nNnTF {#1} = 1
16639   { \prg_break:n { \exp_not:n {#3} } }
16640   { \exp_args:Nf \__seq_item:nwn { \int_eval:n { #1 - 1 } } }
16641 }
16642 \cs_generate_variant:Nn \seq_item:Nn { c }

```

(End definition for `\seq_item:Nn` and others. This function is documented on page 145.)

`\seq_rand_item:N` Importantly, `\seq_item:Nn` only evaluates its argument once.

```

\seq_rand_item:c 16643 \cs_new:Npn \seq_rand_item:N #1
                  16644 {
                  16645     \seq_if_empty:NF #1
                  16646     { \seq_item:Nn #1 { \int_rand:nn { 1 } { \seq_count:N #1 } } }
                  16647 }
                  16648 \cs_generate_variant:Nn \seq_rand_item:N { c }

```

(End definition for `\seq_rand_item:N`. This function is documented on page 145.)

56.6 Mapping over sequences

`\seq_map_break:` To break a function, the special token `\prg_break_point:Nn` is used to find the end of the code. Any ending code is then inserted before the return value of `\seq_map_break:n` is inserted.

```

16649 \cs_new:Npn \seq_map_break:
16650 { \prg_map_break:Nn \seq_map_break: { } }
16651 \cs_new:Npn \seq_map_break:n
16652 { \prg_map_break:Nn \seq_map_break: }

```

(End definition for `\seq_map_break:` and `\seq_map_break:n`. These functions are documented on page 149.)

`\seq_map_function:NN` The idea here is to apply the code of #2 to each item in the sequence without altering the definition of `__seq_item:n`. The even-numbered arguments of `__seq_map_function:Nw` delimited by `__seq_item:n` are almost always empty, except at the end of the loop where it is `\prg_break:`. This allows to break the loop without needing to do a (relatively-expensive) quark test.

```

16653 \cs_new:Npn \seq_map_function:NN #1#2
16654 {
16655     \exp_after:wN \use_i_ii:nnn
16656     \exp_after:wN \__seq_map_function:Nw
16657     \exp_after:wN #2
16658     #1
16659     \prg_break:
16660     \__seq_item:n { } \__seq_item:n { } \__seq_item:n { } \__seq_item:n { }
16661     \prg_break_point:
16662     \prg_break_point:Nn \seq_map_break: { }
16663 }
16664 \cs_new:Npn \__seq_map_function:Nw #1
16665     #2 \__seq_item:n #3
16666     #4 \__seq_item:n #5
16667     #6 \__seq_item:n #7
16668     #8 \__seq_item:n #9
16669 {
16670     #2 #1 {#3}
16671     #4 #1 {#5}
16672     #6 #1 {#7}
16673     #8 #1 {#9}
16674     \__seq_map_function:Nw #1
16675 }
16676 \cs_generate_variant:Nn \seq_map_function:NN { c }

```

(End definition for `\seq_map_function:Nn` and `__seq_map_function:Nw`. This function is documented on page 148.)

`__seq_push_item_def:n` The definition of `__seq_item:n` needs to be saved and restored at various points within the mapping and manipulation code. That is handled here: as always, this approach uses global assignments.

```

\__seq_push_item_def:x
\__seq_push_item_def:
\__seq_pop_item_def:
16677 \cs_new_protected:Npn \__seq_push_item_def:n
16678 {
16679   \__seq_push_item_def:
16680   \cs_gset:Npn \__seq_item:n ##1
16681 }
16682 \cs_new_protected:Npn \__seq_push_item_def:x
16683 {
16684   \__seq_push_item_def:
16685   \cs_gset:Npx \__seq_item:n ##1
16686 }
16687 \cs_new_protected:Npn \__seq_push_item_def:
16688 {
16689   \int_gincr:N \g__kernel_prg_map_int
16690   \cs_gset_eq:cN { __seq_map_ \int_use:N \g__kernel_prg_map_int :w }
16691   \__seq_item:n
16692 }
16693 \cs_new_protected:Npn \__seq_pop_item_def:
16694 {
16695   \cs_gset_eq:Nc \__seq_item:n
16696   { __seq_map_ \int_use:N \g__kernel_prg_map_int :w }
16697   \int_gdecr:N \g__kernel_prg_map_int
16698 }

```

(End definition for `__seq_push_item_def:n`, `__seq_push_item_def:`, and `__seq_pop_item_def:`.)

`\seq_map_inline:Nn` The idea here is that `__seq_item:n` is already “applied” to each item in a sequence, and so an in-line mapping is just a case of redefining `__seq_item:n`.

```

\seq_map_inline:cn
16699 \cs_new_protected:Npn \seq_map_inline:Nn #1#2
16700 {
16701   \__seq_push_item_def:n {#2}
16702   #1
16703   \prg_break_point:Nn \seq_map_break: { \__seq_pop_item_def: }
16704 }
16705 \cs_generate_variant:Nn \seq_map_inline:Nn { c }

```

(End definition for `\seq_map_inline:Nn`. This function is documented on page 148.)

`\seq_map_tokens:Nn` This is based on the function mapping but using the same tricks as described for `\prop_map_tokens:Nn`. The idea is to remove the leading `\s__seq` and apply the tokens such that they are safe with the break points, hence the `\use:n`.

```

\seq_map_tokens:cn
\__seq_map_tokens:nw
16706 \cs_new:Npn \seq_map_tokens:Nn #1#2
16707 {
16708   \exp_last_unbraced:Nno
16709   \use_i:nn { \__seq_map_tokens:nw {#2} } #1
16710   \prg_break:
16711   \__seq_item:n { } \__seq_item:n { } \__seq_item:n { } \__seq_item:n { }
16712   \prg_break_point:
16713   \prg_break_point:Nn \seq_map_break: { }

```

```

16714 }
16715 \cs_generate_variant:Nn \seq_map_tokens:Nn { c }
16716 \cs_new:Npn \__seq_map_tokens:nw #1
16717   #2 \__seq_item:n #3
16718   #4 \__seq_item:n #5
16719   #6 \__seq_item:n #7
16720   #8 \__seq_item:n #9
16721 {
16722   #2 \use:n {#1} {#3}
16723   #4 \use:n {#1} {#5}
16724   #6 \use:n {#1} {#7}
16725   #8 \use:n {#1} {#9}
16726   \__seq_map_tokens:nw {#1}
16727 }

```

(End definition for `\seq_map_tokens:Nn` and `__seq_map_tokens:nw`. This function is documented on page 148.)

`\seq_map_variable:NNn`
`\seq_map_variable:Ncn`
`\seq_map_variable:cNn`
`\seq_map_variable:ccn`

This is just a specialised version of the in-line mapping function, using an `x`-type expansion for the code set up so that the number of `#` tokens required is as expected.

```

16728 \cs_new_protected:Npn \seq_map_variable:NNn #1#2#3
16729 {
16730   \__seq_push_item_def:x
16731   {
16732     \tl_set:Nn \exp_not:N #2 {##1}
16733     \exp_not:n {#3}
16734   }
16735   #1
16736   \prg_break_point:Nn \seq_map_break: { \__seq_pop_item_def: }
16737 }
16738 \cs_generate_variant:Nn \seq_map_variable:NNn { Nc }
16739 \cs_generate_variant:Nn \seq_map_variable:NNn { c , cc }

```

(End definition for `\seq_map_variable:NNn`. This function is documented on page 148.)

`\seq_map_indexed_function:NN`
`\seq_map_indexed_inline:Nn`
`__seq_map_indexed:nNN`
`__seq_map_indexed:Nw`

Similar to `\seq_map_function:NN` but we keep track of the item index as a `;`-delimited argument of `__seq_map_indexed:Nw`.

```

16740 \cs_new:Npn \seq_map_indexed_function:NN #1#2
16741 {
16742   \__seq_map_indexed:NN #1#2
16743   \prg_break_point:Nn \seq_map_break: { }
16744 }
16745 \cs_new_protected:Npn \seq_map_indexed_inline:Nn #1#2
16746 {
16747   \int_gincr:N \g__kernel_pr_g_map_int
16748   \cs_gset_protected:cpn
16749   { \__seq_map_ \int_use:N \g__kernel_pr_g_map_int :w } ##1##2 {#2}
16750   \exp_args:NNc \__seq_map_indexed:NN #1
16751   { \__seq_map_ \int_use:N \g__kernel_pr_g_map_int :w }
16752   \prg_break_point:Nn \seq_map_break:
16753   { \int_gdecr:N \g__kernel_pr_g_map_int }
16754 }
16755 \cs_new:Npn \__seq_map_indexed:NN #1#2
16756 {

```

```

16757 \exp_after:wN \__seq_map_indexed:Nw
16758 \exp_after:wN #2
16759 \int_value:w 1
16760 \exp_after:wN \use_i:nn
16761 \exp_after:wN ;
16762 #1
16763 \prg_break: \__seq_item:n { } \prg_break_point:
16764 }
16765 \cs_new:Npn \__seq_map_indexed:Nw #1#2 ; #3 \__seq_item:n #4
16766 {
16767 #3
16768 #1 {#2} {#4}
16769 \exp_after:wN \__seq_map_indexed:Nw
16770 \exp_after:wN #1
16771 \int_value:w \int_eval:w 1 + #2 ;
16772 }

```

(End definition for `\seq_map_indexed_function:NN` and others. These functions are documented on page 148.)

`\seq_set_map_x:NNn` Very similar to `\seq_set_filter:NNn`. We could actually merge the two within a single function, but it would have weird semantics.

```

\seq_gset_map_x:NNn
\__seq_set_map_x:NNNn
16773 \cs_new_protected:Npn \seq_set_map_x:NNn
16774 { \__seq_set_map_x:NNNn \__kernel_tl_set:Nx }
16775 \cs_new_protected:Npn \seq_gset_map_x:NNn
16776 { \__seq_set_map_x:NNNn \__kernel_tl_gset:Nx }
16777 \cs_new_protected:Npn \__seq_set_map_x:NNNn #1#2#3#4
16778 {
16779 \__seq_push_item_def:n { \exp_not:N \__seq_item:n {#4} }
16780 #1 #2 { #3 }
16781 \__seq_pop_item_def:
16782 }

```

(End definition for `\seq_set_map_x:NNn`, `\seq_gset_map_x:NNn`, and `__seq_set_map_x:NNNn`. These functions are documented on page 150.)

`\seq_set_map:NNn` Similar to `\seq_set_map_x:NNn`, but prevents expansion of the `<inline function>`.

```

\seq_gset_map:NNn
\__seq_set_map:NNNn
16783 \cs_new_protected:Npn \seq_set_map:NNn
16784 { \__seq_set_map:NNNn \__kernel_tl_set:Nx }
16785 \cs_new_protected:Npn \seq_gset_map:NNn
16786 { \__seq_set_map:NNNn \__kernel_tl_gset:Nx }
16787 \cs_new_protected:Npn \__seq_set_map:NNNn #1#2#3#4
16788 {
16789 \__seq_push_item_def:n { \exp_not:n { \__seq_item:n {#4} } }
16790 #1 #2 { #3 }
16791 \__seq_pop_item_def:
16792 }

```

(End definition for `\seq_set_map:NNn`, `\seq_gset_map:NNn`, and `__seq_set_map:NNNn`. These functions are documented on page 150.)

`\seq_count:N` Since counting the items in a sequence is quite common, we optimize it by grabbing 8 items at a time and correspondingly adding 8 to an integer expression. At the end of the loop, #9 is `__seq_count_end:w` instead of being empty. It removes 8+ and instead

```

\seq_count:c
\__seq_count:w
\__seq_count_end:w

```

places the number of `__seq_item:n` that `__seq_count:w` grabbed before reaching the end of the sequence.

```

16793 \cs_new:Npn \seq_count:N #1
16794 {
16795   \int_eval:n
16796   {
16797     \exp_after:wN \use_i:nn
16798     \exp_after:wN \__seq_count:w
16799     #1
16800     \__seq_count_end:w \__seq_item:n 7
16801     \__seq_count_end:w \__seq_item:n 6
16802     \__seq_count_end:w \__seq_item:n 5
16803     \__seq_count_end:w \__seq_item:n 4
16804     \__seq_count_end:w \__seq_item:n 3
16805     \__seq_count_end:w \__seq_item:n 2
16806     \__seq_count_end:w \__seq_item:n 1
16807     \__seq_count_end:w \__seq_item:n 0
16808     \prg_break_point:
16809   }
16810 }
16811 \cs_new:Npn \__seq_count:w
16812   #1 \__seq_item:n #2 \__seq_item:n #3 \__seq_item:n #4 \__seq_item:n
16813   #5 \__seq_item:n #6 \__seq_item:n #7 \__seq_item:n #8 #9 \__seq_item:n
16814   { #9 8 + \__seq_count:w }
16815 \cs_new:Npn \__seq_count_end:w 8 + \__seq_count:w #1#2 \prg_break_point: {#1}
16816 \cs_generate_variant:Nn \seq_count:N { c }

```

(End definition for `\seq_count:N`, `__seq_count:w`, and `__seq_count_end:w`. This function is documented on page 150.)

56.7 Using sequences

<pre> \seq_use:Nnnn \seq_use:cnnn __seq_use:NNnNnn __seq_use_setup:w __seq_use:nwwwnwn __seq_use:nwwn \seq_use:Nn \seq_use:cn </pre>	<p>See <code>\clist_use:Nnnn</code> for a general explanation. The main difference is that we use <code>__seq_item:n</code> as a delimiter rather than commas. We also need to add <code>__seq_item:n</code> at various places, and <code>\s__seq</code>.</p> <pre> 16817 \cs_new:Npn \seq_use:Nnnn #1#2#3#4 16818 { 16819 \seq_if_exist:NTF #1 16820 { 16821 \int_case:nnF { \seq_count:N #1 } 16822 { 16823 { 0 } { } 16824 { 1 } { \exp_after:wN __seq_use:NNnNnn #1 ? { } { } } 16825 { 2 } { \exp_after:wN __seq_use:NNnNnn #1 {#2} } 16826 } 16827 { 16828 \exp_after:wN __seq_use_setup:w #1 __seq_item:n 16829 \s__seq_mark { __seq_use:nwwwnwn {#3} } 16830 \s__seq_mark { __seq_use:nwwn {#4} } 16831 \s__seq_stop { } 16832 } 16833 } 16834 { </pre>
--	---

```

16835         \msg_expandable_error:nnn
16836         { kernel } { bad-variable } {#1}
16837     }
16838 }
16839 \cs_generate_variant:Nn \seq_use:Nnnn { c }
16840 \cs_new:Npn \__seq_use:NNnNnn #1#2#3#4#5#6 { \exp_not:n { #3 #6 #5 } }
16841 \cs_new:Npn \__seq_use_setup:w \s__seq { \__seq_use:nwwwnwn { } }
16842 \cs_new:Npn \__seq_use:nwwwnwn
16843     #1 \__seq_item:n #2 \__seq_item:n #3 \__seq_item:n #4#5
16844     \s__seq_mark #6#7 \s__seq_stop #8
16845     {
16846         #6 \__seq_item:n {#3} \__seq_item:n {#4} #5
16847         \s__seq_mark {#6} #7 \s__seq_stop { #8 #1 #2 }
16848     }
16849 \cs_new:Npn \__seq_use:nwnn #1 \__seq_item:n #2 #3 \s__seq_stop #4
16850     { \exp_not:n { #4 #1 #2 } }
16851 \cs_new:Npn \seq_use:Nn #1#2
16852     { \seq_use:Nnnn #1 {#2} {#2} {#2} }
16853 \cs_generate_variant:Nn \seq_use:Nn { c }

```

(End definition for `\seq_use:Nnnn` and others. These functions are documented on page 150.)

56.8 Sequence stacks

The same functions as for sequences, but with the correct naming.

`\seq_push:Nn` Pushing to a sequence is the same as adding on the left.

```

\seq_push:NV 16854 \cs_new_eq:NN \seq_push:Nn \seq_put_left:Nn
\seq_push:Nv 16855 \cs_new_eq:NN \seq_push:Nv \seq_put_left:Nv
\seq_push:No 16856 \cs_new_eq:NN \seq_push:Nv \seq_put_left:Nv
\seq_push:Nx 16857 \cs_new_eq:NN \seq_push:No \seq_put_left:No
\seq_push:cn 16858 \cs_new_eq:NN \seq_push:Nx \seq_put_left:Nx
\seq_push:cV 16859 \cs_new_eq:NN \seq_push:cn \seq_put_left:cn
\seq_push:cV 16860 \cs_new_eq:NN \seq_push:cV \seq_put_left:cV
\seq_push:cv 16861 \cs_new_eq:NN \seq_push:cv \seq_put_left:cv
\seq_push:co 16862 \cs_new_eq:NN \seq_push:co \seq_put_left:co
\seq_push:cx 16863 \cs_new_eq:NN \seq_push:cx \seq_put_left:cx
\seq_gpush:Nn 16864 \cs_new_eq:NN \seq_gpush:Nn \seq_gput_left:Nn
\seq_gpush:NV 16865 \cs_new_eq:NN \seq_gpush:Nv \seq_gput_left:Nv
\seq_gpush:Nv 16866 \cs_new_eq:NN \seq_gpush:Nv \seq_gput_left:Nv
\seq_gpush:No 16867 \cs_new_eq:NN \seq_gpush:No \seq_gput_left:No
\seq_gpush:Nx 16868 \cs_new_eq:NN \seq_gpush:Nx \seq_gput_left:Nx
\seq_gpush:cn 16869 \cs_new_eq:NN \seq_gpush:cn \seq_gput_left:cn
\seq_gpush:cV 16870 \cs_new_eq:NN \seq_gpush:cV \seq_gput_left:cV
\seq_gpush:cv 16871 \cs_new_eq:NN \seq_gpush:cv \seq_gput_left:cv
\seq_gpush:co 16872 \cs_new_eq:NN \seq_gpush:co \seq_gput_left:co
\seq_gpush:cx 16873 \cs_new_eq:NN \seq_gpush:cx \seq_gput_left:cx

```

(End definition for `\seq_push:Nn` and `\seq_gpush:Nn`. These functions are documented on page 152.)

`\seq_get:NN` In most cases, getting items from the stack does not need to specify that this is from the left. So alias are provided.

```

\seq_get:cn 16874 \cs_new_eq:NN \seq_get:NN \seq_get_left:NN
\seq_pop:cn
\seq_gpop:NN
\seq_gpop:cn

```

```

16875 \cs_new_eq:NN \seq_get:cN \seq_get_left:cN
16876 \cs_new_eq:NN \seq_pop:NN \seq_pop_left:NN
16877 \cs_new_eq:NN \seq_pop:cN \seq_pop_left:cN
16878 \cs_new_eq:NN \seq_gpop:NN \seq_gpop_left:NN
16879 \cs_new_eq:NN \seq_gpop:cN \seq_gpop_left:cN

```

(End definition for `\seq_get:NN`, `\seq_pop:NN`, and `\seq_gpop:NN`. These functions are documented on page 151.)

```

\seq_get:NNTF More copies.
\seq_get:cNTF 16880 \prg_new_eq_conditional:NNn \seq_get:NN \seq_get_left:NN { T , F , TF }
\seq_pop:NNTF 16881 \prg_new_eq_conditional:NNn \seq_get:cN \seq_get_left:cN { T , F , TF }
\seq_pop:cNTF 16882 \prg_new_eq_conditional:NNn \seq_pop:NN \seq_pop_left:NN { T , F , TF }
\seq_gpop:NNTF 16883 \prg_new_eq_conditional:NNn \seq_pop:cN \seq_pop_left:cN { T , F , TF }
\seq_gpop:cNTF 16884 \prg_new_eq_conditional:NNn \seq_gpop:NN \seq_gpop_left:NN { T , F , TF }
16885 \prg_new_eq_conditional:NNn \seq_gpop:cN \seq_gpop_left:cN { T , F , TF }

```

(End definition for `\seq_get:NNTF`, `\seq_pop:NNTF`, and `\seq_gpop:NNTF`. These functions are documented on page 151.)

56.9 Viewing sequences

```

\seq_show:N Apply the general \__kernel_chk_tl_type:NnnT.
\seq_show:c 16886 \cs_new_protected:Npn \seq_show:N { \__seq_show:NN \msg_show:nnxxxx }
\seq_log:N 16887 \cs_generate_variant:Nn \seq_show:N { c }
\seq_log:c 16888 \cs_new_protected:Npn \seq_log:N { \__seq_show:NN \msg_log:nnxxxx }
__seq_show:NN 16889 \cs_generate_variant:Nn \seq_log:N { c }
__seq_show_validate:nn 16890 \cs_new_protected:Npn \__seq_show:NN #1#2
16891 {
16892   \__kernel_chk_tl_type:NnnT #2 { seq }
16893   {
16894     \s_seq
16895     \exp_after:wN \use_i:nn \exp_after:wN \__seq_show_validate:nn #2
16896     \q_recursion_tail \q_recursion_tail \q_recursion_stop
16897   }
16898   {
16899     #1 { seq } { show }
16900     { \token_to_str:N #2 }
16901     { \seq_map_function:NN #2 \msg_show_item:n }
16902     { } { }
16903   }
16904 }
16905 \cs_new:Npn \__seq_show_validate:nn #1#2
16906 {
16907   \quark_if_recursion_tail_stop:n {#2}
16908   \__seq_wrap_item:n {#2}
16909   \__seq_show_validate:nn
16910 }

```

(End definition for `\seq_show:N` and others. These functions are documented on page 154.)

56.10 Scratch sequences

`\l_tmpa_seq` Temporary comma list variables.

`\l_tmpb_seq` 16911 `\seq_new:N \l_tmpa_seq`

`\g_tmpa_seq` 16912 `\seq_new:N \l_tmpb_seq`

`\g_tmpb_seq` 16913 `\seq_new:N \g_tmpa_seq`

16914 `\seq_new:N \g_tmpb_seq`

(End definition for `\l_tmpa_seq` and others. These variables are documented on page [154](#).)

16915 `\</package>`

Chapter 57

l3int implementation

```
16916 <*package>
```

```
16917 <@@=int>
```

The following test files are used for this code: m3int001,m3int002,m3int03.

\c_max_register_int Done in l3basics.

(End definition for \c_max_register_int. This variable is documented on page 166.)

__int_to_roman:w Done in l3basics.

\if_int_compare:w *(End definition for __int_to_roman:w and \if_int_compare:w. This function is documented on page 167.)*

\or: Done in l3basics.

(End definition for \or:. This function is documented on page 167.)

\int_value:w Here are the remaining primitives for number comparisons and expressions.

__int_eval:w	16918 \cs_new_eq:NN \int_value:w	\tex_number:D
__int_eval_end:	16919 \cs_new_eq:NN __int_eval:w	\tex_numexpr:D
\if_int_odd:w	16920 \cs_new_eq:NN __int_eval_end:	\tex_relax:D
\if_case:w	16921 \cs_new_eq:NN \if_int_odd:w	\tex_ifodd:D
	16922 \cs_new_eq:NN \if_case:w	\tex_ifcase:D

(End definition for \int_value:w and others. These functions are documented on page 167.)

\s__int_mark Scan marks used throughout the module.

\s__int_stop	16923 \scan_new:N \s__int_mark
	16924 \scan_new:N \s__int_stop

(End definition for \s__int_mark and \s__int_stop.)

__int_use_none_delimit_by_s_stop:w Function to gobble until a scan mark.

```
16925 \cs_new:Npn \__int_use_none_delimit_by_s_stop:w #1 \s__int_stop { }
```

(End definition for __int_use_none_delimit_by_s_stop:w.)

\q__int_recursion_tail Quarks for recursion.

\q__int_recursion_stop	16926 \quark_new:N \q__int_recursion_tail
	16927 \quark_new:N \q__int_recursion_stop

(End definition for `\q__int_recursion_tail` and `\q__int_recursion_stop`.)

`__int_if_recursion_tail_stop_do:Nn`
`__int_if_recursion_tail_stop:N`

Functions to query quarks.

```
16928 \__kernel_quark_new_test:N \__int_if_recursion_tail_stop_do:Nn
16929 \__kernel_quark_new_test:N \__int_if_recursion_tail_stop:N
```

(End definition for `__int_if_recursion_tail_stop_do:Nn` and `__int_if_recursion_tail_stop:N`.)

57.1 Integer expressions

`\int_eval:n` Wrapper for `__int_eval:w`: can be used in an integer expression or directly in the input stream. It is very slightly faster to use `\the` rather than `\number` to turn the expression to a number. When debugging, we introduce parentheses to catch early termination (see `l3debug`).

```
16930 \cs_new:Npn \int_eval:n #1
16931 { \tex_the:D \__int_eval:w #1 \__int_eval_end: }
16932 \cs_new:Npn \int_eval:w { \tex_the:D \__int_eval:w }
```

(End definition for `\int_eval:n` and `\int_eval:w`. These functions are documented on page 156.)

`\int_sign:n`
`__int_sign:Nw`

See `\int_abs:n`. Evaluate the expression once (and when debugging is enabled, check that the expression is well-formed), then test the first character to determine the sign. This is wrapped in `\int_value:w ... \exp_stop_f:` to ensure a fixed number of expansions and to avoid dealing with closing the conditionals.

```
16933 \cs_new:Npn \int_sign:n #1
16934 {
16935   \int_value:w \exp_after:wN \__int_sign:Nw
16936   \int_value:w \__int_eval:w #1 \__int_eval_end: ;
16937   \exp_stop_f:
16938 }
16939 \cs_new:Npn \__int_sign:Nw #1#2 ;
16940 {
16941   \if_meaning:w 0 #1
16942   0
16943   \else:
16944     \if_meaning:w - #1 - \fi: 1
16945   \fi:
16946 }
```

(End definition for `\int_sign:n` and `__int_sign:Nw`. This function is documented on page 156.)

`\int_abs:n`
`__int_abs:N`

Functions for min, max, and absolute value with only one evaluation. The absolute value is obtained by removing a leading sign if any. All three functions expand in two steps.

`\int_max:nn`
`\int_min:nn`
`__int_maxmin:wwN`

```
16947 \cs_new:Npn \int_abs:n #1
16948 {
16949   \int_value:w \exp_after:wN \__int_abs:N
16950   \int_value:w \__int_eval:w #1 \__int_eval_end:
16951   \exp_stop_f:
16952 }
16953 \cs_new:Npn \__int_abs:N #1
16954 { \if_meaning:w - #1 \else: \exp_after:wN #1 \fi: }
16955 \cs_set:Npn \int_max:nn #1#2
16956 {
```

```

16957 \int_value:w \exp_after:wN \__int_maxmin:wwN
16958 \int_value:w \__int_eval:w #1 \exp_after:wN ;
16959 \int_value:w \__int_eval:w #2 ;
16960 >
16961 \exp_stop_f:
16962 }
16963 \cs_set:Npn \int_min:nn #1#2
16964 {
16965 \int_value:w \exp_after:wN \__int_maxmin:wwN
16966 \int_value:w \__int_eval:w #1 \exp_after:wN ;
16967 \int_value:w \__int_eval:w #2 ;
16968 <
16969 \exp_stop_f:
16970 }
16971 \cs_new:Npn \__int_maxmin:wwN #1 ; #2 ; #3
16972 {
16973 \if_int_compare:w #1 #3 #2 ~
16974 #1
16975 \else:
16976 #2
16977 \fi:
16978 }

```

(End definition for `\int_abs:n` and others. These functions are documented on page 156.)

`\int_div_truncate:nn` As `__int_eval:w` rounds the result of a division we also provide a version that truncates the result. We use an auxiliary to make sure numerator and denominator are only evaluated once: this comes in handy when those are more expressions are expensive to evaluate (e.g., `\tl_count:n`). If the numerator `#1#2` is 0, then we divide 0 by the denominator (this ensures that 0/0 is correctly reported as an error). Otherwise, shift the numerator `#1#2` towards 0 by $(|\#3\#4| - 1)/2$, which we round away from zero. It turns out that this quantity exactly compensates the difference between ε -TeX's rounding and the truncating behaviour that we want. The details are thanks to Heiko Oberdiek: getting things right in all cases is not so easy.

```

16979 \cs_new:Npn \int_div_truncate:nn #1#2
16980 {
16981 \int_value:w \__int_eval:w
16982 \exp_after:wN \__int_div_truncate:NwNw
16983 \int_value:w \__int_eval:w #1 \exp_after:wN ;
16984 \int_value:w \__int_eval:w #2 ;
16985 \__int_eval_end:
16986 }
16987 \cs_new:Npn \__int_div_truncate:NwNw #1#2; #3#4;
16988 {
16989 \if_meaning:w 0 #1
16990 0
16991 \else:
16992 (
16993 #1#2
16994 \if_meaning:w - #1 + \else: - \fi:
16995 ( \if_meaning:w - #3 - \fi: #3#4 - 1 ) / 2
16996 )
16997 \fi:
16998 / #3#4

```

```
16999 }
```

For the sake of completeness:

```
17000 \cs_new:Npn \int_div_round:nn #1#2
17001 { \int_value:w \__int_eval:w ( #1 ) / ( #2 ) \__int_eval_end: }
```

Finally there's the modulus operation.

```
17002 \cs_new:Npn \int_mod:nn #1#2
17003 {
17004   \int_value:w \__int_eval:w \exp_after:wN \__int_mod:ww
17005   \int_value:w \__int_eval:w #1 \exp_after:wN ;
17006   \int_value:w \__int_eval:w #2 ;
17007   \__int_eval_end:
17008 }
17009 \cs_new:Npn \__int_mod:ww #1; #2;
17010 { #1 - ( \__int_div_truncate:NwNw #1 ; #2 ; ) * #2 }
```

(End definition for `\int_div_truncate:nn` and others. These functions are documented on page 156.)

`__kernel_int_add:nnn`

Equivalent to `\int_eval:n {#1+#2+#3}` except that overflow only occurs if the final result overflows $[-2^{31} + 1, 2^{31} - 1]$. The idea is to choose the order in which the three numbers are added together. If #1 and #2 have opposite signs (one is in $[-2^{31} + 1, -1]$ and the other in $[0, 2^{31} - 1]$) then #1+#2 cannot overflow so we compute the result as #1+#2+#3. If they have the same sign, then either #3 has the same sign and the order does not matter, or #3 has the opposite sign and any order in which #3 is not last will work. We use #1+#3+#2.

```
17011 \cs_new:Npn \__kernel_int_add:nnn #1#2#3
17012 {
17013   \int_value:w \__int_eval:w #1
17014   \if_int_compare:w #2 < \c_zero_int \exp_after:wN \reverse_if:N \fi:
17015   \if_int_compare:w #1 < \c_zero_int + #2 + #3 \else: + #3 + #2 \fi:
17016   \__int_eval_end:
17017 }
```

(End definition for `__kernel_int_add:nnn`.)

57.2 Creating and initialising integers

`\int_new:N`
`\int_new:c`

Two ways to do this: one for the format and one for the L^AT_EX 2_ε package. In plain T_EX, `\newcount` (and other allocators) are `\outer:` to allow the code here to work in “generic” mode this is therefore accessed by name. (The same applies to `\newbox`, `\newdimen` and so on.)

```
17018 \cs_new_protected:Npn \int_new:N #1
17019 {
17020   \__kernel_chk_if_free_cs:N #1
17021   \cs:w newcount \cs_end: #1
17022 }
17023 \cs_generate_variant:Nn \int_new:N { c }
```

(End definition for `\int_new:N`. This function is documented on page 157.)

`\int_const:Nn` As stated, most constants can be defined as `\chardef` or `\mathchardef` but that's engine dependent. As a result, there is some set up code to determine what can be done. No full engine testing just yet so everything is a little awkward. We cannot use `\int_gset:Nn` because (when `check-declarations` is enabled) this runs some checks that constants would fail.

```

17024 \cs_new_protected:Npn \int_const:Nn #1#2
17025 { \exp_args:Nx \__int_const:nN { \int_eval:n {#2} } #1 }
17026 \cs_new_protected:Npn \__int_const:nN #1#2
17027 {
17028   \int_compare:nNnTF {#1} < \c_zero_int
17029   {
17030     \int_new:N #2
17031     \tex_global:D
17032   }
17033   {
17034     \int_compare:nNnTF {#1} > \c__int_max_constdef_int
17035     {
17036       \int_new:N #2
17037       \tex_global:D
17038     }
17039     {
17040       \__kernel_chk_if_free_cs:N #2
17041       \tex_global:D \__int_constdef:Nw
17042     }
17043   }
17044   #2 = \__int_eval:w #1 \__int_eval_end:
17045 }
17046 \cs_generate_variant:Nn \int_const:Nn { c }
17047 \if_int_odd:w 0
17048   \cs_if_exist:NT \tex_luatexversion:D { 1 }
17049   \cs_if_exist:NT \tex_omathchardef:D { 1 }
17050   \cs_if_exist:NT \tex_XeTeXversion:D { 1 } ~
17051   \cs_if_exist:NTF \tex_omathchardef:D
17052   { \cs_new_eq:NN \__int_constdef:Nw \tex_omathchardef:D }
17053   { \cs_new_eq:NN \__int_constdef:Nw \tex_chardef:D }
17054   \__int_constdef:Nw \c__int_max_constdef_int 1114111 ~
17055 \else:
17056   \cs_new_eq:NN \__int_constdef:Nw \tex_mathchardef:D
17057   \tex_mathchardef:D \c__int_max_constdef_int 32767 ~
17058 \fi:

```

(End definition for `\int_const:Nn` and others. This function is documented on page 157.)

`\int_zero:N` Functions that reset an *integer* register to zero.

`\int_zero:c` 17059 \cs_new_protected:Npn \int_zero:N #1 { #1 = \c_zero_int }

`\int_gzero:N` 17060 \cs_new_protected:Npn \int_gzero:N #1 { \tex_global:D #1 = \c_zero_int }

`\int_gzero:c` 17061 \cs_generate_variant:Nn \int_zero:N { c }

17062 \cs_generate_variant:Nn \int_gzero:N { c }

(End definition for `\int_zero:N` and `\int_gzero:N`. These functions are documented on page 157.)

`\int_zero_new:N` Create a register if needed, otherwise clear it.

`\int_zero_new:c` 17063 \cs_new_protected:Npn \int_zero_new:N #1

`\int_gzero_new:N` 17064 { \int_if_exist:NTF #1 { \int_zero:N #1 } { \int_new:N #1 } }

`\int_gzero_new:c`

```

17065 \cs_new_protected:Npn \int_gzero_new:N #1
17066 { \int_if_exist:NTF #1 { \int_gzero:N #1 } { \int_new:N #1 } }
17067 \cs_generate_variant:Nn \int_zero_new:N { c }
17068 \cs_generate_variant:Nn \int_gzero_new:N { c }

```

(End definition for `\int_zero_new:N` and `\int_gzero_new:N`. These functions are documented on page 157.)

`\int_set_eq:NN` Setting equal means using one integer inside the set function of another. Check that assigned integer is local/global. No need to check that the other one is defined as `\TeX` does it for us.

```

17069 \cs_new_protected:Npn \int_set_eq:NN #1#2 { #1 = #2 }
17070 \cs_generate_variant:Nn \int_set_eq:NN { c , Nc , cc }
17071 \cs_new_protected:Npn \int_gset_eq:NN #1#2 { \tex_global:D #1 = #2 }
17072 \cs_generate_variant:Nn \int_gset_eq:NN { c , Nc , cc }

```

(End definition for `\int_set_eq:NN` and `\int_gset_eq:NN`. These functions are documented on page 157.)

`\int_if_exist_p:N` Copies of the `\cs` functions defined in `l3basics`.

```

17073 \prg_new_eq_conditional:NNn \int_if_exist:N \cs_if_exist:N
17074 { TF , T , F , p }
17075 \prg_new_eq_conditional:NNn \int_if_exist:c \cs_if_exist:c
17076 { TF , T , F , p }

```

(End definition for `\int_if_exist:NTF`. This function is documented on page 157.)

57.3 Setting and incrementing integers

`\int_add:Nn` Adding and subtracting to and from a counter. Including here the optional `by` would slow down these operations by a few percent.

```

17077 \cs_new_protected:Npn \int_add:Nn #1#2
17078 { \tex_advance:D #1 \__int_eval:w #2 \__int_eval_end: }
17079 \cs_new_protected:Npn \int_sub:Nn #1#2
17080 { \tex_advance:D #1 - \__int_eval:w #2 \__int_eval_end: }
17081 \cs_new_protected:Npn \int_gadd:Nn #1#2
17082 { \tex_global:D \tex_advance:D #1 \__int_eval:w #2 \__int_eval_end: }
17083 \cs_new_protected:Npn \int_gsub:Nn #1#2
17084 { \tex_global:D \tex_advance:D #1 - \__int_eval:w #2 \__int_eval_end: }
17085 \cs_generate_variant:Nn \int_add:Nn { c }
17086 \cs_generate_variant:Nn \int_gadd:Nn { c }
17087 \cs_generate_variant:Nn \int_sub:Nn { c }
17088 \cs_generate_variant:Nn \int_gsub:Nn { c }

```

(End definition for `\int_add:Nn` and others. These functions are documented on page 158.)

`\int_incr:N` Incrementing and decrementing of integer registers is done with the following functions.

```

17089 \cs_new_protected:Npn \int_incr:N #1
17090 { \tex_advance:D #1 \c_one_int }
17091 \cs_new_protected:Npn \int_decr:N #1
17092 { \tex_advance:D #1 - \c_one_int }
17093 \cs_new_protected:Npn \int_gincr:N #1
17094 { \tex_global:D \tex_advance:D #1 \c_one_int }
17095 \cs_new_protected:Npn \int_gdecr:N #1

```

```

17096 { \tex_global:D \tex_advance:D #1 - \c_one_int }
17097 \cs_generate_variant:Nn \int_incr:N { c }
17098 \cs_generate_variant:Nn \int_decr:N { c }
17099 \cs_generate_variant:Nn \int_gincr:N { c }
17100 \cs_generate_variant:Nn \int_gdecr:N { c }

```

(End definition for `\int_incr:N` and others. These functions are documented on page 158.)

`\int_set:Nn` As integers are register-based TeX issues an error if they are not defined. While the = sign is optional, this version with = is slightly quicker than without, while adding the optional space after = slows things down minutely.

```

\int_set:cn
\int_gset:Nn
\int_gset:cn
17101 \cs_new_protected:Npn \int_set:Nn #1#2
17102 { #1 = \__int_eval:w #2 \__int_eval_end: }
17103 \cs_new_protected:Npn \int_gset:Nn #1#2
17104 { \tex_global:D #1 = \__int_eval:w #2 \__int_eval_end: }
17105 \cs_generate_variant:Nn \int_set:Nn { c }
17106 \cs_generate_variant:Nn \int_gset:Nn { c }

```

(End definition for `\int_set:Nn` and `\int_gset:Nn`. These functions are documented on page 158.)

57.4 Using integers

`\int_use:N` Here is how counters are accessed. We hand-code the `c` variant for some speed gain.

```

\int_use:c
17107 \cs_new_eq:NN \int_use:N \tex_the:D
17108 \cs_new:Npn \int_use:c #1 { \tex_the:D \cs:w #1 \cs_end: }

```

(End definition for `\int_use:N`. This function is documented on page 158.)

57.5 Integer expression conditionals

`__int_compare_error:` Those functions are used for comparison tests which use a simple syntax where only one set of braces is required and additional operators such as `!=` and `>=` are supported. `__int_compare_error:Nw` The tests first evaluate their left-hand side, with a trailing `__int_compare_error:`. This marker is normally not expanded, but if the relation symbol is missing from the test's argument, then the marker inserts = (and itself) after triggering the relevant TeX error. If the first token which appears after evaluating and removing the left-hand side is not a known relation symbol, then a judiciously placed `__int_compare_error:Nw` gets expanded, cleaning up the end of the test and telling the user what the problem was.

```

17109 \cs_new_protected:Npn \__int_compare_error:
17110 {
17111   \if_int_compare:w \c_zero_int \c_zero_int \fi:
17112   =
17113   \__int_compare_error:
17114 }
17115 \cs_new:Npn \__int_compare_error:Nw
17116 #1#2 \s__int_stop
17117 {
17118   { }
17119   \c_zero_int \fi:
17120   \msg_expandable_error:nnn
17121     { kernel } { unknown-comparison } {#1}
17122   \prg_return_false:
17123 }

```


(End definition for `_int_compare_error:` and `_int_compare_error:Nw`.)

```

\int_compare_p:n Comparison tests using a simple syntax where only one set of braces is required, additional
\int_compare:nTF operators such as != and >= are supported, and multiple comparisons can be performed
\_int_compare:w at once, for instance 0 < 5 <= 1. The idea is to loop through the argument, finding one
\_int_compare:Nw operand at a time, and comparing it to the previous one. The looping auxiliary \_int_
\_int_compare:NNw compare:Nw reads one <operand> and one <comparison> symbol, and leaves roughly
\_int_compare:nnN <operand> \prg_return_false: \fi:
\_int_compare_end=:NNw \reverse_if:N \if_int_compare:w <operand> <comparison>
\_int_compare=:NNw \_int_compare:Nw
\_int_compare_<:NNw
\_int_compare_>:NNw
\_int_compare_=:NNw
\_int_compare_!=:NNw
\_int_compare_<=:NNw
\_int_compare_>=:NNw

```

in the input stream. Each call to this auxiliary provides the second operand of the last call's `\if_int_compare:w`. If one of the *<comparisons>* is `false`, the `true` branch of the `TEX` conditional is taken (because of `\reverse_if:N`), immediately returning `false` as the result of the test. There is no `TEX` conditional waiting the first operand, so we add an `\if_false:` and expand by hand with `\int_value:w`, thus skipping `\prg_return_false:` on the first iteration.

Before starting the loop, the first step is to make sure that there is at least one relation symbol. We first let `TEX` evaluate this left hand side of the (in)equality using `_int_eval:w`. Since the relation symbols `<`, `>`, `=` and `!` are not allowed in integer expressions, they would terminate the expression. If the argument contains no relation symbol, `_int_compare_error:` is expanded, inserting `=` and itself after an error. In all cases, `_int_compare:w` receives as its argument an integer, a relation symbol, and some more tokens. We then setup the loop, which is ended by the two odd-looking items `e` and `{=nd_}`, with a trailing `\s__int_stop` used to grab the entire argument when necessary.

```

17124 \prg_new_conditional:Npnn \int_compare:n #1 { p , T , F , TF }
17125 {
17126   \exp_after:wN \_int_compare:w
17127   \int_value:w \_int_eval:w #1 \_int_compare_error:
17128 }
17129 \cs_new:Npn \_int_compare:w #1 \_int_compare_error:
17130 {
17131   \exp_after:wN \if_false: \int_value:w
17132   \_int_compare:Nw #1 e { = nd_ } \s__int_stop
17133 }

```

The goal here is to find an *<operand>* and a *<comparison>*. The *<operand>* is already evaluated, but we cannot yet grab it as an argument. To access the following relation symbol, we remove the number by applying `_int_to_roman:w`, after making sure that the argument becomes non-positive: its roman numeral representation is then empty. Then probe the first two tokens with `_int_compare:NNw` to determine the relation symbol, building a control sequence from it (`\token_to_str:N` gives better errors if `#1` is not a character). All the extended forms have an extra `=` hence the test for that as a second token. If the relation symbol is unknown, then the control sequence is turned by `TEX` into `\scan_stop:`, ignored thanks to `\unexpanded`, and `_int_compare_error:Nw` raises an error.

```

17134 \cs_new:Npn \_int_compare:Nw #1#2 \s__int_stop
17135 {
17136   \exp_after:wN \_int_compare:NNw
17137   \_int_to_roman:w - 0 #2 \s__int_mark

```

```

17138     #1#2 \s__int_stop
17139   }
17140 \cs_new:Npn \__int_compare:NNw #1#2#3 \s__int_mark
17141   {
17142     \__kernel_exp_not:w
17143     \use:c
17144     {
17145       __int_compare_ \token_to_str:N #1
17146       \if_meaning:w = #2 = \fi:
17147       :NNw
17148     }
17149     \__int_compare_error:Nw #1
17150   }

```

When the last $\langle operand \rangle$ is seen, `__int_compare:NNw` receives `e` and `=nd_` as arguments, hence calling `__int_compare_end_=:NNw` to end the loop: return the result of the last comparison (involving the operand that we just found). When a normal relation is found, the appropriate auxiliary calls `__int_compare:nnN` where `#1` is `\if_int_compare:w` or `\reverse_if:N \if_int_compare:w`, `#2` is the $\langle operand \rangle$, and `#3` is one of `<`, `=`, or `>`. As announced earlier, we leave the $\langle operand \rangle$ for the previous conditional. If this conditional is true the result of the test is known, so we remove all tokens and return `false`. Otherwise, we apply the conditional `#1` to the $\langle operand \rangle$ `#2` and the comparison `#3`, and call `__int_compare:Nw` to look for additional operands, after evaluating the following expression.

```

17151 \cs_new:cpn { __int_compare_end_=:NNw } #1#2#3 e #4 \s__int_stop
17152   {
17153     {#3} \exp_stop_f:
17154     \prg_return_false: \else: \prg_return_true: \fi:
17155   }
17156 \cs_new:Npn \__int_compare:nnN #1#2#3
17157   {
17158     {#2} \exp_stop_f:
17159     \prg_return_false: \exp_after:wN \__int_use_none_delimit_by_s_stop:w
17160     \fi:
17161     #1 #2 #3 \exp_after:wN \__int_compare:Nw \int_value:w \__int_eval:w
17162   }

```

The actual comparisons are then simple function calls, using the relation as delimiter for a delimited argument and discarding `__int_compare_error:Nw` $\langle token \rangle$ responsible for error detection.

```

17163 \cs_new:cpn { __int_compare_=:NNw } #1#2#3 =
17164   { \__int_compare:nnN { \reverse_if:N \if_int_compare:w } {#3} = }
17165 \cs_new:cpn { __int_compare_<:NNw } #1#2#3 <
17166   { \__int_compare:nnN { \reverse_if:N \if_int_compare:w } {#3} < }
17167 \cs_new:cpn { __int_compare_>:NNw } #1#2#3 >
17168   { \__int_compare:nnN { \reverse_if:N \if_int_compare:w } {#3} > }
17169 \cs_new:cpn { __int_compare_=:NNw } #1#2#3 ==
17170   { \__int_compare:nnN { \reverse_if:N \if_int_compare:w } {#3} = }
17171 \cs_new:cpn { __int_compare_!=:NNw } #1#2#3 !=
17172   { \__int_compare:nnN { \if_int_compare:w } {#3} = }
17173 \cs_new:cpn { __int_compare_<=:NNw } #1#2#3 <=
17174   { \__int_compare:nnN { \if_int_compare:w } {#3} > }
17175 \cs_new:cpn { __int_compare_>=:NNw } #1#2#3 >=
17176   { \__int_compare:nnN { \if_int_compare:w } {#3} < }

```

(End definition for `\int_compare:nTF` and others. This function is documented on page 159.)

`\int_compare_p:nNn` More efficient but less natural in typing.

```
\int_compare:nNnTF 17177 \prg_new_conditional:Npnn \int_compare:nNn #1#2#3 { p , T , F , TF }
17178 {
17179   \if_int_compare:w \__int_eval:w #1 #2 \__int_eval:w #3 \__int_eval_end:
17180   \prg_return_true:
17181   \else:
17182   \prg_return_false:
17183   \fi:
17184 }
```

(End definition for `\int_compare:nNnTF`. This function is documented on page 159.)

`\int_case:nn` For integer cases, the first task to fully expand the check condition. The over all idea is then much the same as for `\tl_case:nnTF` as described in l3tl.

```
\int_case:nnTF 17185 \cs_new:Npn \int_case:nnTF #1
17186 {
17187   \exp:w
17188   \exp_args:Nf \__int_case:nnTF { \int_eval:n {#1} }
17189 }
17190 \cs_new:Npn \int_case:nnT #1#2#3
17191 {
17192   \exp:w
17193   \exp_args:Nf \__int_case:nnTF { \int_eval:n {#1} } {#2} {#3} { }
17194 }
17195 \cs_new:Npn \int_case:nnF #1#2
17196 {
17197   \exp:w
17198   \exp_args:Nf \__int_case:nnTF { \int_eval:n {#1} } {#2} { }
17199 }
17200 \cs_new:Npn \int_case:nn #1#2
17201 {
17202   \exp:w
17203   \exp_args:Nf \__int_case:nnTF { \int_eval:n {#1} } {#2} { } { }
17204 }
17205 \cs_new:Npn \__int_case:nnTF #1#2#3#4
17206 { \__int_case:nw {#1} #2 {#1} { } \s__int_mark {#3} \s__int_mark {#4} \s__int_stop }
17207 \cs_new:Npn \__int_case:nw #1#2#3
17208 {
17209   \int_compare:nNnTF {#1} = {#2}
17210   { \__int_case_end:nw {#3} }
17211   { \__int_case:nw {#1} }
17212 }
17213 \cs_new:Npn \__int_case_end:nw #1#2#3 \s__int_mark #4#5 \s__int_stop
17214 { \exp_end: #1 #4 }
```

(End definition for `\int_case:nnTF` and others. This function is documented on page 160.)

`\int_if_odd_p:n` A predicate function.

```
\int_if_odd:nTF 17215 \prg_new_conditional:Npnn \int_if_odd:n #1 { p , T , F , TF}
17216 {
17217   \if_int_odd:w \__int_eval:w #1 \__int_eval_end:
17218   \prg_return_true:
```

```

17219     \else:
17220         \prg_return_false:
17221     \fi:
17222 }
17223 \prg_new_conditional:Npnn \int_if_even:n #1 { p , T , F , TF}
17224 {
17225     \reverse_if:N \if_int_odd:w \__int_eval:w #1 \__int_eval_end:
17226     \prg_return_true:
17227 }
17228 \else:
17229     \prg_return_false:
17230 \fi:
17231 }

```

(End definition for `\int_if_odd:nTF` and `\int_if_even:nTF`. These functions are documented on page 160.)

57.6 Integer expression loops

`\int_while_do:nn` These are quite easy given the above functions. The `while` versions test first and then execute the body. The `do_while` does it the other way round.

```

\int_until_do:nn
\int_do_while:nn
\int_do_until:nn
17231 \cs_new:Npn \int_while_do:nn #1#2
17232 {
17233     \int_compare:nT {#1}
17234     {
17235         #2
17236         \int_while_do:nn {#1} {#2}
17237     }
17238 }
17239 \cs_new:Npn \int_until_do:nn #1#2
17240 {
17241     \int_compare:nF {#1}
17242     {
17243         #2
17244         \int_until_do:nn {#1} {#2}
17245     }
17246 }
17247 \cs_new:Npn \int_do_while:nn #1#2
17248 {
17249     #2
17250     \int_compare:nT {#1}
17251     { \int_do_while:nn {#1} {#2} }
17252 }
17253 \cs_new:Npn \int_do_until:nn #1#2
17254 {
17255     #2
17256     \int_compare:nF {#1}
17257     { \int_do_until:nn {#1} {#2} }
17258 }

```

(End definition for `\int_while_do:nn` and others. These functions are documented on page 161.)

`\int_while_do:nNnn` As above but not using the more natural syntax.

```

17259 \cs_new:Npn \int_while_do:nNnn #1#2#3#4

```

```

17260 {
17261     \int_compare:nNnT {#1} #2 {#3}
17262     {
17263         #4
17264         \int_while_do:nNnn {#1} #2 {#3} {#4}
17265     }
17266 }
17267 \cs_new:Npn \int_until_do:nNnn #1#2#3#4
17268 {
17269     \int_compare:nNnF {#1} #2 {#3}
17270     {
17271         #4
17272         \int_until_do:nNnn {#1} #2 {#3} {#4}
17273     }
17274 }
17275 \cs_new:Npn \int_do_while:nNnn #1#2#3#4
17276 {
17277     #4
17278     \int_compare:nNnT {#1} #2 {#3}
17279     { \int_do_while:nNnn {#1} #2 {#3} {#4} }
17280 }
17281 \cs_new:Npn \int_do_until:nNnn #1#2#3#4
17282 {
17283     #4
17284     \int_compare:nNnF {#1} #2 {#3}
17285     { \int_do_until:nNnn {#1} #2 {#3} {#4} }
17286 }

```

(End definition for `\int_while_do:nNnn` and others. These functions are documented on page 161.)

57.7 Integer step functions

`\int_step_function:nnnN` Before all else, evaluate the initial value, step, and final value. Repeating a function by steps first needs a check on the direction of the steps. After that, do the function for the start value then step and loop around. It would be more symmetrical to test for a step size of zero before checking the sign, but we optimize for the most frequent case (positive step).

```

17287 \cs_new:Npn \int_step_function:nnnN #1#2#3
17288 {
17289     \exp_after:wN \__int_step:wwwN
17290     \int_value:w \__int_eval:w #1 \exp_after:wN ;
17291     \int_value:w \__int_eval:w #2 \exp_after:wN ;
17292     \int_value:w \__int_eval:w #3 ;
17293 }
17294 \cs_new:Npn \__int_step:wwwN #1; #2; #3; #4
17295 {
17296     \int_compare:nNnTF {#2} > \c_zero_int
17297     { \__int_step:NwnnnN > }
17298     {
17299         \int_compare:nNnTF {#2} = \c_zero_int
17300         {
17301             \msg_expandable_error:nnn
17302             { kernel } { zero-step } {#4}

```

```

17303         \prg_break:
17304     }
17305     { \__int_step:NwnnN < }
17306 }
17307 #1 ; {#2} {#3} #4
17308 \prg_break_point:
17309 }
17310 \cs_new:Npn \__int_step:NwnnN #1#2 ; #3#4#5
17311 {
17312     \if_int_compare:w #2 #1 #4 \exp_stop_f:
17313         \prg_break:n
17314     \fi:
17315     #5 {#2}
17316     \exp_after:wN \__int_step:NwnnN
17317     \exp_after:wN #1
17318     \int_value:w \__int_eval:w #2 + #3 ; {#3} {#4} #5
17319 }
17320 \cs_new:Npn \int_step_function:nN
17321 { \int_step_function:nnnN { 1 } { 1 } }
17322 \cs_new:Npn \int_step_function:nnN #1
17323 { \int_step_function:nnnN {#1} { 1 } }

```

(End definition for `\int_step_function:nnnN` and others. These functions are documented on page 162.)

```

\int_step_inline:nn
\int_step_inline:nnn
\int_step_inline:nnnn
\int_step_variable:nNn
\int_step_variable:nnNn
\int_step_variable:nnnNn
\__int_step:NNnnnn

```

The approach here is to build a function, with a global integer required to make the nesting safe (as seen in other in line functions), and map that function using `\int_step_function:nnnN`. We put a `\prg_break_point:Nn` so that `map_break` functions from other modules correctly decrement `\g__kernel_prg_map_int` before looking for their own break point. The first argument is `\scan_stop:`, so that no breaking function recognizes this break point as its own.

```

17324 \cs_new_protected:Npn \int_step_inline:nn
17325 { \int_step_inline:nnnn { 1 } { 1 } }
17326 \cs_new_protected:Npn \int_step_inline:nnn #1
17327 { \int_step_inline:nnnn {#1} { 1 } }
17328 \cs_new_protected:Npn \int_step_inline:nnnn
17329 {
17330     \int_gincr:N \g__kernel_prg_map_int
17331     \exp_args:NNc \__int_step:NNnnnn
17332     \cs_gset_protected:Npn
17333     { __int_map_ \int_use:N \g__kernel_prg_map_int :w }
17334 }
17335 \cs_new_protected:Npn \int_step_variable:nNn
17336 { \int_step_variable:nnnNn { 1 } { 1 } }
17337 \cs_new_protected:Npn \int_step_variable:nnNn #1
17338 { \int_step_variable:nnnNn {#1} { 1 } }
17339 \cs_new_protected:Npn \int_step_variable:nnnNn #1#2#3#4#5
17340 {
17341     \int_gincr:N \g__kernel_prg_map_int
17342     \exp_args:NNc \__int_step:NNnnnn
17343     \cs_gset_protected:Npx
17344     { __int_map_ \int_use:N \g__kernel_prg_map_int :w }
17345     {#1}{#2}{#3}
17346 }

```

```

17347         \tl_set:Nn \exp_not:N #4 {##1}
17348         \exp_not:n {#5}
17349     }
17350 }
17351 \cs_new_protected:Npn \__int_step:NNnnnn #1#2#3#4#5#6
17352 {
17353     #1 #2 ##1 {#6}
17354     \int_step_function:nnnN {#3} {#4} {#5} #2
17355     \prg_break_point:Nn \scan_stop: { \int_gdecr:N \g__kernel_prg_map_int }
17356 }

```

(End definition for `\int_step_inline:nn` and others. These functions are documented on page 162.)

57.8 Formatting integers

`\int_to_arabic:n` Nothing exciting here.

```

17357 \cs_new_eq:NN \int_to_arabic:n \int_eval:n

```

(End definition for `\int_to_arabic:n`. This function is documented on page 163.)

`\int_to_symbols:nnn` For conversion of integers to arbitrary symbols the method is in general as follows. The input number (#1) is compared to the total number of symbols available at each place (#2). If the input is larger than the total number of symbols available then the modulus is needed, with one added so that the positions don't have to number from zero. Using an f-type expansion, this is done so that the system is recursive. The actual conversion function therefore gets a 'nice' number at each stage. Of course, if the initial input was small enough then there is no problem and everything is easy.

```

17358 \cs_new:Npn \int_to_symbols:nnn #1#2#3
17359 {
17360     \int_compare:nNnTF {#1} > {#2}
17361     {
17362         \exp_args:NNo \exp_args:No \__int_to_symbols:nnnn
17363         {
17364             \int_case:nn
17365             { 1 + \int_mod:nn { #1 - 1 } {#2} }
17366             {#3}
17367         }
17368         {#1} {#2} {#3}
17369     }
17370     { \int_case:nn {#1} {#3} }
17371 }
17372 \cs_new:Npn \__int_to_symbols:nnnn #1#2#3#4
17373 {
17374     \exp_args:Nf \int_to_symbols:nnn
17375     { \int_div_truncate:nn { #2 - 1 } {#3} } {#3} {#4}
17376     #1
17377 }

```

(End definition for `\int_to_symbols:nnn` and `__int_to_symbols:nnnn`. This function is documented on page 163.)

`\int_to_alpha:n` These both use the above function with input functions that make sense for the alphabet in English.

`\int_to_Alpha:n`

```

17378 \cs_new:Npn \int_to_alph:n #1
17379 {
17380   \int_to_symbols:nnn {#1} { 26 }
17381   {
17382     { 1 } { a }
17383     { 2 } { b }
17384     { 3 } { c }
17385     { 4 } { d }
17386     { 5 } { e }
17387     { 6 } { f }
17388     { 7 } { g }
17389     { 8 } { h }
17390     { 9 } { i }
17391     { 10 } { j }
17392     { 11 } { k }
17393     { 12 } { l }
17394     { 13 } { m }
17395     { 14 } { n }
17396     { 15 } { o }
17397     { 16 } { p }
17398     { 17 } { q }
17399     { 18 } { r }
17400     { 19 } { s }
17401     { 20 } { t }
17402     { 21 } { u }
17403     { 22 } { v }
17404     { 23 } { w }
17405     { 24 } { x }
17406     { 25 } { y }
17407     { 26 } { z }
17408   }
17409 }
17410 \cs_new:Npn \int_to_Alph:n #1
17411 {
17412   \int_to_symbols:nnn {#1} { 26 }
17413   {
17414     { 1 } { A }
17415     { 2 } { B }
17416     { 3 } { C }
17417     { 4 } { D }
17418     { 5 } { E }
17419     { 6 } { F }
17420     { 7 } { G }
17421     { 8 } { H }
17422     { 9 } { I }
17423     { 10 } { J }
17424     { 11 } { K }
17425     { 12 } { L }
17426     { 13 } { M }
17427     { 14 } { N }
17428     { 15 } { O }
17429     { 16 } { P }
17430     { 17 } { Q }
17431     { 18 } { R }

```



```

17432      { 19 } { S }
17433      { 20 } { T }
17434      { 21 } { U }
17435      { 22 } { V }
17436      { 23 } { W }
17437      { 24 } { X }
17438      { 25 } { Y }
17439      { 26 } { Z }
17440    }
17441  }

```

(End definition for `\int_to_alph:n` and `\int_to_Alph:n`. These functions are documented on page 163.)

```

\int_to_base:nn Converting from base ten (#1) to a second base (#2) starts with computing #1: if it is
\int_to_Base:nn a complicated calculation, we shouldn't perform it twice. Then check the sign, store it,
\__int_to_base:nn either - or \c_empty_tl, and feed the absolute value to the next auxiliary function.
\__int_to_Base:nn
\__int_to_base:nnN 17442 \cs_new:Npn \int_to_base:nn #1
\__int_to_Base:nnN 17443 { \exp_args:Nf \__int_to_base:nn { \int_eval:n {#1} } }
\__int_to_base:nnN 17444 \cs_new:Npn \int_to_Base:nn #1
\__int_to_Base:nnN 17445 { \exp_args:Nf \__int_to_Base:nn { \int_eval:n {#1} } }
\__int_to_base:nnN 17446 \cs_new:Npn \__int_to_base:nn #1#2
\__int_to_Base:nnN 17447 {
\__int_to_letter:n 17448   \int_compare:nNnTF {#1} < 0
\__int_to_Letter:n 17449     { \exp_args:No \__int_to_base:nnN { \use_none:n #1 } {#2} - }
17450     { \__int_to_base:nnN {#1} {#2} \c_empty_tl }
17451   }
17452 \cs_new:Npn \__int_to_Base:nn #1#2
17453 {
17454   \int_compare:nNnTF {#1} < 0
17455     { \exp_args:No \__int_to_Base:nnN { \use_none:n #1 } {#2} - }
17456     { \__int_to_Base:nnN {#1} {#2} \c_empty_tl }
17457 }

```

Here, the idea is to provide a recursive system to deal with the input. The output is built up after the end of the function. At each pass, the value in `#1` is checked to see if it is less than the new base (`#2`). If it is, then it is converted directly, putting the sign back in front. On the other hand, if the value to convert is greater than or equal to the new base then the modulus and remainder values are found. The modulus is converted to a symbol and put on the right, and the remainder is carried forward to the next round.

```

17458 \cs_new:Npn \__int_to_base:nnN #1#2#3
17459 {
17460   \int_compare:nNnTF {#1} < {#2}
17461     { \exp_last_unbraced:Nf #3 { \__int_to_letter:n {#1} } }
17462     {
17463       \exp_args:Nf \__int_to_base:nnnN
17464         { \__int_to_letter:n { \int_mod:nn {#1} {#2} } }
17465         {#1}
17466         {#2}
17467         #3
17468     }
17469 }
17470 \cs_new:Npn \__int_to_base:nnnN #1#2#3#4
17471 {
17472   \exp_args:Nf \__int_to_base:nnN

```

```

17473     { \int_div_truncate:nn {#2} {#3} }
17474     {#3}
17475     #4
17476   #1
17477 }
17478 \cs_new:Npn \__int_to_Base:nnN #1#2#3
17479 {
17480   \int_compare:nNnTF {#1} < {#2}
17481   { \exp_last_unbraced:Nf #3 { \__int_to_Letter:n {#1} } }
17482   {
17483     \exp_args:Nf \__int_to_Base:nnnN
17484     { \__int_to_Letter:n { \int_mod:nn {#1} {#2} } }
17485     {#1}
17486     {#2}
17487     #3
17488   }
17489 }
17490 \cs_new:Npn \__int_to_Base:nnnN #1#2#3#4
17491 {
17492   \exp_args:Nf \__int_to_Base:nnN
17493   { \int_div_truncate:nn {#2} {#3} }
17494   {#3}
17495   #4
17496   #1
17497 }

```

Convert to a letter only if necessary, otherwise simply return the value unchanged. It would be cleaner to use `\int_case:nn`, but in our case, the cases are contiguous, so it is forty times faster to use the `\if_case:w` primitive. The first `\exp_after:wN` expands the conditional, jumping to the correct case, the second one expands after the resulting character to close the conditional. Since `#1` might be an expression, and not directly a single digit, we need to evaluate it properly, and expand the trailing `\fi:`.

```

17498 \cs_new:Npn \__int_to_letter:n #1
17499 {
17500   \exp_after:wN \exp_after:wN
17501   \if_case:w \__int_eval:w #1 - 10 \__int_eval_end:
17502   a
17503   \or: b
17504   \or: c
17505   \or: d
17506   \or: e
17507   \or: f
17508   \or: g
17509   \or: h
17510   \or: i
17511   \or: j
17512   \or: k
17513   \or: l
17514   \or: m
17515   \or: n
17516   \or: o
17517   \or: p
17518   \or: q
17519   \or: r

```

```

17520     \or: s
17521     \or: t
17522     \or: u
17523     \or: v
17524     \or: w
17525     \or: x
17526     \or: y
17527     \or: z
17528     \else: \int_value:w \__int_eval:w #1 \exp_after:wN \__int_eval_end:
17529     \fi:
17530 }
17531 \cs_new:Npn \__int_to_Letter:n #1
17532 {
17533   \exp_after:wN \exp_after:wN
17534   \if_case:w \__int_eval:w #1 - 10 \__int_eval_end:
17535     A
17536     \or: B
17537     \or: C
17538     \or: D
17539     \or: E
17540     \or: F
17541     \or: G
17542     \or: H
17543     \or: I
17544     \or: J
17545     \or: K
17546     \or: L
17547     \or: M
17548     \or: N
17549     \or: O
17550     \or: P
17551     \or: Q
17552     \or: R
17553     \or: S
17554     \or: T
17555     \or: U
17556     \or: V
17557     \or: W
17558     \or: X
17559     \or: Y
17560     \or: Z
17561     \else: \int_value:w \__int_eval:w #1 \exp_after:wN \__int_eval_end:
17562     \fi:
17563 }

```

(End definition for `\int_to_base:nn` and others. These functions are documented on page 164.)

`\int_to_bin:n` Wrappers around the generic function.

```

\int_to_hex:n 17564 \cs_new:Npn \int_to_bin:n #1
\int_to_hex:n 17565 { \int_to_base:nn {#1} { 2 } }
\int_to_hex:n 17566 \cs_new:Npn \int_to_hex:n #1
\int_to_hex:n 17567 { \int_to_base:nn {#1} { 16 } }
\int_to_hex:n 17568 \cs_new:Npn \int_to_hex:n #1
\int_to_hex:n 17569 { \int_to_Base:nn {#1} { 16 } }

```

```

17570 \cs_new:Npn \int_to_oct:n #1
17571 { \int_to_base:nn {#1} { 8 } }

```

(End definition for `\int_to_bin:n` and others. These functions are documented on page 164.)

```

\int_to_roman:n The \__int_to_roman:w primitive creates tokens of category code 12 (other). Usually,
\int_to_Roman:n what is actually wanted is letters. The approach here is to convert the output of the
\__int_to_roman:N primitive into letters using appropriate control sequence names. That keeps everything
\__int_to_roman:N expandable. The loop is terminated by the conversion of the Q.
\__int_to_roman_i:w 17572 \cs_new:Npn \int_to_roman:n #1
\__int_to_roman_v:w 17573 {
\__int_to_roman_x:w 17574 \exp_after:wN \__int_to_roman:N
\__int_to_roman_l:w 17575 \__int_to_roman:w \int_eval:n {#1} Q
\__int_to_roman_c:w 17576 }
\__int_to_roman_d:w 17577 \cs_new:Npn \__int_to_roman:N #1
\__int_to_roman_m:w 17578 {
\__int_to_roman_Q:w 17579 \use:c { __int_to_roman_ #1 :w }
\__int_to_Roman_i:w 17580 \__int_to_roman:N
\__int_to_Roman_v:w 17581 }
\__int_to_Roman_x:w 17582 \cs_new:Npn \int_to_Roman:n #1
\__int_to_Roman_l:w 17583 {
\__int_to_Roman_c:w 17584 \exp_after:wN \__int_to_Roman_aux:N
\__int_to_Roman_d:w 17585 \__int_to_roman:w \int_eval:n {#1} Q
\__int_to_Roman_m:w 17586 }
\__int_to_Roman_Q:w 17587 \cs_new:Npn \__int_to_Roman_aux:N #1
17588 {
17589 \use:c { __int_to_Roman_ #1 :w }
17590 \__int_to_Roman_aux:N
17591 }
17592 \cs_new:Npn \__int_to_roman_i:w { i }
17593 \cs_new:Npn \__int_to_roman_v:w { v }
17594 \cs_new:Npn \__int_to_roman_x:w { x }
17595 \cs_new:Npn \__int_to_roman_l:w { l }
17596 \cs_new:Npn \__int_to_roman_c:w { c }
17597 \cs_new:Npn \__int_to_roman_d:w { d }
17598 \cs_new:Npn \__int_to_roman_m:w { m }
17599 \cs_new:Npn \__int_to_roman_Q:w #1 { }
17600 \cs_new:Npn \__int_to_Roman_i:w { I }
17601 \cs_new:Npn \__int_to_Roman_v:w { V }
17602 \cs_new:Npn \__int_to_Roman_x:w { X }
17603 \cs_new:Npn \__int_to_Roman_l:w { L }
17604 \cs_new:Npn \__int_to_Roman_c:w { C }
17605 \cs_new:Npn \__int_to_Roman_d:w { D }
17606 \cs_new:Npn \__int_to_Roman_m:w { M }
17607 \cs_new:Npn \__int_to_Roman_Q:w #1 { }

```

(End definition for `\int_to_roman:n` and others. These functions are documented on page 164.)

57.9 Converting from other formats to integers

`__int_pass_signs:wn` Called as `__int_pass_signs:wn <signs and digits> \s__int_stop {<code>}`, this function leaves in the input stream any sign it finds, then inserts the `<code>` before the first

non-sign token (and removes `\s__int_stop`). More precisely, it deletes any + and passes any - to the input stream, hence should be called in an integer expression.

```

17608 \cs_new:Npn \__int_pass_signs:wn #1
17609 {
17610   \if:w + \if:w - \exp_not:N #1 + \fi: \exp_not:N #1
17611   \exp_after:wN \__int_pass_signs:wn
17612   \else:
17613     \exp_after:wN \__int_pass_signs_end:wn
17614     \exp_after:wN #1
17615   \fi:
17616 }
17617 \cs_new:Npn \__int_pass_signs_end:wn #1 \s__int_stop #2 { #2 #1 }

```

(End definition for `__int_pass_signs:wn` and `__int_pass_signs_end:wn`.)

`\int_from_alph:n` First take care of signs then loop through the input using the recursion quarks. The `__int_from_alph:nN` auxiliary collects in its first argument the value obtained so far, and the auxiliary `__int_from_alph:N` converts one letter to an expression which evaluates to the correct number.

```

17618 \cs_new:Npn \int_from_alph:n #1
17619 {
17620   \int_eval:n
17621   {
17622     \exp_after:wN \__int_pass_signs:wn \tl_to_str:n {#1}
17623     \s__int_stop { \__int_from_alph:nN { 0 } }
17624     \q__int_recursion_tail \q__int_recursion_stop
17625   }
17626 }
17627 \cs_new:Npn \__int_from_alph:nN #1#2
17628 {
17629   \__int_if_recursion_tail_stop_do:Nn #2 {#1}
17630   \exp_args:Nf \__int_from_alph:nN
17631   { \int_eval:n { #1 * 26 + \__int_from_alph:N #2 } }
17632 }
17633 \cs_new:Npn \__int_from_alph:N #1
17634 { '#1 - \int_compare:nNnTF { '#1 } < { 91 } { 64 } { 96 } }

```

(End definition for `\int_from_alph:n`, `__int_from_alph:nN`, and `__int_from_alph:N`. This function is documented on page 164.)

`\int_from_base:nn` Leave the signs into the integer expression, then loop through characters, collecting the value found so far in the first argument of `__int_from_base:nnN`. To convert a single character, `__int_from_base:N` checks first for digits, then distinguishes lower from upper case letters, turning them into the appropriate number. Note that this auxiliary does not use `\int_eval:n`, hence is not safe for general use.

```

17635 \cs_new:Npn \int_from_base:nn #1#2
17636 {
17637   \int_eval:n
17638   {
17639     \exp_after:wN \__int_pass_signs:wn \tl_to_str:n {#1}
17640     \s__int_stop { \__int_from_base:nnN { 0 } {#2} }
17641     \q__int_recursion_tail \q__int_recursion_stop
17642   }
17643 }

```

```

17644 \cs_new:Npn \__int_from_base:nnN #1#2#3
17645 {
17646   \__int_if_recursion_tail_stop_do:Nn #3 {#1}
17647   \exp_args:Nf \__int_from_base:nnN
17648     { \int_eval:n { #1 * #2 + \__int_from_base:N #3 } }
17649     {#2}
17650 }
17651 \cs_new:Npn \__int_from_base:N #1
17652 {
17653   \int_compare:nNnTF { '#1 } < { 58 }
17654     {#1}
17655     { '#1 - \int_compare:nNnTF { '#1 } < { 91 } { 55 } { 87 } }
17656 }

```

(End definition for `\int_from_base:nn`, `__int_from_base:nnN`, and `__int_from_base:N`. This function is documented on page 165.)

`\int_from_bin:n` Wrappers around the generic function.

```

17657 \cs_new:Npn \int_from_bin:n #1
17658 { \int_from_base:nn {#1} { 2 } }
17659 \cs_new:Npn \int_from_hex:n #1
17660 { \int_from_base:nn {#1} { 16 } }
17661 \cs_new:Npn \int_from_oct:n #1
17662 { \int_from_base:nn {#1} { 8 } }

```

(End definition for `\int_from_bin:n`, `\int_from_hex:n`, and `\int_from_oct:n`. These functions are documented on page 165.)

`\c__int_from_roman_i_int` Constants used to convert from Roman numerals to integers.

```

17663 \int_const:cn { c__int_from_roman_i_int } { 1 }
17664 \int_const:cn { c__int_from_roman_v_int } { 5 }
17665 \int_const:cn { c__int_from_roman_x_int } { 10 }
17666 \int_const:cn { c__int_from_roman_l_int } { 50 }
17667 \int_const:cn { c__int_from_roman_c_int } { 100 }
17668 \int_const:cn { c__int_from_roman_d_int } { 500 }
17669 \int_const:cn { c__int_from_roman_m_int } { 1000 }
17670 \int_const:cn { c__int_from_roman_I_int } { 1 }
17671 \int_const:cn { c__int_from_roman_V_int } { 5 }
17672 \int_const:cn { c__int_from_roman_X_int } { 10 }
17673 \int_const:cn { c__int_from_roman_L_int } { 50 }
17674 \int_const:cn { c__int_from_roman_C_int } { 100 }
17675 \int_const:cn { c__int_from_roman_D_int } { 500 }
17676 \int_const:cn { c__int_from_roman_M_int } { 1000 }

```

(End definition for `\c__int_from_roman_i_int` and others.)

`\int_from_roman:n` The method here is to iterate through the input, finding the appropriate value for each letter and building up a sum. This is then evaluated by \TeX . If any unknown letter is found, skip to the closing parenthesis and insert `*0-1` afterwards, to replace the value by `-1`.

```

17677 \cs_new:Npn \int_from_roman:n #1
17678 {
17679   \int_eval:n
17680     {
17681     (

```

```

17682         0
17683         \exp_after:wN \__int_from_roman:NN \tl_to_str:n {#1}
17684         \q__int_recursion_tail \q__int_recursion_tail \q__int_recursion_stop
17685     )
17686 }
17687 }
17688 \cs_new:Npn \__int_from_roman:NN #1#2
17689 {
17690     \__int_if_recursion_tail_stop:N #1
17691     \int_if_exist:cF { c__int_from_roman_ #1 _int }
17692     { \__int_from_roman_error:w }
17693     \__int_if_recursion_tail_stop_do:Nn #2
17694     { + \use:c { c__int_from_roman_ #1 _int } }
17695     \int_if_exist:cF { c__int_from_roman_ #2 _int }
17696     { \__int_from_roman_error:w }
17697     \int_compare:nNnTF
17698     { \use:c { c__int_from_roman_ #1 _int } }
17699     <
17700     { \use:c { c__int_from_roman_ #2 _int } }
17701     {
17702         + \use:c { c__int_from_roman_ #2 _int }
17703         - \use:c { c__int_from_roman_ #1 _int }
17704         \__int_from_roman:NN
17705     }
17706     {
17707         + \use:c { c__int_from_roman_ #1 _int }
17708         \__int_from_roman:NN #2
17709     }
17710 }
17711 \cs_new:Npn \__int_from_roman_error:w #1 \q__int_recursion_stop #2
17712 { #2 * 0 - 1 }

```

(End definition for `\int_from_roman:n`, `__int_from_roman:NN`, and `__int_from_roman_error:w`. This function is documented on page 165.)

57.10 Viewing integer

`\int_show:N` Diagnostics.

```

\int_show:c 17713 \cs_new_eq:NN \int_show:N \__kernel_register_show:N
\__int_show:nN 17714 \cs_generate_variant:Nn \int_show:N { c }

```

(End definition for `\int_show:N` and `__int_show:nN`. This function is documented on page 166.)

`\int_show:n` We don't use the TeX primitive `\showthe` to show integer expressions: this gives a more unified output.

```

17715 \cs_new_protected:Npn \int_show:n
17716 { \msg_show_eval:Nn \int_eval:n }

```

(End definition for `\int_show:n`. This function is documented on page 166.)

`\int_log:N` Diagnostics.

```

\int_log:c 17717 \cs_new_eq:NN \int_log:N \__kernel_register_log:N
17718 \cs_generate_variant:Nn \int_log:N { c }

```

(End definition for `\int_log:n`. This function is documented on page 166.)

`\int_log:n` Similar to `\int_show:n`.

```
17719 \cs_new_protected:Npn \int_log:n
17720 { \msg_log_eval:Nn \int_eval:n }
```

(End definition for `\int_log:n`. This function is documented on page 166.)

57.11 Random integers

`\int_rand:nn` Defined in `l3fp-random`.

(End definition for `\int_rand:nn`. This function is documented on page 165.)

57.12 Constant integers

`\c_zero_int` The zero is defined in `l3basics`.

`\c_one_int`

```
17721 \int_const:Nn \c_one_int { 1 }
```

(End definition for `\c_zero_int` and `\c_one_int`. These variables are documented on page 166.)

`\c_max_int` The largest number allowed is $2^{31} - 1$

```
17722 \int_const:Nn \c_max_int { 2 147 483 647 }
```

(End definition for `\c_max_int`. This variable is documented on page 166.)

`\c_max_char_int` The largest character code is 1114111 (hexadecimal 10FFFF) in \XeTeX and \LuaTeX and 255 in other engines. In many places \pTeX and \upTeX support larger character codes but for instance the values of `\lccode` are restricted to $[0, 255]$.

```
17723 \int_const:Nn \c_max_char_int
17724 {
17725   \if_int_odd:w 0
17726     \cs_if_exist:NT \tex luatexversion:D { 1 }
17727     \cs_if_exist:NT \tex XeTeXversion:D { 1 } ~
17728     "10FFFF
17729   \else:
17730     "FF
17731   \fi:
17732 }
```

(End definition for `\c_max_char_int`. This variable is documented on page 166.)

57.13 Scratch integers

`\l_tmpa_int` We provide two local and two global scratch counters, maybe we need more or less.

```
\l_tmpb_int 17733 \int_new:N \l_tmpa_int
\g_tmpa_int 17734 \int_new:N \l_tmpb_int
\g_tmpb_int 17735 \int_new:N \g_tmpa_int
17736 \int_new:N \g_tmpb_int
```

(End definition for `\l_tmpa_int` and others. These variables are documented on page 166.)

57.14 Integers for earlier modules

<@@=seq>

\l__int_internal_a_int

\l__int_internal_b_int

17737 \int_new:N \l__int_internal_a_int

17738 \int_new:N \l__int_internal_b_int

(End definition for \l__int_internal_a_int and \l__int_internal_b_int.)

17739 \</package>

Chapter 58

l3flag implementation

```
17740 <*package>
```

```
17741 <@@=flag>
```

The following test files are used for this code: m3flag001.

58.1 Non-expandable flag commands

The height h of a flag (initially zero) is stored by setting control sequences of the form `\flag <name> <integer>` to `\relax` for $0 \leq \langle integer \rangle < h$. When a flag is raised, a “trap” function `\flag <name>` is called. The existence of this function is also used to test for the existence of a flag.

\flag_new:n For each flag, we define a “trap” function, which by default simply increases the flag by 1 by letting the appropriate control sequence to `\relax`. This can be done expandably!

```
17742 \cs_new_protected:Npn \flag_new:n #1
17743 {
17744   \cs_new:cpn { flag~#1 } ##1 ;
17745   { \exp_after:wN \use_none:n \cs:w flag~#1~##1 \cs_end: }
17746 }
```

(End definition for \flag_new:n. This function is documented on page 169.)

\flag_clear:n **__flag_clear:wn** Undefine control sequences, starting from the 0 flag, upwards, until reaching an undefined control sequence. We don’t use `\cs_undefine:c` because that would act globally. When the option `check-declarations` is used, check for the function defined by `\flag_new:n`.

```
17747 \cs_new_protected:Npn \flag_clear:n #1 { \__flag_clear:wn 0 ; {#1} }
17748 \cs_new_protected:Npn \__flag_clear:wn #1 ; #2
17749 {
17750   \if_cs_exist:w flag~#2~#1 \cs_end:
17751   \cs_set_eq:cN { flag~#2~#1 } \tex_undefined:D
17752   \exp_after:wN \__flag_clear:wn
17753   \int_value:w \int_eval:w 1 + #1
17754   \else:
17755     \use_i:nnn
17756   \fi:
17757   ; {#2}
17758 }
```

(End definition for \flag_clear:n and __flag_clear:wn. This function is documented on page 169.)

\flag_clear_new:n As for other datatypes, clear the $\langle flag \rangle$ or create a new one, as appropriate.

```

17759 \cs_new_protected:Npn \flag_clear_new:n #1
17760 { \flag_if_exist:nTF {#1} { \flag_clear:n } { \flag_new:n } {#1} }

```

(End definition for \flag_clear_new:n. This function is documented on page 170.)

\flag_show:n Show the height (terminal or log file) using appropriate l3msg auxiliaries.
\flag_log:n
__flag_show:Nn

```

17761 \cs_new_protected:Npn \flag_show:n { \__flag_show:Nn \tl_show:n }
17762 \cs_new_protected:Npn \flag_log:n { \__flag_show:Nn \tl_log:n }
17763 \cs_new_protected:Npn \__flag_show:Nn #1#2
17764 {
17765   \exp_args:Nc \__kernel_chk_defined:NT { flag~#2 }
17766   {
17767     \exp_args:Nx #1
17768     { \tl_to_str:n { flag~#2~height } = \flag_height:n {#2} }
17769   }
17770 }

```

(End definition for \flag_show:n, \flag_log:n, and __flag_show:Nn. These functions are documented on page 170.)

58.2 Expandable flag commands

\flag_if_exist_p:n A flag exist if the corresponding trap \flag $\langle flag\ name \rangle$:n is defined.
\flag_if_exist:nTF

```

17771 \prg_new_conditional:Npnn \flag_if_exist:n #1 { p , T , F , TF }
17772 {
17773   \cs_if_exist:cTF { flag~#1 }
17774   { \prg_return_true: } { \prg_return_false: }
17775 }

```

(End definition for \flag_if_exist:nTF. This function is documented on page 170.)

\flag_if_raised_p:n Test if the flag has a non-zero height, by checking the 0 control sequence.
\flag_if_raised:nTF

```

17776 \prg_new_conditional:Npnn \flag_if_raised:n #1 { p , T , F , TF }
17777 {
17778   \if_cs_exist:w flag~#1~0 \cs_end:
17779   \prg_return_true:
17780   \else:
17781   \prg_return_false:
17782   \fi:
17783 }

```

(End definition for \flag_if_raised:nTF. This function is documented on page 170.)

\flag_height:n Extract the value of the flag by going through all of the control sequences starting from 0.
__flag_height_loop:wn
__flag_height_end:wn

```

17784 \cs_new:Npn \flag_height:n #1 { \__flag_height_loop:wn 0; {#1} }
17785 \cs_new:Npn \__flag_height_loop:wn #1 ; #2
17786 {
17787   \if_cs_exist:w flag~#2~#1 \cs_end:
17788   \exp_after:wN \__flag_height_loop:wn \int_value:w \int_eval:w 1 +
17789   \else:

```

```

17790     \exp_after:wN \_flag_height_end:wn
17791     \fi:
17792     #1 ; {#2}
17793   }
17794 \cs_new:Npn \_flag_height_end:wn #1 ; #2 {#1}

```

(End definition for \flag_height:n, _flag_height_loop:wn, and _flag_height_end:wn. This function is documented on page 170.)

\flag_raise:n Simply apply the trap to the height, after expanding the latter.

```

17795 \cs_new:Npn \flag_raise:n #1
17796 {
17797   \cs:w flag~#1 \exp_after:wN \cs_end:
17798   \int_value:w \flag_height:n {#1} ;
17799 }

```

(End definition for \flag_raise:n. This function is documented on page 170.)

```

17800 </package>

```

Chapter 59

l3clist implementation

The following test files are used for this code: m3clist002.

```
17801 <*package>
17802 <@@=clist>
```

\c_empty_clist An empty comma list is simply an empty token list.

```
17803 \cs_new_eq:NN \c_empty_clist \c_empty_tl
```

(End definition for \c_empty_clist. This variable is documented on page 180.)

\l__clist_internal_clist Scratch space for various internal uses. This comma list variable cannot be declared as such because it comes before \clist_new:N

```
17804 \tl_new:N \l__clist_internal_clist
```

(End definition for \l__clist_internal_clist.)

\s__clist_mark Internal scan marks.

```
\s__clist_stop
17805 \scan_new:N \s__clist_mark
17806 \scan_new:N \s__clist_stop
```

(End definition for \s__clist_mark and \s__clist_stop.)

__clist_use_none_delimit_by_s_mark:w Functions to gobble up to a scan mark.

```
\__clist_use_none_delimit_by_s_stop:w
17807 \cs_new:Npn \__clist_use_none_delimit_by_s_mark:w #1 \s__clist_mark { }
\__clist_use_i_delimit_by_s_stop:nw
17808 \cs_new:Npn \__clist_use_none_delimit_by_s_stop:w #1 \s__clist_stop { }
17809 \cs_new:Npn \__clist_use_i_delimit_by_s_stop:nw #1 #2 \s__clist_stop {#1}
```

(End definition for __clist_use_none_delimit_by_s_mark:w, __clist_use_none_delimit_by_s_stop:w, and __clist_use_i_delimit_by_s_stop:nw.)

__clist_tmp:w A temporary function for various purposes.

```
17810 \cs_new_protected:Npn \__clist_tmp:w { }
```

(End definition for __clist_tmp:w.)

59.1 Removing spaces around items

`__clist_trim_next:w` Called as `\exp:w __clist_trim_next:w \prg_do_nothing: <comma list> ...` it expands to `{<trimmed item>}` where the `<trimmed item>` is the first non-empty result from removing spaces from both ends of comma-delimited items in the `<comma list>`. The `\prg_do_nothing:` marker avoids losing braces. The test for blank items is a somewhat optimized `\tl_if_empty:oTF` construction; if blank, another item is sought, otherwise trim spaces.

```

17811 \cs_new:Npn \__clist_trim_next:w #1 ,
17812 {
17813   \tl_if_empty:oTF { \use_none:nn #1 ? }
17814   { \__clist_trim_next:w \prg_do_nothing: }
17815   { \tl_trim_spaces_apply:oN {#1} \exp_end: }
17816 }
```

(End definition for `__clist_trim_next:w`.)

`__clist_sanitize:n` The auxiliary `__clist_sanitize:Nn` receives a delimiter (`\c_empty_tl` the first time, afterwards a comma) and that item as arguments. Unless we are done with the loop it calls `__clist_wrap_item:w` to unbrace the item (using a comma delimiter is safe since `#2` came from removing spaces from an argument delimited by a comma) and possibly re-brace it if needed.

```

\__clist_sanitize:n
\__clist_sanitize:Nn

17817 \cs_new:Npn \__clist_sanitize:n #1
17818 {
17819   \exp_after:wN \__clist_sanitize:Nn \exp_after:wN \c_empty_tl
17820   \exp:w \__clist_trim_next:w \prg_do_nothing:
17821   #1 , \s__clist_stop \prg_break: , \prg_break_point:
17822 }
17823 \cs_new:Npn \__clist_sanitize:Nn #1#2
17824 {
17825   \__clist_use_none_delimit_by_s_stop:w #2 \s__clist_stop
17826   #1 \__clist_wrap_item:w #2 ,
17827   \exp_after:wN \__clist_sanitize:Nn \exp_after:wN ,
17828   \exp:w \__clist_trim_next:w \prg_do_nothing:
17829 }
```

(End definition for `__clist_sanitize:n` and `__clist_sanitize:Nn`.)

`__clist_if_wrap:nTF` True if the argument must be wrapped to avoid getting altered by some clist operations.
`__clist_if_wrap:w` That is the case whenever the argument

- starts or end with a space or contains a comma,
- is empty, or
- consists of a single braced group.

If the argument starts or ends with a space or contains a comma then one of the three arguments of `__clist_if_wrap:w` will have its end delimiter (partly) in one of the three copies of `#1` in `__clist_if_wrap:nTF`; this has a knock-on effect meaning that the result of the expansion is not empty; in that case, wrap. Otherwise, the argument is safe unless it starts with a brace group (or is empty) and it is empty or consists of a single n-type argument.

```

17830 \prg_new_conditional:Npnn \__clist_if_wrap:n #1 { TF }
```

```

17831 {
17832   \tl_if_empty:oTF
17833   {
17834     \__clist_if_wrap:w
17835     \s__clist_mark ? #1 ~ \s__clist_mark ? ~ #1
17836     \s__clist_mark , ~ \s__clist_mark #1 ,
17837   }
17838   {
17839     \tl_if_head_is_group:nTF { #1 { } }
17840     {
17841       \tl_if_empty:nTF {#1}
17842       { \prg_return_true: }
17843       {
17844         \tl_if_empty:oTF { \use_none:n #1}
17845         { \prg_return_true: }
17846         { \prg_return_false: }
17847       }
17848     }
17849     { \prg_return_false: }
17850   }
17851   { \prg_return_true: }
17852 }
17853 \cs_new:Npn \__clist_if_wrap:w #1 \s__clist_mark ? ~ #2 ~ \s__clist_mark #3 , { }

```

(End definition for `__clist_if_wrap:nTF` and `__clist_if_wrap:w`.)

`__clist_wrap_item:w` Safe items are put in `\exp_not:n`, otherwise we put an extra set of braces.

```

17854 \cs_new:Npn \__clist_wrap_item:w #1 ,
17855   { \__clist_if_wrap:nTF {#1} { \exp_not:n { {#1} } } { \exp_not:n {#1} } }

```

(End definition for `__clist_wrap_item:w`.)

59.2 Allocation and initialisation

`\clist_new:N` Internally, comma lists are just token lists.

```

\clist_new:c
17856 \cs_new_eq:NN \clist_new:N \tl_new:N
17857 \cs_new_eq:NN \clist_new:c \tl_new:c

```

(End definition for `\clist_new:N`. This function is documented on page 172.)

`\clist_const:Nn` Creating and initializing a constant comma list is done by sanitizing all items (stripping spaces and braces).

```

\clist_const:cn
\clist_const:Nx
\clist_const:cx
17858 \cs_new_protected:Npn \clist_const:Nn #1#2
17859   { \tl_const:Nx #1 { \__clist_sanitize:n {#2} } }
17860 \cs_generate_variant:Nn \clist_const:Nn { c , Nx , cx }

```

(End definition for `\clist_const:Nn`. This function is documented on page 172.)

`\clist_clear:N` Clearing comma lists is just the same as clearing token lists.

```

\clist_clear:c
17861 \cs_new_eq:NN \clist_clear:N \tl_clear:N
\clist_gclear:N
17862 \cs_new_eq:NN \clist_clear:c \tl_clear:c
\clist_gclear:c
17863 \cs_new_eq:NN \clist_gclear:N \tl_gclear:N
17864 \cs_new_eq:NN \clist_gclear:c \tl_gclear:c

```

(End definition for `\clist_clear:N` and `\clist_gclear:N`. These functions are documented on page 172.)

```
\clist_clear_new:N Once again a copy from the token list functions.
\clist_clear_new:c 17865 \cs_new_eq:NN \clist_clear_new:N \tl_clear_new:N
\clist_gclear_new:N 17866 \cs_new_eq:NN \clist_clear_new:c \tl_clear_new:c
\clist_gclear_new:c 17867 \cs_new_eq:NN \clist_gclear_new:N \tl_gclear_new:N
17868 \cs_new_eq:NN \clist_gclear_new:c \tl_gclear_new:c
```

(End definition for `\clist_clear_new:N` and `\clist_gclear_new:N`. These functions are documented on page 172.)

```
\clist_set_eq:NN Once again, these are simple copies from the token list functions.
\clist_set_eq:cN 17869 \cs_new_eq:NN \clist_set_eq:NN \tl_set_eq:NN
\clist_set_eq:Nc 17870 \cs_new_eq:NN \clist_set_eq:Nc \tl_set_eq:Nc
\clist_set_eq:cc 17871 \cs_new_eq:NN \clist_set_eq:cN \tl_set_eq:cN
\clist_gset_eq:NN 17872 \cs_new_eq:NN \clist_set_eq:cc \tl_set_eq:cc
\clist_gset_eq:cN 17873 \cs_new_eq:NN \clist_gset_eq:NN \tl_gset_eq:NN
\clist_gset_eq:Nc 17874 \cs_new_eq:NN \clist_gset_eq:Nc \tl_gset_eq:Nc
\clist_gset_eq:cN 17875 \cs_new_eq:NN \clist_gset_eq:cN \tl_gset_eq:cN
17876 \cs_new_eq:NN \clist_gset_eq:cc \tl_gset_eq:cc
```

(End definition for `\clist_set_eq:NN` and `\clist_gset_eq:NN`. These functions are documented on page 172.)

```
\clist_set_from_seq:NN Setting a comma list from a comma-separated list is done using a simple mapping. Safe
\clist_set_from_seq:cN items are put in \exp_not:n, otherwise we put an extra set of braces. The first comma
\clist_set_from_seq:Nc must be removed, except in the case of an empty comma-list.
\clist_set_from_seq:cc 17877 \cs_new_protected:Npn \clist_set_from_seq:NN
\clist_gset_from_seq:NN 17878 { \__clist_set_from_seq:NNNN \clist_clear:N \__kernel_tl_set:Nx }
\clist_gset_from_seq:cN 17879 \cs_new_protected:Npn \clist_gset_from_seq:NN
\clist_gset_from_seq:Nc 17880 { \__clist_set_from_seq:NNNN \clist_gclear:N \__kernel_tl_gset:Nx }
\clist_gset_from_seq:cc 17881 \cs_new_protected:Npn \__clist_set_from_seq:NNNN #1#2#3#4
\__clist_set_from_seq:NNNN 17882 {
\__clist_set_from_seq:n 17883 \seq_if_empty:NTF #4
17884 { #1 #3 }
17885 {
17886 #2 #3
17887 {
17888 \exp_after:wN \use_none:n \exp:w \exp_end_continue_f:w
17889 \seq_map_function:NN #4 \__clist_set_from_seq:n
17890 }
17891 }
17892 }
17893 \cs_new:Npn \__clist_set_from_seq:n #1
17894 {
17895 ,
17896 \__clist_if_wrap:NTF {#1}
17897 { \exp_not:n { {#1} } }
17898 { \exp_not:n {#1} }
17899 }
17900 \cs_generate_variant:Nn \clist_set_from_seq:NN { Nc }
17901 \cs_generate_variant:Nn \clist_set_from_seq:NN { c , cc }
17902 \cs_generate_variant:Nn \clist_gset_from_seq:NN { Nc }
17903 \cs_generate_variant:Nn \clist_gset_from_seq:NN { c , cc }
```


(End definition for `\clist_set_from_seq:NN` and others. These functions are documented on page 172.)

```

\clist_concat:NNN
\clist_concat:ccc
\clist_gconcat:NNN
\clist_gconcat:ccc
__clist_concat:NNNN
17904 \cs_new_protected:Npn \clist_concat:NNN
17905   { __clist_concat:NNNN __kernel_tl_set:Nx }
17906 \cs_new_protected:Npn \clist_gconcat:NNN
17907   { __clist_concat:NNNN __kernel_tl_gset:Nx }
17908 \cs_new_protected:Npn __clist_concat:NNNN #1#2#3#4
17909   {
17910     #1 #2
17911     {
17912       \exp_not:o #3
17913       \clist_if_empty:NF #3 { \clist_if_empty:NF #4 { , } }
17914       \exp_not:o #4
17915     }
17916   }
17917 \cs_generate_variant:Nn \clist_concat:NNN { ccc }
17918 \cs_generate_variant:Nn \clist_gconcat:NNN { ccc }

```

(End definition for `\clist_concat:NNN`, `\clist_gconcat:NNN`, and `__clist_concat:NNNN`. These functions are documented on page 173.)

```

\clist_if_exist_p:N
\clist_if_exist_p:c
\clist_if_exist:NTF
\clist_if_exist:cTF
17919 \prg_new_eq_conditional:NNn \clist_if_exist:N \cs_if_exist:N
17920   { TF , T , F , p }
17921 \prg_new_eq_conditional:NNn \clist_if_exist:c \cs_if_exist:c
17922   { TF , T , F , p }

```

(End definition for `\clist_if_exist:NTF`. This function is documented on page 173.)

59.3 Adding data to comma lists

```

\clist_set:Nn
\clist_set:NV
\clist_set:No
\clist_set:Nx
\clist_set:cn
\clist_set:cV
\clist_set:co
\clist_set:cx
\clist_gset:Nn
\clist_gset:NV
\clist_gset:No
\clist_gset:Nx
\clist_gset:cn
\clist_gset:cV
\clist_gset:co
\clist_gset:cx
\clist_gput_left:Nn
\clist_gput_left:NV
\clist_gput_left:No
\clist_gput_left:Nx
\clist_gput_left:cn
\clist_gput_left:cV
\clist_gput_left:co
\clist_gput_left:cx
__clist_gput_left:NNNn

```

(End definition for `\clist_set:Nn` and `\clist_gset:Nn`. These functions are documented on page 173.)

Everything is based on concatenation after storing in `\l__clist_internal_clist`. This avoids having to worry here about space-trimming and so on.

```

17929 \cs_new_protected:Npn \clist_put_left:Nn
17930   { __clist_put_left:NNNn \clist_concat:NNN \clist_set:Nn }
17931 \cs_new_protected:Npn \clist_gput_left:Nn
17932   { __clist_put_left:NNNn \clist_gconcat:NNN \clist_set:Nn }
17933 \cs_new_protected:Npn __clist_gput_left:NNNn #1#2#3#4
17934   {
17935     #2 \l__clist_internal_clist {#4}
17936     #1 #3 \l__clist_internal_clist #3

```

```

17937 }
17938 \cs_generate_variant:Nn \clist_put_left:Nn { NV , No , Nx }
17939 \cs_generate_variant:Nn \clist_put_left:Nn { c , cV , co , cx }
17940 \cs_generate_variant:Nn \clist_gput_left:Nn { NV , No , Nx }
17941 \cs_generate_variant:Nn \clist_gput_left:Nn { c , cV , co , cx }

```

(End definition for `\clist_put_left:Nn`, `\clist_gput_left:Nn`, and `__clist_put_left:NNNn`. These functions are documented on page 173.)

```

\clist_put_right:Nn
\clist_put_right:NV
\clist_put_right:No
\clist_put_right:Nx
\clist_put_right:cn
\clist_put_right:cV
\clist_put_right:co
\clist_put_right:cx
\clist_gput_right:Nn
\clist_gput_right:NV
\clist_gput_right:No
\clist_gput_right:Nx
\clist_gput_right:cn
\clist_gput_right:cV
\clist_gput_right:co
\clist_gput_right:cx
\__clist_put_right:NNNn

```

```

17942 \cs_new_protected:Npn \clist_put_right:Nn
17943 { \__clist_put_right:NNNn \clist_concat:NNN \clist_set:Nn }
17944 \cs_new_protected:Npn \clist_gput_right:Nn
17945 { \__clist_put_right:NNNn \clist_gconcat:NNN \clist_set:Nn }
17946 \cs_new_protected:Npn \__clist_put_right:NNNn #1#2#3#4
17947 {
17948   #2 \l__clist_internal_clist {#4}
17949   #1 #3 #3 \l__clist_internal_clist
17950 }
17951 \cs_generate_variant:Nn \clist_put_right:Nn { NV , No , Nx }
17952 \cs_generate_variant:Nn \clist_put_right:Nn { c , cV , co , cx }
17953 \cs_generate_variant:Nn \clist_gput_right:Nn { NV , No , Nx }
17954 \cs_generate_variant:Nn \clist_gput_right:Nn { c , cV , co , cx }

```

(End definition for `\clist_put_right:Nn`, `\clist_gput_right:Nn`, and `__clist_put_right:NNNn`. These functions are documented on page 173.)

59.4 Comma lists as stacks

\clist_get:NN Getting an item from the left of a comma list is pretty easy: just trim off the first item using the comma. No need to trim spaces as comma-list *variables* are assumed to have “cleaned-up” items. (Note that grabbing a comma-delimited item removes an outer pair of braces if present, exactly as needed to uncover the underlying item.)

```

17955 \cs_new_protected:Npn \clist_get:NN #1#2
17956 {
17957   \if_meaning:w #1 \c_empty_clist
17958     \tl_set:Nn #2 { \q_no_value }
17959   \else:
17960     \exp_after:wN \__clist_get:wN #1 , \s__clist_stop #2
17961   \fi:
17962 }
17963 \cs_new_protected:Npn \__clist_get:wN #1 , #2 \s__clist_stop #3
17964 { \tl_set:Nn #3 {#1} }
17965 \cs_generate_variant:Nn \clist_get:NN { c }

```

(End definition for `\clist_get:NN` and `__clist_get:wN`. This function is documented on page 178.)

\clist_pop:NN An empty clist leads to `\q_no_value`, otherwise grab until the first comma and assign to the variable. The second argument of `__clist_pop:wwNNN` is a comma list ending in a comma and `\s__clist_mark`, unless the original clist contained exactly one item: then the argument is just `\s__clist_mark`. The next auxiliary picks either `\exp_not:n` or `\use_none:n` as #2, ensuring that the result can safely be an empty comma list.

\clist_gpop:NN

__clist_pop:NNN

__clist_pop:wwNNN

__clist_pop:wN

```

17966 \cs_new_protected:Npn \clist_pop:NN

```

```

17967 { \__clist_pop:NNN \__kernel_tl_set:Nx }
17968 \cs_new_protected:Npn \clist_gpop:NN
17969 { \__clist_pop:NNN \__kernel_tl_gset:Nx }
17970 \cs_new_protected:Npn \__clist_pop:NNN #1#2#3
17971 {
17972   \if_meaning:w #2 \c_empty_clist
17973     \tl_set:Nn #3 { \q_no_value }
17974   \else:
17975     \exp_after:wN \__clist_pop:wwNNN #2 , \s__clist_mark \s__clist_stop #1#2#3
17976   \fi:
17977 }
17978 \cs_new_protected:Npn \__clist_pop:wwNNN #1 , #2 \s__clist_stop #3#4#5
17979 {
17980   \tl_set:Nn #5 {#1}
17981   #3 #4
17982   {
17983     \__clist_pop:wN \prg_do_nothing:
17984     #2 \exp_not:o
17985     , \s__clist_mark \use_none:n
17986     \s__clist_stop
17987   }
17988 }
17989 \cs_new:Npn \__clist_pop:wN #1 , \s__clist_mark #2 #3 \s__clist_stop { #2 {#1} }
17990 \cs_generate_variant:Nn \clist_pop:NN { c }
17991 \cs_generate_variant:Nn \clist_gpop:NN { c }

```

(End definition for \clist_pop:NN and others. These functions are documented on page 178.)

```

\clist_get:NNTF The same, as branching code: very similar to the above.
\clist_get:cNTF 17992 \prg_new_protected_conditional:Npnn \clist_get:NN #1#2 { T , F , TF }
\clist_pop:NNTF 17993 {
\clist_pop:cNTF 17994   \if_meaning:w #1 \c_empty_clist
\clist_gpop:NNTF 17995   \prg_return_false:
\clist_gpop:cNTF 17996   \else:
\__clist_pop_TF:NNN 17997     \exp_after:wN \__clist_get:wN #1 , \s__clist_stop #2
17998     \prg_return_true:
17999   \fi:
18000 }
18001 \prg_generate_conditional_variant:Nnn \clist_get:NN { c } { T , F , TF }
18002 \prg_new_protected_conditional:Npnn \clist_pop:NN #1#2 { T , F , TF }
18003 { \__clist_pop_TF:NNN \__kernel_tl_set:Nx #1 #2 }
18004 \prg_new_protected_conditional:Npnn \clist_gpop:NN #1#2 { T , F , TF }
18005 { \__clist_pop_TF:NNN \__kernel_tl_gset:Nx #1 #2 }
18006 \cs_new_protected:Npn \__clist_pop_TF:NNN #1#2#3
18007 {
18008   \if_meaning:w #2 \c_empty_clist
18009     \prg_return_false:
18010   \else:
18011     \exp_after:wN \__clist_pop:wwNNN #2 , \s__clist_mark \s__clist_stop #1#2#3
18012     \prg_return_true:
18013   \fi:
18014 }
18015 \prg_generate_conditional_variant:Nnn \clist_pop:NN { c } { T , F , TF }
18016 \prg_generate_conditional_variant:Nnn \clist_gpop:NN { c } { T , F , TF }

```

(End definition for `\clist_get:NNTF` and others. These functions are documented on page 178.)

```

\clist_push:Nn Pushing to a comma list is the same as adding on the left.
\clist_push:NV 18017 \cs_new_eq:NN \clist_push:Nn \clist_put_left:Nn
\clist_push:No 18018 \cs_new_eq:NN \clist_push:NV \clist_put_left:NV
\clist_push:Nx 18019 \cs_new_eq:NN \clist_push:No \clist_put_left:No
\clist_push:cn 18020 \cs_new_eq:NN \clist_push:Nx \clist_put_left:Nx
\clist_push:cV 18021 \cs_new_eq:NN \clist_push:cn \clist_put_left:cn
\clist_push:co 18022 \cs_new_eq:NN \clist_push:cV \clist_put_left:cV
\clist_push:cx 18023 \cs_new_eq:NN \clist_push:co \clist_put_left:co
\clist_push:cx 18024 \cs_new_eq:NN \clist_push:cx \clist_put_left:cx
\clist_gpush:Nn 18025 \cs_new_eq:NN \clist_gpush:Nn \clist_gput_left:Nn
\clist_gpush:NV 18026 \cs_new_eq:NN \clist_gpush:NV \clist_gput_left:NV
\clist_gpush:No 18027 \cs_new_eq:NN \clist_gpush:No \clist_gput_left:No
\clist_gpush:Nx 18028 \cs_new_eq:NN \clist_gpush:Nx \clist_gput_left:Nx
\clist_gpush:cn 18029 \cs_new_eq:NN \clist_gpush:cn \clist_gput_left:cn
\clist_gpush:cV 18030 \cs_new_eq:NN \clist_gpush:cV \clist_gput_left:cV
\clist_gpush:co 18031 \cs_new_eq:NN \clist_gpush:co \clist_gput_left:co
\clist_gpush:cx 18032 \cs_new_eq:NN \clist_gpush:cx \clist_gput_left:cx

```

(End definition for `\clist_push:Nn` and `\clist_gpush:Nn`. These functions are documented on page 179.)

59.5 Modifying comma lists

```

\l__clist_internal_remove_clist An internal comma list and a sequence for the removal routines.
\l__clist_internal_remove_seq 18033 \clist_new:N \l__clist_internal_remove_clist
18034 \seq_new:N \l__clist_internal_remove_seq

```

(End definition for `\l__clist_internal_remove_clist` and `\l__clist_internal_remove_seq`.)

```

\clist_remove_duplicates:N Removing duplicates means making a new list then copying it.
\clist_remove_duplicates:c 18035 \cs_new_protected:Npn \clist_remove_duplicates:N
\clist_gremove_duplicates:N 18036 { \__clist_remove_duplicates:NN \clist_set_eq:NN }
\clist_gremove_duplicates:c 18037 \cs_new_protected:Npn \clist_gremove_duplicates:N
\__clist_remove_duplicates:NN 18038 { \__clist_remove_duplicates:NN \clist_gset_eq:NN }
18039 \cs_new_protected:Npn \__clist_remove_duplicates:NN #1#2
18040 {
18041   \clist_clear:N \l__clist_internal_remove_clist
18042   \clist_map_inline:Nn #2
18043   {
18044     \clist_if_in:NnF \l__clist_internal_remove_clist {##1}
18045     {
18046       \tl_put_right:Nx \l__clist_internal_remove_clist
18047       {
18048         \clist_if_empty:NF \l__clist_internal_remove_clist { , }
18049         \__clist_if_wrap:nTF {##1} { \exp_not:n { {##1} } } { \exp_not:n {##1} }
18050       }
18051     }
18052   }
18053   #1 #2 \l__clist_internal_remove_clist
18054 }
18055 \cs_generate_variant:Nn \clist_remove_duplicates:N { c }
18056 \cs_generate_variant:Nn \clist_gremove_duplicates:N { c }

```

(End definition for `\clist_remove_duplicates:N`, `\clist_gremove_duplicates:N`, and `__clist_remove_duplicates:NN`. These functions are documented on page 174.)

```

\clist_remove_all:Nn
\clist_remove_all:cn
\clist_remove_all:NV
\clist_remove_all:cV
\clist_gremove_all:Nn
\clist_gremove_all:cn
\clist_gremove_all:NV
\clist_gremove_all:cV
\__clist_remove_all:NNNn
\__clist_remove_all:w
\__clist_remove_all:

```

The method used here for safe items is very similar to `\tl_replace_all:Nnn`. However, if the item contains commas or leading/trailing spaces, or is empty, or consists of a single brace group, we know that it can only appear within braces so the code would fail; instead just convert to a sequence and do the removal with `l3seq` code (it involves somewhat elaborate code to do most of the work expandably but the final token list comparisons non-expandably).

For “safe” items, build a function delimited by the $\langle item \rangle$ that should be removed, surrounded with commas, and call that function followed by the expanded comma list, and another copy of the $\langle item \rangle$. The loop is controlled by the argument grabbed by `__clist_remove_all:w`: when the item was found, the `\s__clist_mark` delimiter used is the one inserted by `__clist_tmp:w`, and `__clist_use_none_delimit_by_s_stop:w` is deleted. At the end, the final $\langle item \rangle$ is grabbed, and the argument of `__clist_tmp:w` contains `\s__clist_mark`: in that case, `__clist_remove_all:w` removes the second `\s__clist_mark` (inserted by `__clist_tmp:w`), and lets `__clist_use_none_delimit_by_s_stop:w` act.

No brace is lost because items are always grabbed with a leading comma. The result of the first assignment has an extra leading comma, which we remove in a second assignment. Two exceptions: if the clist lost all of its elements, the result is empty, and we shouldn’t remove anything; if the clist started up empty, the first step happens to turn it into a single comma, and the second step removes it.

```

18057 \cs_new_protected:Npn \clist_remove_all:Nn
18058   { \__clist_remove_all:NNNn \clist_set_from_seq:NN \__kernel_tl_set:Nx }
18059 \cs_new_protected:Npn \clist_gremove_all:Nn
18060   { \__clist_remove_all:NNNn \clist_gset_from_seq:NN \__kernel_tl_gset:Nx }
18061 \cs_new_protected:Npn \__clist_remove_all:NNNn #1#2#3#4
18062   {
18063     \__clist_if_wrap:nTF {#4}
18064     {
18065       \seq_set_from_clist:NN \l__clist_internal_remove_seq #3
18066       \seq_remove_all:Nn \l__clist_internal_remove_seq {#4}
18067       #1 #3 \l__clist_internal_remove_seq
18068     }
18069     {
18070       \cs_set:Npn \__clist_tmp:w ##1 , #4 ,
18071       {
18072         ##1
18073         , \s__clist_mark , \__clist_use_none_delimit_by_s_stop:w ,
18074         \__clist_remove_all:
18075       }
18076       #2 #3
18077       {
18078         \exp_after:wN \__clist_remove_all:
18079         #3 , \s__clist_mark , #4 , \s__clist_stop
18080       }
18081       \clist_if_empty:NF #3
18082       {
18083         #2 #3
18084         {
18085           \exp_args:No \exp_not:o

```

```

18086         { \exp_after:wN \use_none:n #3 }
18087     }
18088 }
18089 }
18090 }
18091 \cs_new:Npn \__clist_remove_all:
18092 { \exp_after:wN \__clist_remove_all:w \__clist_tmp:w , }
18093 \cs_new:Npn \__clist_remove_all:w #1 , \s__clist_mark , #2 , { \exp_not:n {#1} }
18094 \cs_generate_variant:Nn \clist_remove_all:Nn { c , NV , cV }
18095 \cs_generate_variant:Nn \clist_gremove_all:Nn { c , NV , cV }

```

(End definition for `\clist_remove_all:Nn` and others. These functions are documented on page 174.)

`\clist_reverse:N` Use `\clist_reverse:n` in an x-expanding assignment. The extra work that `\clist_reverse:n` does to preserve braces and spaces would not be needed for the well-controlled case of N-type comma lists, but the slow-down is not too bad.

`\clist_reverse:c`
`\clist_greverse:N`
`\clist_greverse:c`

```

18096 \cs_new_protected:Npn \clist_reverse:N #1
18097 { \__kernel_tl_set:Nx #1 { \exp_args:No \clist_reverse:n {#1} } }
18098 \cs_new_protected:Npn \clist_greverse:N #1
18099 { \__kernel_tl_gset:Nx #1 { \exp_args:No \clist_reverse:n {#1} } }
18100 \cs_generate_variant:Nn \clist_reverse:N { c }
18101 \cs_generate_variant:Nn \clist_greverse:N { c }

```

(End definition for `\clist_reverse:N` and `\clist_greverse:N`. These functions are documented on page 174.)

`\clist_reverse:n` The reversed token list is built one item at a time, and stored between `\s__clist_stop` and `\s__clist_mark`, in the form of ? followed by zero or more instances of “`<item>`,”. We start from a comma list “`<item1>, ..., <itemn>`”. During the loop, the auxiliary `__clist_reverse:wwNww` receives “`?<itemi>`” as #1, “`<itemi+1>, ..., <itemn>`” as #2, `__clist_reverse:wwNww` as #3, what remains until `\s__clist_stop` as #4, and “`<itemi-1>, ..., <item1>`,” as #5. The auxiliary moves #1 just before #5, with a comma, and calls itself (#3). After the last item is moved, `__clist_reverse:wwNww` receives “`\s__clist_mark __clist_reverse:wwNww !`” as its argument #1, thus `__clist_reverse_end:ww` as its argument #3. This second auxiliary cleans up until the marker !, removes the trailing comma (introduced when the first item was moved after `\s__clist_stop`), and leaves its argument #1 within `\exp_not:n`. There is also a need to remove a leading comma, hence `\exp_not:o` and `\use_none:n`.

```

18102 \cs_new:Npn \clist_reverse:n #1
18103 {
18104     \__clist_reverse:wwNww ? #1 ,
18105     \s__clist_mark \__clist_reverse:wwNww ! ,
18106     \s__clist_mark \__clist_reverse_end:ww
18107     \s__clist_stop ? \s__clist_mark
18108 }
18109 \cs_new:Npn \__clist_reverse:wwNww
18110 #1 , #2 \s__clist_mark #3 #4 \s__clist_stop ? #5 \s__clist_mark
18111 { #3 ? #2 \s__clist_mark #3 #4 \s__clist_stop #1 , #5 \s__clist_mark }
18112 \cs_new:Npn \__clist_reverse_end:ww #1 ! #2 , \s__clist_mark
18113 { \exp_not:o { \use_none:n #2 } }

```

(End definition for `\clist_reverse:n`, `__clist_reverse:wwNww`, and `__clist_reverse_end:ww`. This function is documented on page 174.)

`\clist_sort:Nn` Implemented in `l3sort`.
`\clist_sort:cn`
`\clist_gsort:Nn` (End definition for `\clist_sort:Nn` and `\clist_gsort:Nn`. These functions are documented on page 175.)
`\clist_gsort:cn`

59.6 Comma list conditionals

`\clist_if_empty_p:N` Simple copies from the token list variable material.
`\clist_if_empty_p:c` 18114 `\prg_new_eq_conditional:Nn \clist_if_empty:N \tl_if_empty:N`
`\clist_if_empty:NTF` 18115 `{ p , T , F , TF }`
`\clist_if_empty:cTF` 18116 `\prg_new_eq_conditional:Nn \clist_if_empty:c \tl_if_empty:c`
18117 `{ p , T , F , TF }`

(End definition for `\clist_if_empty:N`. This function is documented on page 175.)

`\clist_if_empty_p:n` As usual, we insert a token (here `?`) before grabbing any argument: this avoids losing
`\clist_if_empty:nTF` braces. The argument of `\tl_if_empty:oTF` is empty if `#1` is `?` followed by blank spaces
`__clist_if_empty_n:w` (besides, this particular variant of the emptiness test is optimized). If the item of the
`__clist_if_empty_n:wNw` comma list is blank, grab the next one. As soon as one item is non-blank, exit: the second
auxiliary grabs `\prg_return_false:` as `#2`, unless every item in the comma list was blank
and the loop actually got broken by the trailing `\s__clist_mark \prg_return_false:`
item.

18118 `\prg_new_conditional:Npnn \clist_if_empty:n #1 { p , T , F , TF }`
18119 `{`
18120 `__clist_if_empty_n:w ? #1`
18121 `, \s__clist_mark \prg_return_false:`
18122 `, \s__clist_mark \prg_return_true:`
18123 `\s__clist_stop`
18124 `}`
18125 `\cs_new:Npn __clist_if_empty_n:w #1 ,`
18126 `{`
18127 `\tl_if_empty:oTF { \use_none:nn #1 ? }`
18128 `{ __clist_if_empty_n:w ? }`
18129 `{ __clist_if_empty_n:wNw }`
18130 `}`
18131 `\cs_new:Npn __clist_if_empty_n:wNw #1 \s__clist_mark #2#3 \s__clist_stop {#2}`

(End definition for `\clist_if_empty:nTF`, `__clist_if_empty_n:w`, and `__clist_if_empty_n:wNw`. This function is documented on page 175.)

`\clist_if_in:NnTF` For “safe” items, we simply surround the comma list, and the item, with commas, then
`\clist_if_in:NvTF` use the same code as for `\tl_if_in:Nn`. For “unsafe” items we follow the same route as
`\clist_if_in:NoTF` `\seq_if_in:Nn`, mapping through the list a comparison function. If found, return true
`\clist_if_in:cnTF` and remove `\prg_return_false:`.
`\clist_if_in:cVTF` 18132 `\prg_new_protected_conditional:Npnn \clist_if_in:Nn #1#2 { T , F , TF }`
`\clist_if_in:coTF` 18133 `{`
`\clist_if_in:nnTF` 18134 `\exp_args:No __clist_if_in_return:nnN #1 {#2} #1`
`\clist_if_in:nVTF` 18135 `}`
`\clist_if_in:noTF` 18136 `\prg_new_protected_conditional:Npnn \clist_if_in:nn #1#2 { T , F , TF }`
`__clist_if_in_return:nnN` 18137 `{`
18138 `\clist_set:Nn \l__clist_internal_clist {#1}`
18139 `\exp_args:No __clist_if_in_return:nnN \l__clist_internal_clist {#2}`
18140 `\l__clist_internal_clist`

```

18141 }
18142 \cs_new_protected:Npn \__clist_if_in_return:nnN #1#2#3
18143 {
18144   \__clist_if_wrap:nTF {#2}
18145   {
18146     \cs_set:Npx \__clist_tmp:w ##1
18147     {
18148       \exp_not:N \tl_if_eq:nnT {##1}
18149       \exp_not:n
18150       {
18151         {#2}
18152         { \clist_map_break:n { \prg_return_true: \use_none:n } }
18153       }
18154     }
18155     \clist_map_function:NN #3 \__clist_tmp:w
18156     \prg_return_false:
18157   }
18158   {
18159     \cs_set:Npn \__clist_tmp:w ##1 ,#2, { }
18160     \tl_if_empty:oTF
18161     { \__clist_tmp:w ,#1, {} {} ,#2, }
18162     { \prg_return_false: } { \prg_return_true: }
18163   }
18164 }
18165 \prg_generate_conditional_variant:Nnn \clist_if_in:Nn
18166 { NV , No , c , cV , co } { T , F , TF }
18167 \prg_generate_conditional_variant:Nnn \clist_if_in:nn
18168 { nV , no } { T , F , TF }

```

(End definition for `\clist_if_in:NnTF`, `\clist_if_in:nnTF`, and `__clist_if_in_return:nnN`. These functions are documented on page 175.)

59.7 Mapping over comma lists

```

\clist_map_function:NN
\clist_map_function:cN
\__clist_map_function:Nw
\__clist_map_function_end:w

```

If the variable is empty, the mapping is skipped (otherwise, that comma-list would be seen as consisting of one empty item). Then loop over the comma-list, grabbing eight comma-delimited items at a time. The end is marked by `\s__clist_stop`, which may not appear in any of the items. Once the last group of eight items has been reached, we go through them more slowly using `__clist_map_function_end:w`. The auxiliary function `__clist_map_function:Nw` is also used in some other clist mappings.

```

18169 \cs_new:Npn \clist_map_function:NN #1#2
18170 {
18171   \clist_if_empty:NF #1
18172   {
18173     \exp_after:wN \__clist_map_function:Nw \exp_after:wN #2 #1 ,
18174     \s__clist_stop , \s__clist_stop , \s__clist_stop , \s__clist_stop ,
18175     \s__clist_stop , \s__clist_stop , \s__clist_stop , \s__clist_stop ,
18176     \prg_break_point:Nn \clist_map_break: { }
18177   }
18178 }
18179 \cs_new:Npn \__clist_map_function:Nw #1 #2, #3, #4, #5, #6, #7, #8, #9,
18180 {
18181   \__clist_use_none_delimit_by_s_stop:w

```



```

18182     #9 \__clist_map_function_end:w \s__clist_stop
18183     #1 {#2} #1 {#3} #1 {#4} #1 {#5} #1 {#6} #1 {#7} #1 {#8} #1 {#9}
18184     \__clist_map_function:Nw #1
18185   }
18186 \cs_new:Npn \__clist_map_function_end:w \s__clist_stop #1#2
18187 {
18188     \__clist_use_none_delimit_by_s_stop:w #2 \clist_map_break: \s__clist_stop
18189     #1 {#2}
18190     \__clist_map_function_end:w \s__clist_stop
18191   }
18192 \cs_generate_variant:Nn \clist_map_function:NN { c }

```

(End definition for `\clist_map_function:NN`, `__clist_map_function:Nw`, and `__clist_map_function_end:w`. This function is documented on page 176.)

`\clist_map_function:nN`
`__clist_map_function_n:Nn`
`__clist_map_unbrace:wn`

The `n`-type mapping function is a bit more awkward, since spaces must be trimmed from each item. Space trimming is again based on `__clist_trim_next:w`. The auxiliary `__clist_map_function_n:Nn` receives as arguments the function, and the next non-empty item (after space trimming but before brace removal). One level of braces is removed by `__clist_map_unbrace:wn`.

```

18193 \cs_new:Npn \clist_map_function:nN #1#2
18194 {
18195     \exp_after:wN \__clist_map_function_n:Nn \exp_after:wN #2
18196     \exp:w \__clist_trim_next:w \prg_do_nothing: #1 ,
18197     \s__clist_stop \clist_map_break: ,
18198     \prg_break_point:Nn \clist_map_break: { }
18199   }
18200 \cs_new:Npn \__clist_map_function_n:Nn #1 #2
18201 {
18202     \__clist_use_none_delimit_by_s_stop:w #2 \s__clist_stop
18203     \__clist_map_unbrace:wn #2 , #1
18204     \exp_after:wN \__clist_map_function_n:Nn \exp_after:wN #1
18205     \exp:w \__clist_trim_next:w \prg_do_nothing:
18206   }
18207 \cs_new:Npn \__clist_map_unbrace:wn #1, #2 { #2 {#1} }

```

(End definition for `\clist_map_function:nN`, `__clist_map_function_n:Nn`, and `__clist_map_unbrace:wn`. This function is documented on page 176.)

`\clist_map_inline:Nn`
`\clist_map_inline:cn`
`\clist_map_inline:nn`

Inline mapping is done by creating a suitable function “on the fly”: this is done globally to avoid any issues with $\mathrm{T}_{\mathrm{E}}\mathrm{X}$ ’s groups. We use a different function for each level of nesting.

Since the mapping is non-expandable, we can perform the space-trimming needed by the `n` version simply by storing the comma-list in a variable. We don’t need a different comma-list for each nesting level: the comma-list is expanded before the mapping starts.

```

18208 \cs_new_protected:Npn \clist_map_inline:Nn #1#2
18209 {
18210     \clist_if_empty:NF #1
18211     {
18212         \int_gincr:N \g__kernel_prg_map_int
18213         \cs_gset_protected:cpn
18214         { \__clist_map_ \int_use:N \g__kernel_prg_map_int :w } ##1 {#2}
18215         \exp_last_unbraced:Nco \__clist_map_function:Nw
18216         { \__clist_map_ \int_use:N \g__kernel_prg_map_int :w }

```

```

18217         #1 ,
18218         \s__clist_stop , \s__clist_stop , \s__clist_stop , \s__clist_stop ,
18219         \s__clist_stop , \s__clist_stop , \s__clist_stop , \s__clist_stop ,
18220         \prg_break_point:Nn \clist_map_break:
18221         { \int_gdecr:N \g__kernel_prg_map_int }
18222     }
18223 }
18224 \cs_new_protected:Npn \clist_map_inline:nn #1
18225 {
18226     \clist_set:Nn \l__clist_internal_clist {#1}
18227     \clist_map_inline:Nn \l__clist_internal_clist
18228 }
18229 \cs_generate_variant:Nn \clist_map_inline:Nn { c }

```

(End definition for `\clist_map_inline:Nn` and `\clist_map_inline:nn`. These functions are documented on page 176.)

`\clist_map_variable:NNn` The N-type version is a straightforward application of `\clist_map_tokens:Nn`, calling `__clist_map_variable:Nnn` for each item to assign the variable and run the user's code. The n-type version is *not* implemented in terms of the n-type function `\clist_map_tokens:Nn`, because here we are allowed to clean up the n-type comma list non-expandably.

```

18230 \cs_new_protected:Npn \clist_map_variable:NNn #1#2#3
18231 { \clist_map_tokens:Nn #1 { \__clist_map_variable:Nnn #2 {#3} } }
18232 \cs_generate_variant:Nn \clist_map_variable:NNn { c }
18233 \cs_new_protected:Npn \__clist_map_variable:Nnn #1#2#3
18234 { \tl_set:Nn #1 {#3} #2 }
18235 \cs_new_protected:Npn \clist_map_variable:nNn #1
18236 {
18237     \clist_set:Nn \l__clist_internal_clist {#1}
18238     \clist_map_variable:NNn \l__clist_internal_clist
18239 }

```

(End definition for `\clist_map_variable:NNn`, `\clist_map_variable:nNn`, and `__clist_map_variable:Nnn`. These functions are documented on page 176.)

`\clist_map_tokens:Nn` Essentially a copy of `\clist_map_function:NN` with braces added.

```

\clist_map_tokens:cn
\__clist_map_tokens:nw
\__clist_map_tokens_end:w
18240 \cs_new:Npn \clist_map_tokens:Nn #1#2
18241 {
18242     \clist_if_empty:NF #1
18243     {
18244         \exp_last_unbraced:Nno \__clist_map_tokens:nw {#2} #1 ,
18245         \s__clist_stop , \s__clist_stop , \s__clist_stop , \s__clist_stop ,
18246         \s__clist_stop , \s__clist_stop , \s__clist_stop , \s__clist_stop ,
18247         \prg_break_point:Nn \clist_map_break: { }
18248     }
18249 }
18250 \cs_new:Npn \__clist_map_tokens:nw #1 #2, #3, #4, #5, #6, #7, #8, #9,
18251 {
18252     \__clist_use_none_delimit_by_s_stop:w
18253     #9 \__clist_map_tokens_end:w \s__clist_stop
18254     \use:n {#1} {#2} \use:n {#1} {#3} \use:n {#1} {#4} \use:n {#1} {#5}
18255     \use:n {#1} {#6} \use:n {#1} {#7} \use:n {#1} {#8} \use:n {#1} {#9}
18256     \__clist_map_tokens:nw {#1}

```

```

18257 }
18258 \cs_new:Npn \__clist_map_tokens_end:w \s__clist_stop \use:n #1#2
18259 {
18260   \__clist_use_none_delimit_by_s_stop:w #2 \clist_map_break: \s__clist_stop
18261   #1 {#2}
18262   \__clist_map_tokens_end:w \s__clist_stop
18263 }
18264 \cs_generate_variant:Nn \clist_map_tokens:Nn { c }

```

(End definition for `\clist_map_tokens:Nn`, `__clist_map_tokens:nw`, and `__clist_map_tokens_end:w`. This function is documented on page 176.)

`\clist_map_tokens:nn` Similar to `\clist_map_function:nN` but with a different way of grabbing items because we cannot use `\exp_after:wN` to pass the `{code}`.
`__clist_map_tokens_n:nw`

```

18265 \cs_new:Npn \clist_map_tokens:nn #1#2
18266 {
18267   \__clist_map_tokens_n:nw {#2}
18268   \prg_do_nothing: #1 , \s__clist_stop \clist_map_break: ,
18269   \prg_break_point:Nn \clist_map_break: { }
18270 }
18271 \cs_new:Npn \__clist_map_tokens_n:nw #1#2 ,
18272 {
18273   \tl_if_empty:oF { \use_none:nn #2 ? }
18274   {
18275     \__clist_use_none_delimit_by_s_stop:w #2 \s__clist_stop
18276     \tl_trim_spaces_apply:oN {#2} \use_ii_i:nn
18277     \__clist_map_unbrace:wn , {#1}
18278   }
18279   \__clist_map_tokens_n:nw {#1} \prg_do_nothing:
18280 }

```

(End definition for `\clist_map_tokens:nn` and `__clist_map_tokens_n:nw`. This function is documented on page 176.)

`\clist_map_break:` The break statements use the general `\prg_map_break:Nn` mechanism.

`\clist_map_break:n`

```

18281 \cs_new:Npn \clist_map_break:
18282 { \prg_map_break:Nn \clist_map_break: { } }
18283 \cs_new:Npn \clist_map_break:n
18284 { \prg_map_break:Nn \clist_map_break: }

```

(End definition for `\clist_map_break:` and `\clist_map_break:n`. These functions are documented on page 176.)

`\clist_count:N` Counting the items in a comma list is done using the same approach as for other token
`\clist_count:c` count functions: turn each entry into a +1 then use integer evaluation to actually do the
`\clist_count:n` mathematics. In the case of an n-type comma-list, we could of course use `\clist_map_`
`__clist_count:n` **`function:nN`**, but that is very slow, because it carefully removes spaces. Instead, we loop
`__clist_count:w` manually, and skip blank items (but not `{}`, hence the extra spaces).

```

18285 \cs_new:Npn \clist_count:N #1
18286 {
18287   \int_eval:n
18288   {
18289     0
18290     \clist_map_function:NN #1 \__clist_count:n
18291   }

```

```

18292 }
18293 \cs_generate_variant:Nn \clist_count:N { c }
18294 \cs_new:Npn \__clist_count:n #1 { + 1 }
18295 \cs_set_protected:Npn \__clist_tmp:w #1
18296 {
18297   \cs_new:Npn \clist_count:n ##1
18298   {
18299     \int_eval:n
18300     {
18301       0
18302       \__clist_count:w #1
18303       ##1 , \s__clist_stop \prg_break: , \prg_break_point:
18304     }
18305   }
18306   \cs_new:Npn \__clist_count:w ##1 ,
18307   {
18308     \__clist_use_none_delimit_by_s_stop:w ##1 \s__clist_stop
18309     \tl_if_blank:nF {##1} { + 1 }
18310     \__clist_count:w #1
18311   }
18312 }
18313 \exp_args:No \__clist_tmp:w \c_space_tl

```

(End definition for `\clist_count:N` and others. These functions are documented on page 177.)

59.8 Using comma lists

`\clist_use:Nnnn` First check that the variable exists. Then count the items in the comma list. If it has none, output nothing. If it has one item, output that item, brace stripped (note that space-trimming has already been done when the comma list was assigned). If it has two, place the *<separator between two>* in the middle.

`__clist_use:nwwwnwn` Otherwise, `__clist_use:nwwwnwn` takes the following arguments; 1: a *<separator>*, 2, 3, 4: three items from the comma list (or quarks), 5: the rest of the comma list, 6: a *<continuation>* function (`use_ii` or `use_iii` with its *<separator>* argument), 7: junk, and 8: the temporary result, which is built in a brace group following `\s__clist_stop`. The *<separator>* and the first of the three items are placed in the result, then we use the *<continuation>*, placing the remaining two items after it. When we begin this loop, the three items really belong to the comma list, the first `\s__clist_mark` is taken as a delimiter to the `use_ii` function, and the continuation is `use_ii` itself. When we reach the last two items of the original token list, `\s__clist_mark` is taken as a third item, and now the second `\s__clist_mark` serves as a delimiter to `use_ii`, switching to the other *<continuation>*, `use_iii`, which uses the *<separator between final two>*.

```

18314 \cs_new:Npn \clist_use:Nnnn #1#2#3#4
18315 {
18316   \clist_if_exist:NTF #1
18317   {
18318     \int_case:nnF { \clist_count:N #1 }
18319     {
18320       { 0 } { }
18321       { 1 } { \exp_after:wN \__clist_use:wnn #1 , , { } }
18322       { 2 } { \exp_after:wN \__clist_use:wnn #1 , {#2} }
18323     }

```

```

18324         {
18325             \exp_after:wN \__clist_use:nwwwnwn
18326             \exp_after:wN { \exp_after:wN } #1 ,
18327             \s__clist_mark , { \__clist_use:nwwwnwn {#3} }
18328             \s__clist_mark , { \__clist_use:nwn {#4} }
18329             \s__clist_stop { }
18330         }
18331     }
18332     {
18333         \msg_expandable_error:nnn
18334         { kernel } { bad-variable } {#1}
18335     }
18336 }
18337 \cs_generate_variant:Nn \clist_use:Nnnn { c }
18338 \cs_new:Npn \__clist_use:wN #1 , #2 , #3 { \exp_not:n { #1 #3 #2 } }
18339 \cs_new:Npn \__clist_use:nwwwnwn
18340     #1#2 , #3 , #4 , #5 \s__clist_mark , #6#7 \s__clist_stop #8
18341     { #6 {#3} , {#4} , #5 \s__clist_mark , {#6} #7 \s__clist_stop { #8 #1 #2 } }
18342 \cs_new:Npn \__clist_use:nwn #1#2 , #3 \s__clist_stop #4
18343     { \exp_not:n { #4 #1 #2 } }
18344 \cs_new:Npn \clist_use:Nn #1#2
18345     { \clist_use:Nnnn #1 {#2} {#2} {#2} }
18346 \cs_generate_variant:Nn \clist_use:Nn { c }

```

(End definition for `\clist_use:Nnnn` and others. These functions are documented on page 177.)

`\clist_use:nnnn` Items are grabbed by `__clist_use:Nw`, which detects blank items with a `\tl_if_empty:oTF` test (in which case it recurses). Non-blank items are either the end of the list, in which case the argument `#1` of `__clist_use:Nw` is used to properly end the list, or are normal items, which must be trimmed and properly unbraced. As we find successive items, the long list of `__clist_use:Nw` calls gets shortened and we end up calling `__clist_use_more:w` once we have found 3 items. This auxiliary leaves the first-found item and the general separator, and calls `__clist_use:Nw` to find more items. A subtlety is that we use `__clist_use_end:w` both in the case of a two-item list and for the last two items of a general list: to get the correct separator, `__clist_use_more:w` replaces the separator-of-two by the last-separator when called, namely as soon as we have found three items.

```

18347 \cs_new:Npn \clist_use:nnnn #1#2#3#4
18348     {
18349         \__clist_use:Nw \__clist_use_none_delimit_by_s_stop:w
18350         \__clist_use:Nw \__clist_use_one:w
18351         \__clist_use:Nw \__clist_use_end:w
18352         \__clist_use_more:w ;
18353         {#2} {#3} {#4} ;
18354         \prg_do_nothing: #1 , \s__clist_mark ,
18355         \s__clist_stop
18356     }
18357 \cs_new:Npn \__clist_use:Nw #1#2 ; #3 ; #4 ,
18358     {
18359         \tl_if_empty:oTF { \use_none:nn #4 ? }
18360         { \__clist_use:Nw #1#2 ; }
18361         {
18362             \__clist_use_none_delimit_by_s_mark:w #4 #1 \s__clist_mark

```

```

18363         \tl_trim_spaces_apply:oN {#4} \use_ii_i:nn
18364         \__clist_map_unbrace:wn , { #2 ; }
18365     }
18366     #3 ; \prg_do_nothing:
18367 }
18368 \cs_new:Npn \__clist_use_one:w \s__clist_mark #1 , #2#3#4 \s__clist_stop
18369 { \exp_not:n {#3} }
18370 \cs_new:Npn \__clist_use_end:w
18371     \s__clist_mark #1 , #2#3#4#5#6 \s__clist_stop
18372 { \exp_not:n { #4 #5 #3 } }
18373 \cs_new:Npn \__clist_use_more:w ; #1#2#3#4#5#6 ;
18374 {
18375     \exp_not:n { #3 #5 }
18376     \__clist_use:Nw \__clist_use_end:w \__clist_use_more:w ;
18377     {#1} {#2} {#6} {#5} {#6} ;
18378 }
18379 \cs_new:Npn \clist_use:nn #1#2 { \clist_use:nnnn {#1} {#2} {#2} {#2} }

```

(End definition for `\clist_use:nnnn` and others. These functions are documented on page 178.)

59.9 Using a single item

`\clist_item:Nn` To avoid needing to test the end of the list at each step, we first compute the $\langle length \rangle$ of the list. If the item number is 0, less than $-\langle length \rangle$, or more than $\langle length \rangle$, the result is empty. If it is negative, but not less than $-\langle length \rangle$, add $\langle length \rangle + 1$ to the item number before performing the loop. The loop itself is very simple, return the item if the counter reached 1, otherwise, decrease the counter and repeat.

```

\__clist_item:nnnN
\__clist_item:ffoN
\__clist_item:ffnN
\__clist_item_N_loop:nw
18380 \cs_new:Npn \clist_item:Nn #1#2
18381 {
18382     \__clist_item:ffoN
18383     { \clist_count:N #1 }
18384     { \int_eval:n {#2} }
18385     #1
18386     \__clist_item_N_loop:nw
18387 }
18388 \cs_new:Npn \__clist_item:nnnN #1#2#3#4
18389 {
18390     \int_compare:nNnTF {#2} < 0
18391     {
18392         \int_compare:nNnTF {#2} < { - #1 }
18393         { \__clist_use_none_delimit_by_s_stop:w }
18394         { \exp_args:Nf #4 { \int_eval:n { #2 + 1 + #1 } } }
18395     }
18396     {
18397         \int_compare:nNnTF {#2} > {#1}
18398         { \__clist_use_none_delimit_by_s_stop:w }
18399         { #4 {#2} }
18400     }
18401     { } , #3 , \s__clist_stop
18402 }
18403 \cs_generate_variant:Nn \__clist_item:nnnN { ffo, ff }
18404 \cs_new:Npn \__clist_item_N_loop:nw #1 #2,
18405 {

```

```

18406 \int_compare:nNnTF {#1} = 0
18407 { \__clist_use_i_delimit_by_s_stop:nw { \exp_not:n {#2} } }
18408 { \exp_args:Nf \__clist_item_N_loop:nw { \int_eval:n { #1 - 1 } } }
18409 }
18410 \cs_generate_variant:Nn \clist_item:Nn { c }

```

(End definition for `\clist_item:Nn`, `__clist_item:nnnN`, and `__clist_item_N_loop:nw`. This function is documented on page 179.)

`\clist_item:nn` This starts in the same way as `\clist_item:Nn` by counting the items of the comma list. The final item should be space-trimmed before being brace-stripped, hence we insert a couple of odd-looking `\prg_do_nothing:` to avoid losing braces. Blank items are ignored.

```

\__clist_item_n:nw
\__clist_item_n_loop:nw
\__clist_item_n_end:n
\__clist_item_n_strip:n
\__clist_item_n_strip:w
18411 \cs_new:Npn \clist_item:nn #1#2
18412 {
18413   \__clist_item:ffnN
18414   { \clist_count:n {#1} }
18415   { \int_eval:n {#2} }
18416   {#1}
18417   \__clist_item_n:nw
18418 }
18419 \cs_new:Npn \__clist_item_n:nw #1
18420 { \__clist_item_n_loop:nw {#1} \prg_do_nothing: }
18421 \cs_new:Npn \__clist_item_n_loop:nw #1 #2,
18422 {
18423   \exp_args:No \tl_if_blank:nTF {#2}
18424   { \__clist_item_n_loop:nw {#1} \prg_do_nothing: }
18425   {
18426     \int_compare:nNnTF {#1} = 0
18427     { \exp_args:No \__clist_item_n_end:n {#2} }
18428     {
18429       \exp_args:Nf \__clist_item_n_loop:nw
18430       { \int_eval:n { #1 - 1 } }
18431       \prg_do_nothing:
18432     }
18433   }
18434 }
18435 \cs_new:Npn \__clist_item_n_end:n #1 #2 \s_clist_stop
18436 { \tl_trim_spaces_apply:nN {#1} \__clist_item_n_strip:n }
18437 \cs_new:Npn \__clist_item_n_strip:n #1 { \__clist_item_n_strip:w #1 , }
18438 \cs_new:Npn \__clist_item_n_strip:w #1 , { \exp_not:n {#1} }

```

(End definition for `\clist_item:nn` and others. This function is documented on page 179.)

`\clist_rand_item:n` The N-type function is not implemented through the n-type function for efficiency: for instance comma-list variables do not require space-trimming of their items. Even testing for emptiness of an n-type comma-list is slow, so we count items first and use that both for the emptiness test and the pseudo-random integer. Importantly, `\clist_item:Nn` and `\clist_item:nn` only evaluate their argument once.

```

18439 \cs_new:Npn \clist_rand_item:n #1
18440 { \exp_args:Nf \__clist_rand_item:nn { \clist_count:n {#1} } {#1} }
18441 \cs_new:Npn \__clist_rand_item:nn #1#2
18442 {
18443   \int_compare:nNnF {#1} = 0
18444   { \clist_item:nn {#2} { \int_rand:nn { 1 } {#1} } }

```

```

18445 }
18446 \cs_new:Npn \clist_rand_item:N #1
18447 {
18448   \clist_if_empty:NF #1
18449   { \clist_item:Nn #1 { \int_rand:nn { 1 } { \clist_count:N #1 } } }
18450 }
18451 \cs_generate_variant:Nn \clist_rand_item:N { c }

```

(End definition for `\clist_rand_item:n`, `\clist_rand_item:N`, and `__clist_rand_item:nn`. These functions are documented on page 180.)

59.10 Viewing comma lists

```

\clist_show:N Apply the general \__kernel_chk_tl_type:NnnT with \exp_not:o #2 serving as a
\clist_show:c dummy code to prevent a check performed by this auxiliary.
\clist_log:N
\clist_log:c
\__clist_show:NN
18452 \cs_new_protected:Npn \clist_show:N { \__clist_show:NN \msg_show:nnxxxx }
18453 \cs_generate_variant:Nn \clist_show:N { c }
18454 \cs_new_protected:Npn \clist_log:N { \__clist_show:NN \msg_log:nnxxxx }
18455 \cs_generate_variant:Nn \clist_log:N { c }
18456 \cs_new_protected:Npn \__clist_show:NN #1#2
18457 {
18458   \__kernel_chk_tl_type:NnnT #2 { clist } { \exp_not:o #2 }
18459   {
18460     \int_compare:nNnTF { \clist_count:N #2 }
18461     = { \exp_args:No \clist_count:n #2 }
18462     {
18463       #1 { clist } { show }
18464       { \token_to_str:N #2 }
18465       { \clist_map_function:NN #2 \msg_show_item:n }
18466       { } { }
18467     }
18468     {
18469       \msg_error:nnxx { clist } { non-clist }
18470       { \token_to_str:N #2 } { \tl_to_str:N #2 }
18471     }
18472   }
18473 }

```

(End definition for `\clist_show:N`, `\clist_log:N`, and `__clist_show:NN`. These functions are documented on page 180.)

```

\clist_show:n A variant of the above: no existence check, empty first argument for the message.
\clist_log:n
\__clist_show:Nn
18474 \cs_new_protected:Npn \clist_show:n { \__clist_show:Nn \msg_show:nnxxxx }
18475 \cs_new_protected:Npn \clist_log:n { \__clist_show:Nn \msg_log:nnxxxx }
18476 \cs_new_protected:Npn \__clist_show:Nn #1#2
18477 {
18478   #1 { clist } { show }
18479   { } { \clist_map_function:nN {#2} \msg_show_item:n } { } { }
18480 }

```

(End definition for `\clist_show:n`, `\clist_log:n`, and `__clist_show:Nn`. These functions are documented on page 180.)

59.11 Scratch comma lists

```
\l_tmpa_clist Temporary comma list variables.  
\l_tmpb_clist 18481 \clist_new:N \l_tmpa_clist  
\g_tmpa_clist 18482 \clist_new:N \l_tmpb_clist  
\g_tmpb_clist 18483 \clist_new:N \g_tmpa_clist  
18484 \clist_new:N \g_tmpb_clist
```

(End definition for \l_tmpa_clist and others. These variables are documented on page 180.)

```
18485 \endpackage
```

Chapter 60

l3token implementation

```
18486 <*package>
18487 <*tex>
18488 <@@=char>
```

60.1 Internal auxiliaries

`\s__char_stop` Internal scan mark.

```
18489 \scan_new:N \s__char_stop
```

(End definition for \s__char_stop.)

`\q__char_no_value` Internal recursion quarks.

```
18490 \quark_new:N \q__char_no_value
```

(End definition for \q__char_no_value.)

`__char_quark_if_no_value_p:N` Functions to query recursion quarks.

```
\__char_quark_if_no_value:NTF 18491 \__kernel_quark_new_conditional:Nn \__char_quark_if_no_value:N { TF }
```

(End definition for __char_quark_if_no_value:NTF.)

60.2 Manipulating and interrogating character tokens

`\char_set_catcode:nn` Simple wrappers around the primitives.

`\char_value_catcode:n`

`\char_show_value_catcode:n`

```
18492 \cs_new_protected:Npn \char_set_catcode:nn #1#2
```

```
18493 { \tex_catcode:D \int_eval:n {#1} = \int_eval:n {#2} \exp_stop_f: }
```

```
18494 \cs_new:Npn \char_value_catcode:n #1
```

```
18495 { \tex_the:D \tex_catcode:D \int_eval:n {#1} \exp_stop_f: }
```

```
18496 \cs_new_protected:Npn \char_show_value_catcode:n #1
```

```
18497 { \exp_args:Nf \tl_show:n { \char_value_catcode:n {#1} } }
```

(End definition for \char_set_catcode:nn, \char_value_catcode:n, and \char_show_value_catcode:n. These functions are documented on page 184.)

```

\char_set_catcode_escape:N
  \char_set_catcode_group_begin:N
  \char_set_catcode_group_end:N
  \char_set_catcode_math_toggle:N
  \char_set_catcode_alignment:N
\char_set_catcode_end_line:N
  \char_set_catcode_parameter:N
  \char_set_catcode_math_superscript:N
  \char_set_catcode_math_subscript:N
\char_set_catcode_ignore:N
\char_set_catcode_space:N
\char_set_catcode_letter:N
\char_set_catcode_other:N
\char_set_catcode_active:N
\char_set_catcode_comment:N
\char_set_catcode_invalid:N

18498 \cs_new_protected:Npn \char_set_catcode_escape:N #1
18499   { \char_set_catcode:nn { '#1 } { 0 } }
18500 \cs_new_protected:Npn \char_set_catcode_group_begin:N #1
18501   { \char_set_catcode:nn { '#1 } { 1 } }
18502 \cs_new_protected:Npn \char_set_catcode_group_end:N #1
18503   { \char_set_catcode:nn { '#1 } { 2 } }
18504 \cs_new_protected:Npn \char_set_catcode_math_toggle:N #1
18505   { \char_set_catcode:nn { '#1 } { 3 } }
18506 \cs_new_protected:Npn \char_set_catcode_alignment:N #1
18507   { \char_set_catcode:nn { '#1 } { 4 } }
18508 \cs_new_protected:Npn \char_set_catcode_end_line:N #1
18509   { \char_set_catcode:nn { '#1 } { 5 } }
18510 \cs_new_protected:Npn \char_set_catcode_parameter:N #1
18511   { \char_set_catcode:nn { '#1 } { 6 } }
18512 \cs_new_protected:Npn \char_set_catcode_math_superscript:N #1
18513   { \char_set_catcode:nn { '#1 } { 7 } }
18514 \cs_new_protected:Npn \char_set_catcode_math_subscript:N #1
18515   { \char_set_catcode:nn { '#1 } { 8 } }
18516 \cs_new_protected:Npn \char_set_catcode_ignore:N #1
18517   { \char_set_catcode:nn { '#1 } { 9 } }
18518 \cs_new_protected:Npn \char_set_catcode_space:N #1
18519   { \char_set_catcode:nn { '#1 } { 10 } }
18520 \cs_new_protected:Npn \char_set_catcode_letter:N #1
18521   { \char_set_catcode:nn { '#1 } { 11 } }
18522 \cs_new_protected:Npn \char_set_catcode_other:N #1
18523   { \char_set_catcode:nn { '#1 } { 12 } }
18524 \cs_new_protected:Npn \char_set_catcode_active:N #1
18525   { \char_set_catcode:nn { '#1 } { 13 } }
18526 \cs_new_protected:Npn \char_set_catcode_comment:N #1
18527   { \char_set_catcode:nn { '#1 } { 14 } }
18528 \cs_new_protected:Npn \char_set_catcode_invalid:N #1
18529   { \char_set_catcode:nn { '#1 } { 15 } }

```

(End definition for `\char_set_catcode_escape:N` and others. These functions are documented on page 183.)

```

\char_set_catcode_escape:n
  \char_set_catcode_group_begin:n
  \char_set_catcode_group_end:n
  \char_set_catcode_math_toggle:n
  \char_set_catcode_alignment:n
\char_set_catcode_end_line:n
  \char_set_catcode_parameter:n
  \char_set_catcode_math_superscript:n
  \char_set_catcode_math_subscript:n
\char_set_catcode_ignore:n
\char_set_catcode_space:n
\char_set_catcode_letter:n
\char_set_catcode_other:n
\char_set_catcode_active:n
\char_set_catcode_comment:n
\char_set_catcode_invalid:n

18530 \cs_new_protected:Npn \char_set_catcode_escape:n #1
18531   { \char_set_catcode:nn {#1} { 0 } }
18532 \cs_new_protected:Npn \char_set_catcode_group_begin:n #1
18533   { \char_set_catcode:nn {#1} { 1 } }
18534 \cs_new_protected:Npn \char_set_catcode_group_end:n #1
18535   { \char_set_catcode:nn {#1} { 2 } }
18536 \cs_new_protected:Npn \char_set_catcode_math_toggle:n #1
18537   { \char_set_catcode:nn {#1} { 3 } }
18538 \cs_new_protected:Npn \char_set_catcode_alignment:n #1
18539   { \char_set_catcode:nn {#1} { 4 } }
18540 \cs_new_protected:Npn \char_set_catcode_end_line:n #1
18541   { \char_set_catcode:nn {#1} { 5 } }
18542 \cs_new_protected:Npn \char_set_catcode_parameter:n #1
18543   { \char_set_catcode:nn {#1} { 6 } }
18544 \cs_new_protected:Npn \char_set_catcode_math_superscript:n #1
18545   { \char_set_catcode:nn {#1} { 7 } }

```

```

18546 \cs_new_protected:Npn \char_set_catcode_math_subscript:n #1
18547 { \char_set_catcode:nn {#1} { 8 } }
18548 \cs_new_protected:Npn \char_set_catcode_ignore:n #1
18549 { \char_set_catcode:nn {#1} { 9 } }
18550 \cs_new_protected:Npn \char_set_catcode_space:n #1
18551 { \char_set_catcode:nn {#1} { 10 } }
18552 \cs_new_protected:Npn \char_set_catcode_letter:n #1
18553 { \char_set_catcode:nn {#1} { 11 } }
18554 \cs_new_protected:Npn \char_set_catcode_other:n #1
18555 { \char_set_catcode:nn {#1} { 12 } }
18556 \cs_new_protected:Npn \char_set_catcode_active:n #1
18557 { \char_set_catcode:nn {#1} { 13 } }
18558 \cs_new_protected:Npn \char_set_catcode_comment:n #1
18559 { \char_set_catcode:nn {#1} { 14 } }
18560 \cs_new_protected:Npn \char_set_catcode_invalid:n #1
18561 { \char_set_catcode:nn {#1} { 15 } }

```

(End definition for `\char_set_catcode_escape:n` and others. These functions are documented on page 184.)

```

\char_set_mathcode:nn Pretty repetitive, but necessary!
\char_value_mathcode:n
\char_show_value_mathcode:n
\char_set_lccode:nn
\char_value_lccode:n
\char_show_value_lccode:n
\char_set_uccode:nn
\char_value_uccode:n
\char_show_value_uccode:n
\char_set_sfcode:nn
\char_value_sfcode:n
\char_show_value_sfcode:n
18562 \cs_new_protected:Npn \char_set_mathcode:nn #1#2
18563 { \tex_mathcode:D \int_eval:n {#1} = \int_eval:n {#2} \exp_stop_f: }
18564 \cs_new:Npn \char_value_mathcode:n #1
18565 { \tex_the:D \tex_mathcode:D \int_eval:n {#1} \exp_stop_f: }
18566 \cs_new_protected:Npn \char_show_value_mathcode:n #1
18567 { \exp_args:Nf \tl_show:n { \char_value_mathcode:n {#1} } }
18568 \cs_new_protected:Npn \char_set_lccode:nn #1#2
18569 { \tex_lccode:D \int_eval:n {#1} = \int_eval:n {#2} \exp_stop_f: }
18570 \cs_new:Npn \char_value_lccode:n #1
18571 { \tex_the:D \tex_lccode:D \int_eval:n {#1} \exp_stop_f: }
18572 \cs_new_protected:Npn \char_show_value_lccode:n #1
18573 { \exp_args:Nf \tl_show:n { \char_value_lccode:n {#1} } }
18574 \cs_new_protected:Npn \char_set_uccode:nn #1#2
18575 { \tex_uccode:D \int_eval:n {#1} = \int_eval:n {#2} \exp_stop_f: }
18576 \cs_new:Npn \char_value_uccode:n #1
18577 { \tex_the:D \tex_uccode:D \int_eval:n {#1} \exp_stop_f: }
18578 \cs_new_protected:Npn \char_show_value_uccode:n #1
18579 { \exp_args:Nf \tl_show:n { \char_value_uccode:n {#1} } }
18580 \cs_new_protected:Npn \char_set_sfcode:nn #1#2
18581 { \tex_sfcode:D \int_eval:n {#1} = \int_eval:n {#2} \exp_stop_f: }
18582 \cs_new:Npn \char_value_sfcode:n #1
18583 { \tex_the:D \tex_sfcode:D \int_eval:n {#1} \exp_stop_f: }
18584 \cs_new_protected:Npn \char_show_value_sfcode:n #1
18585 { \exp_args:Nf \tl_show:n { \char_value_sfcode:n {#1} } }

```

(End definition for `\char_set_mathcode:nn` and others. These functions are documented on page 185.)

`\l_char_active_seq` Two sequences for dealing with special characters. The first is characters which may be active, the second longer list is for “special” characters more generally. Both lists are escaped so that for example bulk code assignments can be carried out. In both cases, the order is by ASCII character code (as is done in for example `\ExplSyntaxOn`).

```

18586 \seq_new:N \l_char_special_seq
18587 \seq_set_split:Nnn \l_char_special_seq { }

```

```

18588 { \ \ " \# \$ \% \& \ \ ^ \_ \{ \} \~ }
18589 \seq_new:N \l_char_active_seq
18590 \seq_set_split:Nnn \l_char_active_seq { }
18591 { \ " \$ \% \^ \_ \~ }

```

(End definition for `\l_char_active_seq` and `\l_char_special_seq`. These variables are documented on page 186.)

60.3 Creating character tokens

`\char_set_active_eq:NN` Four simple functions with very similar definitions, so set up using an auxiliary. These are similar to LuaTeX's `\letcharcode` primitive.

```

\char_set_active_eq:Nc
\char_gset_active_eq:NN
\char_gset_active_eq:Nc
\char_set_active_eq:nN
\char_set_active_eq:nc
\char_gset_active_eq:nN
\char_gset_active_eq:nc
18592 \group_begin:
18593 \char_set_catcode_active:N \^^@
18594 \cs_set_protected:Npn \__char_tmp:nN #1#2
18595 {
18596   \cs_new_protected:cpn { #1 :nN } ##1
18597   {
18598     \group_begin:
18599     \char_set_lccode:nn { '^^@ } { ##1 }
18600     \tex_lowercase:D { \group_end: #2 ^^@ }
18601   }
18602   \cs_new_protected:cpx { #1 :NN } ##1
18603   { \exp_not:c { #1 : nN } { '##1 } }
18604 }
18605 \__char_tmp:nN { char_set_active_eq } \cs_set_eq:NN
18606 \__char_tmp:nN { char_gset_active_eq } \cs_gset_eq:NN
18607 \group_end:
18608 \cs_generate_variant:Nn \char_set_active_eq:NN { Nc }
18609 \cs_generate_variant:Nn \char_gset_active_eq:NN { Nc }
18610 \cs_generate_variant:Nn \char_set_active_eq:nN { nc }
18611 \cs_generate_variant:Nn \char_gset_active_eq:nN { nc }

```

(End definition for `\char_set_active_eq:NN` and others. These functions are documented on page 182.)

`__char_int_to_roman:w` For efficiency in 8-bit engines, we use the faster primitive approach to making roman numerals.

```

18612 \cs_new_eq:NN \__char_int_to_roman:w \tex_romannumeral:D

```

(End definition for `__char_int_to_roman:w`.)

`\char_generate:nn` The aim here is to generate characters of (broadly) arbitrary category code. Where possible, that is done using engine support (XeTeX, LuaTeX). There are though various issues which are covered below. At the interface layer, turn the two arguments into integers up-front so this is only done once.

```

\__char_generate_aux:nn
\__char_generate_aux:nnw
\__char_generate_auxii:nnw
\l__char_tmp_tl
\__char_generate_invalid_catcode:
18613 \cs_new:Npn \char_generate:nn #1#2
18614 {
18615   \exp:w \exp_after:wN \__char_generate_aux:w
18616   \int_value:w \int_eval:n {#1} \exp_after:wN ;
18617   \int_value:w \int_eval:n {#2} ;
18618 }

```

Before doing any actual conversion, first some special case filtering. Spaces are out here as LuaTeX emulation only makes normal (charcode 32 spaces). However, $\sim\@$ is filtered out separately as that can't be done with macro emulation either, so is flagged up separately. That done, hand off to the engine-dependent part.

```

18619 \cs_new:Npn \__char_generate_aux:w #1 ; #2 ;
18620 {
18621   \if_int_compare:w #2 = 10 \exp_stop_f:
18622     \if_int_compare:w #1 = \c_zero_int
18623       \msg_expandable_error:nn { char } { null-space }
18624     \else:
18625       \msg_expandable_error:nn { char } { space }
18626     \fi:
18627   \else:
18628     \if_int_odd:w 0
18629       \if_int_compare:w #2 < 1 \exp_stop_f: 1 \fi:
18630       \if_int_compare:w #2 = 5 \exp_stop_f: 1 \fi:
18631       \if_int_compare:w #2 = 9 \exp_stop_f: 1 \fi:
18632       \if_int_compare:w #2 > 13 \exp_stop_f: 1 \fi: \exp_stop_f:
18633       \msg_expandable_error:nn { char }
18634         { invalid-catcode }
18635     \else:
18636       \if_int_odd:w 0
18637         \if_int_compare:w #1 < \c_zero_int 1 \fi:
18638         \if_int_compare:w #1 > \c_max_char_int 1 \fi: \exp_stop_f:
18639         \msg_expandable_error:nn { char }
18640           { out-of-range }
18641       \else:
18642         \__char_generate_aux:nnw {#1} {#2}
18643       \fi:
18644     \fi:
18645   \fi:
18646   \exp_end:
18647 }
18648 \tl_new:N \l__char_tmp_tl

```

Engine-dependent definitions are now needed for the implementation. For LuaTeX and XeTeX there is engine-level support. They can do cases that macro emulation can't. All of those are filtered out here using a primitive-based boolean expression to avoid fixing the category code of the null character used in the false branch (for 8-bit engines). The final level is the basic definition at the engine level: the arguments here are integers so there is no need to worry about them too much. Older versions of XeTeX cannot generate active characters so we filter that: at some future stage that may change: the slightly odd ordering of auxiliaries reflects that.

```

18649 \group_begin:
18650   \char_set_catcode_active:N \sim
18651   \cs_set:Npn \sim { }
18652   \char_set_catcode_other:n { 0 }
18653   \if_int_odd:w 0
18654     \sys_if_engine luatex:T { 1 }
18655     \sys_if_engine xetex:T { 1 } \exp_stop_f:
18656     \cs_new:Npn \__char_generate_aux:nnw #1#2#3 \exp_end:
18657       {
18658         #3

```

```

18659         \exp_after:wN \exp_end:
18660         \tex_Ucharcat:D #1 \exp_stop_f: #2 \exp_stop_f:
18661     }
18662     \cs_if_exist:NF \tex_expanded:D
18663     {
18664         \cs_new_eq:NN \__char_generate_auxii:nnw \__char_generate_aux:nnw
18665         \cs_gset:Npn \__char_generate_aux:nnw #1#2#3 \exp_end:
18666         {
18667             #3
18668             \if_int_compare:w #2 = 13 \exp_stop_f:
18669                 \msg_expandable_error:nn { char } { active }
18670             \else:
18671                 \__char_generate_auxii:nnw {#1} {#2}
18672             \fi:
18673             \exp_end:
18674         }
18675     }
18676     \else:

```

For engines where \Ucharcat isn't available or emulated, we have to work in macros, and cover only the 8-bit range. The first stage is to build up a `tl` containing `^^@` with each category code that can be accessed in this way, with an error set up for the other cases. This is all done such that it can be quickly accessed using a `\if_case:w` low-level conditional. There are a few things to notice here. As `^^L` is `\outer` we need to locally set it to avoid a problem. To get open/close braces into the list, they are set up using `\if_false:` pairing and are then x-type expanded together into the desired form.

```

18677     \tl_set:Nn \l__char_tmp_tl { \exp_not:N \or: }
18678     \char_set_catcode_group_begin:n { 0 } % {
18679     \tl_put_right:Nn \l__char_tmp_tl { ^^@ \if_false: } }
18680     \char_set_catcode_group_end:n { 0 }
18681     \tl_put_right:Nn \l__char_tmp_tl { { \fi: \exp_not:N \or: ^^@ } % }
18682     \__kernel_tl_set:Nx \l__char_tmp_tl { \l__char_tmp_tl }
18683     \char_set_catcode_math_toggle:n { 0 }
18684     \tl_put_right:Nn \l__char_tmp_tl { \or: ^^@ }
18685     \char_set_catcode_alignment:n { 0 }
18686     \tl_put_right:Nn \l__char_tmp_tl { \or: ^^@ }
18687     \tl_put_right:Nn \l__char_tmp_tl { \or: }
18688     \char_set_catcode_parameter:n { 0 }
18689     \tl_put_right:Nn \l__char_tmp_tl { \or: ^^@ }
18690     \char_set_catcode_math_superscript:n { 0 }
18691     \tl_put_right:Nn \l__char_tmp_tl { \or: ^^@ }
18692     \char_set_catcode_math_subscript:n { 0 }
18693     \tl_put_right:Nn \l__char_tmp_tl { \or: ^^@ }
18694     \tl_put_right:Nn \l__char_tmp_tl { \or: }

```

For making spaces, there needs to be an o-type expansion of a `\use:n` (or some other tokenization) to avoid dropping the space.

```

18695     \char_set_catcode_space:n { 0 }
18696     \tl_put_right:No \l__char_tmp_tl { \use:n { \or: } ^^@ }
18697     \char_set_catcode_letter:n { 0 }
18698     \tl_put_right:Nn \l__char_tmp_tl { \or: ^^@ }
18699     \char_set_catcode_other:n { 0 }
18700     \tl_put_right:Nn \l__char_tmp_tl { \or: ^^@ }
18701     \char_set_catcode_active:n { 0 }

```

```
18702 \tl_put_right:Nn \l__char_tmp_tl { \or: ^^@ }
```

Convert the above temporary list into a series of constant token lists, one for each character code, using `\tex_lowercase:D` to convert `^^@` in each case. The x-type expansion ensures that `\tex_lowercase:D` receives the contents of the token list. `^^L` is awkward hence this is done in three parts: up to `^^L`, `^^L` itself and above `^^L`. Notice that at this stage `^^@` is active.

```
18703 \cs_set_protected:Npn \__char_tmp:n #1
18704 {
18705   \char_set_lccode:nn { 0 } {#1}
18706   \char_set_lccode:nn { 32 } {#1}
18707   \exp_args:Nx \tex_lowercase:D
18708   {
18709     \tl_const:Nn
18710       \exp_not:c { c__char_ \__char_int_to_roman:w #1 _tl }
18711       { \exp_not:o \l__char_tmp_tl }
18712   }
18713 }
18714 \int_step_function:nnN { 0 } { 11 } \__char_tmp:n
18715 \group_begin:
18716   \tl_replace_once:Nnn \l__char_tmp_tl { ^^@ } { \ERROR }
18717   \__char_tmp:n { 12 }
18718 \group_end:
18719 \int_step_function:nnN { 13 } { 255 } \__char_tmp:n
```

As TeX is very unhappy if it finds an alignment character inside a primitive `\halign` even when skipping false branches, some precautions are required. TeX is happy if the token is hidden between braces within `\if_false: ... \fi:`.

```
18720 \cs_new:Npn \__char_generate_aux:nnw #1#2#3 \exp_end:
18721 {
18722   #3
18723   \if_false: { \fi:
18724     \exp_after:wN \exp_after:wN
18725     \exp_after:wN \exp_end:
18726     \exp_after:wN \exp_after:wN
18727     \if_case:w #2
18728       \exp_last_unbraced:Nv \exp_stop_f:
18729       { c__char_ \__char_int_to_roman:w #1 _tl }
18730     \or: }
18731     \fi:
18732   }
18733   \fi:
18734 \group_end:
```

(End definition for `\char_generate:nn` and others. This function is documented on page 182.)

`\char_to_utfviii_bytes:n`

This code converts a codepoint into the correct UTF-8 representation. In terms of the algorithm itself, see <https://en.wikipedia.org/wiki/UTF-8> for the octet pattern.

```
\__char_to_utfviii_bytes_auxi:n
\__char_to_utfviii_bytes_auxii:Nnn
\__char_to_utfviii_bytes_auxiii:n
\__char_to_utfviii_bytes_outputi:nw
\__char_to_utfviii_bytes_outputii:nw
\__char_to_utfviii_bytes_outputiii:nw
\__char_to_utfviii_bytes_outputiv:nw
\__char_to_utfviii_bytes_outputv:nnn
\__char_to_utfviii_bytes_output:fnn
\__char_to_utfviii_bytes_end:
18735 \cs_new:Npn \char_to_utfviii_bytes:n #1
18736 {
18737   \exp_args:Nf \__char_to_utfviii_bytes_auxi:n
18738   { \int_eval:n {#1} }
18739 }
18740 \cs_new:Npn \__char_to_utfviii_bytes_auxi:n #1
```



```

18741 {
18742     \if_int_compare:w #1 > "80 \exp_stop_f:
18743     \if_int_compare:w #1 < "800 \exp_stop_f:
18744         \__char_to_utfviii_bytes_outputi:nw
18745         { \__char_to_utfviii_bytes_auxii:Nnn C {#1} { 64 } }
18746         \__char_to_utfviii_bytes_outputii:nw
18747         { \__char_to_utfviii_bytes_auxiii:n {#1} }
18748     \else:
18749         \if_int_compare:w #1 < "10000 \exp_stop_f:
18750         \__char_to_utfviii_bytes_outputi:nw
18751         { \__char_to_utfviii_bytes_auxii:Nnn E {#1} { 64 * 64 } }
18752         \__char_to_utfviii_bytes_outputii:nw
18753         {
18754             \__char_to_utfviii_bytes_auxiii:n
18755             { \int_div_truncate:nn {#1} { 64 } }
18756         }
18757         \__char_to_utfviii_bytes_outputiii:nw
18758         { \__char_to_utfviii_bytes_auxiii:n {#1} }
18759     \else:
18760         \__char_to_utfviii_bytes_outputi:nw
18761         {
18762             \__char_to_utfviii_bytes_auxii:Nnn F
18763             {#1} { 64 * 64 * 64 }
18764         }
18765         \__char_to_utfviii_bytes_outputii:nw
18766         {
18767             \__char_to_utfviii_bytes_auxiii:n
18768             { \int_div_truncate:nn {#1} { 64 * 64 } }
18769         }
18770         \__char_to_utfviii_bytes_outputiii:nw
18771         {
18772             \__char_to_utfviii_bytes_auxiii:n
18773             { \int_div_truncate:nn {#1} { 64 } }
18774         }
18775         \__char_to_utfviii_bytes_outputiv:nw
18776         { \__char_to_utfviii_bytes_auxiii:n {#1} }
18777     \fi:
18778     \fi:
18779     \else:
18780         \__char_to_utfviii_bytes_outputi:nw {#1}
18781     \fi:
18782     \__char_to_utfviii_bytes_end: { } { } { } { }
18783 }
18784 \cs_new:Npn \__char_to_utfviii_bytes_auxii:Nnn #1#2#3
18785 { "#10 + \int_div_truncate:nn {#2} {#3} }
18786 \cs_new:Npn \__char_to_utfviii_bytes_auxiii:n #1
18787 { \int_mod:nn {#1} { 64 } + 128 }
18788 \cs_new:Npn \__char_to_utfviii_bytes_outputi:nw
18789 #1 #2 \__char_to_utfviii_bytes_end: #3
18790 { \__char_to_utfviii_bytes_output:fnn { \int_eval:n {#1} } { } {#2} }
18791 \cs_new:Npn \__char_to_utfviii_bytes_outputii:nw
18792 #1 #2 \__char_to_utfviii_bytes_end: #3#4
18793 { \__char_to_utfviii_bytes_output:fnn { \int_eval:n {#1} } { {#3} } {#2} }
18794 \cs_new:Npn \__char_to_utfviii_bytes_outputiii:nw

```

```

18795 #1 #2 \_char_to_utfviii_bytes_end: #3#4#5
18796 {
18797     \_char_to_utfviii_bytes_output:fnn
18798     { \int_eval:n {#1} } { {#3} {#4} } {#2}
18799 }
18800 \cs_new:Npn \_char_to_utfviii_bytes_outputiv:nw
18801 #1 #2 \_char_to_utfviii_bytes_end: #3#4#5#6
18802 {
18803     \_char_to_utfviii_bytes_output:fnn
18804     { \int_eval:n {#1} } { {#3} {#4} {#5} } {#2}
18805 }
18806 \cs_new:Npn \_char_to_utfviii_bytes_output:nnn #1#2#3
18807 {
18808     #3
18809     \_char_to_utfviii_bytes_end: #2 {#1}
18810 }
18811 \cs_generate_variant:Nn \_char_to_utfviii_bytes_output:nnn { f }
18812 \cs_new:Npn \_char_to_utfviii_bytes_end: { }

```

(End definition for `\char_to_utfviii_bytes:n` and others. This function is documented on page 308.)

```

\_char_to_nfd:N Look up any NFD and recursively produce the result.
\_char_to_nfd:n
\_char_to_nfd:Nw
18813 \cs_new:Npn \char_to_nfd:N #1
18814 {
18815     \cs_if_exist:cTF { c__char_nfd_ \token_to_str:N #1 _ t1 }
18816     {
18817         \exp_after:wN \exp_after:wN \exp_after:wN \_char_to_nfd:Nw
18818         \exp_after:wN \exp_after:wN \exp_after:wN #1
18819         \cs:w c__char_nfd_ \token_to_str:N #1 _ t1 \cs_end:
18820         \s__char_stop
18821     }
18822     { \exp_not:n {#1} }
18823 }
18824 \cs_set_eq:NN \_char_to_nfd:n \char_to_nfd:N
18825 \cs_new:Npn \_char_to_nfd:Nw #1#2#3 \s__char_stop
18826 {
18827     \exp_args:Ne \_char_to_nfd:n
18828     { \char_generate:nn { '#2 } { \_char_change_case_catcode:N #1 } }
18829     \tl_if_blank:nF {#3}
18830     {
18831         \exp_args:Ne \_char_to_nfd:n
18832         { \char_generate:nn { '#3 } { \char_value_catcode:n { '#3 } } }
18833     }
18834 }

```

(End definition for `\char_to_nfd:N`, `_char_to_nfd:n`, and `_char_to_nfd:Nw`. This function is documented on page 308.)

`\char_lowercase:N` To ensure that the category codes produced are predictable, every character is re-generated even if it is otherwise unchanged. This makes life a little interesting when we might have multiple output characters: we have to grab each of them and case change them in reverse order to maintain f-type expandability.

```

\_char_change_case:nNN
\_char_change_case:nN
\_char_change_case_multi:nN
\_char_change_case_multi:vN
\_char_change_case_multi:NNNw
\_char_change_case:NNN
\_char_change_case:NNNN
\_char_change_case:NN
\_char_change_case_catcode:N
\_char_str_lowercase:N
\_char_str_uppercase:N
\_char_str_titlecase:N
\_char_str_foldcase:N

```

```

18837 \cs_new:Npn \char_uppercase:N #1
18838 { \__char_change_case:nNN { upper } \char_value_uccode:n #1 }
18839 \cs_new:Npn \char_titlecase:N #1
18840 {
18841   \tl_if_exist:cTF { c__char_titlecase_ \token_to_str:N #1 _tl }
18842   {
18843     \__char_change_case_multi:vN
18844     { c__char_titlecase_ \token_to_str:N #1 _tl } #1
18845   }
18846   { \char_uppercase:N #1 }
18847 }
18848 \cs_new:Npn \char_foldcase:N #1
18849 { \__char_change_case:nNN { fold } \char_value_lccode:n #1 }
18850 \cs_new:Npn \__char_change_case:nNN #1#2#3
18851 {
18852   \tl_if_exist:cTF { c__char_ #1 case_ \token_to_str:N #3 _tl }
18853   {
18854     \__char_change_case_multi:vN
18855     { c__char_ #1 case_ \token_to_str:N #3 _tl } #3
18856   }
18857   { \exp_args:Nf \__char_change_case:nN { #2 { '#3 } } #3 }
18858 }
18859 \cs_new:Npn \__char_change_case:nN #1#2
18860 {
18861   \int_compare:nNnTF {#1} = 0
18862   { #2 }
18863   { \char_generate:nn {#1} { \__char_change_case_catcode:N #2 } }
18864 }
18865 \cs_new:Npn \__char_change_case_multi:nN #1#2
18866 { \__char_change_case_multi:NNNNw #2 #1 \q__char_no_value \q__char_no_value \s__char_stop
18867 \cs_generate_variant:Nn \__char_change_case_multi:nN { v }
18868 \cs_new:Npn \__char_change_case_multi:NNNNw #1#2#3#4#5 \s__char_stop
18869 {
18870   \__char_quark_if_no_value:NTF #4
18871   {
18872     \__char_quark_if_no_value:NTF #3
18873     { \__char_change_case:NN #1 #2 }
18874     { \__char_change_case:NNN #1 #2#3 }
18875   }
18876   { \__char_change_case:NNNN #1 #2#3#4 }
18877 }
18878 \cs_new:Npn \__char_change_case:NNN #1#2#3
18879 {
18880   \exp_args:Nnf \use:nn
18881   { \__char_change_case:NN #1 #2 }
18882   { \__char_change_case:NN #1 #3 }
18883 }
18884 \cs_new:Npn \__char_change_case:NNNN #1#2#3#4
18885 {
18886   \exp_args:Nnff \use:nnn
18887   { \__char_change_case:NN #1 #2 }
18888   { \__char_change_case:NN #1 #3 }
18889   { \__char_change_case:NN #1 #4 }
18890 }

```

```

18891 \cs_new:Npn \__char_change_case:NN #1#2
18892 { \char_generate:nn { '#2 } { \__char_change_case_catcode:N #1 } }
18893 \cs_new:Npn \__char_change_case_catcode:N #1
18894 {
18895   \if_catcode:w \exp_not:N #1 \c_math_toggle_token
18896   3
18897   \else:
18898     \if_catcode:w \exp_not:N #1 \c_alignment_token
18899     4
18900     \else:
18901       \if_catcode:w \exp_not:N #1 \c_math_superscript_token
18902       7
18903       \else:
18904         \if_catcode:w \exp_not:N #1 \c_math_subscript_token
18905         8
18906         \else:
18907           \if_catcode:w \exp_not:N #1 \c_space_token
18908           10
18909           \else:
18910             \if_catcode:w \exp_not:N #1 \c_catcode_letter_token
18911             11
18912             \else:
18913               \if_catcode:w \exp_not:N #1 \c_catcode_other_token
18914               12
18915               \else:
18916                 13
18917                 \fi:
18918                 \fi:
18919                 \fi:
18920                 \fi:
18921                 \fi:
18922                 \fi:
18923                 \fi:
18924 }

```

Same story for the string version, except category code is easier to follow. This of course makes this version significantly faster.

```

18925 \cs_new:Npn \char_str_lowercase:N #1
18926 { \__char_str_change_case:nNN { lower } \char_value_lccode:n #1 }
18927 \cs_new:Npn \char_str_uppercase:N #1
18928 { \__char_str_change_case:nNN { upper } \char_value_uccode:n #1 }
18929 \cs_new:Npn \char_str_titlecase:N #1
18930 {
18931   \tl_if_exist:cTF { c__char_titlecase_ \token_to_str:N #1 _tl }
18932   { \tl_to_str:c { c__char_titlecase_ \token_to_str:N #1 _tl } }
18933   { \char_str_uppercase:N #1 }
18934 }
18935 \cs_new:Npn \char_str_foldcase:N #1
18936 { \__char_str_change_case:nNN { fold } \char_value_lccode:n #1 }
18937 \cs_new:Npn \__char_str_change_case:nNN #1#2#3
18938 {
18939   \tl_if_exist:cTF { c__char_ #1 case_ \token_to_str:N #3 _tl }
18940   { \tl_to_str:c { c__char_ #1 case_ \token_to_str:N #3 _tl } }
18941   { \exp_args:Nf \__char_str_change_case:nN { #2 { '#3 } } #3 }

```

```

18942 }
18943 \cs_new:Npn \__char_str_change_case:nN #1#2
18944 {
18945     \int_compare:nNnTF {#1} = 0
18946     { \tl_to_str:n {#2} }
18947     { \char_generate:nn {#1} { 12 } }
18948 }
18949 \bool_lazy_or:nnF
18950 { \cs_if_exist_p:N \tex_luatexversion:D }
18951 { \cs_if_exist_p:N \tex_XeTeXversion:D }
18952 {
18953     \cs_set:Npn \__char_str_change_case:nN #1#2
18954     { \tl_to_str:n {#2} }
18955 }

```

(End definition for `\char_lowercase:N` and others. These functions are documented on page 183.)

`\c_catcode_other_space_tl` Create a space with category code 12: an “other” space.

```

18956 \tl_const:Nx \c_catcode_other_space_tl { \char_generate:nn { '\ } { 12 } }

```

(End definition for `\c_catcode_other_space_tl`. This function is documented on page 183.)

60.4 Generic tokens

```

18957 <@@=token>

```

`\s_token_mark` Internal scan marks.

```

\s_token_stop
18958 \scan_new:N \s_token_mark
18959 \scan_new:N \s_token_stop

```

(End definition for `\s_token_mark` and `\s_token_stop`.)

`\token_to_meaning:N` These are all defined in `l3basics`, as they are needed “early”. This is just a reminder!

```

\token_to_meaning:c
\token_to_str:N
\token_to_str:c

```

(End definition for `\token_to_meaning:N` and `\token_to_str:N`. These functions are documented on page 187.)

`\c_group_begin_token`
`\c_group_end_token`
`\c_math_toggle_token`
`\c_alignment_token`
`\c_parameter_token`
`\c_math_superscript_token`
`\c_math_subscript_token`
`\c_space_token`
`\c_catcode_letter_token`
`\c_catcode_other_token`

We define these useful tokens. For the brace and space tokens things have to be done by hand: the formal argument spec. for `\cs_new_eq:NN` does not cover them so we do things by hand. (As currently coded it would *work* with `\cs_new_eq:NN` but that’s not really a great idea to show off: we want people to stick to the defined interfaces and that includes us.) So that these few odd names go into the log when appropriate there is a need to hand-apply the `__kernel_chk_if_free_cs:N` check.

```

18960 \group_begin:
18961     \__kernel_chk_if_free_cs:N \c_group_begin_token
18962     \tex_global:D \tex_let:D \c_group_begin_token {
18963     \__kernel_chk_if_free_cs:N \c_group_end_token
18964     \tex_global:D \tex_let:D \c_group_end_token }
18965     \char_set_catcode_math_toggle:N \*
18966     \cs_new_eq:NN \c_math_toggle_token *
18967     \char_set_catcode_alignment:N \#
18968     \cs_new_eq:NN \c_alignment_token #
18969     \cs_new_eq:NN \c_parameter_token #
18970     \cs_new_eq:NN \c_math_superscript_token ^

```

```

18971 \char_set_catcode_math_subscript:N \*
18972 \cs_new_eq:NN \c_math_subscript_token *
18973 \__kernel_chk_if_free_cs:N \c_space_token
18974 \use:n { \tex_global:D \tex_let:D \c_space_token = ~ } ~
18975 \cs_new_eq:NN \c_catcode_letter_token a
18976 \cs_new_eq:NN \c_catcode_other_token 1
18977 \group_end:

```

(End definition for `\c_group_begin_token` and others. These functions are documented on page 186.)

`\c_catcode_active_tl` Not an implicit token!

```

18978 \group_begin:
18979 \char_set_catcode_active:N \*
18980 \tl_const:Nn \c_catcode_active_tl { \exp_not:N * }
18981 \group_end:

```

(End definition for `\c_catcode_active_tl`. This variable is documented on page 186.)

60.5 Token conditionals

`\token_if_group_begin_p:N` Check if token is a begin group token. We use the constant `\c_group_begin_token` for this.

`\token_if_group_begin:N \underline{TF}`

```

18982 \prg_new_conditional:Npnn \token_if_group_begin:N #1 { p , T , F , TF }
18983 {
18984     \if_catcode:w \exp_not:N #1 \c_group_begin_token
18985     \prg_return_true: \else: \prg_return_false: \fi:
18986 }

```

(End definition for `\token_if_group_begin:N \underline{TF}` . This function is documented on page 187.)

`\token_if_group_end_p:N` Check if token is a end group token. We use the constant `\c_group_end_token` for this.

`\token_if_group_end:N \underline{TF}`

```

18987 \prg_new_conditional:Npnn \token_if_group_end:N #1 { p , T , F , TF }
18988 {
18989     \if_catcode:w \exp_not:N #1 \c_group_end_token
18990     \prg_return_true: \else: \prg_return_false: \fi:
18991 }

```

(End definition for `\token_if_group_end:N \underline{TF}` . This function is documented on page 187.)

`\token_if_math_toggle_p:N` Check if token is a math shift token. We use the constant `\c_math_toggle_token` for this.

`\token_if_math_toggle:N \underline{TF}`

```

18992 \prg_new_conditional:Npnn \token_if_math_toggle:N #1 { p , T , F , TF }
18993 {
18994     \if_catcode:w \exp_not:N #1 \c_math_toggle_token
18995     \prg_return_true: \else: \prg_return_false: \fi:
18996 }

```

(End definition for `\token_if_math_toggle:N \underline{TF}` . This function is documented on page 187.)

`\token_if_alignment_p:N` Check if token is an alignment tab token. We use the constant `\c_alignment_token` for this.

`\token_if_alignment:N \underline{TF}`

```

18997 \prg_new_conditional:Npnn \token_if_alignment:N #1 { p , T , F , TF }
18998 {
18999     \if_catcode:w \exp_not:N #1 \c_alignment_token
19000     \prg_return_true: \else: \prg_return_false: \fi:
19001 }

```

(End definition for `\token_if_alignment:NTF`. This function is documented on page 188.)

`\token_if_parameter_p:N` Check if token is a parameter token. We use the constant `\c_parameter_token` for this.
`\token_if_parameter:N \underline{TF}` We have to trick TeX a bit to avoid an error message: within a group we prevent `\c_parameter_token` from behaving like a macro parameter character. The definitions of `\prg_new_conditional:Npnn` are global, so they remain after the group.

```

19002 \group_begin:
19003 \cs_set_eq:NN \c_parameter_token \scan_stop:
19004 \prg_new_conditional:Npnn \token_if_parameter:N #1 { p , T , F , TF }
19005 {
19006     \if_catcode:w \exp_not:N #1 \c_parameter_token
19007     \prg_return_true: \else: \prg_return_false: \fi:
19008 }
19009 \group_end:

```

(End definition for `\token_if_parameter:NTF`. This function is documented on page 188.)

`\token_if_math_superscript_p:N` Check if token is a math superscript token. We use the constant `\c_math_superscript_token` for this.
`\token_if_math_superscript:N \underline{TF}`

```

19010 \prg_new_conditional:Npnn \token_if_math_superscript:N #1
19011 { p , T , F , TF }
19012 {
19013     \if_catcode:w \exp_not:N #1 \c_math_superscript_token
19014     \prg_return_true: \else: \prg_return_false: \fi:
19015 }

```

(End definition for `\token_if_math_superscript:NTF`. This function is documented on page 188.)

`\token_if_math_subscript_p:N` Check if token is a math subscript token. We use the constant `\c_math_subscript_token` for this.
`\token_if_math_subscript:N \underline{TF}`

```

19016 \prg_new_conditional:Npnn \token_if_math_subscript:N #1 { p , T , F , TF }
19017 {
19018     \if_catcode:w \exp_not:N #1 \c_math_subscript_token
19019     \prg_return_true: \else: \prg_return_false: \fi:
19020 }

```

(End definition for `\token_if_math_subscript:NTF`. This function is documented on page 188.)

`\token_if_space_p:N` Check if token is a space token. We use the constant `\c_space_token` for this.

`\token_if_space:N \underline{TF}`

```

19021 \prg_new_conditional:Npnn \token_if_space:N #1 { p , T , F , TF }
19022 {
19023     \if_catcode:w \exp_not:N #1 \c_space_token
19024     \prg_return_true: \else: \prg_return_false: \fi:
19025 }

```

(End definition for `\token_if_space:NTF`. This function is documented on page 188.)

`\token_if_letter_p:N` Check if token is a letter token. We use the constant `\c_catcode_letter_token` for this.

`\token_if_letter:N \underline{TF}`

```

19026 \prg_new_conditional:Npnn \token_if_letter:N #1 { p , T , F , TF }
19027 {
19028     \if_catcode:w \exp_not:N #1 \c_catcode_letter_token
19029     \prg_return_true: \else: \prg_return_false: \fi:
19030 }

```

(End definition for `\token_if_letter:NTF`. This function is documented on page 188.)

`\token_if_other_p:N` Check if token is an other char token. We use the constant `\c_catcode_other_token`
`\token_if_other:N \underline{TF}` for this.

```

19031 \prg_new_conditional:Npnn \token_if_other:N #1 { p , T , F , TF }
19032 {
19033     \if_catcode:w \exp_not:N #1 \c_catcode_other_token
19034     \prg_return_true: \else: \prg_return_false: \fi:
19035 }

```

(End definition for `\token_if_other:N \underline{TF}` . This function is documented on page 188.)

`\token_if_active_p:N` Check if token is an active char token. We use the constant `\c_catcode_active_tl` for
`\token_if_active:N \underline{TF}` this. A technical point is that `\c_catcode_active_tl` is in fact a macro expanding to
`\exp_not:N *`, where `*` is active.

```

19036 \prg_new_conditional:Npnn \token_if_active:N #1 { p , T , F , TF }
19037 {
19038     \if_catcode:w \exp_not:N #1 \c_catcode_active_tl
19039     \prg_return_true: \else: \prg_return_false: \fi:
19040 }

```

(End definition for `\token_if_active:N \underline{TF}` . This function is documented on page 188.)

`\token_if_eq_meaning_p:NN` Check if the tokens #1 and #2 have same meaning.

`\token_if_eq_meaning:NN \underline{TF}`

```

19041 \prg_new_conditional:Npnn \token_if_eq_meaning:NN #1#2 { p , T , F , TF }
19042 {
19043     \if_meaning:w #1 #2
19044     \prg_return_true: \else: \prg_return_false: \fi:
19045 }

```

(End definition for `\token_if_eq_meaning:NN \underline{TF}` . This function is documented on page 189.)

`\token_if_eq_catcode_p:NN` Check if the tokens #1 and #2 have same category code.

`\token_if_eq_catcode:NN \underline{TF}`

```

19046 \prg_new_conditional:Npnn \token_if_eq_catcode:NN #1#2 { p , T , F , TF }
19047 {
19048     \if_catcode:w \exp_not:N #1 \exp_not:N #2
19049     \prg_return_true: \else: \prg_return_false: \fi:
19050 }

```

(End definition for `\token_if_eq_catcode:NN \underline{TF}` . This function is documented on page 188.)

`\token_if_eq_charcode_p:NN` Check if the tokens #1 and #2 have same character code.

`\token_if_eq_charcode:NN \underline{TF}`

```

19051 \prg_new_conditional:Npnn \token_if_eq_charcode:NN #1#2 { p , T , F , TF }
19052 {
19053     \if_charcode:w \exp_not:N #1 \exp_not:N #2
19054     \prg_return_true: \else: \prg_return_false: \fi:
19055 }

```

(End definition for `\token_if_eq_charcode:NN \underline{TF}` . This function is documented on page 188.)

`\token_if_macro_p:N` When a token is a macro, `\token_to_meaning:N` always outputs something like
`\token_if_macro:N \underline{TF}` `\long macro:#1->#1` so we could naively check to see if the meaning contains `->`.
`__token_if_macro_p:w` However, this can fail the five `\...mark` primitives, whose meaning has the form
`\...mark:<user material>`. The problem is that the `<user material>` can contain `->`.

However, only characters, macros, and marks can contain the colon character. The idea is thus to grab until the first `:`, and analyse what is left. However, macros can have

any combination of `\long`, `\protected` or `\outer` (not used in L^AT_EX3) before the string `macro:`. We thus only select the part of the meaning between the first `ma` and the first following `:`. If this string is `cro`, then we have a macro. If the string is `rk`, then we have a mark. The string can also be `cro parameter character` for a colon with a weird category code (namely the usual category code of `#`). Otherwise, it is empty.

This relies on the fact that `\long`, `\protected`, `\outer` cannot contain `ma`, regardless of the escape character, even if the escape character is `m...`.

Both `ma` and `:` must be of category code 12 (other), so are detokenized.

```

19056 \use:x
19057 {
19058   \prg_new_conditional:Npnn \exp_not:N \token_if_macro:N ##1
19059   { p , T , F , TF }
19060   {
19061     \exp_not:N \exp_after:wN \exp_not:N \__token_if_macro_p:w
19062     \exp_not:N \token_to_meaning:N ##1 \tl_to_str:n { ma : }
19063     \s__token_stop
19064   }
19065   \cs_new:Npn \exp_not:N \__token_if_macro_p:w
19066   ##1 \tl_to_str:n { ma } ##2 \c_colon_str ##3 \s__token_stop
19067 }
19068 {
19069   \str_if_eq:nnTF { #2 } { cro }
19070   { \prg_return_true: }
19071   { \prg_return_false: }
19072 }

```

(End definition for `\token_if_macro:N` and `__token_if_macro_p:w`. This function is documented on page 189.)

`\token_if_cs_p:N` Check if token has same catcode as a control sequence. This follows the same pattern as `\token_if_letter:N` etc. We use `\scan_stop:` for this.

`\token_if_cs:N` TF

```

19073 \prg_new_conditional:Npnn \token_if_cs:N #1 { p , T , F , TF }
19074 {
19075   \if_catcode:w \exp_not:N #1 \scan_stop:
19076   \prg_return_true: \else: \prg_return_false: \fi:
19077 }

```

(End definition for `\token_if_cs:N`. This function is documented on page 189.)

`\token_if_expandable_p:N` Check if token is expandable. We use the fact that T_EX temporarily converts `\exp_not:N` $\langle token \rangle$ into `\scan_stop:` if $\langle token \rangle$ is expandable. An undefined token is not considered as expandable. No problem nesting the conditionals, since the third `#1` is only skipped if it is non-expandable (hence not part of T_EX's conditional apparatus).

`\token_if_expandable:N` TF

```

19078 \prg_new_conditional:Npnn \token_if_expandable:N #1 { p , T , F , TF }
19079 {
19080   \exp_after:wN \if_meaning:w \exp_not:N #1 #1
19081   \prg_return_false:
19082   \else:
19083     \if_cs_exist:N #1
19084     \prg_return_true:
19085     \else:
19086     \prg_return_false:
19087   \fi:

```

```

19088     \fi:
19089   }

```

(End definition for `\token_if_expandable:NTF`. This function is documented on page 189.)

```

\__token_delimit_by_char:w
\__token_delimit_by_count:w
\__token_delimit_by_dimen:w
\__token_delimit_by_font:w
\__token_delimit_by_macro:w
\__token_delimit_by_muskip:w
\__token_delimit_by_skip:w
\__token_delimit_by_toks:w

```

These auxiliary functions are used below to define some conditionals which detect whether the `\meaning` of their argument begins with a particular string. Each auxiliary takes an argument delimited by a string, a second one delimited by `\s__token_stop`, and returns the first one and its delimiter. This result is eventually compared to another string. Note that the “font” auxiliary is delimited by a space followed by “font”. This avoids an unnecessary check for the `\font` primitive below.

```

19090 \group_begin:
19091 \cs_set_protected:Npn \__token_tmp:w #1
19092 {
19093   \use:x
19094   {
19095     \cs_new:Npn \exp_not:c { __token_delimit_by_ #1 :w }
19096     #####1 \tl_to_str:n {#1} #####2 \s__token_stop
19097     { #####1 \tl_to_str:n {#1} }
19098   }
19099 }
19100 \__token_tmp:w { char" }
19101 \__token_tmp:w { count }
19102 \__token_tmp:w { dimen }
19103 \__token_tmp:w { ~ font }
19104 \__token_tmp:w { macro }
19105 \__token_tmp:w { muskip }
19106 \__token_tmp:w { skip }
19107 \__token_tmp:w { toks }
19108 \group_end:

```

(End definition for `__token_delimit_by_char:w` and others.)

```

\token_if_chardef_p:N
\token_if_chardef:NTF
\token_if_mathchardef_p:N
\token_if_mathchardef:NTF
\token_if_long_macro_p:N
\token_if_long_macro:NTF
\token_if_protected_macro_p:N
\token_if_protected_macro:NTF
\token_if_protected_long_macro_p:N
\token_if_protected_long_macro:NTF
\token_if_font_selection_p:N
\token_if_font_selection:NTF
\token_if_dim_register_p:N
\token_if_dim_register:NTF
\token_if_int_register_p:N
\token_if_int_register:NTF
\token_if_muskip_register_p:N
\token_if_muskip_register:NTF
\token_if_skip_register_p:N
\token_if_skip_register:NTF
\token_if_toks_register_p:N
\token_if_toks_register:NTF

```

Each of these conditionals tests whether its argument’s `\meaning` starts with a given string. This is essentially done by having an auxiliary grab an argument delimited by the string and testing whether the argument was empty. Of course, a copy of this string must first be added to the end of the `\meaning` to avoid a runaway argument in case it does not contain the string. Two complications arise. First, the escape character is not fixed, and cannot be included in the delimiter of the auxiliary function (this function cannot be defined on the fly because tests must remain expandable): instead the first argument of the auxiliary (plus the delimiter to avoid complications with trailing spaces) is compared using `\str_if_eq:eeTF` to the result of applying `\token_to_str:N` to a control sequence. Second, the `\meaning` of primitives such as `\dimen` or `\dimendef` starts in the same way as registers such as `\dimen123`, so they must be tested for.

Characters used as delimiters must have catcode 12 and are obtained through `\tl_to_str:n`. This requires doing all definitions within x-expansion. The temporary function `__token_tmp:w` used to define each conditional receives three arguments: the name of the conditional, the auxiliary’s delimiter (also used to name the auxiliary), and the string to which one compares the auxiliary’s result. Note that the `\meaning` of a protected long macro starts with `\protected\long macro`, with no space after `\protected` but a space after `\long`, hence the mixture of `\token_to_str:N` and `\tl_to_str:n`.

For the first six conditionals, `\cs_if_exist:cT` turns out to be false (thanks to the leading space for font), and the code boils down to a string comparison between

the result of the auxiliary on the `\meaning` of the conditional's argument `####1`, and `#3`. Both are evaluated at run-time, as this is important to get the correct escape character.

The other five conditionals have additional code that compares the argument `####1` to two `TeX` primitives which would wrongly be recognized as registers otherwise. Despite using `TeX`'s primitive conditional construction, this does not break when `####1` is itself a conditional, because branches of the conditionals are only skipped if `####1` is one of the two primitives that are tested for (which are not `TeX` conditionals).

```

19109 \group_begin:
19110 \cs_set_protected:Npn \__token_tmp:w #1#2#3
19111 {
19112   \use:x
19113   {
19114     \prg_new_conditional:Npnn \exp_not:c { token_if_ #1 :N } ####1
19115     { p , T , F , TF }
19116     {
19117       \cs_if_exist:cT { tex_ #2 :D }
19118       {
19119         \exp_not:N \if_meaning:w ####1 \exp_not:c { tex_ #2 :D }
19120         \exp_not:N \prg_return_false:
19121         \exp_not:N \else:
19122         \exp_not:N \if_meaning:w ####1 \exp_not:c { tex_ #2 def:D }
19123         \exp_not:N \prg_return_false:
19124         \exp_not:N \else:
19125       }
19126       \exp_not:N \str_if_eq:eeTF
19127       {
19128         \exp_not:N \exp_after:wN
19129         \exp_not:c { __token_delimit_by_ #2 :w }
19130         \exp_not:N \token_to_meaning:N ####1
19131         ? \tl_to_str:n {#2} \s__token_stop
19132       }
19133       { \exp_not:n {#3} }
19134       { \exp_not:N \prg_return_true: }
19135       { \exp_not:N \prg_return_false: }
19136       \cs_if_exist:cT { tex_ #2 :D }
19137       {
19138         \exp_not:N \fi:
19139         \exp_not:N \fi:
19140       }
19141     }
19142   }
19143 }
19144 \__token_tmp:w { chardef } { char" } { \token_to_str:N \char" }
19145 \__token_tmp:w { mathchardef } { char" } { \token_to_str:N \mathchar" }
19146 \__token_tmp:w { long_macro } { macro } { \tl_to_str:n { \long } macro }
19147 \__token_tmp:w { protected_macro } { macro }
19148   { \tl_to_str:n { \protected } macro }
19149 \__token_tmp:w { protected_long_macro } { macro }
19150   { \token_to_str:N \protected \tl_to_str:n { \long } macro }
19151 \__token_tmp:w { font_selection } { ~ font } { select ~ font }
19152 \__token_tmp:w { dim_register } { dimen } { \token_to_str:N \dimen }
19153 \__token_tmp:w { int_register } { count } { \token_to_str:N \count }
19154 \__token_tmp:w { muskip_register } { muskip } { \token_to_str:N \muskip }

```

```

19155 \__token_tmp:w { skip_register } { skip } { \token_to_str:N \skip }
19156 \__token_tmp:w { toks_register } { toks } { \token_to_str:N \toks }
19157 \group_end:

```

(End definition for `\token_if_chardef:NTF` and others. These functions are documented on page 189.)

```

\token_if_primitive_p:N
\token_if_primitive:NTF
\__token_if_primitive:NNw
  \__token_if_primitive_space:w
  \__token_if_primitive_nullfont:N
\__token_if_primitive_loop:N
  \__token_if_primitive:Nw
  \__token_if_primitive_undefined:N
\__token_if_primitive_lua:N

```

We filter out macros first, because they cause endless trouble later otherwise.

Primitives are almost distinguished by the fact that the result of `\token_to_meaning:N` is formed from letters only. Every other token has either a space (e.g., the letter A), a digit (e.g., `\count123`) or a double quote (e.g., `\char"A`).

Ten exceptions: on the one hand, `\tex_undefined:D` is not a primitive, but its meaning is undefined, only letters; on the other hand, `\space`, `\italiccorr`, `\hyphen`, `\firstmark`, `\topmark`, `\botmark`, `\splitfirstmark`, `\splitbotmark`, and `\nullfont` are primitives, but have non-letters in their meaning.

We start by removing the two first (non-space) characters from the meaning. This removes the escape character (which may be nonexistent depending on `\endlinechar`), and takes care of three of the exceptions: `\space`, `\italiccorr` and `\hyphen`, whose meaning is at most two characters. This leaves a string terminated by some `:`, and `\s__token_stop`.

The meaning of each one of the five `\...mark` primitives has the form $\langle letters \rangle : \langle user\ material \rangle$. In other words, the first non-letter is a colon. We remove everything after the first colon.

We are now left with a string, which we must analyze. For primitives, it contains only letters. For non-primitives, it contains either `"`, or a space, or a digit. Two exceptions remain: `\tex_undefined:D`, which is not a primitive, and `\nullfont`, which is a primitive.

Spaces cannot be grabbed in an undelimited way, so we check them separately. If there is a space, we test for `\nullfont`. Otherwise, we go through characters one by one, and stop at the first character less than `'A'` (this is not quite a test for “only letters”, but is close enough to work in this context). If this first character is `:` then we have a primitive, or `\tex_undefined:D`, and if it is `"` or a digit, then the token is not a primitive.

For LuaTeX we use a different implementation which just looks at the command code for the token and compares it to a list of non-primitives. Again, `\nullfont` is a special case because it is the only primitive with the normally non-primitive `set_font` command code.

In LuaMetaTeX some of the command names are different, so we check for both versions. The first one is always the LuaTeX version.

```

19158 \sys_if_engine luatex:TF
19159 {
19160   </tex>
19161   <lua>
19162   do
19163     local get_next = token.get_next
19164     local get_command = token.get_command
19165     local get_index = token.get_index
19166     local get_mode = token.get_mode or token.get_index
19167     local cmd = command_id
19168     local set_font = cmd'get_font'
19169     local biggest_char = token.biggest_char and token.biggest_char()
19170                           or status.getconstants().max_character_code
19171

```

```

19172 local mode_below_biggest_char = {}
19173 local index_not_nil = {}
19174 local mode_not_null = {}
19175 local non_primitive = {
19176   [cmd'left_brace'] = true,
19177   [cmd'right_brace'] = true,
19178   [cmd'math_shift'] = true,
19179   [cmd'mac_param' or cmd'parameter'] = mode_below_biggest_char,
19180   [cmd'sup_mark' or cmd'superscript'] = true,
19181   [cmd'sub_mark' or cmd'subscript'] = true,
19182   [cmd'endv' or cmd'ignore'] = true,
19183   [cmd'spacer'] = true,
19184   [cmd'letter'] = true,
19185   [cmd'other_char'] = true,
19186   [cmd'tab_mark' or cmd'alignment_tab'] = mode_below_biggest_char,
19187   [cmd'char_given'] = true,
19188   [cmd'math_given' or 'math_char_given'] = true,
19189   [cmd'xmath_given' or 'math_char_xgiven'] = true,
19190   [cmd'set_font'] = mode_not_null,
19191   [cmd'undefined_cs'] = true,
19192   [cmd'call'] = true,
19193   [cmd'long_call' or cmd'protected_call'] = true,
19194   [cmd'outer_call' or cmd'tolerant_call'] = true,
19195   [cmd'long_outer_call' or cmd'tolerant_protected_call'] = true,
19196   [cmd'assign_glue' or cmd'register_glue'] = index_not_nil,
19197   [cmd'assign_mu_glue' or cmd'register_mu_glue'] = index_not_nil,
19198   [cmd'assign_toks' or cmd'register_toks'] = index_not_nil,
19199   [cmd'assign_int' or cmd'register_int'] = index_not_nil,
19200   [cmd'assign_attr' or cmd'register_attribute'] = true,
19201   [cmd'assign_dimen' or cmd'register_dimen'] = index_not_nil,
19202 }
19203
19204 luacmd("__token_if_primitive_lua:N", function()
19205   local tok = get_next()
19206   local is_non_primitive = non_primitive[get_command(tok)]
19207   return put_next(
19208     is_non_primitive == true
19209     and false_tok
19210   or is_non_primitive == nil
19211     and true_tok
19212   or is_non_primitive == mode_not_null
19213     and (get_mode(tok) == 0 and true_tok or false_tok)
19214   or is_non_primitive == index_not_nil
19215     and (get_index(tok) and false_tok or true_tok)
19216   or is_non_primitive == mode_below_biggest_char
19217     and (get_mode(tok) > biggest_char and true_tok or false_tok))
19218   end, "global")
19219 end
19220 </lua>
19221 <*tex>
19222 \prg_new_conditional:Npnn \token_if_primitive:N #1 { p , T , F , TF }
19223 {
19224   \__token_if_primitive_lua:N #1
19225 }

```

```

19226 }
19227 {
19228   \tex_chardef:D \c__token_A_int = 'A ~ %
19229   \use:x
19230   {
19231     \prg_new_conditional:Npnn \exp_not:N \token_if_primitive:N ##1
19232     { p , T , F , TF }
19233     {
19234       \exp_not:N \token_if_macro:NTF ##1
19235       \exp_not:N \prg_return_false:
19236       {
19237         \exp_not:N \exp_after:wN \exp_not:N \__token_if_primitive:NNw
19238         \exp_not:N \token_to_meaning:N ##1
19239         \tl_to_str:n { : : : } \s__token_stop ##1
19240       }
19241     }
19242     \cs_new:Npn \exp_not:N \__token_if_primitive:NNw
19243     ##1##2 ##3 \c_colon_str ##4 \s__token_stop
19244     {
19245       \exp_not:N \tl_if_empty:oTF
19246       { \exp_not:N \__token_if_primitive_space:w ##3 ~ }
19247       {
19248         \exp_not:N \__token_if_primitive_loop:N ##3
19249         \c_colon_str \s__token_stop
19250       }
19251       { \exp_not:N \__token_if_primitive_nullfont:N }
19252     }
19253   }
19254   \cs_new:Npn \__token_if_primitive_space:w #1 ~ { }
19255   \cs_new:Npn \__token_if_primitive_nullfont:N #1
19256   {
19257     \if_meaning:w \tex_nullfont:D #1
19258     \prg_return_true:
19259     \else:
19260     \prg_return_false:
19261     \fi:
19262   }
19263   \cs_new:Npn \__token_if_primitive_loop:N #1
19264   {
19265     \if_int_compare:w '#1 < \c__token_A_int %
19266     \exp_after:wN \__token_if_primitive:Nw
19267     \exp_after:wN #1
19268     \else:
19269     \exp_after:wN \__token_if_primitive_loop:N
19270     \fi:
19271   }
19272   \cs_new:Npn \__token_if_primitive:Nw #1 #2 \s__token_stop
19273   {
19274     \if:w : #1
19275     \exp_after:wN \__token_if_primitive_undefined:N
19276     \else:
19277     \prg_return_false:
19278     \exp_after:wN \use_none:n
19279     \fi:

```

```

19280     }
19281     \cs_new:Npn \__token_if_primitive_undefined:N #1
19282     {
19283         \if_cs_exist:N #1
19284         \prg_return_true:
19285         \else:
19286         \prg_return_false:
19287         \fi:
19288     }
19289 }

```

(End definition for \token_if_primitive:NTF and others. This function is documented on page 190.)

```

\token_case_catcode:Nn
\token_case_catcode:NnTF
\token_case_charcode:Nn
\token_case_charcode:NnTF
\token_case_meaning:Nn
\token_case_meaning:NnTF
  \__token_case:NNnTF
  \__token_case:NNw
  \__token_case_end:nw

```

The aim here is to allow the case statement to be evaluated using a known number of expansion steps (two), and without needing to use an explicit “end of recursion” marker. That is achieved by using the test input as the final case, as this is always true. The trick is then to tidy up the output such that the appropriate case code plus either the true or false branch code is inserted.

```

19290 \cs_new:Npn \token_case_catcode:Nn #1#2
19291   { \exp:w \__token_case:NNnTF \token_if_eq_catcode:NNTF #1 {#2} { } { } }
19292 \cs_new:Npn \token_case_catcode:NnT #1#2#3
19293   { \exp:w \__token_case:NNnTF \token_if_eq_catcode:NNTF #1 {#2} {#3} { } }
19294 \cs_new:Npn \token_case_catcode:NnF #1#2
19295   { \exp:w \__token_case:NNnTF \token_if_eq_catcode:NNTF #1 {#2} { } }
19296 \cs_new:Npn \token_case_catcode:NnTF
19297   { \exp:w \__token_case:NNnTF \token_if_eq_catcode:NNTF }
19298 \cs_new:Npn \token_case_charcode:Nn #1#2
19299   { \exp:w \__token_case:NNnTF \token_if_eq_charcode:NNTF #1 {#2} { } { } }
19300 \cs_new:Npn \token_case_charcode:NnT #1#2#3
19301   { \exp:w \__token_case:NNnTF \token_if_eq_charcode:NNTF #1 {#2} {#3} { } }
19302 \cs_new:Npn \token_case_charcode:NnF #1#2
19303   { \exp:w \__token_case:NNnTF \token_if_eq_charcode:NNTF #1 {#2} { } }
19304 \cs_new:Npn \token_case_charcode:NnTF
19305   { \exp:w \__token_case:NNnTF \token_if_eq_charcode:NNTF }
19306 \cs_new:Npn \token_case_meaning:Nn #1#2
19307   { \exp:w \__token_case:NNnTF \token_if_eq_meaning:NNTF #1 {#2} { } { } }
19308 \cs_new:Npn \token_case_meaning:NnT #1#2#3
19309   { \exp:w \__token_case:NNnTF \token_if_eq_meaning:NNTF #1 {#2} {#3} { } }
19310 \cs_new:Npn \token_case_meaning:NnF #1#2
19311   { \exp:w \__token_case:NNnTF \token_if_eq_meaning:NNTF #1 {#2} { } }
19312 \cs_new:Npn \token_case_meaning:NnTF
19313   { \exp:w \__token_case:NNnTF \token_if_eq_meaning:NNTF }
19314 \cs_new:Npn \__token_case:NNnTF #1#2#3#4#5
19315   {
19316     \__token_case:NNw #1 #2 #3 #2 { }
19317     \s_token_mark {#4}
19318     \s_token_mark {#5}
19319     \s_token_stop
19320   }
19321 \cs_new:Npn \__token_case:NNw #1#2#3#4
19322   {
19323     #1 #2 #3
19324     { \__token_case_end:nw {#4} }
19325     { \__token_case:NNw #1 #2 }

```

```
19326 }
```

To tidy up the recursion, there are two outcomes. If there was a hit to one of the cases searched for, then #1 is the code to insert, #2 is the *next* case to check on and #3 is all of the rest of the cases code. That means that #4 is the **true** branch code, and #5 tidies up the spare `\s__token_mark` and the **false** branch. On the other hand, if none of the cases matched then we arrive here using the “termination” case of comparing the search with itself. That means that #1 is empty, #2 is the first `\s__token_mark` and so #4 is the **false** code (the **true** code is mopped up by #3).

```
19327 \cs_new:Npn \__token_case_end:nw #1#2#3 \s__token_mark #4#5 \s__token_stop
19328 { \exp_end: #1 #4 }
```

(End definition for `\token_case_catcode:NnTF` and others. These functions are documented on page 191.)

60.6 Peeking ahead at the next token

```
19329 (@@=peek)
```

Peeking ahead is implemented using a two part mechanism. The outer level provides a defined interface to the lower level material. This allows a large amount of code to be shared. There are four cases:

1. peek at the next token;
2. peek at the next non-space token;
3. peek at the next token and remove it;
4. peek at the next non-space token and remove it.

`\l_peek_token` Storage tokens which are publicly documented: the token peeked.

```
\g_peek_token 19330 \cs_new_eq:NN \l_peek_token ?
19331 \cs_new_eq:NN \g_peek_token ?
```

(End definition for `\l_peek_token` and `\g_peek_token`. These variables are documented on page 191.)

`\l__peek_search_token` The token to search for as an implicit token: cf. `\l__peek_search_tl`.

```
19332 \cs_new_eq:NN \l__peek_search_token ?
```

(End definition for `\l__peek_search_token`.)

`\l__peek_search_tl` The token to search for as an explicit token: cf. `\l__peek_search_token`.

```
19333 \tl_new:N \l__peek_search_tl
```

(End definition for `\l__peek_search_tl`.)

`__peek_true:w` Functions used by the branching and space-stripping code.

```
\__peek_true_aux:w 19334 \cs_new:Npn \__peek_true:w { }
\__peek_false:w 19335 \cs_new:Npn \__peek_true_aux:w { }
\__peek_tmp:w 19336 \cs_new:Npn \__peek_false:w { }
19337 \cs_new:Npn \__peek_tmp:w { }
```

(End definition for `__peek_true:w` and others.)

`\s__peek_mark` Internal scan marks.

`\s__peek_stop` 19338 `\scan_new:N \s__peek_mark`
19339 `\scan_new:N \s__peek_stop`

(End definition for `\s__peek_mark` and `\s__peek_stop`.)

`__peek_use_none_delimit_by_s_stop:w` Functions to gobble up to a scan mark.

19340 `\cs_new:Npn __peek_use_none_delimit_by_s_stop:w #1 \s__peek_stop { }`

(End definition for `__peek_use_none_delimit_by_s_stop:w`.)

`\peek_after:Nw` Simple wrappers for `\futurelet`: no arguments absorbed here.

`\peek_gafter:Nw` 19341 `\cs_new_protected:Npn \peek_after:Nw`
19342 `{ \tex_futurelet:D \l_peek_token }`
19343 `\cs_new_protected:Npn \peek_gafter:Nw`
19344 `{ \tex_global:D \tex_futurelet:D \g_peek_token }`

(End definition for `\peek_after:Nw` and `\peek_gafter:Nw`. These functions are documented on page 191.)

`__peek_true_remove:w` A function to remove the next token and then regain control.

19345 `\cs_new_protected:Npn __peek_true_remove:w`
19346 `{`
19347 `\tex_afterassignment:D __peek_true_aux:w`
19348 `\cs_set_eq:NN __peek_tmp:w`
19349 `}`

(End definition for `__peek_true_remove:w`.)

`\peek_remove_spaces:n` Repeatedly use `__peek_true_remove:w` to remove a space and call `__peek_true_`
`__peek_remove_spaces:` `aux:w`.

19350 `\cs_new_protected:Npn \peek_remove_spaces:n #1`
19351 `{`
19352 `\cs_set:Npx __peek_false:w { \exp_not:n {#1} }`
19353 `\group_align_safe_begin:`
19354 `\cs_set:Npn __peek_true_aux:w { \peek_after:Nw __peek_remove_spaces: }`
19355 `__peek_true_aux:w`
19356 `}`
19357 `\cs_new_protected:Npn __peek_remove_spaces:`
19358 `{`
19359 `\if_meaning:w \l_peek_token \c_space_token`
19360 `\exp_after:wN __peek_true_remove:w`
19361 `\else:`
19362 `\group_align_safe_end:`
19363 `\exp_after:wN __peek_false:w`
19364 `\fi:`
19365 `}`

(End definition for `\peek_remove_spaces:n` and `__peek_remove_spaces:`. This function is documented on page 192.)

`\peek_remove_filler:n` Here we expand the input, removing spaces and `\scan_stop:` tokens until we reach a non-expandable token. At that stage we re-insert the payload. To deal with the problem of `&` tokens, we have to put the align-safe group in the correct place.

```

19366 \cs_new_protected:Npn \peek_remove_filler:n #1
19367 {
19368   \cs_set:Npn \__peek_true_aux:w { \__peek_remove_filler:w }
19369   \cs_set:Npx \__peek_false:w
19370   {
19371     \exp_not:N \group_align_safe_end:
19372     \exp_not:n {#1}
19373   }
19374   \group_align_safe_begin:
19375   \__peek_remove_filler:w
19376 }
19377 \cs_new_protected:Npn \__peek_remove_filler:w
19378 {
19379   \exp_after:wN \peek_after:Nw \exp_after:wN \__peek_remove_filler:
19380   \exp:w \exp_end_continue_f:w
19381 }

```

Here we can nest conditionals as `\l_peek_token` is only skipped over in the nested one if it's a space: no problems with conditionals or outer tokens.

```

19382 \cs_new_protected:Npn \__peek_remove_filler:
19383 {
19384   \if_catcode:w \exp_not:N \l_peek_token \c_space_token
19385   \exp_after:wN \__peek_true_remove:w
19386   \else:
19387     \if_meaning:w \l_peek_token \scan_stop:
19388     \exp_after:wN \exp_after:wN \exp_after:wN
19389     \__peek_true_remove:w
19390   \else:
19391     \exp_after:wN \exp_after:wN \exp_after:wN
19392     \__peek_remove_filler_expand:w
19393   \fi:
19394   \fi:
19395 }

```

To deal with undefined control sequences in the same way \TeX does, we need to check for expansion manually.

```

19396 \cs_new_protected:Npn \__peek_remove_filler_expand:w
19397 {
19398   \exp_after:wN \if_meaning:w \exp_not:N \l_peek_token \l_peek_token
19399   \exp_after:wN \__peek_false:w
19400   \else:
19401     \exp_after:wN \__peek_remove_filler:w
19402   \fi:
19403 }

```

(End definition for `\peek_remove_filler:n` and others. This function is documented on page 193.)

`__peek_token_generic_aux:NNWTF` The generic functions store the test token in both implicit and explicit modes, and the `true` and `false` code as token lists, more or less. The two branches have to be absorbed here as the input stream needs to be cleared for the peek function itself. Here, `#1` is `__peek_true_remove:w` when removing the token and `__peek_true_aux:w` otherwise.

```

19404 \cs_new_protected:Npn \__peek_token_generic_aux:NNNTF #1#2#3#4#5
19405 {
19406   \group_align_safe_begin:
19407   \cs_set_eq:NN \l__peek_search_token #3
19408   \tl_set:Nn \l__peek_search_tl {#3}
19409   \cs_set:Npx \__peek_true_aux:w
19410   {
19411     \exp_not:N \group_align_safe_end:
19412     \exp_not:n {#4}
19413   }
19414   \cs_set_eq:NN \__peek_true:w #1
19415   \cs_set:Npx \__peek_false:w
19416   {
19417     \exp_not:N \group_align_safe_end:
19418     \exp_not:n {#5}
19419   }
19420   \peek_after:Nw #2
19421 }

```

(End definition for __peek_token_generic_aux:NNNTF.)

__peek_token_generic:NNTF For token removal there needs to be a call to the auxiliary function which does the work.

```

\__peek_token_remove_generic:NNTF
19422 \cs_new_protected:Npn \__peek_token_generic:NNTF
19423 { \__peek_token_generic_aux:NNNTF \__peek_true_aux:w }
19424 \cs_new_protected:Npn \__peek_token_generic:NNT #1#2#3
19425 { \__peek_token_generic:NNTF #1 #2 {#3} { } }
19426 \cs_new_protected:Npn \__peek_token_generic:NNTF #1#2#3
19427 { \__peek_token_generic:NNTF #1 #2 { } {#3} }
19428 \cs_new_protected:Npn \__peek_token_remove_generic:NNTF
19429 { \__peek_token_generic_aux:NNNTF \__peek_true_remove:w }
19430 \cs_new_protected:Npn \__peek_token_remove_generic:NNT #1#2#3
19431 { \__peek_token_remove_generic:NNTF #1 #2 {#3} { } }
19432 \cs_new_protected:Npn \__peek_token_remove_generic:NNTF #1#2#3
19433 { \__peek_token_remove_generic:NNTF #1 #2 { } {#3} }

```

(End definition for __peek_token_generic:NNTF and __peek_token_remove_generic:NNTF.)

__peek_execute_branches_meaning: The meaning test is straight forward.

```

19434 \cs_new:Npn \__peek_execute_branches_meaning:
19435 {
19436   \if_meaning:w \l__peek_token \l__peek_search_token
19437   \exp_after:wN \__peek_true:w
19438   \else:
19439   \exp_after:wN \__peek_false:w
19440   \fi:
19441 }

```

(End definition for __peek_execute_branches_meaning:.)

__peek_execute_branches_catcode: The catcode and charcode tests are very similar, and in order to use the same auxiliaries we do something a little bit odd, firing \if_catcode:w and \if_charcode:w before finding the operands for those tests, which are only given in the auxii:N and auxiii: auxiliaries. For our purposes, three kinds of tokens may follow the peeking function:

__peek_execute_branches_catcode_aux:
 __peek_execute_branches_catcode_auxii:N
 __peek_execute_branches_catcode_auxiii:

- control sequences which are not equal to a non-active character token (*e.g.*, macro, primitive);
- active characters which are not equal to a non-active character token (*e.g.*, macro, primitive);
- explicit non-active character tokens, or control sequences or active characters set equal to a non-active character token.

The first two cases are not distinguishable simply using \TeX 's `\futurelet`, because we can only access the `\meaning` of tokens in that way. In those cases, detected thanks to a comparison with `\scan_stop:`, we grab the following token, and compare it explicitly with the explicit search token stored in `\l__peek_search_tl`. The `\exp_not:N` prevents outer macros (coming from non- \LaTeX 3 code) from blowing up. In the third case, `\l__peek_token` is good enough for the test, and we compare it again with the explicit search token. Just like the peek token, the search token may be of any of the three types above, hence the need to use the explicit token that was given to the peek function.

```

19442 \cs_new:Npn \__peek_execute_branches_catcode:
19443   { \if_catcode:w \__peek_execute_branches_catcode_aux: }
19444 \cs_new:Npn \__peek_execute_branches_charcode:
19445   { \if_charcode:w \__peek_execute_branches_catcode_aux: }
19446 \cs_new:Npn \__peek_execute_branches_catcode_aux:
19447   {
19448     \if_catcode:w \exp_not:N \l__peek_token \scan_stop:
19449       \exp_after:wN \exp_after:wN
19450       \exp_after:wN \__peek_execute_branches_catcode_auxiii:
19451       \exp_after:wN \exp_not:N
19452     \else:
19453       \exp_after:wN \__peek_execute_branches_catcode_auxiii:
19454     \fi:
19455   }
19456 \cs_new:Npn \__peek_execute_branches_catcode_auxii:N #1
19457   {
19458     \exp_not:N #1
19459     \exp_after:wN \exp_not:N \l__peek_search_tl
19460     \exp_after:wN \__peek_true:w
19461   \else:
19462     \exp_after:wN \__peek_false:w
19463   \fi:
19464   #1
19465   }
19466 \cs_new:Npn \__peek_execute_branches_catcode_auxiii:
19467   {
19468     \exp_not:N \l__peek_token
19469     \exp_after:wN \exp_not:N \l__peek_search_tl
19470     \exp_after:wN \__peek_true:w
19471   \else:
19472     \exp_after:wN \__peek_false:w
19473   \fi:
19474   }

```

(End definition for `__peek_execute_branches_catcode:` and others.)

The public functions themselves cannot be defined using `\prg_new_conditional:Npnn`. Instead, the TF, T, F variants are defined in terms of corresponding variants of

```

\peek_catcode:NTF
\peek_catcode_remove:NTF
\peek_charcode:NTF
\peek_charcode_remove:NTF
\peek_meaning:NTF
\peek_meaning_remove:NTF

```

_peek_token_generic:NNTF or _peek_token_remove_generic:NNTF, with first argument one of _peek_execute_branches_catcode:, _peek_execute_branches_charcode:, or _peek_execute_branches_meaning:.

```

19475 \tl_map_inline:nn { { catcode } { charcode } { meaning } }
19476 {
19477   \tl_map_inline:nn { { } { _remove } }
19478   {
19479     \tl_map_inline:nn { { TF } { T } { F } }
19480     {
19481       \cs_new_protected:cpx { peek_ #1 ##1 :N ####1 }
19482       {
19483         \exp_not:c { __peek_token ##1 _generic:NN ####1 }
19484         \exp_not:c { __peek_execute_branches_ #1 : }
19485       }
19486     }
19487   }
19488 }

```

(End definition for \peek_catcode:NNTF and others. These functions are documented on page 192.)

\peek_N_type:TF All tokens are N-type tokens, except in four cases: begin-group tokens, end-group tokens, space tokens with character code 32, and outer tokens. Since \l_peek_token might be outer, we cannot use the convenient \bool_if:NNTF function, and must resort to the old trick of using \ifodd to expand a set of tests. The false branch of this test is taken if the token is one of the first three kinds of non-N-type tokens (explicit or implicit), thus we call _peek_false:w. In the true branch, we must detect outer tokens, without impacting performance too much for non-outer tokens. The first filter is to search for outer in the meaning of \l_peek_token. If that is absent, _peek_use_none_delimit_by_s_stop:w cleans up, and we call _peek_true:w. Otherwise, the token can be a non-outer macro or a primitive mark whose parameter or replacement text contains outer, it can be the primitive \outer, or it can be an outer token. Macros and marks would have ma in the part before the first occurrence of outer; the meaning of \outer has nothing after outer, contrarily to outer macros; and that covers all cases, calling _peek_true:w or _peek_false:w as appropriate. Here, there is no *search token*, so we feed a dummy \scan_stop: to the _peek_token_generic:NNTF function.

```

19489 \group_begin:
19490   \cs_set_protected:Npn \_peek_tmp:w #1 \s_peek_stop
19491   {
19492     \cs_new_protected:Npn \_peek_execute_branches_N_type:
19493     {
19494       \if_int_odd:w
19495         \if_catcode:w \exp_not:N \l_peek_token { \c_zero_int \fi:
19496         \if_catcode:w \exp_not:N \l_peek_token } \c_zero_int \fi:
19497         \if_meaning:w \l_peek_token \c_space_token \c_zero_int \fi:
19498         \c_one_int
19499         \exp_after:wN \_peek_N_type:w
19500         \token_to_meaning:N \l_peek_token
19501         \s_peek_mark \_peek_N_type_aux:nnw
19502         #1 \s_peek_mark \_peek_use_none_delimit_by_s_stop:w
19503         \s_peek_stop
19504         \exp_after:wN \_peek_true:w
19505       \else:
19506         \exp_after:wN \_peek_false:w

```

```

19507         \fi:
19508     }
19509     \cs_new_protected:Npn \__peek_N_type:w ##1 #1 ##2 \s__peek_mark ##3
19510     { ##3 {##1} {##2} }
19511 }
19512 \exp_after:wN \__peek_tmp:w \tl_to_str:n { outer } \s__peek_stop
19513 \group_end:
19514 \cs_new_protected:Npn \__peek_N_type_aux:nnw #1 #2 #3 \fi:
19515 {
19516     \fi:
19517     \tl_if_in:noTF {#1} { \tl_to_str:n {ma} }
19518     { \__peek_true:w }
19519     { \tl_if_empty:nTF {#2} { \__peek_true:w } { \__peek_false:w } }
19520 }
19521 \cs_new_protected:Npn \peek_N_type:TF
19522 {
19523     \__peek_token_generic:NNTF
19524     \__peek_execute_branches_N_type: \scan_stop:
19525 }
19526 \cs_new_protected:Npn \peek_N_type:T
19527 { \__peek_token_generic:NNT \__peek_execute_branches_N_type: \scan_stop: }
19528 \cs_new_protected:Npn \peek_N_type:F
19529 { \__peek_token_generic:NNTF \__peek_execute_branches_N_type: \scan_stop: }

(End definition for \peek_N_type:TF and others. This function is documented on page 193.)

19530 </tex>
19531 </package>

```

Chapter 61

l3prop implementation

The following test files are used for this code: `m3prop001`, `m3prop002`, `m3prop003`, `m3prop004`, `m3show001`.

```
19532 <*package>
```

```
19533 <@@=prop>
```

A property list is a macro whose top-level expansion is of the form

```
\s__prop \__prop_pair:wn <key1> \s__prop {<value1>}  
...  
\__prop_pair:wn <keyn> \s__prop {<valuen>}
```

where `\s__prop` is a scan mark (equal to `\scan_stop:`), and `__prop_pair:wn` can be used to map through the property list.

`\s__prop` The internal token used at the beginning of property lists. This is also used after each `<key>` (see `__prop_pair:wn`).

(End definition for `\s__prop`.)

`__prop_pair:wn` `__prop_pair:wn <key> \s__prop {<item>}`

The internal token used to begin each key–value pair in the property list. If expanded outside of a mapping or manipulation function, an error is raised. The definition should always be set globally.

(End definition for `__prop_pair:wn`.)

`\l__prop_internal_tl` Token list used to store new key–value pairs to be inserted by functions of the `\prop_put:Nnn` family.

(End definition for `\l__prop_internal_tl`.)

`__prop_split:NnTF`

Updated: 2013-01-08

`__prop_split:NnTF` $\langle\textit{property list}\rangle$ $\{\langle\textit{key}\rangle\}$ $\{\langle\textit{true code}\rangle\}$ $\{\langle\textit{false code}\rangle\}$

Splits the $\langle\textit{property list}\rangle$ at the $\langle\textit{key}\rangle$, giving three token lists: the $\langle\textit{extract}\rangle$ of $\langle\textit{property list}\rangle$ before the $\langle\textit{key}\rangle$, the $\langle\textit{value}\rangle$ associated with the $\langle\textit{key}\rangle$ and the $\langle\textit{extract}\rangle$ of the $\langle\textit{property list}\rangle$ after the $\langle\textit{value}\rangle$. Both $\langle\textit{extracts}\rangle$ retain the internal structure of a property list, and the concatenation of the two $\langle\textit{extracts}\rangle$ is a property list. If the $\langle\textit{key}\rangle$ is present in the $\langle\textit{property list}\rangle$ then the $\langle\textit{true code}\rangle$ is left in the input stream, with #1, #2, and #3 replaced by the first $\langle\textit{extract}\rangle$, the $\langle\textit{value}\rangle$, and the second $\langle\textit{extract}\rangle$. If the $\langle\textit{key}\rangle$ is not present in the $\langle\textit{property list}\rangle$ then the $\langle\textit{false code}\rangle$ is left in the input stream, with no trailing material. Both $\langle\textit{true code}\rangle$ and $\langle\textit{false code}\rangle$ are used in the replacement text of a macro defined internally, hence macro parameter characters should be doubled, except #1, #2, and #3 which stand in the $\langle\textit{true code}\rangle$ for the three extracts from the property list. The $\langle\textit{key}\rangle$ comparison takes place as described for `\str_if_eq:nn`.

`\s__prop` A private scan mark is used as a marker after each key, and at the very beginning of the property list.

19534 `\scan_new:N \s__prop`

(End definition for `\s__prop`.)

`__prop_pair:wn` The delimiter is always defined, but when misused simply triggers an error and removes its argument.

19535 `\cs_new:Npn __prop_pair:wn #1 \s__prop #2`

19536 `{ \msg_expandable_error:nn { prop } { misused } }`

(End definition for `__prop_pair:wn`.)

`\l__prop_internal_tl` Token list used to store the new key–value pair inserted by `\prop_put:Nnn` and friends.

19537 `\tl_new:N \l__prop_internal_tl`

(End definition for `\l__prop_internal_tl`.)

`\c_empty_prop` An empty prop.

19538 `\tl_const:Nn \c_empty_prop { \s__prop }`

(End definition for `\c_empty_prop`. This variable is documented on page 205.)

61.1 Internal auxiliaries

`\s__prop_mark` Internal scan marks.

`\s__prop_stop` 19539 `\scan_new:N \s__prop_mark`

19540 `\scan_new:N \s__prop_stop`

(End definition for `\s__prop_mark` and `\s__prop_stop`.)

`\q__prop_recursion_tail` Internal recursion quarks.

`\q__prop_recursion_stop` 19541 `\quark_new:N \q__prop_recursion_tail`

19542 `\quark_new:N \q__prop_recursion_stop`

(End definition for `\q__prop_recursion_tail` and `\q__prop_recursion_stop`.)

`__prop_if_recursion_tail_stop:n` Functions to query recursion quarks.

`__prop_if_recursion_tail_stop:o` 19543 `__kernel_quark_new_test:N __prop_if_recursion_tail_stop:n`

19544 `\cs_generate_variant:Nn __prop_if_recursion_tail_stop:n { o }`

(End definition for `__prop_if_recursion_tail_stop:n` and `__prop_if_recursion_tail_stop:o`.)

61.2 Allocation and initialisation

\prop_new:N Property lists are initialized with the value `\c_empty_prop`.

```
\prop_new:c
19545 \cs_new_protected:Npn \prop_new:N #1
19546 {
19547   \__kernel_chk_if_free_cs:N #1
19548   \cs_gset_eq:NN #1 \c_empty_prop
19549 }
19550 \cs_generate_variant:Nn \prop_new:N { c }
```

(End definition for `\prop_new:N`. This function is documented on page 198.)

\prop_clear:N The same idea for clearing.

```
\prop_clear:c
19551 \cs_new_protected:Npn \prop_clear:N #1
\prop_gclear:N
19552 { \prop_set_eq:NN #1 \c_empty_prop }
\prop_gclear:c
19553 \cs_generate_variant:Nn \prop_clear:N { c }
19554 \cs_new_protected:Npn \prop_gclear:N #1
19555 { \prop_gset_eq:NN #1 \c_empty_prop }
19556 \cs_generate_variant:Nn \prop_gclear:N { c }
```

(End definition for `\prop_clear:N` and `\prop_gclear:N`. These functions are documented on page 198.)

\prop_clear_new:N Once again a simple variation of the token list functions.

```
\prop_clear_new:c
19557 \cs_new_protected:Npn \prop_clear_new:N #1
\prop_gclear_new:N
19558 { \prop_if_exist:NTF #1 { \prop_clear:N #1 } { \prop_new:N #1 } }
\prop_gclear_new:c
19559 \cs_generate_variant:Nn \prop_clear_new:N { c }
19560 \cs_new_protected:Npn \prop_gclear_new:N #1
19561 { \prop_if_exist:NTF #1 { \prop_gclear:N #1 } { \prop_new:N #1 } }
19562 \cs_generate_variant:Nn \prop_gclear_new:N { c }
```

(End definition for `\prop_clear_new:N` and `\prop_gclear_new:N`. These functions are documented on page 198.)

\prop_set_eq:NN These are simply copies from the token list functions.

```
\prop_set_eq:cN
19563 \cs_new_eq:NN \prop_set_eq:NN \tl_set_eq:NN
\prop_set_eq:Nc
19564 \cs_new_eq:NN \prop_set_eq:Nc \tl_set_eq:Nc
\prop_set_eq:cc
19565 \cs_new_eq:NN \prop_set_eq:cN \tl_set_eq:cN
\prop_gset_eq:NN
19566 \cs_new_eq:NN \prop_set_eq:cc \tl_set_eq:cc
\prop_gset_eq:cN
19567 \cs_new_eq:NN \prop_gset_eq:NN \tl_gset_eq:NN
\prop_gset_eq:Nc
19568 \cs_new_eq:NN \prop_gset_eq:Nc \tl_gset_eq:Nc
\prop_gset_eq:cN
19569 \cs_new_eq:NN \prop_gset_eq:cN \tl_gset_eq:cN
\prop_gset_eq:cc
19570 \cs_new_eq:NN \prop_gset_eq:cc \tl_gset_eq:cc
```

(End definition for `\prop_set_eq:NN` and `\prop_gset_eq:NN`. These functions are documented on page 199.)

\l_tmpa_prop We can now initialize the scratch variables.

```
\l_tmpb_prop
19571 \prop_new:N \l_tmpa_prop
\g_tmpa_prop
19572 \prop_new:N \l_tmpb_prop
\g_tmpb_prop
19573 \prop_new:N \g_tmpa_prop
19574 \prop_new:N \g_tmpb_prop
```

(End definition for `\l_tmpa_prop` and others. These variables are documented on page 205.)

\l__prop_internal_prop Property list used by `\prop_concat:NNN`, `\prop_set_from_keyval:Nn` and others.

```
19575 \prop_new:N \l__prop_internal_prop
```

(End definition for \l__prop_internal_prop.)

```

\prop_concat:NNN Combine two property lists. We cannot use a simple \tl_concat:NNN because there may
\prop_concat:ccc be some duplicate keys between the two property lists.
\prop_gconcat:NNN
\prop_gconcat:ccc
\__prop_concat:NNNN
19576 \cs_new_protected:Npn \prop_concat:NNN
19577 { \__prop_concat:NNNN \prop_set_eq:NN }
19578 \cs_generate_variant:Nn \prop_concat:NNN { ccc }
19579 \cs_new_protected:Npn \prop_gconcat:NNN
19580 { \__prop_concat:NNNN \prop_gset_eq:NN }
19581 \cs_generate_variant:Nn \prop_gconcat:NNN { ccc }
19582 \cs_new_protected:Npn \__prop_concat:NNNN #1#2#3#4
19583 {
19584   \prop_set_eq:NN \l__prop_internal_prop #3
19585   \prop_map_inline:Nn #4 { \prop_put:Nnn \l__prop_internal_prop {##1} {##2} }
19586   #1 #2 \l__prop_internal_prop
19587 }

```

(End definition for \prop_concat:NNN, \prop_gconcat:NNN, and __prop_concat:NNNN. These functions are documented on page 200.)

```

\prop_set_from_keyval:Nn To avoid tracking throughout the loop the variable name and whether the assignment is
\prop_set_from_keyval:cn local/global, do everything in a scratch variable and empty it afterwards to avoid wasting
\prop_gset_from_keyval:Nn memory. Loop through items separated by commas, with \prg_do_nothing: to avoid
\prop_gset_from_keyval:cn losing braces. After checking for termination, split the item at the first and then at the
\prop_const_from_keyval:Nn second = (which ought to be the first of the trailing = that we added). For both splits
\prop_const_from_keyval:cn trim spaces and call a function (first \__prop_from_keyval_key:w then \__prop_from_
\prop_put_from_keyval:Nn keyval_value:w), followed by the trimmed material, \s__prop_mark, the subsequent
\prop_put_from_keyval:cn part of the item, and the trailing =’s and \s__prop_stop. After finding the <key> just
\prop_gput_from_keyval:Nn store it after \s__prop_stop. After finding the <value> ignore completely empty items
\prop_gput_from_keyval:cn (both trailing = were used as delimiters and all parts are empty); if the remaining part #2
\__prop_missing_eq:Nn consists exactly of the second trailing = (namely there was exactly one = in the item)
then output one key–value pair for the property list; otherwise complain about a missing
or extra =.

```

```

19588 \cs_new_protected:Npn \prop_set_from_keyval:Nn #1
19589 {
19590   \prop_clear:N #1
19591   \prop_put_from_keyval:Nn #1
19592 }
19593 \cs_generate_variant:Nn \prop_set_from_keyval:Nn { c }
19594 \cs_new_protected:Npn \prop_gset_from_keyval:Nn #1
19595 {
19596   \prop_gclear:N #1
19597   \prop_gput_from_keyval:Nn #1
19598 }
19599 \cs_generate_variant:Nn \prop_gset_from_keyval:Nn { c }
19600 \cs_new_protected:Npn \prop_const_from_keyval:Nn #1#2
19601 {
19602   \prop_set_from_keyval:Nn \l__prop_internal_prop {#2}
19603   \tl_const:Nx #1 { \exp_not:o \l__prop_internal_prop
19604     \prop_clear:N \l__prop_internal_prop
19605   }
19606   \cs_generate_variant:Nn \prop_const_from_keyval:Nn { c }
19607   \cs_new_protected:Npn \prop_put_from_keyval:Nn

```

```

19608 {
19609   \bool_if:NTF \l__kernel_keyval_allow_blank_keys_bool
19610   { \__prop_keyval_parse:NNNn \c_true_bool }
19611   { \__prop_keyval_parse:NNNn \c_false_bool }
19612   \prop_put:Nnn
19613 }
19614 \cs_generate_variant:Nn \prop_put_from_keyval:Nn { c }
19615 \cs_new_protected:Npn \prop_gput_from_keyval:Nn
19616 {
19617   \bool_if:NTF \l__kernel_keyval_allow_blank_keys_bool
19618   { \__prop_keyval_parse:NNNn \c_true_bool }
19619   { \__prop_keyval_parse:NNNn \c_false_bool }
19620   \prop_gput:Nnn
19621 }
19622 \cs_generate_variant:Nn \prop_gput_from_keyval:Nn { c }
19623 \cs_new_protected:Npn \__prop_missing_eq:n
19624 { \msg_error:nnn { prop } { prop-keyval } }
19625 \cs_new_protected:Npn \__prop_keyval_parse:NNNn #1#2#3#4
19626 {
19627   \bool_set_eq:NN \l__kernel_keyval_allow_blank_keys_bool \c_true_bool
19628   \keyval_parse:nnn \__prop_missing_eq:n { #2 #3 } {#4}
19629   \bool_set_eq:NN \l__kernel_keyval_allow_blank_keys_bool #1
19630 }

```

(End definition for `\prop_set_from_keyval:Nn` and others. These functions are documented on page 199.)

61.3 Accessing data in property lists

```

\__prop_split:NnTF
\__prop_split_aux:NnTF
\__prop_split_aux:w

```

This function is used by most of the module, and hence must be fast. It receives a *property list*, a *key*, a *true code* and a *false code*. The aim is to split the *property list* at the given *key* into the *extract₁* before the key–value pair, the *value* associated with the *key* and the *extract₂* after the key–value pair. This is done using a delimited function, whose definition is as follows, where the *key* is turned into a string.

```

\cs_set:Npn \__prop_split_aux:w #1
\__prop_pair:wn <key> \s__prop #2
#3 \s__prop_mark #4 #5 \s__prop_stop
{ #4 {<true code>} {<false code>}}

```

If the *key* is present in the property list, `__prop_split_aux:w`’s #1 is the part before the *key*, #2 is the *value*, #3 is the part after the *key*, #4 is `\use_i:nn`, and #5 is additional tokens that we do not care about. The *true code* is left in the input stream, and can use the parameters #1, #2, #3 for the three parts of the property list as desired. Namely, the original property list is in this case #1 `__prop_pair:wn <key> \s__prop {#2} #3`.

If the *key* is not there, then the *function* is `\use_ii:nn`, which keeps the *false code*.

```

19631 \cs_new_protected:Npn \__prop_split:NnTF #1#2
19632 { \exp_args:NNo \__prop_split_aux:NnTF #1 { \tl_to_str:n {#2} } }
19633 \cs_new_protected:Npn \__prop_split_aux:NnTF #1#2#3#4
19634 {
19635   \cs_set:Npn \__prop_split_aux:w ##1

```

```

19636     \__prop_pair:wn #2 \s__prop ##2 ##3 \s__prop_mark ##4 ##5 \s__prop_stop
19637     { ##4 {#3} {#4} }
19638     \exp_after:wN \__prop_split_aux:w #1 \s__prop_mark \use_i:nn
19639     \__prop_pair:wn #2 \s__prop { } \s__prop_mark \use_ii:nn \s__prop_stop
19640 }
19641 \cs_new:Npn \__prop_split_aux:w { }

```

(End definition for `__prop_split:NnTF`, `__prop_split_aux:NnTF`, and `__prop_split_aux:w`.)

`\prop_remove:Nn` Deleting from a property starts by splitting the list. If the key is present in the property list, the returned value is ignored. If the key is missing, nothing happens.

```

\prop_remove:NV
\prop_remove:cn 19642 \cs_new_protected:Npn \prop_remove:Nn #1#2
\prop_remove:cV 19643 {
19644     \__prop_split:NnTF #1 {#2}
\prop_gremove:Nn 19645     { \tl_set:Nn #1 { ##1 ##3 } }
\prop_gremove:NV 19646     { }
\prop_gremove:cn 19647 }
\prop_gremove:cV 19648 \cs_new_protected:Npn \prop_gremove:Nn #1#2
19649 {
19650     \__prop_split:NnTF #1 {#2}
19651     { \tl_gset:Nn #1 { ##1 ##3 } }
19652     { }
19653 }
19654 \cs_generate_variant:Nn \prop_remove:Nn { NV }
19655 \cs_generate_variant:Nn \prop_remove:Nn { c , cV }
19656 \cs_generate_variant:Nn \prop_gremove:Nn { NV }
19657 \cs_generate_variant:Nn \prop_gremove:Nn { c , cV }

```

(End definition for `\prop_remove:Nn` and `\prop_gremove:Nn`. These functions are documented on page 201.)

`\prop_get:NnN` Getting an item from a list is very easy: after splitting, if the key is in the property list, just set the token list variable to the return value, otherwise to `\q_no_value`.

```

\prop_get:NVN
\prop_get:NoN 19658 \cs_new_protected:Npn \prop_get:NnN #1#2#3
\prop_get:cnN 19659 {
\prop_get:cVN 19660     \__prop_split:NnTF #1 {#2}
\prop_get:coN 19661     { \tl_set:Nn #3 {##2} }
19662     { \tl_set:Nn #3 { \q_no_value } }
19663 }
19664 \cs_generate_variant:Nn \prop_get:NnN { NV , Nv , No }
19665 \cs_generate_variant:Nn \prop_get:NnN { c , cV , cv , co }

```

(End definition for `\prop_get:NnN`. This function is documented on page 200.)

`\prop_pop:NnN` Popping a value also starts by doing the split. If the key is present, save the value in the token list and update the property list as when deleting. If the key is missing, save `\q_no_value` in the token list.

```

\prop_pop:NoN 19666 \cs_new_protected:Npn \prop_pop:NnN #1#2#3
\prop_pop:cnN 19667 {
\prop_pop:coN 19668     \__prop_split:NnTF #1 {#2}
\prop_gpop:Nn 19669     {
\prop_gpop:NoN 19670     \tl_set:Nn #3 {##2}
\prop_gpop:cnN 19671     \tl_set:Nn #1 { ##1 ##3 }
19672     }

```

```

19673     { \tl_set:Nn #3 { \q_no_value } }
19674   }
19675 \cs_new_protected:Npn \prop_gpop:NnN #1#2#3
19676 {
19677   \__prop_split:NnTF #1 {#2}
19678   {
19679     \tl_set:Nn #3 {##2}
19680     \tl_gset:Nn #1 { ##1 ##3 }
19681   }
19682   { \tl_set:Nn #3 { \q_no_value } }
19683 }
19684 \cs_generate_variant:Nn \prop_pop:NnN { No }
19685 \cs_generate_variant:Nn \prop_pop:NnN { c , co }
19686 \cs_generate_variant:Nn \prop_gpop:NnN { No }
19687 \cs_generate_variant:Nn \prop_gpop:NnN { c , co }

```

(End definition for `\prop_pop:NnN` and `\prop_gpop:NnN`. These functions are documented on page 200.)

`\prop_item:Nn` Getting the value corresponding to a key in a property list in an expandable fashion simply uses `\prop_map_tokens:Nn` to go through the property list. The auxiliary `__prop_item:nnn` receives the search string #1, the key #2 and the value #3 and returns as appropriate.

```

19688 \cs_new:Npn \prop_item:Nn #1#2
19689 {
19690   \exp_args:NNo \prop_map_tokens:Nn #1
19691   { \exp_after:wN \__prop_item:nnn \exp_after:wN { \tl_to_str:n {#2} } }
19692 }
19693 \cs_new:Npn \__prop_item:nnn #1#2#3
19694 {
19695   \str_if_eq:eeT {#1} {#2}
19696   { \prop_map_break:n { \exp_not:n {#3} } }
19697 }
19698 \cs_generate_variant:Nn \prop_item:Nn { c }

```

(End definition for `\prop_item:Nn` and `__prop_item:nnn`. This function is documented on page 201.)

`\prop_count:N` Counting the key–value pairs in a property list is done using the same approach as for other count functions: turn each entry into a +1 then use integer evaluation to actually do the mathematics.

```

19699 \cs_new:Npn \prop_count:N #1
19700 {
19701   \int_eval:n
19702   {
19703     0
19704     \prop_map_function:NN #1 \__prop_count:nn
19705   }
19706 }
19707 \cs_new:Npn \__prop_count:nn #1#2 { + 1 }
19708 \cs_generate_variant:Nn \prop_count:N { c }

```

(End definition for `\prop_count:N` and `__prop_count:nn`. This function is documented on page 201.)

`\prop_to_keyval:N` Each property name and value pair will be returned in the form `_{}{<name>}=_{}{<value>}`.
`__prop_to_keyval_exp_after:wN` As one of the main use cases for this macro is to pass the *<property list>* on to a key–
`__prop_to_keyval:nn` value parser, we have to make sure that the behaviour is as good as possible. Using a
`__prop_to_keyval:nnw`

space before the opening brace we get the correct brace stripping behaviour for most of the key-value parsers available in L^AT_EX. If `\tex_expanded:D` is available this function makes use of it, so there are two different implementations here. They both start with `__kernel_exp_not:w` to start the expansion context to expand in two steps. If the \langle *property list* \rangle is empty they just leave an empty set of braces in the input stream for `__kernel_exp_not:w`.

```
19709 \cs_if_exist:NTF \tex_expanded:D
19710 {
```

The variant using `\tex_expanded:D` can just iterate over the \langle *property list* \rangle and remove the leading comma afterwards. Only the value has to be protected in `__kernel_exp_not:w` as the property name is always a string. After the loop the leading comma is removed by `\use_none:n` and afterwards `__kernel_exp_not:w` eventually finds the opening brace of its argument.

```
19711 \cs_new:Npn \prop_to_keyval:N #1
19712 {
19713   \__kernel_exp_not:w
19714   \prop_if_empty:NTF #1
19715   { {} }
19716   {
19717     \exp_after:wN \exp_after:wN \exp_after:wN
19718     {
19719       \tex_expanded:D
19720       {
19721         \__kernel_exp_not:w { \use_none:n }
19722         \prop_map_function:NN #1 \__prop_to_keyval:nn
19723       }
19724     }
19725   }
19726 }
19727 \cs_new:Npn \__prop_to_keyval:nn #1#2
19728 { , ~ {#1} =~ { \__kernel_exp_not:w {#2} } }
19729 }
```

The other variant will iterate over the \langle *property list* \rangle and has to output the result in a group after the marker `__prop_to_keyval_exp_after:wN`. As a result this is considerably slower than the `\tex_expanded:D` using variant as it has to read the entire contents of the \langle *property list* \rangle for each item. Since the marker is just `\exp_after:wN` with another name, after the loop the leading comma is gobbled by `\use_none:n`, leaving the result as the argument to `__kernel_exp_not:w`.

```
19730 {
19731   \cs_new:Npn \prop_to_keyval:N #1
19732   {
19733     \__kernel_exp_not:w
19734     \prop_if_empty:NTF #1
19735     { {} }
19736     {
19737       \prop_map_function:NN #1 \__prop_to_keyval:nnw
19738       \__prop_to_keyval_exp_after:wN { \use_none:n }
19739     }
19740   }
19741   \cs_new_eq:NN \__prop_to_keyval_exp_after:wN \exp_after:wN
19742   \cs_new:Npn \__prop_to_keyval:nnw #1#2#3 \__prop_to_keyval_exp_after:wN #4
```

```

19743     { #3 \__prop_to_keyval_exp_after:wn { #4 , ~ {#1} =~ {#2} } }
19744 }

```

(End definition for \prop_to_keyval:N and others. This function is documented on page 201.)

\prop_pop:NnNTF Popping an item from a property list, keeping track of whether the key was present or not, is implemented as a conditional. If the key was missing, neither the property list, nor the token list are altered. Otherwise, \prg_return_true: is used after the assignments.

\prop_gpop:NnNTF

```

19745 \prg_new_protected_conditional:Npnn \prop_pop:NnN #1#2#3 { T , F , TF }
19746 {
19747   \__prop_split:NnTF #1 {#2}
19748   {
19749     \tl_set:Nn #3 {##2}
19750     \tl_set:Nn #1 { ##1 ##3 }
19751     \prg_return_true:
19752   }
19753   { \prg_return_false: }
19754 }
19755 \prg_new_protected_conditional:Npnn \prop_gpop:NnN #1#2#3 { T , F , TF }
19756 {
19757   \__prop_split:NnTF #1 {#2}
19758   {
19759     \tl_set:Nn #3 {##2}
19760     \tl_gset:Nn #1 { ##1 ##3 }
19761     \prg_return_true:
19762   }
19763   { \prg_return_false: }
19764 }
19765 \prg_generate_conditional_variant:Nnn \prop_pop:NnN { c } { T , F , TF }
19766 \prg_generate_conditional_variant:Nnn \prop_gpop:NnN { c } { T , F , TF }

```

(End definition for \prop_pop:NnNTF and \prop_gpop:NnNTF. These functions are documented on page 202.)

\prop_put:Nnn Since the branches of __prop_split:NnTF are used as the replacement text of an internal macro, and since the *<key>* and new *<value>* may contain arbitrary tokens, it is not safe to include them in the argument of __prop_split:NnTF. We thus start by storing in \l__prop_internal_tl tokens which (after x-expansion) encode the key–value pair. This variable can safely be used in __prop_split:NnTF. If the *<key>* was absent, append the new key–value to the list. Otherwise concatenate the extracts ##1 and ##3 with the new key–value pair \l__prop_internal_tl. The updated entry is placed at the same spot as the original *<key>* in the property list, preserving the order of entries.

\prop_put:NnV

\prop_put:Nno

\prop_put:Nnx

\prop_put:Nvn

\prop_put:Nvv

\prop_put:Nvx

\prop_put:Non

\prop_put:Noo

\prop_put:Nxx

\prop_put:cnn

\prop_put:cnV

\prop_put:cno

\prop_put:cnx

\prop_put:cVn

\prop_put:cVV

\prop_put:cVx

\prop_put:cvx

\prop_put:con

\prop_put:coo

\prop_put:cxx

```

19767 \cs_new_protected:Npn \prop_put:Nnn { \__prop_put:Nnnn \__kernel_tl_set:Nx }
19768 \cs_new_protected:Npn \prop_gput:Nnn { \__prop_put:Nnnn \__kernel_tl_gset:Nx }
19769 \cs_new_protected:Npn \__prop_put:Nnnn #1#2#3#4
19770 {
19771   \tl_set:Nn \l__prop_internal_tl
19772   {
19773     \exp_not:N \__prop_pair:wn \tl_to_str:n {#3}
19774     \s__prop { \exp_not:n {#4} }
19775   }
19776   \__prop_split:NnTF #2 {#3}
19777   { #1 #2 { \exp_not:n {##1} \l__prop_internal_tl \exp_not:n {##3} } }
19778   { #1 #2 { \exp_not:o {#2} \l__prop_internal_tl } }

```

\prop_gput:Nnn

\prop_gput:NnV

\prop_gput:Nno

\prop_gput:Nnx

\prop_gput:Nvn

\prop_gput:Nvv

\prop_gput:Non

```

19779 }
19780 \cs_generate_variant:Nn \prop_put:Nnn
19781 { NnV , Nno , Nnx , NV , NVV , NVx , Nvx , No , Noo , Nxx }
19782 \cs_generate_variant:Nn \prop_put:Nnn
19783 { c , cnV , cno , cnx , cV , cVV , cVx , cvx , co , coo , cxx }
19784 \cs_generate_variant:Nn \prop_gput:Nnn
19785 { NnV , Nno , Nnx , NV , NVV , NVx , Nvx , No , Noo , Nxx }
19786 \cs_generate_variant:Nn \prop_gput:Nnn
19787 { c , cnV , cno , cnx , cV , cVV , cVx , cvx , co , coo , cxx }

```

(End definition for `\prop_put:Nnn` and others. These functions are documented on page 199.)

`\prop_put_if_new:Nnn`
`\prop_put_if_new:cnn`
`\prop_gput_if_new:Nnn`
`\prop_gput_if_new:cnn`
`__prop_put_if_new:NNnn`

Adding conditionally also splits. If the key is already present, the three brace groups given by `__prop_split:NnTF` are removed. If the key is new, then the value is added, being careful to convert the key to a string using `\tl_to_str:n`.

```

19788 \cs_new_protected:Npn \prop_put_if_new:Nnn
19789 { \__prop_put_if_new:NNnn \__kernel_tl_set:Nx }
19790 \cs_new_protected:Npn \prop_gput_if_new:Nnn
19791 { \__prop_put_if_new:NNnn \__kernel_tl_gset:Nx }
19792 \cs_new_protected:Npn \__prop_put_if_new:NNnn #1#2#3#4
19793 {
19794   \tl_set:Nn \l__prop_internal_tl
19795   {
19796     \exp_not:N \__prop_pair:wn \tl_to_str:n {#3}
19797     \s__prop \exp_not:n { {#4} }
19798   }
19799   \__prop_split:NnTF #2 {#3}
19800   { }
19801   { #1 #2 { \exp_not:o {#2} \l__prop_internal_tl } }
19802 }
19803 \cs_generate_variant:Nn \prop_put_if_new:Nnn { c }
19804 \cs_generate_variant:Nn \prop_gput_if_new:Nnn { c }

```

(End definition for `\prop_put_if_new:Nnn`, `\prop_gput_if_new:Nnn`, and `__prop_put_if_new:NNnn`. These functions are documented on page 200.)

61.4 Property list conditionals

`\prop_if_exist_p:N`
`\prop_if_exist_p:c`
`\prop_if_exist:NTF`
`\prop_if_exist:cTF`

Copies of the `cs` functions defined in `l3basics`.

```

19805 \prg_new_eq_conditional:NNn \prop_if_exist:N \cs_if_exist:N
19806 { TF , T , F , p }
19807 \prg_new_eq_conditional:NNn \prop_if_exist:c \cs_if_exist:c
19808 { TF , T , F , p }

```

(End definition for `\prop_if_exist:N`. This function is documented on page 201.)

`\prop_if_empty_p:N`
`\prop_if_empty_p:c`
`\prop_if_empty:NTF`
`\prop_if_empty:cTF`

Same test as for token lists.

```

19809 \prg_new_conditional:Npnn \prop_if_empty:N #1 { p , T , F , TF }
19810 {
19811   \tl_if_eq:NNTF #1 \c_empty_prop
19812   \prg_return_true: \prg_return_false:
19813 }
19814 \prg_generate_conditional_variant:Nnn \prop_if_empty:N
19815 { c } { p , T , F , TF }

```


(End definition for `\prop_if_empty:NTF`. This function is documented on page 202.)

`\prop_if_in_p:Nn` Testing expandably if a key is in a property list requires to go through the key–value pairs one by one. This is rather slow, and a faster test would be

```

\prop_if_in_p:NV
\prop_if_in_p:No
\prop_if_in_p:cn
\prop_if_in_p:cV
\prop_if_in_p:co
\prop_if_in:NnTF
\prop_if_in:NVTF
\prop_if_in:NoTF
\prop_if_in:cnTF
\prop_if_in:cVTF
\prop_if_in:coTF
__prop_if_in:nnn

```

but `__prop_split:NnTF` is non-expandable. Instead, we use `\prop_map_tokens:Nn` to compare the search key to each key in turn using `\str_if_eq:ee`, which is expandable.

```

19816 \prg_new_conditional:Npnn \prop_if_in:Nn #1#2 { p , T , F , TF }
19817 {
19818     \exp_args:NNo \prop_map_tokens:Nn #1
19819     { \exp_after:wN \__prop_if_in:nnn \exp_after:wN { \tl_to_str:n {#2} } }
19820     \prg_return_false:
19821 }
19822 \cs_new:Npn \__prop_if_in:nnn #1#2#3
19823 {
19824     \str_if_eq:eeT {#1} {#2}
19825     { \prop_map_break:n { \use_i:nn \prg_return_true: } }
19826 }
19827 \prg_generate_conditional_variant:Nnn \prop_if_in:Nn
19828 { NV , No , c , cV , co } { p , T , F , TF }

```

(End definition for `\prop_if_in:NnTF` and `__prop_if_in:nnn`. This function is documented on page 202.)

61.5 Recovering values from property lists with branching

`\prop_get:NnNTF` Getting the value corresponding to a key, keeping track of whether the key was present or not, is implemented as a conditional (with side effects). If the key was absent, the token list is not altered.

```

\prop_get:NVNTF
\prop_get:NoNTF
\prop_get:cnNTF
\prop_get:cVNTF
\prop_get:coNTF
19829 \prg_new_protected_conditional:Npnn \prop_get:NnN #1#2#3 { T , F , TF }
19830 {
19831     \__prop_split:NnTF #1 {#2}
19832     {
19833         \tl_set:Nn #3 {##2}
19834         \prg_return_true:
19835     }
19836     { \prg_return_false: }
19837 }
19838 \prg_generate_conditional_variant:Nnn \prop_get:NnN
19839 { NV , Nv , No , c , cV , cv , co } { T , F , TF }

```

(End definition for `\prop_get:NnNTF`. This function is documented on page 202.)

61.6 Mapping over property lists

\prop_map_function:NN The even-numbered arguments of `__prop_map_function:Nw` are keys, hence have string catcodes, except at the end where they are `\fi: \prop_map_break:.` The `\fi:` ends the
\prop_map_function:Nc `\if_false: #⟨even⟩ \fi:` construction and we jump out of the loop. No need for any
\prop_map_function:cN quark test.
\prop_map_function:cc
__prop_map_function:Nw

```

19840 \cs_new:Npn \prop_map_function:NN #1#2
19841 {
19842   \exp_after:wN \use_i_ii:nnn
19843   \exp_after:wN \__prop_map_function:Nw
19844   \exp_after:wN #2
19845   #1
19846   \__prop_pair:wn \fi: \prop_map_break: \s__prop { }
19847   \__prop_pair:wn \fi: \prop_map_break: \s__prop { }
19848   \__prop_pair:wn \fi: \prop_map_break: \s__prop { }
19849   \__prop_pair:wn \fi: \prop_map_break: \s__prop { }
19850   \prg_break_point:Nn \prop_map_break: { }
19851 }
19852 \cs_new:Npn \__prop_map_function:Nw #1
19853   \__prop_pair:wn #2 \s__prop #3
19854   \__prop_pair:wn #4 \s__prop #5
19855   \__prop_pair:wn #6 \s__prop #7
19856   \__prop_pair:wn #8 \s__prop #9
19857 {
19858   \if_false: #2 \fi: #1 {#2} {#3}
19859   \if_false: #4 \fi: #1 {#4} {#5}
19860   \if_false: #6 \fi: #1 {#6} {#7}
19861   \if_false: #8 \fi: #1 {#8} {#9}
19862   \__prop_map_function:Nw #1
19863 }
19864 \cs_generate_variant:Nn \prop_map_function:NN { Nc , c , cc }

```

(End definition for `\prop_map_function:NN` and `__prop_map_function:Nw`. This function is documented on page 203.)

\prop_map_inline:Nn Mapping in line requires a nesting level counter. Store the current definition of `__prop_pair:wn`, and define it anew. At the end of the loop, revert to the earlier definition. Note that besides pairs of the form `__prop_pair:wn ⟨key⟩ \s__prop {⟨value⟩}`, there are a leading and a trailing tokens, but both are equal to `\scan_stop:`, hence have no effect in such inline mapping. Such `\scan_stop:` could have affected ligatures if they appeared during the mapping.

\prop_map_inline:cn

```

19865 \cs_new_protected:Npn \prop_map_inline:Nn #1#2
19866 {
19867   \cs_gset_eq:cN
19868     { __prop_map_ \int_use:N \g__kernel_prg_map_int :wn } \__prop_pair:wn
19869   \int_gincr:N \g__kernel_prg_map_int
19870   \cs_gset_protected:Npn \__prop_pair:wn ##1 \s__prop ##2 {#2}
19871   #1
19872   \prg_break_point:Nn \prop_map_break:
19873   {
19874     \int_gdecr:N \g__kernel_prg_map_int
19875     \cs_gset_eq:Nc \__prop_pair:wn
19876       { __prop_map_ \int_use:N \g__kernel_prg_map_int :wn }

```

```

19877     }
19878   }
19879   \cs_generate_variant:Nn \prop_map_inline:Nn { c }

```

(End definition for `\prop_map_inline:Nn`. This function is documented on page 203.)

`\prop_map_tokens:Nn` The mapping is very similar to `\prop_map_function:NN`. The `\use_i:nn` removes the
`\prop_map_tokens:cn` leading `\s__prop`. The odd construction `\use:n {#1}` allows #1 to contain any token
`__prop_map_tokens:nw` without interfering with `\prop_map_break:.` The loop stops when the $\langle key \rangle$ between
`__prop_pair:wn` and `\s__prop` is `\fi: \prop_map_break:` instead of being a string.

```

19880 \cs_new:Npn \prop_map_tokens:Nn #1#2
19881 {
19882   \exp_last_unbraced:Nno
19883   \use_i:nn { \__prop_map_tokens:nw {#2} } #1
19884   \__prop_pair:wn \fi: \prop_map_break: \s__prop { }
19885   \__prop_pair:wn \fi: \prop_map_break: \s__prop { }
19886   \__prop_pair:wn \fi: \prop_map_break: \s__prop { }
19887   \__prop_pair:wn \fi: \prop_map_break: \s__prop { }
19888   \prg_break_point:Nn \prop_map_break: { }
19889 }
19890 \cs_new:Npn \__prop_map_tokens:nw #1
19891   \__prop_pair:wn #2 \s__prop #3
19892   \__prop_pair:wn #4 \s__prop #5
19893   \__prop_pair:wn #6 \s__prop #7
19894   \__prop_pair:wn #8 \s__prop #9
19895 {
19896   \if_false: #2 \fi: \use:n {#1} {#2} {#3}
19897   \if_false: #4 \fi: \use:n {#1} {#4} {#5}
19898   \if_false: #6 \fi: \use:n {#1} {#6} {#7}
19899   \if_false: #8 \fi: \use:n {#1} {#8} {#9}
19900   \__prop_map_tokens:nw {#1}
19901 }
19902 \cs_generate_variant:Nn \prop_map_tokens:Nn { c }

```

(End definition for `\prop_map_tokens:Nn` and `__prop_map_tokens:nw`. This function is documented on page 203.)

`\prop_map_break:` The break statements are based on the general `\prg_map_break:Nn`.
`\prop_map_break:n`

```

19903 \cs_new:Npn \prop_map_break:
19904 { \prg_map_break:Nn \prop_map_break: { } }
19905 \cs_new:Npn \prop_map_break:n
19906 { \prg_map_break:Nn \prop_map_break: }

```

(End definition for `\prop_map_break:` and `\prop_map_break:n`. These functions are documented on page 204.)

61.7 Viewing property lists

`\prop_show:N` Apply the general `__kernel_chk_tl_type:NnnT`. Contrarily to sequences and comma
`\prop_show:c` lists, we use `\msg_show_item:nn` to format both the key and the value for each pair.
`\prop_log:N`

```
19907 \cs_new_protected:Npn \prop_show:N { \__prop_show:NN \msg_show:nnxxxx }
```


`\prop_log:c`

```
19908 \cs_generate_variant:Nn \prop_show:N { c }
```


`__prop_show:NN`

```
19909 \cs_new_protected:Npn \prop_log:N { \__prop_show:NN \msg_log:nnxxxx }
```


`__prop_show_validate:w`

```

19910 \cs_generate_variant:Nn \prop_log:N { c }
19911 \cs_new_protected:Npn \__prop_show:NN #1#2
19912 {
19913   \__kernel_chk_tl_type:NnnT #2 { prop }
19914   {
19915     \s__prop
19916     \exp_after:wN \use_i:nn \exp_after:wN \__prop_show_validate:w #2
19917     \__prop_pair:wn \q_recursion_tail \s__prop { } \q_recursion_stop
19918   }
19919   {
19920     #1 { prop } { show }
19921     { \token_to_str:N #2 }
19922     { \prop_map_function:NN #2 \msg_show_item:nn }
19923     { } { }
19924   }
19925 }
19926 \cs_new:Npn \__prop_show_validate:w #1 \__prop_pair:wn #2 \s__prop #3
19927 {
19928   \quark_if_recursion_tail_stop:n {#2}
19929   \exp_not:N \__prop_pair:wn \tl_to_str:n {#2} \s__prop \exp_not:n { {#3} }
19930   \__prop_show_validate:w
19931 }

```

(End definition for `\prop_show:N` and others. These functions are documented on page 204.)

```

19932 \endpackage

```

Chapter 62

l3skip implementation

```
19933 <*package>
19934 <@@=dim>
```

62.1 Length primitives renamed

```
\if_dim:w Primitives renamed.
  \__dim_eval:w 19935 \cs_new_eq:NN \if_dim:w      \tex_ifdim:D
  \__dim_eval_end: 19936 \cs_new_eq:NN \__dim_eval:w    \tex_dimexpr:D
                  19937 \cs_new_eq:NN \__dim_eval_end:  \tex_relax:D
```

(End definition for `\if_dim:w`, `__dim_eval:w`, and `__dim_eval_end:`. This function is documented on page 220.)

62.2 Internal auxiliaries

```
\s__dim_mark Internal scan marks.
\s__dim_stop 19938 \scan_new:N \s__dim_mark
              19939 \scan_new:N \s__dim_stop
```

(End definition for `\s__dim_mark` and `\s__dim_stop`.)

```
\_dim_use_none_delimit_by_s_stop:w Functions to gobble up to a scan mark.
19940 \cs_new:Npn \_dim_use_none_delimit_by_s_stop:w #1 \s__dim_stop { }
```

(End definition for `_dim_use_none_delimit_by_s_stop:w`.)

62.3 Creating and initialising dim variables

```
\dim_new:N Allocating  $\langle dim \rangle$  registers ...
\dim_new:c 19941 \cs_new_protected:Npn \dim_new:N #1
              19942 {
              19943   \__kernel_chk_if_free_cs:N #1
              19944   \cs:w newdimen \cs_end: #1
              19945   }
              19946 \cs_generate_variant:Nn \dim_new:N { c }
```

(End definition for `\dim_new:N`. This function is documented on page 206.)

`\dim_const:Nn` Contrarily to integer constants, we cannot avoid using a register, even for constants. We cannot use `\dim_gset:Nn` because debugging code would complain that the constant is not a global variable. Since `\dim_const:Nn` does not need to be fast, use `\dim_eval:n` to avoid needing a debugging patch that wraps the expression in checking code.

```
19947 \cs_new_protected:Npn \dim_const:Nn #1#2
19948 {
19949   \dim_new:N #1
19950   \tex_global:D #1 = \dim_eval:n {#2} \scan_stop:
19951 }
19952 \cs_generate_variant:Nn \dim_const:Nn { c }
```

(End definition for `\dim_const:Nn`. This function is documented on page 206.)

`\dim_zero:N` Reset the register to zero. Using `\c_zero_skip` deals with the case where the variable passed is incorrectly a skip (for example a $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X} 2_{\epsilon}$ length). Besides, these functions are then simply copied for `\skip_zero:N` and related functions.

```
\dim_zero:c
\dim_gzero:N
\dim_gzero:c
19953 \cs_new_protected:Npn \dim_zero:N #1 { #1 = \c_zero_skip }
19954 \cs_new_protected:Npn \dim_gzero:N #1
19955 { \tex_global:D #1 = \c_zero_skip }
19956 \cs_generate_variant:Nn \dim_zero:N { c }
19957 \cs_generate_variant:Nn \dim_gzero:N { c }
```

(End definition for `\dim_zero:N` and `\dim_gzero:N`. These functions are documented on page 206.)

`\dim_zero_new:N` Create a register if needed, otherwise clear it.

```
\dim_zero_new:c
\dim_gzero_new:N
\dim_gzero_new:c
19958 \cs_new_protected:Npn \dim_zero_new:N #1
19959 { \dim_if_exist:NTF #1 { \dim_zero:N #1 } { \dim_new:N #1 } }
19960 \cs_new_protected:Npn \dim_gzero_new:N #1
19961 { \dim_if_exist:NTF #1 { \dim_gzero:N #1 } { \dim_new:N #1 } }
19962 \cs_generate_variant:Nn \dim_zero_new:N { c }
19963 \cs_generate_variant:Nn \dim_gzero_new:N { c }
```

(End definition for `\dim_zero_new:N` and `\dim_gzero_new:N`. These functions are documented on page 206.)

`\dim_if_exist_p:N` Copies of the cs functions defined in `l3basics`.

```
\dim_if_exist_p:c
\dim_if_exist:NTF
\dim_if_exist:cTF
19964 \prg_new_eq_conditional:NNn \dim_if_exist:N \cs_if_exist:N
19965 { TF , T , F , p }
19966 \prg_new_eq_conditional:NNn \dim_if_exist:c \cs_if_exist:c
19967 { TF , T , F , p }
```

(End definition for `\dim_if_exist:NTF`. This function is documented on page 207.)

62.4 Setting dim variables

`\dim_set:Nn` Setting dimensions is easy enough but when debugging we want both to check that the variable is correctly local/global and to wrap the expression in some code. The `\scan_stop:` deals with the case where the variable passed is a skip (for example a $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X} 2_{\epsilon}$ length).

```
19968 \cs_new_protected:Npn \dim_set:Nn #1#2
19969 { #1 = \__dim_eval:w #2 \__dim_eval_end: \scan_stop: }
```

```

19970 \cs_new_protected:Npn \dim_gset:Nn #1#2
19971 { \tex_global:D #1 = \__dim_eval:w #2 \__dim_eval_end: \scan_stop: }
19972 \cs_generate_variant:Nn \dim_set:Nn { c }
19973 \cs_generate_variant:Nn \dim_gset:Nn { c }

```

(End definition for `\dim_set:Nn` and `\dim_gset:Nn`. These functions are documented on page 207.)

`\dim_set_eq:NN` All straightforward, with a `\scan_stop:` to deal with the case where #1 is (incorrectly) a skip.

```

\dim_set_eq:cn
\dim_set_eq:Nc
\dim_set_eq:cc
19974 \cs_new_protected:Npn \dim_set_eq:NN #1#2
19975 { #1 = #2 \scan_stop: }
\dim_gset_eq:NN
19976 \cs_generate_variant:Nn \dim_set_eq:NN { c , Nc , cc }
\dim_gset_eq:cn
19977 \cs_new_protected:Npn \dim_gset_eq:NN #1#2
\dim_gset_eq:Nc
19978 { \tex_global:D #1 = #2 \scan_stop: }
\dim_gset_eq:cc
19979 \cs_generate_variant:Nn \dim_gset_eq:NN { c , Nc , cc }

```

(End definition for `\dim_set_eq:NN` and `\dim_gset_eq:NN`. These functions are documented on page 207.)

`\dim_add:Nn` Using by here would slow things down just to detect nonsensical cases such as passing `\dimen 123` as the first argument. Using `\scan_stop:` deals with skip variables. Since debugging checks that the variable is correctly local/global, the global versions cannot be defined as `\tex_global:D` followed by the local versions.

```

\dim_add:cn
\dim_gadd:Nn
\dim_gadd:cn
\dim_sub:Nn
19980 \cs_new_protected:Npn \dim_add:Nn #1#2
\dim_sub:cn
19981 { \tex_advance:D #1 \__dim_eval:w #2 \__dim_eval_end: \scan_stop: }
\dim_gsub:Nn
19982 \cs_new_protected:Npn \dim_gadd:Nn #1#2
\dim_gsub:cn
19983 {
19984   \tex_global:D \tex_advance:D #1
19985   \__dim_eval:w #2 \__dim_eval_end: \scan_stop:
19986 }
19987 \cs_generate_variant:Nn \dim_add:Nn { c }
19988 \cs_generate_variant:Nn \dim_gadd:Nn { c }
19989 \cs_new_protected:Npn \dim_sub:Nn #1#2
19990 { \tex_advance:D #1 - \__dim_eval:w #2 \__dim_eval_end: \scan_stop: }
19991 \cs_new_protected:Npn \dim_gsub:Nn #1#2
19992 {
19993   \tex_global:D \tex_advance:D #1
19994   -\__dim_eval:w #2 \__dim_eval_end: \scan_stop:
19995 }
19996 \cs_generate_variant:Nn \dim_sub:Nn { c }
19997 \cs_generate_variant:Nn \dim_gsub:Nn { c }

```

(End definition for `\dim_add:Nn` and others. These functions are documented on page 207.)

62.5 Utilities for dimension calculations

`\dim_abs:n` Functions for min, max, and absolute value with only one evaluation. The absolute value is evaluated by removing a leading - if present.

```

\__dim_abs:N
\dim_max:nn
\dim_min:nn
\__dim_maxmin:wwN
19998 \cs_new:Npn \dim_abs:n #1
19999 {
20000   \exp_after:wN \__dim_abs:N
20001   \dim_use:N \__dim_eval:w #1 \__dim_eval_end:
20002 }

```

```

20003 \cs_new:Npn \__dim_abs:N #1
20004 { \if_meaning:w - #1 \else: \exp_after:wN #1 \fi: }
20005 \cs_new:Npn \dim_max:nn #1#2
20006 {
20007   \dim_use:N \__dim_eval:w \exp_after:wN \__dim_maxmin:wwN
20008   \dim_use:N \__dim_eval:w #1 \exp_after:wN ;
20009   \dim_use:N \__dim_eval:w #2 ;
20010   >
20011   \__dim_eval_end:
20012 }
20013 \cs_new:Npn \dim_min:nn #1#2
20014 {
20015   \dim_use:N \__dim_eval:w \exp_after:wN \__dim_maxmin:wwN
20016   \dim_use:N \__dim_eval:w #1 \exp_after:wN ;
20017   \dim_use:N \__dim_eval:w #2 ;
20018   <
20019   \__dim_eval_end:
20020 }
20021 \cs_new:Npn \__dim_maxmin:wwN #1 ; #2 ; #3
20022 {
20023   \if_dim:w #1 #3 #2 ~
20024   #1
20025   \else:
20026   #2
20027   \fi:
20028 }

```

(End definition for `\dim_abs:n` and others. These functions are documented on page 207.)

`\dim_ratio:nn` With dimension expressions, something like `10 pt * (5 pt / 10 pt)` does not work. Instead, the ratio part needs to be converted to an integer expression. Using `\int_value:w` forces everything into `sp`, avoiding any decimal parts.

```

20029 \cs_new:Npn \dim_ratio:nn #1#2
20030 { \__dim_ratio:n {#1} / \__dim_ratio:n {#2} }
20031 \cs_new:Npn \__dim_ratio:n #1
20032 { \int_value:w \__dim_eval:w (#1) \__dim_eval_end: }

```

(End definition for `\dim_ratio:nn` and `__dim_ratio:n`. This function is documented on page 208.)

62.6 Dimension expression conditionals

`\dim_compare_p:nNn` Simple comparison.

`\dim_compare:nNnTF`

```

20033 \prg_new_conditional:Npnn \dim_compare:nNn #1#2#3 { p , T , F , TF }
20034 {
20035   \if_dim:w \__dim_eval:w #1 #2 \__dim_eval:w #3 \__dim_eval_end:
20036   \prg_return_true: \else: \prg_return_false: \fi:
20037 }

```

(End definition for `\dim_compare:nNnTF`. This function is documented on page 208.)

`\dim_compare_p:n` This code is adapted from the `\int_compare:nTF` function. First make sure that there is at least one relation operator, by evaluating a dimension expression with a trailing `__dim_compare_error:.` Just like for integers, the looping auxiliary `__dim_compare:wNN` closes a primitive conditional and opens a new one. It is actually easier to

`\dim_compare:nTF`

```

\__dim_compare:w
\__dim_compare:wNN
\__dim_compare:=:w
\__dim_compare!:w
\__dim_compare<:w
\__dim_compare>:w
\__dim_compare_error:

```


grab a dimension operand than an integer one, because once evaluated, dimensions all end with `pt` (with category `other`). Thus we do not need specific auxiliaries for the three “simple” relations `<`, `=`, and `>`.

```

20038 \prg_new_conditional:Npnn \dim_compare:n #1 { p , T , F , TF }
20039 {
20040   \exp_after:wN \__dim_compare:w
20041   \dim_use:N \__dim_eval:w #1 \__dim_compare_error:
20042 }
20043 \cs_new:Npn \__dim_compare:w #1 \__dim_compare_error:
20044 {
20045   \exp_after:wN \if_false: \exp:w \exp_end_continue_f:w
20046   \__dim_compare:wNN #1 ? { = \__dim_compare_end:w \else: } \s__dim_stop
20047 }
20048 \exp_args:Nno \use:nn
20049 { \cs_new:Npn \__dim_compare:wNN #1 } { \tl_to_str:n {pt} #2#3 }
20050 {
20051   \if_meaning:w = #3
20052   \use:c { __dim_compare_#2:w }
20053   \fi:
20054   #1 pt \exp_stop_f:
20055   \prg_return_false:
20056   \exp_after:wN \__dim_use_none_delimit_by_s_stop:w
20057   \fi:
20058   \reverse_if:N \if_dim:w #1 pt #2
20059   \exp_after:wN \__dim_compare:wNN
20060   \dim_use:N \__dim_eval:w #3
20061 }
20062 \cs_new:cpn { __dim_compare_ ! :w }
20063   #1 \reverse_if:N #2 ! #3 = { #1 #2 = #3 }
20064 \cs_new:cpn { __dim_compare_ = :w }
20065   #1 \__dim_eval:w = { #1 \__dim_eval:w }
20066 \cs_new:cpn { __dim_compare_ < :w }
20067   #1 \reverse_if:N #2 < #3 = { #1 #2 > #3 }
20068 \cs_new:cpn { __dim_compare_ > :w }
20069   #1 \reverse_if:N #2 > #3 = { #1 #2 < #3 }
20070 \cs_new:Npn \__dim_compare_end:w #1 \prg_return_false: #2 \s__dim_stop
20071 { #1 \prg_return_false: \else: \prg_return_true: \fi: }
20072 \cs_new_protected:Npn \__dim_compare_error:
20073 {
20074   \if_int_compare:w \c_zero_int \c_zero_int \fi:
20075   =
20076   \__dim_compare_error:
20077 }

```

(End definition for `\dim_compare:nTF` and others. This function is documented on page 209.)

<pre> \dim_case:nn \dim_case:nnTF __dim_case:nnTF __dim_case:nw __dim_case_end:nw </pre>	<p>For dimension cases, the first task to fully expand the check condition. The over all idea is then much the same as for <code>\str_case:nnTF</code> as described in l3basics.</p> <pre> 20078 \cs_new:Npn \dim_case:nnTF #1 20079 { 20080 \exp:w 20081 \exp_args:Nf __dim_case:nnTF { \dim_eval:n {#1} } 20082 } 20083 \cs_new:Npn \dim_case:nnT #1#2#3 </pre>
---	--

```

20084 {
20085   \exp:w
20086   \exp_args:Nf \__dim_case:nnTF { \dim_eval:n {#1} } {#2} {#3} { }
20087 }
20088 \cs_new:Npn \dim_case:nnF #1#2
20089 {
20090   \exp:w
20091   \exp_args:Nf \__dim_case:nnTF { \dim_eval:n {#1} } {#2} { }
20092 }
20093 \cs_new:Npn \dim_case:nn #1#2
20094 {
20095   \exp:w
20096   \exp_args:Nf \__dim_case:nnTF { \dim_eval:n {#1} } {#2} { } { }
20097 }
20098 \cs_new:Npn \__dim_case:nnTF #1#2#3#4
20099 { \__dim_case:nw {#1} #2 {#1} { } \s__dim_mark {#3} \s__dim_mark {#4} \s__dim_stop }
20100 \cs_new:Npn \__dim_case:nw #1#2#3
20101 {
20102   \dim_compare:nNnTF {#1} = {#2}
20103   { \__dim_case_end:nw {#3} }
20104   { \__dim_case:nw {#1} }
20105 }
20106 \cs_new:Npn \__dim_case_end:nw #1#2#3 \s__dim_mark #4#5 \s__dim_stop
20107 { \exp_end: #1 #4 }

```

(End definition for `\dim_case:nnTF` and others. This function is documented on page [210](#).)

62.7 Dimension expression loops

`\dim_while_do:nn` `\dim_until_do:nn` `\dim_do_while:nn` `\dim_do_until:nn` `while_do` and `do_while` functions for dimensions. Same as for the `int` type only the names have changed.

```

20108 \cs_new:Npn \dim_while_do:nn #1#2
20109 {
20110   \dim_compare:nT {#1}
20111   {
20112     #2
20113     \dim_while_do:nn {#1} {#2}
20114   }
20115 }
20116 \cs_new:Npn \dim_until_do:nn #1#2
20117 {
20118   \dim_compare:nF {#1}
20119   {
20120     #2
20121     \dim_until_do:nn {#1} {#2}
20122   }
20123 }
20124 \cs_new:Npn \dim_do_while:nn #1#2
20125 {
20126   #2
20127   \dim_compare:nT {#1}
20128   { \dim_do_while:nn {#1} {#2} }
20129 }

```

```

20130 \cs_new:Npn \dim_do_until:nn #1#2
20131 {
20132     #2
20133     \dim_compare:nF {#1}
20134     { \dim_do_until:nn {#1} {#2} }
20135 }

```

(End definition for `\dim_while_do:nn` and others. These functions are documented on page 211.)

`\dim_while_do:nNnn` `\dim_while_do` and `do_while` functions for dimensions. Same as for the `int` type only the names have changed.

```

\dim_until_do:nNnn
\dim_do_while:nNnn
\dim_do_until:nNnn
20136 \cs_new:Npn \dim_while_do:nNnn #1#2#3#4
20137 {
20138     \dim_compare:nNtT {#1} #2 {#3}
20139     {
20140         #4
20141         \dim_while_do:nNnn {#1} #2 {#3} {#4}
20142     }
20143 }
20144 \cs_new:Npn \dim_until_do:nNnn #1#2#3#4
20145 {
20146     \dim_compare:nNf {#1} #2 {#3}
20147     {
20148         #4
20149         \dim_until_do:nNnn {#1} #2 {#3} {#4}
20150     }
20151 }
20152 \cs_new:Npn \dim_do_while:nNnn #1#2#3#4
20153 {
20154     #4
20155     \dim_compare:nNtT {#1} #2 {#3}
20156     { \dim_do_while:nNnn {#1} #2 {#3} {#4} }
20157 }
20158 \cs_new:Npn \dim_do_until:nNnn #1#2#3#4
20159 {
20160     #4
20161     \dim_compare:nNf {#1} #2 {#3}
20162     { \dim_do_until:nNnn {#1} #2 {#3} {#4} }
20163 }

```

(End definition for `\dim_while_do:nNnn` and others. These functions are documented on page 211.)

62.8 Dimension step functions

`\dim_step_function:nnnN` Before all else, evaluate the initial value, step, and final value. Repeating a function by steps first needs a check on the direction of the steps. After that, do the function for the start value then step and loop around. It would be more symmetrical to test for a step size of zero before checking the sign, but we optimize for the most frequent case (positive step).

```

20164 \cs_new:Npn \dim_step_function:nnnN #1#2#3
20165 {
20166     \exp_after:wN \_dim_step:wwwN
20167     \tex_the:D \_dim_eval:w #1 \exp_after:wN ;

```

```

20168 \tex_the:D \__dim_eval:w #2 \exp_after:wN ;
20169 \tex_the:D \__dim_eval:w #3 ;
20170 }
20171 \cs_new:Npn \__dim_step:wwwN #1; #2; #3; #4
20172 {
20173 \dim_compare:nNnTF {#2} > \c_zero_dim
20174 { \__dim_step:NnnnN > }
20175 {
20176 \dim_compare:nNnTF {#2} = \c_zero_dim
20177 {
20178 \msg_expandable_error:nnn { kernel } { zero-step } {#4}
20179 \use_none:nnnn
20180 }
20181 { \__dim_step:NnnnN < }
20182 }
20183 {#1} {#2} {#3} #4
20184 }
20185 \cs_new:Npn \__dim_step:NnnnN #1#2#3#4#5
20186 {
20187 \dim_compare:nNnF {#2} #1 {#4}
20188 {
20189 #5 {#2}
20190 \exp_args:NNf \__dim_step:NnnnN
20191 #1 { \dim_eval:n { #2 + #3 } } {#3} {#4} #5
20192 }
20193 }

```

(End definition for `\dim_step_function:nnnN`, `__dim_step:wwwN`, and `__dim_step:NnnnN`. This function is documented on page 211.)

`\dim_step_inline:nnnn`
`\dim_step_variable:nnnNn`
`__dim_step:NNnnnn`

The approach here is to build a function, with a global integer required to make the nesting safe (as seen in other in line functions), and map that function using `\dim_step_function:nnnN`. We put a `\prg_break_point:Nn` so that `map_break` functions from other modules correctly decrement `\g__kernel_prg_map_int` before looking for their own break point. The first argument is `\scan_stop:`, so that no breaking function recognizes this break point as its own.

```

20194 \cs_new_protected:Npn \dim_step_inline:nnnn
20195 {
20196 \int_gincr:N \g__kernel_prg_map_int
20197 \exp_args:NNc \__dim_step:NNnnnn
20198 \cs_gset_protected:Npn
20199 { __dim_map_ \int_use:N \g__kernel_prg_map_int :w }
20200 }
20201 \cs_new_protected:Npn \dim_step_variable:nnnNn #1#2#3#4#5
20202 {
20203 \int_gincr:N \g__kernel_prg_map_int
20204 \exp_args:NNc \__dim_step:NNnnnn
20205 \cs_gset_protected:Npx
20206 { __dim_map_ \int_use:N \g__kernel_prg_map_int :w }
20207 {#1}{#2}{#3}
20208 {
20209 \tl_set:Nn \exp_not:N #4 {##1}
20210 \exp_not:n {#5}
20211 }

```

```

20212 }
20213 \cs_new_protected:Npn \__dim_step:NNnnnn #1#2#3#4#5#6
20214 {
20215   #1 #2 ##1 {#6}
20216   \dim_step_function:nnnN {#3} {#4} {#5} #2
20217   \prg_break_point:Nn \scan_stop: { \int_gdecr:N \g__kernel_prg_map_int }
20218 }

```

(End definition for `\dim_step_inline:nnnn`, `\dim_step_variable:nnnNn`, and `__dim_step:NNnnnn`. These functions are documented on page 211.)

62.9 Using dim expressions and variables

`\dim_eval:n` Evaluating a dimension expression expandably.

```

20219 \cs_new:Npn \dim_eval:n #1
20220 { \dim_use:N \__dim_eval:w #1 \__dim_eval_end: }

```

(End definition for `\dim_eval:n`. This function is documented on page 212.)

`\dim_sign:n` See `\dim_abs:n`. Contrarily to `\int_sign:n` the case of a zero dimension cannot be distinguished from a positive dimension by looking only at the first character, since `0.2pt` and `0pt` start the same way. We need explicit comparisons. We start by distinguishing the most common case of a positive dimension.

`__dim_sign:Nw`

```

20221 \cs_new:Npn \dim_sign:n #1
20222 {
20223   \int_value:w \exp_after:wN \__dim_sign:Nw
20224   \dim_use:N \__dim_eval:w #1 \__dim_eval_end: ;
20225   \exp_stop_f:
20226 }
20227 \cs_new:Npn \__dim_sign:Nw #1#2 ;
20228 {
20229   \if_dim:w #1#2 > \c_zero_dim
20230     1
20231   \else:
20232     \if_meaning:w - #1
20233       -1
20234     \else:
20235       0
20236     \fi:
20237   \fi:
20238 }

```

(End definition for `\dim_sign:n` and `__dim_sign:Nw`. This function is documented on page 212.)

`\dim_use:N` Accessing a $\langle dim \rangle$. We hand-code the c variant for some speed gain.

```

\dim_use:c
20239 \cs_new_eq:NN \dim_use:N \tex_the:D
20240 \cs_new:Npn \dim_use:c #1 { \tex_the:D \cs:w #1 \cs_end: }

```

(End definition for `\dim_use:N`. This function is documented on page 212.)

`\dim_to_decimal:n` A function which comes up often enough to deserve a place in the kernel. Evaluate the dimension expression `#1` then remove the trailing `pt`. When debugging is enabled, the argument is put in parentheses as this prevents the dimension expression from terminating early and leaving extra tokens lying around. This is used a lot by low-level manipulations.

`__dim_to_decimal:w`

```

20241 \cs_new:Npn \dim_to_decimal:n #1
20242 {
20243   \exp_after:wN
20244   \__dim_to_decimal:w \dim_use:N \__dim_eval:w #1 \__dim_eval_end:
20245 }
20246 \use:x
20247 {
20248   \cs_new:Npn \exp_not:N \__dim_to_decimal:w
20249   ##1 . ##2 \tl_to_str:n { pt }
20250 }
20251 {
20252   \int_compare:nNnTF {#2} > \c_zero_int
20253   { #1 . #2 }
20254   { #1 }
20255 }

```

(End definition for `\dim_to_decimal:n` and `__dim_to_decimal:w`. This function is documented on page [212](#).)

`\dim_to_decimal_in_bp:n` Conversion to big points is done using a scaling inside `__dim_eval:w` as ϵ -TeX does that using 64-bit precision. Here, 800/803 is the integer fraction for 72/72.27. This is a common case so is hand-coded for accuracy (and speed).

```

20256 \cs_new:Npn \dim_to_decimal_in_bp:n #1
20257 { \dim_to_decimal:n { ( #1 ) * 800 / 803 } }

```

(End definition for `\dim_to_decimal_in_bp:n`. This function is documented on page [213](#).)

`\dim_to_decimal_in_sp:n` Another hard-coded conversion: this one is necessary to avoid things going off-scale.

```

20258 \cs_new:Npn \dim_to_decimal_in_sp:n #1
20259 { \int_value:w \__dim_eval:w #1 \__dim_eval_end: }

```

(End definition for `\dim_to_decimal_in_sp:n`. This function is documented on page [213](#).)

`\dim_to_decimal_in_unit:nn` An analogue of `\dim_ratio:nn` that produces a decimal number as its result, rather than a rational fraction for use within dimension expressions.

```

20260 \cs_new:Npn \dim_to_decimal_in_unit:nn #1#2
20261 {
20262   \dim_to_decimal:n
20263   {
20264     1pt *
20265     \dim_ratio:nn {#1} {#2}
20266   }
20267 }

```

(End definition for `\dim_to_decimal_in_unit:nn`. This function is documented on page [213](#).)

`\dim_to_fp:n` Defined in `l3fp-convert`, documented [here](#).

(End definition for `\dim_to_fp:n`. This function is documented on page [213](#).)

62.10 Viewing dim variables

`\dim_show:N` Diagnostics.

`\dim_show:c`

```
20268 \cs_new_eq:NN \dim_show:N \__kernel_register_show:N
20269 \cs_generate_variant:Nn \dim_show:N { c }
```

(End definition for `\dim_show:N`. This function is documented on page 213.)

`\dim_show:n` Diagnostics. We don't use the `\showthe` primitive to show dimension expressions: this gives a more unified output.

```
20270 \cs_new_protected:Npn \dim_show:n
20271 { \msg_show_eval:Nn \dim_eval:n }
```

(End definition for `\dim_show:n`. This function is documented on page 214.)

`\dim_log:N` Diagnostics. Redirect output of `\dim_show:n` to the log.

`\dim_log:c`

`\dim_log:n`

```
20272 \cs_new_eq:NN \dim_log:N \__kernel_register_log:N
20273 \cs_new_eq:NN \dim_log:c \__kernel_register_log:c
20274 \cs_new_protected:Npn \dim_log:n
20275 { \msg_log_eval:Nn \dim_eval:n }
```

(End definition for `\dim_log:N` and `\dim_log:n`. These functions are documented on page 214.)

62.11 Constant dimensions

`\c_zero_dim` Constant dimensions.

`\c_max_dim`

```
20276 \dim_const:Nn \c_zero_dim { 0 pt }
20277 \dim_const:Nn \c_max_dim { 16383.99999 pt }
```

(End definition for `\c_zero_dim` and `\c_max_dim`. These variables are documented on page 214.)

62.12 Scratch dimensions

`\l_tmpa_dim` We provide two local and two global scratch registers, maybe we need more or less.

`\l_tmpb_dim`

`\g_tmpa_dim`

`\g_tmpb_dim`

```
20278 \dim_new:N \l_tmpa_dim
20279 \dim_new:N \l_tmpb_dim
20280 \dim_new:N \g_tmpa_dim
20281 \dim_new:N \g_tmpb_dim
```

(End definition for `\l_tmpa_dim` and others. These variables are documented on page 214.)

62.13 Creating and initialising skip variables

```
20282 <@@=skip>
```

`\s__skip_stop` Internal scan marks.

```
20283 \scan_new:N \s__skip_stop
```

(End definition for `\s__skip_stop`.)

\skip_new:N Allocation of a new internal registers.

```
\skip_new:c
20284 \cs_new_protected:Npn \skip_new:N #1
20285 {
20286   \__kernel_chk_if_free_cs:N #1
20287   \cs:w newskip \cs_end: #1
20288 }
20289 \cs_generate_variant:Nn \skip_new:N { c }
```

(End definition for \skip_new:N. This function is documented on page 214.)

\skip_const:Nn Contrarily to integer constants, we cannot avoid using a register, even for constants. See **\dim_const:Nn** for why we cannot use **\skip_gset:Nn**.

```
\skip_const:cn
20290 \cs_new_protected:Npn \skip_const:Nn #1#2
20291 {
20292   \skip_new:N #1
20293   \tex_global:D #1 = \skip_eval:n {#2} \scan_stop:
20294 }
20295 \cs_generate_variant:Nn \skip_const:Nn { c }
```

(End definition for \skip_const:Nn. This function is documented on page 215.)

\skip_zero:N Reset the register to zero.

```
\skip_zero:c
20296 \cs_new_eq:NN \skip_zero:N \dim_zero:N
\skip_gzero:N
20297 \cs_new_eq:NN \skip_gzero:N \dim_gzero:N
\skip_gzero:c
20298 \cs_generate_variant:Nn \skip_zero:N { c }
20299 \cs_generate_variant:Nn \skip_gzero:N { c }
```

(End definition for \skip_zero:N and \skip_gzero:N. These functions are documented on page 215.)

\skip_zero_new:N Create a register if needed, otherwise clear it.

```
\skip_zero_new:c
20300 \cs_new_protected:Npn \skip_zero_new:N #1
\skip_gzero_new:N
20301 { \skip_if_exist:NTF #1 { \skip_zero:N #1 } { \skip_new:N #1 } }
\skip_gzero_new:c
20302 \cs_new_protected:Npn \skip_gzero_new:N #1
20303 { \skip_if_exist:NTF #1 { \skip_gzero:N #1 } { \skip_new:N #1 } }
20304 \cs_generate_variant:Nn \skip_zero_new:N { c }
20305 \cs_generate_variant:Nn \skip_gzero_new:N { c }
```

(End definition for \skip_zero_new:N and \skip_gzero_new:N. These functions are documented on page 215.)

\skip_if_exist_p:N Copies of the cs functions defined in l3basics.

```
\skip_if_exist_p:c
20306 \prg_new_eq_conditional:NNn \skip_if_exist:N \cs_if_exist:N
\skip_if_exist:NTF
20307 { TF , T , F , p }
\skip_if_exist:cTF
20308 \prg_new_eq_conditional:NNn \skip_if_exist:c \cs_if_exist:c
20309 { TF , T , F , p }
```

(End definition for \skip_if_exist:N~~TF~~. This function is documented on page 215.)

62.14 Setting skip variables

`\skip_set:Nn` Much the same as for dimensions.

```
\skip_set:cn 20310 \cs_new_protected:Npn \skip_set:Nn #1#2
\skip_gset:Nn 20311 { #1 = \tex_glueexpr:D #2 \scan_stop: }
\skip_gset:cn 20312 \cs_new_protected:Npn \skip_gset:Nn #1#2
20313 { \tex_global:D #1 = \tex_glueexpr:D #2 \scan_stop: }
20314 \cs_generate_variant:Nn \skip_set:Nn { c }
20315 \cs_generate_variant:Nn \skip_gset:Nn { c }
```

(End definition for `\skip_set:Nn` and `\skip_gset:Nn`. These functions are documented on page 215.)

`\skip_set_eq:NN` All straightforward.

```
\skip_set_eq:cn 20316 \cs_new_protected:Npn \skip_set_eq:NN #1#2 { #1 = #2 }
\skip_set_eq:Nc 20317 \cs_generate_variant:Nn \skip_set_eq:NN { c , Nc , cc }
\skip_set_eq:cc 20318 \cs_new_protected:Npn \skip_gset_eq:NN #1#2 { \tex_global:D #1 = #2 }
\skip_gset_eq:cn 20319 \cs_generate_variant:Nn \skip_gset_eq:NN { c , Nc , cc }
```

(End definition for `\skip_set_eq:NN` and `\skip_gset_eq:NN`. These functions are documented on page 215.)

`\skip_add:Nn` Using by here deals with the (incorrect) case `\skip123`.

```
\skip_add:cn 20320 \cs_new_protected:Npn \skip_add:Nn #1#2
\skip_gadd:Nn 20321 { \tex_advance:D #1 \tex_glueexpr:D #2 \scan_stop: }
\skip_gadd:cn 20322 \cs_new_protected:Npn \skip_gadd:Nn #1#2
\skip_sub:Nn 20323 { \tex_global:D \tex_advance:D #1 \tex_glueexpr:D #2 \scan_stop: }
\skip_sub:cn 20324 \cs_generate_variant:Nn \skip_add:Nn { c }
\skip_gsub:Nn 20325 \cs_generate_variant:Nn \skip_gadd:Nn { c }
\skip_gsub:cn 20326 \cs_new_protected:Npn \skip_sub:Nn #1#2
20327 { \tex_advance:D #1 - \tex_glueexpr:D #2 \scan_stop: }
20328 \cs_new_protected:Npn \skip_gsub:Nn #1#2
20329 { \tex_global:D \tex_advance:D #1 - \tex_glueexpr:D #2 \scan_stop: }
20330 \cs_generate_variant:Nn \skip_sub:Nn { c }
20331 \cs_generate_variant:Nn \skip_gsub:Nn { c }
```

(End definition for `\skip_add:Nn` and others. These functions are documented on page 215.)

62.15 Skip expression conditionals

`\skip_if_eq_p:nn` Comparing skips means doing two expansions to make strings, and then testing them.
`\skip_if_eq:nnTF` As a result, only equality is tested.

```
20332 \prg_new_conditional:Npnn \skip_if_eq:nn #1#2 { p , T , F , TF }
20333 {
20334   \str_if_eq:eeTF { \skip_eval:n {#1} } { \skip_eval:n {#2} }
20335   { \prg_return_true: }
20336   { \prg_return_false: }
20337 }
```

(End definition for `\skip_if_eq:nnTF`. This function is documented on page 216.)

`\skip_if_finite:p:n` With ε -TeX, we have an easy access to the order of infinities of the stretch and shrink components of a skip. However, to access both, we either need to evaluate the expression twice, or evaluate it, then call an auxiliary to extract both pieces of information from the result. Since we are going to need an auxiliary anyways, it is quicker to make it search for the string `fil` which characterizes infinite glue.

`\skip_if_finite:nTF`

`__skip_if_finite:wwNw`

```

20338 \cs_set_protected:Npn \__skip_tmp:w #1
20339 {
20340   \prg_new_conditional:Npnn \skip_if_finite:n ##1 { p , T , F , TF }
20341   {
20342     \exp_after:wN \__skip_if_finite:wwNw
20343     \skip_use:N \tex_glueexpr:D ##1 ; \prg_return_false:
20344     #1 ; \prg_return_true: \s__skip_stop
20345   }
20346   \cs_new:Npn \__skip_if_finite:wwNw ##1 #1 ##2 ; ##3 ##4 \s__skip_stop {##3}
20347 }
20348 \exp_args:No \__skip_tmp:w { \tl_to_str:n { fil } }

```

(End definition for `\skip_if_finite:nTF` and `__skip_if_finite:wwNw`. This function is documented on page 216.)

62.16 Using skip expressions and variables

`\skip_eval:n` Evaluating a skip expression expandably.

```

20349 \cs_new:Npn \skip_eval:n #1
20350 { \skip_use:N \tex_glueexpr:D #1 \scan_stop: }

```

(End definition for `\skip_eval:n`. This function is documented on page 216.)

`\skip_use:N` Accessing a $\langle skip \rangle$.

`\skip_use:c`

```

20351 \cs_new_eq:NN \skip_use:N \dim_use:N
20352 \cs_new_eq:NN \skip_use:c \dim_use:c

```

(End definition for `\skip_use:N`. This function is documented on page 216.)

62.17 Inserting skips into the output

`\skip_horizontal:N` Inserting skips.

```

\skip_horizontal:c 20353 \cs_new_eq:NN \skip_horizontal:N \tex_hskip:D
\skip_horizontal:n 20354 \cs_new:Npn \skip_horizontal:n #1
\skip_vertical:N   20355 { \skip_horizontal:N \tex_glueexpr:D #1 \scan_stop: }
\skip_vertical:c   20356 \cs_new_eq:NN \skip_vertical:N \tex_vskip:D
\skip_vertical:n   20357 \cs_new:Npn \skip_vertical:n #1
20358 { \skip_vertical:N \tex_glueexpr:D #1 \scan_stop: }
20359 \cs_generate_variant:Nn \skip_horizontal:N { c }
20360 \cs_generate_variant:Nn \skip_vertical:N { c }

```

(End definition for `\skip_horizontal:N` and others. These functions are documented on page 217.)

62.18 Viewing skip variables

`\skip_show:N` Diagnostics.

```
\skip_show:c 20361 \cs_new_eq:NN \skip_show:N \__kernel_register_show:N
20362 \cs_generate_variant:Nn \skip_show:N { c }
```

(End definition for \skip_show:N. This function is documented on page 216.)

`\skip_show:n` Diagnostics. We don't use the T_EX primitive `\showthe` to show skip expressions: this gives a more unified output.

```
20363 \cs_new_protected:Npn \skip_show:n
20364 { \msg_show_eval:Nn \skip_eval:n }
```

(End definition for \skip_show:n. This function is documented on page 216.)

`\skip_log:N` Diagnostics. Redirect output of `\skip_show:n` to the log.

```
\skip_log:c 20365 \cs_new_eq:NN \skip_log:N \__kernel_register_log:N
\skip_log:n 20366 \cs_new_eq:NN \skip_log:c \__kernel_register_log:c
20367 \cs_new_protected:Npn \skip_log:n
20368 { \msg_log_eval:Nn \skip_eval:n }
```

(End definition for \skip_log:N and \skip_log:n. These functions are documented on page 217.)

62.19 Constant skips

`\c_zero_skip` Skips with no rubber component are just dimensions but need to terminate correctly.

```
\c_max_skip 20369 \skip_const:Nn \c_zero_skip { \c_zero_dim }
20370 \skip_const:Nn \c_max_skip { \c_max_dim }
```

(End definition for \c_zero_skip and \c_max_skip. These functions are documented on page 217.)

62.20 Scratch skips

`\l_tmpa_skip` We provide two local and two global scratch registers, maybe we need more or less.

```
\l_tmpb_skip 20371 \skip_new:N \l_tmpa_skip
\g_tmpa_skip 20372 \skip_new:N \l_tmpb_skip
\g_tmpb_skip 20373 \skip_new:N \g_tmpa_skip
20374 \skip_new:N \g_tmpb_skip
```

(End definition for \l_tmpa_skip and others. These variables are documented on page 217.)

62.21 Creating and initialising muskip variables

`\muskip_new:N` And then we add muskips.

```
\muskip_new:c 20375 \cs_new_protected:Npn \muskip_new:N #1
20376 {
20377   \__kernel_chk_if_free_cs:N #1
20378   \cs:w newmuskip \cs_end: #1
20379 }
20380 \cs_generate_variant:Nn \muskip_new:N { c }
```

(End definition for \muskip_new:N. This function is documented on page 218.)

`\muskip_const:Nn` See `\skip_const:Nn`.
`\muskip_const:cn`

```

20381 \cs_new_protected:Npn \muskip_const:Nn #1#2
20382 {
20383   \muskip_new:N #1
20384   \tex_global:D #1 = \muskip_eval:n {#2} \scan_stop:
20385 }
20386 \cs_generate_variant:Nn \muskip_const:Nn { c }

```

(End definition for `\muskip_const:Nn`. This function is documented on page 218.)

`\muskip_zero:N` Reset the register to zero.
`\muskip_zero:c`
`\muskip_gzero:N`
`\muskip_gzero:c`

```

20387 \cs_new_protected:Npn \muskip_zero:N #1
20388 { #1 = \c_zero_muskip }
20389 \cs_new_protected:Npn \muskip_gzero:N #1
20390 { \tex_global:D #1 = \c_zero_muskip }
20391 \cs_generate_variant:Nn \muskip_zero:N { c }
20392 \cs_generate_variant:Nn \muskip_gzero:N { c }

```

(End definition for `\muskip_zero:N` and `\muskip_gzero:N`. These functions are documented on page 218.)

`\muskip_zero_new:N` Create a register if needed, otherwise clear it.
`\muskip_zero_new:c`
`\muskip_gzero_new:N`
`\muskip_gzero_new:c`

```

20393 \cs_new_protected:Npn \muskip_zero_new:N #1
20394 { \muskip_if_exist:NTF #1 { \muskip_zero:N #1 } { \muskip_new:N #1 } }
20395 \cs_new_protected:Npn \muskip_gzero_new:N #1
20396 { \muskip_if_exist:NTF #1 { \muskip_gzero:N #1 } { \muskip_new:N #1 } }
20397 \cs_generate_variant:Nn \muskip_zero_new:N { c }
20398 \cs_generate_variant:Nn \muskip_gzero_new:N { c }

```

(End definition for `\muskip_zero_new:N` and `\muskip_gzero_new:N`. These functions are documented on page 218.)

`\muskip_if_exist_p:N` Copies of the cs functions defined in l3basics.
`\muskip_if_exist_p:c`
`\muskip_if_exist:NTF`
`\muskip_if_exist:cTF`

```

20399 \prg_new_eq_conditional:NNn \muskip_if_exist:N \cs_if_exist:N
20400 { TF , T , F , p }
20401 \prg_new_eq_conditional:NNn \muskip_if_exist:c \cs_if_exist:c
20402 { TF , T , F , p }

```

(End definition for `\muskip_if_exist:NTF`. This function is documented on page 218.)

62.22 Setting muskip variables

`\muskip_set:Nn` This should be pretty familiar.
`\muskip_set:cn`
`\muskip_gset:Nn`
`\muskip_gset:cn`

```

20403 \cs_new_protected:Npn \muskip_set:Nn #1#2
20404 { #1 = \tex_muexpr:D #2 \scan_stop: }
20405 \cs_new_protected:Npn \muskip_gset:Nn #1#2
20406 { \tex_global:D #1 = \tex_muexpr:D #2 \scan_stop: }
20407 \cs_generate_variant:Nn \muskip_set:Nn { c }
20408 \cs_generate_variant:Nn \muskip_gset:Nn { c }

```

(End definition for `\muskip_set:Nn` and `\muskip_gset:Nn`. These functions are documented on page 219.)

`\muskip_set_eq:NN` All straightforward.

```

\muskip_set_eq:cN 20409 \cs_new_protected:Npn \muskip_set_eq:NN #1#2 { #1 = #2 }
\muskip_set_eq:Nc 20410 \cs_generate_variant:Nn \muskip_set_eq:NN { c , Nc , cc }
\muskip_set_eq:cc 20411 \cs_new_protected:Npn \muskip_gset_eq:NN #1#2 { \tex_global:D #1 = #2 }
\muskip_gset_eq:NN 20412 \cs_generate_variant:Nn \muskip_gset_eq:NN { c , Nc , cc }
\muskip_gset_eq:cN
\muskip_gset_eq:Nc
\muskip_gset_eq:cc

```

(End definition for `\muskip_set_eq:NN` and `\muskip_gset_eq:NN`. These functions are documented on page 219.)

`\muskip_add:Nn` Using by here deals with the (incorrect) case `\muskip123`.

```

\muskip_add:cN 20413 \cs_new_protected:Npn \muskip_add:Nn #1#2
\muskip_gadd:Nn 20414 { \tex_advance:D #1 \tex_muexpr:D #2 \scan_stop: }
\muskip_gadd:cN 20415 \cs_new_protected:Npn \muskip_gadd:Nn #1#2
\muskip_sub:Nn 20416 { \tex_global:D \tex_advance:D #1 \tex_muexpr:D #2 \scan_stop: }
\muskip_sub:cN 20417 \cs_generate_variant:Nn \muskip_add:Nn { c }
\muskip_gsub:Nn 20418 \cs_generate_variant:Nn \muskip_gadd:Nn { c }
\muskip_gsub:cN 20419 \cs_new_protected:Npn \muskip_sub:Nn #1#2
20420 { \tex_advance:D #1 - \tex_muexpr:D #2 \scan_stop: }
20421 \cs_new_protected:Npn \muskip_gsub:Nn #1#2
20422 { \tex_global:D \tex_advance:D #1 - \tex_muexpr:D #2 \scan_stop: }
20423 \cs_generate_variant:Nn \muskip_sub:Nn { c }
20424 \cs_generate_variant:Nn \muskip_gsub:Nn { c }

```

(End definition for `\muskip_add:Nn` and others. These functions are documented on page 218.)

62.23 Using muskip expressions and variables

`\muskip_eval:n` Evaluating a muskip expression expandably.

```

20425 \cs_new:Npn \muskip_eval:n #1
20426 { \muskip_use:N \tex_muexpr:D #1 \scan_stop: }

```

(End definition for `\muskip_eval:n`. This function is documented on page 219.)

`\muskip_use:N` Accessing a $\langle muskip \rangle$.

```

\muskip_use:c 20427 \cs_new_eq:NN \muskip_use:N \dim_use:N
20428 \cs_new_eq:NN \muskip_use:c \dim_use:c

```

(End definition for `\muskip_use:N`. This function is documented on page 219.)

62.24 Viewing muskip variables

`\muskip_show:N` Diagnostics.

```

\muskip_show:c 20429 \cs_new_eq:NN \muskip_show:N \__kernel_register_show:N
20430 \cs_generate_variant:Nn \muskip_show:N { c }

```

(End definition for `\muskip_show:N`. This function is documented on page 219.)

`\muskip_show:n` Diagnostics. We don't use the $\mathrm{T}_{\mathrm{E}}\mathrm{X}$ primitive `\showthe` to show muskip expressions: this gives a more unified output.

```

20431 \cs_new_protected:Npn \muskip_show:n
20432 { \msg_show_eval:Nn \muskip_eval:n }

```

(End definition for `\muskip_show:n`. This function is documented on page 220.)

`\muskip_log:N` Diagnostics. Redirect output of `\muskip_show:n` to the log.
`\muskip_log:c` 20433 `\cs_new_eq:NN \muskip_log:N __kernel_register_log:N`
`\muskip_log:n` 20434 `\cs_new_eq:NN \muskip_log:c __kernel_register_log:c`
20435 `\cs_new_protected:Npn \muskip_log:n`
20436 `{ \msg_log_eval:Nn \muskip_eval:n }`

(End definition for `\muskip_log:N` and `\muskip_log:n`. These functions are documented on page 220.)

62.25 Constant muskips

`\c_zero_muskip` Constant muskips given by their value.
`\c_max_muskip` 20437 `\muskip_const:Nn \c_zero_muskip { 0 mu }`
20438 `\muskip_const:Nn \c_max_muskip { 16383.99999 mu }`

(End definition for `\c_zero_muskip` and `\c_max_muskip`. These functions are documented on page 220.)

62.26 Scratch muskips

`\l_tmpa_muskip` We provide two local and two global scratch registers, maybe we need more or less.
`\l_tmpb_muskip` 20439 `\muskip_new:N \l_tmpa_muskip`
`\g_tmpa_muskip` 20440 `\muskip_new:N \l_tmpb_muskip`
`\g_tmpb_muskip` 20441 `\muskip_new:N \g_tmpa_muskip`
20442 `\muskip_new:N \g_tmpb_muskip`

(End definition for `\l_tmpa_muskip` and others. These variables are documented on page 220.)

20443 `\</package>`

Chapter 63

l3keys Implementation

20444 $\langle *package \rangle$

63.1 Low-level interface

The low-level key parser’s implementation is based heavily on `expkv`. Compared to `keyval` it adds a number of additional “safety” requirements and allows to process the parsed list of key–value pairs in a variety of ways. The net result is that this code needs around one and a half the amount of time as `keyval` to parse the same list of keys. To optimise speed as far as reasonably practical, a number of lower-level approaches are taken rather than using the higher-level `expl3` interfaces.

20445 $\langle @@=keyval \rangle$

```
\s__keyval_nil
\s__keyval_mark
\s__keyval_stop
\s__keyval_tail
20446 \scan_new:N \s__keyval_nil
20447 \scan_new:N \s__keyval_mark
20448 \scan_new:N \s__keyval_stop
20449 \scan_new:N \s__keyval_tail
```

(End definition for `\s__keyval_nil` and others.)

`\l__kernel_keyval_allow_blank_keys_bool`

The general behavior of the `l3keys` module is to throw an error on blank key names. However to support the usage of `\keyval_parse:nnn` in the `l3prop` module we allow this error to be switched off temporarily and just ignore blank names.

20450 \backslash bool_new:N \backslash l__kernel_keyval_allow_blank_keys_bool

(End definition for `\l__kernel_keyval_allow_blank_keys_bool`.)

This temporary macro will be used since some of the definitions will need an active comma or equals sign. Inside of this macro `#1` will be the active comma and `#2` will be the active equals sign.

```
20451 \group_begin:
20452   \cs_set_protected:Npn \__keyval_tmp:w #1#2
20453   {
```

`\keyval_parse:nnn`
`\keyval_parse:NNn`

The main function starts the first of two loops. The outer loop splits the key–value list at active commas, the inner loop will do so at other commas. The use of `\s__keyval_mark` here prevents loss of braces from the key argument.

```

20454 \cs_if_exist:NTF \tex_expanded:D
20455 {
20456   \cs_new:Npn \keyval_parse:nnn ##1 ##2 ##3
20457   {
20458     \__kernel_exp_not:w \tex_expanded:D
20459     {
20460       {
20461         \__keyval_loop_active:nnw {##1} {##2}
20462         \s__keyval_mark ##3 #1 \s__keyval_tail #1
20463       }
20464     }
20465   }
20466 {
20467   \cs_new:Npn \keyval_parse:NNn ##1 ##2 ##3
20468   {
20469     \group_align_safe_begin:
20470     \__keyval_loop_active:nnw {##1} {##2}
20471     \s__keyval_mark ##3 #1 \s__keyval_tail #1
20472     \group_align_safe_end:
20473   }
20474 }
20475 }
20476 \cs_new_eq:NN \keyval_parse:NNn \keyval_parse:nnn

```

(End definition for `\keyval_parse:nnn` and `\keyval_parse:NNn`. These functions are documented on page 234.)

`__keyval_loop_active:nnw` First a fast test for the end of the loop is done, it'll gobble everything up to a `\s__keyval_tail`. The loop ending macro will gobble everything to the last comma in this definition. If the end isn't reached yet, start the second loop splitting at other commas, the next iteration of this first loop will be inserted by the end of `__keyval_loop_other:nnw`.

```

20477 \cs_new:Npn \__keyval_loop_active:nnw ##1 ##2 ##3 #1
20478 {
20479   \__keyval_if_recursion_tail:w ##3
20480   \__keyval_end_loop_active:w \s__keyval_tail
20481   \__keyval_loop_other:nnw {##1} {##2} ##3 , \s__keyval_tail ,
20482 }

```

(End definition for `__keyval_loop_active:nnw`.)

`__keyval_split_other:w` These two macros allow to split at the first equals sign of category 12 or 13. At the same time they also execute branching by inserting the first token following `\s__keyval_mark` that followed the equals sign. Hence they also test for the presence of such an equals sign simultaneously.

```

20483 \cs_new:Npn \__keyval_split_other:w ##1 = ##2 \s__keyval_mark ##3
20484 { ##3 ##1 \s__keyval_stop \s__keyval_mark ##2 }
20485 \cs_new:Npn \__keyval_split_active:w ##1 #2 ##2 \s__keyval_mark ##3
20486 { ##3 ##1 \s__keyval_stop \s__keyval_mark ##2 }

```

(End definition for `__keyval_split_other:w` and `__keyval_split_active:w`.)

`__keyval_loop_other:nnw` The second loop uses the same test for its end as the first loop, next it splits at the first active equals sign using `__keyval_split_active:w`. The `\s__keyval_nil` prevents

accidental brace stripping and acts as a delimiter in the next steps. First testing for an active equals sign will reduce the number of necessary expansion steps for the expected average use case of other equals signs and hence perform better on average.

```

20487 \cs_new:Npn \__keyval_loop_other:nnw ##1 ##2 ##3 ,
20488 {
20489 \__keyval_if_recursion_tail:w ##3
20490 \__keyval_end_loop_other:w \s__keyval_tail
20491 \__keyval_split_active:w ##3 \s__keyval_nil
20492 \s__keyval_mark \__keyval_split_active_auxi:w
20493 #2 \s__keyval_mark \__keyval_clean_up_active:w
20494 {##1} {##2}
20495 \s__keyval_mark
20496 }

```

(End definition for __keyval_loop_other:nnw.)

```

\__keyval_split_active_auxi:w
\__keyval_split_active_auxii:w
\__keyval_split_active_auxiii:w
\__keyval_split_active_auxiv:w
\__keyval_split_active_auxv:w

```

After __keyval_split_active:w the following will only be called if there was at least one active equals sign in the current key–value pair. Therefore this is the execution branch for a key–value pair with an active equals sign. ##1 will be everything up to the first active equals sign. First it tests for other equals signs in the key name, which will eventually throw an error via __keyval_misplaced_equal_after_active_error:w. If none was found we forward the key to __keyval_split_active_auxii:w.

```

20497 \cs_new:Npn \__keyval_split_active_auxi:w ##1 \s__keyval_stop
20498 {
20499 \__keyval_split_other:w ##1 \s__keyval_nil
20500 \s__keyval_mark \__keyval_misplaced_equal_after_active_error:w
20501 = \s__keyval_mark \__keyval_split_active_auxii:w
20502 }

```

__keyval_split_active_auxii:w gets the correct key name with a leading \s__keyval_mark as ##1. It has to sanitise the remainder of the previous test and trims the key name which will be forwarded to __keyval_split_active_auxiii:w.

```

20503 \cs_new:Npn \__keyval_split_active_auxii:w
20504 ##1 \s__keyval_nil \s__keyval_mark \__keyval_misplaced_equal_after_active_error:w
20505 \s__keyval_stop \s__keyval_mark
20506 ##2 \s__keyval_nil #2 \s__keyval_mark \__keyval_clean_up_active:w
20507 { \__keyval_trim:nN {##1} \__keyval_split_active_auxiii:w ##2 \s__keyval_nil }

```

Next we test for a misplaced active equals sign in the value, if none is found __keyval_split_active_auxiv:w will be called.

```

20508 \cs_new:Npn \__keyval_split_active_auxiii:w ##1 ##2 \s__keyval_nil
20509 {
20510 \__keyval_split_active:w ##2 \s__keyval_nil
20511 \s__keyval_mark \__keyval_misplaced_equal_in_split_error:w
20512 #2 \s__keyval_mark \__keyval_split_active_auxiv:w
20513 {##1}
20514 }

```

This runs the last test after sanitising the remainder of the previous one. This time test for a misplaced equals sign of category 12 in the value. Finally the last auxiliary macro will be called.

```

20515 \cs_new:Npn \__keyval_split_active_auxiv:w
20516 ##1 \s__keyval_nil \s__keyval_mark \__keyval_misplaced_equal_in_split_error:w
20517 \s__keyval_stop \s__keyval_mark

```

```

20518     {
20519         \__keyval_split_other:w ##1 \s__keyval_nil
20520         \s__keyval_mark \__keyval_misplaced_equal_in_split_error:w
20521         = \s__keyval_mark \__keyval_split_active_auxv:w
20522     }

```

This last macro in this execution branch sanitises the last test, trims the value and passes it to `__keyval_pair:nnnn`.

```

20523     \cs_new:Npn \__keyval_split_active_auxv:w
20524         ##1 \s__keyval_nil \s__keyval_mark \__keyval_misplaced_equal_in_split_error:w
20525         \s__keyval_stop \s__keyval_mark
20526         { \__keyval_trim:nN { ##1 } \__keyval_pair:nnnn }

```

(End definition for __keyval_split_active_auxi:w and others.)

`__keyval_clean_up_active:w`

The following is the branch taken if the key–value pair doesn’t contain an active equals sign. The remainder of that test will be cleaned up by `__keyval_clean_up_active:w` which will then split at an equals sign of category other.

```

20527     \cs_new:Npn \__keyval_clean_up_active:w
20528         ##1 \s__keyval_nil \s__keyval_mark \__keyval_split_active_auxi:w \s__keyval_stop
20529     {
20530         \__keyval_split_other:w ##1 \s__keyval_nil
20531         \s__keyval_mark \__keyval_split_other_auxi:w
20532         = \s__keyval_mark \__keyval_clean_up_other:w
20533     }

```

(End definition for __keyval_clean_up_active:w.)

`__keyval_split_other_auxi:w`

`__keyval_split_other_auxii:w`

`__keyval_split_other_auxiii:w`

This is executed if the key–value pair doesn’t contain an active equals sign but at least one other. `##1` of `__keyval_split_other_auxi:w` will contain the complete key name, which is trimmed and forwarded to the next auxiliary macro.

```

20534     \cs_new:Npn \__keyval_split_other_auxi:w ##1 \s__keyval_stop
20535     { \__keyval_trim:nN { ##1 } \__keyval_split_other_auxii:w }

```

We know that the value doesn’t contain misplaced active equals signs but we have to test for others. Also we need to sanitise the previous test, which is done here and not earlier to avoid superfluous argument grabbing.

```

20536     \cs_new:Npn \__keyval_split_other_auxii:w
20537         ##1 ##2 \s__keyval_nil = \s__keyval_mark \__keyval_clean_up_other:w
20538     {
20539         \__keyval_split_other:w ##2 \s__keyval_nil
20540         \s__keyval_mark \__keyval_misplaced_equal_in_split_error:w
20541         = \s__keyval_mark \__keyval_split_other_auxiii:w
20542         { ##1 }
20543     }

```

`__keyval_split_other_auxiii:w` sanitises the test for other equals signs, trims the value and forwards it to `__keyval_pair:nnnn`.

```

20544     \cs_new:Npn \__keyval_split_other_auxiii:w
20545         ##1 \s__keyval_nil \s__keyval_mark \__keyval_misplaced_equal_in_split_error:w
20546         \s__keyval_stop \s__keyval_mark
20547         { \__keyval_trim:nN { ##1 } \__keyval_pair:nnnn }

```

(End definition for __keyval_split_other_auxi:w, __keyval_split_other_auxii:w, and __keyval_split_other_auxiii:w.)

`__keyval_clean_up_other:w` `__keyval_clean_up_other:w` is the last branch that might exist. It is called if no equals sign was found, hence the only possibilities left are a blank list element, which is to be skipped, or a lonely key. If it's no empty list element this will trim the key name and forward it to `__keyval_key:nn`.

```

20548 \cs_new:Npn \__keyval_clean_up_other:w
20549     ##1 \s__keyval_nil \s__keyval_mark \__keyval_split_other_auxi:w \s__keyval_stop \
20550 {
20551     \__keyval_if_blank:w ##1 \s__keyval_nil \s__keyval_stop \__keyval_blank_true:w
20552     \s__keyval_mark \s__keyval_stop
20553     \__keyval_trim:nN { ##1 } \__keyval_key:nn
20554 }

```

(End definition for `__keyval_clean_up_other:w`.)

`keyval_misplaced_equal_after_active_error:w` `__keyval_misplaced_equal_in_split_error:w` All these two macros do is gobble the remainder of the current other loop execution and throw an error. Afterwards they have to insert the next loop iteration.

```

20555 \cs_new:Npn \__keyval_misplaced_equal_after_active_error:w
20556     \s__keyval_mark ##1 \s__keyval_stop \s__keyval_mark ##2 \s__keyval_nil
20557     = \s__keyval_mark \__keyval_split_active_auxii:w
20558     \s__keyval_mark ##3 \s__keyval_nil
20559     #2 \s__keyval_mark \__keyval_clean_up_active:w
20560 {
20561     \msg_expandable_error:nn
20562     { keyval } { misplaced-equals-sign }
20563     \__keyval_loop_other:nnw
20564 }
20565 \cs_new:Npn \__keyval_misplaced_equal_in_split_error:w
20566     \s__keyval_mark ##1 \s__keyval_stop \s__keyval_mark ##2 \s__keyval_nil
20567     ##3 \s__keyval_mark ##4 ##5
20568 {
20569     \msg_expandable_error:nn
20570     { keyval } { misplaced-equals-sign }
20571     \__keyval_loop_other:nnw
20572 }

```

(End definition for `__keyval_misplaced_equal_after_active_error:w` and `__keyval_misplaced_equal_in_split_error:w`.)

`__keyval_end_loop_other:w` `__keyval_end_loop_active:w` All that's left for the parsing loops are the macros which end the recursion. Both just gobble the remaining tokens of the respective loop including the next recursion call. `__keyval_end_loop_other:w` also has to insert the next iteration of the active loop.

```

20573 \cs_new:Npn \__keyval_end_loop_other:w
20574     \s__keyval_tail
20575     \__keyval_split_active:w
20576     \s__keyval_mark \s__keyval_tail
20577     \s__keyval_nil \s__keyval_mark
20578     \__keyval_split_active_auxi:w
20579     #2 \s__keyval_mark \__keyval_clean_up_active:w
20580     { \__keyval_loop_active:nnw }
20581 \cs_new:Npn \__keyval_end_loop_active:w
20582     \s__keyval_tail
20583     \__keyval_loop_other:nnw ##1 \s__keyval_mark \s__keyval_tail , \s__keyval_tail ,
20584     { }

```

(End definition for `_keyval_end_loop_other:w` and `_keyval_end_loop_active:w`.)

The parsing loops are done, so here ends the definition of `_keyval_tmp:w`, which will finally set up the macros.

```

20585     }
20586     \char_set_catcode_active:n { '\, }
20587     \char_set_catcode_active:n { '\= }
20588     \_keyval_tmp:w , =
20589 \group_end:

```

`_keyval_pair:nnnn` These macros will be called on the parsed keys and values of the key–value list. All arguments are completely trimmed. They test for blank key names and call the functions passed to `\keyval_parse:nnn` inside of `\exp_not:n` with the correct arguments. Afterwards they insert the next iteration of the other loop.

```

20590 \group_begin:
20591   \cs_set_protected:Npn \_keyval_tmp:w #1#2
20592   {
20593     \cs_new:Npn \_keyval_pair:nnnn ##1 ##2 ##3 ##4
20594     {
20595       \_keyval_if_blank:w \s__keyval_mark ##2 \s__keyval_nil \s__keyval_stop \_keyval
20596       \s__keyval_mark \s__keyval_stop
20597       #1
20598       \exp_not:n { ##4 {##2} {##1} }
20599       #2
20600       \_keyval_loop_other:nnw {##3} {##4}
20601     }
20602     \cs_new:Npn \_keyval_key:nn ##1 ##2
20603     {
20604       \_keyval_if_blank:w \s__keyval_mark ##1 \s__keyval_nil \s__keyval_stop \_keyval
20605       \s__keyval_mark \s__keyval_stop
20606       #1
20607       \exp_not:n { ##2 {##1} }
20608       #2
20609       \_keyval_loop_other:nnw {##2}
20610     }
20611   }
20612   \cs_if_exist:NTF \tex_expanded:D
20613   { \_keyval_tmp:w { } { } }
20614   { \_keyval_tmp:w \group_align_safe_end: \group_align_safe_begin: }
20615 \group_end:

```

(End definition for `_keyval_pair:nnnn` and `_keyval_key:nn`.)

`_keyval_if_empty:w` All these tests work by gobbling tokens until a certain combination is met, which makes them pretty fast. The test for a blank argument should be called with an arbitrary token following the argument. Each of these utilize the fact that the argument will contain a leading `\s__keyval_mark`.

```

20616 \cs_new:Npn \_keyval_if_empty:w #1 \s__keyval_mark \s__keyval_stop { }
20617 \cs_new:Npn \_keyval_if_blank:w \s__keyval_mark #1 { \_keyval_if_empty:w \s__keyval_mark
20618 \cs_new:Npn \_keyval_if_recursion_tail:w \s__keyval_mark #1 \s__keyval_tail { }

```

(End definition for `_keyval_if_empty:w`, `_keyval_if_blank:w`, and `_keyval_if_recursion_tail:w`.)

```

    \_keyval_blank_true:w
\_keyval_blank_key_error:w

```

These macros will be called if the tests above didn't gobble them, they execute the branching.

```

20619 \cs_new:Npn \_keyval_blank_true:w \s_keyval_mark \s_keyval_stop \_keyval_trim:nN #1 \_
20620 { \_keyval_loop_other:nnw }
20621 \cs_new:Npn \_keyval_blank_key_error:w \s_keyval_mark \s_keyval_stop #1 \_keyval_loop_o
20622 {
20623     \bool_if:NTF \l_kernel_keyval_allow_blank_keys_bool
20624     { #1 }
20625     { \msg_expandable_error:nn { keyval } { blank-key-name } }
20626     \_keyval_loop_other:nnw
20627 }

```

(End definition for _keyval_blank_true:w and _keyval_blank_key_error:w.)

Two messages for the low level parsing system.

```

20628 \msg_new:nnn { keyval } { misplaced-equals-sign }
20629 { Misplaced~'=~in~key-value-input~\msg_line_context: }
20630 \msg_new:nnn { keyval } { blank-key-name }
20631 { Blank~key~name~in~key-value-input~\msg_line_context: }
20632 \prop_gput:Nnn \g_msg_module_name_prop { keyval } { LaTeX3 }
20633 \prop_gput:Nnn \g_msg_module_type_prop { keyval } { }

```

```

    \_keyval_trim:nN
\_keyval_trim_auxi:w
\_keyval_trim_auxii:w
\_keyval_trim_auxiii:w
\_keyval_trim_auxiv:w

```

And an adapted version of _tl_trim_spaces:nn which is a bit faster for our use case, as it can strip the braces at the end. This is pretty much the same concept, so I won't comment on it here. The speed gain by using this instead of \tl_trim_spaces_apply:nN is about 10 % of the total time for \keyval_parse:Nn with one key and one key-value pair, so I think it's worth it.

```

20634 \group_begin:
20635     \cs_set_protected:Npn \_keyval_tmp:w #1
20636     {
20637         \cs_new:Npn \_keyval_trim:nN ##1
20638         {
20639             \_keyval_trim_auxi:w
20640             ##1
20641             \s_keyval_nil
20642             \s_keyval_mark #1 { }
20643             \s_keyval_mark \_keyval_trim_auxii:w
20644             \_keyval_trim_auxiii:w
20645             #1 \s_keyval_nil
20646             \_keyval_trim_auxiv:w
20647         }
20648         \cs_new:Npn \_keyval_trim_auxi:w ##1 \s_keyval_mark #1 ##2 \s_keyval_mark ##3
20649         {
20650             ##3
20651             \_keyval_trim_auxi:w
20652             \s_keyval_mark
20653             ##2
20654             \s_keyval_mark #1 {##1}
20655         }
20656         \cs_new:Npn \_keyval_trim_auxii:w \_keyval_trim_auxi:w \s_keyval_mark \s_keyval_m
20657         {
20658             \_keyval_trim_auxiii:w
20659             ##1
20660         }

```

```

20661 \cs_new:Npn \__keyval_trim_auxiii:w ##1 #1 \s__keyval_nil ##2
20662 {
20663     ##2
20664     ##1 \s__keyval_nil
20665     \__keyval_trim_auxiii:w
20666 }

```

This is the one macro which differs from the original definition.

```

20667 \cs_new:Npn \__keyval_trim_auxiv:w
20668     \s__keyval_mark ##1 \s__keyval_nil
20669     \__keyval_trim_auxiii:w \s__keyval_nil \__keyval_trim_auxiii:w
20670     ##2
20671     { ##2 { ##1 } }
20672 }
20673 \__keyval_tmp:w { ~ }
20674 \group_end:

```

(End definition for `__keyval_trim:nN` and others.)

63.2 Constants and variables

```

20675 <@@=keys>

```

Various storage areas for the different data which make up keys.

```

\c__keys_code_root_str
\c__keys_check_root_str
\c__keys_default_root_str
\c__keys_groups_root_str
\c__keys_inherit_root_str
\c__keys_type_root_str
20676 \str_const:Nn \c__keys_code_root_str { key~code~>~ }
20677 \str_const:Nn \c__keys_check_root_str { key~check~>~ }
20678 \str_const:Nn \c__keys_default_root_str { key~default~>~ }
20679 \str_const:Nn \c__keys_groups_root_str { key~groups~>~ }
20680 \str_const:Nn \c__keys_inherit_root_str { key~inherit~>~ }
20681 \str_const:Nn \c__keys_type_root_str { key~type~>~ }

```

(End definition for `\c__keys_code_root_str` and others.)

`\c__keys_props_root_str` The prefix for storing properties.

```

20682 \str_const:Nn \c__keys_props_root_str { key~prop~>~ }

```

(End definition for `\c__keys_props_root_str`.)

`\l_keys_choice_int` `\l_keys_choice_tl` Publicly accessible data on which choice is being used when several are generated as a set.

```

20683 \int_new:N \l_keys_choice_int
20684 \tl_new:N \l_keys_choice_tl

```

(End definition for `\l_keys_choice_int` and `\l_keys_choice_tl`. These variables are documented on page 228.)

`\l__keys_groups_clist` Used for storing and recovering the list of groups which apply to a key: set as a comma list but at one point we have to use this for a token list recovery.

```

20685 \clist_new:N \l__keys_groups_clist

```

(End definition for `\l__keys_groups_clist`.)

`\l_keys_key_str` `\l_keys_key_tl` The name of a key itself: needed when setting keys. The `tl` version is deprecated but has to be handled manually.

```

20686 \str_new:N \l_keys_key_str
20687 \tl_new:N \l_keys_key_tl

```

(End definition for `\l_keys_key_str` and `\l_keys_key_tl`. These variables are documented on page 230.)

`\l__keys_module_str` The module for an entire set of keys.

20688 `\str_new:N \l__keys_module_str`

(End definition for `\l__keys_module_str`.)

`\l__keys_no_value_bool` A marker is needed internally to show if only a key or a key plus a value was seen: this is recorded here.

20689 `\bool_new:N \l__keys_no_value_bool`

(End definition for `\l__keys_no_value_bool`.)

`\l__keys_only_known_bool` Used to track if only “known” keys are being set.

20690 `\bool_new:N \l__keys_only_known_bool`

(End definition for `\l__keys_only_known_bool`.)

`\l_keys_path_str` The “path” of the current key is stored here: this is available to the programmer and so is public. The older version is deprecated but has to be handled manually.

`\l_keys_path_tl`

20691 `\str_new:N \l_keys_path_str`

20692 `\tl_new:N \l_keys_path_tl`

(End definition for `\l_keys_path_str` and `\l_keys_path_tl`. These variables are documented on page 230.)

`\l__keys_inherit_str`

20693 `\str_new:N \l__keys_inherit_str`

(End definition for `\l__keys_inherit_str`.)

`\l__keys_relative_tl` The relative path for passing keys back to the user. As this can be explicitly no-value, it must be a token list.

20694 `\tl_new:N \l__keys_relative_tl`

20695 `\tl_set:Nn \l__keys_relative_tl { \q__keys_no_value }`

(End definition for `\l__keys_relative_tl`.)

`\l__keys_property_str` The “property” begin set for a key at definition time is stored here.

20696 `\str_new:N \l__keys_property_str`

(End definition for `\l__keys_property_str`.)

`\l__keys_selective_bool` Two flags for using key groups: one to indicate that “selective” setting is active, a second to specify which type (“opt-in” or “opt-out”).

`\l__keys_filtered_bool`

20697 `\bool_new:N \l__keys_selective_bool`

20698 `\bool_new:N \l__keys_filtered_bool`

(End definition for `\l__keys_selective_bool` and `\l__keys_filtered_bool`.)

`\l__keys_selective_seq` The list of key groups being filtered in or out during selective setting.

20699 `\seq_new:N \l__keys_selective_seq`

(End definition for `\l__keys_selective_seq`.)

`\l__keys_unused_clist` Used when setting only some keys to store those left over.

20700 `\tl_new:N \l__keys_unused_clist`

(End definition for `\l__keys_unused_clist`.)

`\l_keys_value_tl` The value given for a key: may be empty if no value was given.

20701 `\tl_new:N \l_keys_value_tl`

(End definition for `\l_keys_value_tl`. This variable is documented on page 230.)

`\l__keys_tmp_bool` Scratch space.

`\l__keys_tmpa_tl` 20702 `\bool_new:N \l__keys_tmp_bool`

`\l__keys_tmpb_tl` 20703 `\tl_new:N \l__keys_tmpa_tl`

20704 `\tl_new:N \l__keys_tmpb_tl`

(End definition for `\l__keys_tmp_bool`, `\l__keys_tmpa_tl`, and `\l__keys_tmpb_tl`.)

`\l_keys_usage_load_prop` Global data for document-level information.

`\l_keys_usage_preamble_prop` 20705 `\prop_new:N \l_keys_usage_load_prop`

20706 `\prop_new:N \l_keys_usage_preamble_prop`

(End definition for `\l_keys_usage_load_prop` and `\l_keys_usage_preamble_prop`. These variables are documented on page 230.)

63.2.1 Internal auxiliaries

`\s__keys_nil` Internal scan marks.

`\s__keys_mark` 20707 `\scan_new:N \s__keys_nil`

`\s__keys_stop` 20708 `\scan_new:N \s__keys_mark`

20709 `\scan_new:N \s__keys_stop`

(End definition for `\s__keys_nil`, `\s__keys_mark`, and `\s__keys_stop`.)

`\q__keys_no_value` Internal quarks.

20710 `\quark_new:N \q__keys_no_value`

(End definition for `\q__keys_no_value`.)

`_keys_quark_if_no_value_p:N` Branching quark conditional.

`_keys_quark_if_no_value:N` ***TF*** 20711 `_kernel_quark_new_conditional:Nn _keys_quark_if_no_value:N { TF }`

(End definition for `_keys_quark_if_no_value:N` ***NTF***.)

63.3 The key defining mechanism

`\keys_define:nn` The public function for definitions is just a wrapper for the lower level mechanism, more or less. The outer function is designed to keep a track of the current module, to allow safe nesting. The module is set removing any leading / (which is not needed here).

```

20712 \cs_new_protected:Npn \keys_define:nn
20713 { \__keys_define:onn \l__keys_module_str }
20714 \cs_new_protected:Npn \__keys_define:nnn #1#2#3
20715 {
20716   \str_set:Nx \l__keys_module_str { \__keys_trim_spaces:n {#2} }
20717   \keyval_parse:NNn \__keys_define:n \__keys_define:nn {#3}
20718   \str_set:Nn \l__keys_module_str {#1}
20719 }
20720 \cs_generate_variant:Nn \__keys_define:nnn { o }

```

(End definition for `\keys_define:nn` and `__keys_define:nnn`. This function is documented on page 222.)

`__keys_define:n` The outer functions here record whether a value was given and then converge on a common internal mechanism. There is first a search for a property in the current key name, then a check to make sure it is known before the code hands off to the next step.

```

20721 \cs_new_protected:Npn \__keys_define:n #1
20722 {
20723   \bool_set_true:N \l__keys_no_value_bool
20724   \__keys_define_aux:nn {#1} { }
20725 }
20726 \cs_new_protected:Npn \__keys_define:nn #1#2
20727 {
20728   \bool_set_false:N \l__keys_no_value_bool
20729   \__keys_define_aux:nn {#1} {#2}
20730 }
20731 \cs_new_protected:Npn \__keys_define_aux:nn #1#2
20732 {
20733   \__keys_property_find:n {#1}
20734   \cs_if_exist:cTF { \c__keys_props_root_str \l__keys_property_str }
20735   { \__keys_define_code:n {#2} }
20736   {
20737     \str_if_empty:NF \l__keys_property_str
20738     {
20739       \msg_error:nnxx { keys } { property-unknown }
20740       \l__keys_property_str \l_keys_path_str
20741     }
20742   }
20743 }

```

(End definition for `__keys_define:n`, `__keys_define:nn`, and `__keys_define_aux:nn`.)

`__keys_property_find:n` Searching for a property means finding the last . in the input, and storing the text before and after it. Everything is turned into strings, so there is no problem using an x-type expansion. Since `__keys_trim_spaces:n` will turn its argument into a string anyway, this function uses `\cs_set_nopar:Npx` instead of `\tl_set:Nx` to gain some speed.

```

20744 \cs_new_protected:Npn \__keys_property_find:n #1
20745 {
20746   \cs_set_nopar:Npx \l__keys_property_str { \__keys_trim_spaces:n { #1 } }

```

```

20747     \exp_after:wN \__keys_property_find_auxi:w \l__keys_property_str
20748     \s__keys_nil \__keys_property_find_auxii:w
20749     . \s__keys_nil \__keys_property_find_err:w
20750 }
20751 \cs_new_protected:Npn \__keys_property_find_auxi:w #1 . #2 \s__keys_nil #3
20752 {
20753     #3 #1 \s__keys_mark #2 \s__keys_nil #3
20754 }
20755 \cs_new_protected:Npn \__keys_property_find_auxii:w
20756     #1 \s__keys_mark #2 \s__keys_nil \__keys_property_find_auxii:w . \s__keys_nil
20757     \__keys_property_find_err:w
20758 {
20759     \cs_set_nopar:Npx \l__keys_path_str
20760     { \str_if_empty:NF \l__keys_module_str { \l__keys_module_str / } #1 }
20761     \__keys_property_find_auxi:w #2 \s__keys_nil \__keys_property_find_auxiii:w . \s__keys_
20762     \__keys_property_find_auxiv:w
20763 }
20764 \cs_new_protected:Npn \__keys_property_find_auxiii:w #1 \s__keys_mark
20765 {
20766     \cs_set_nopar:Npx \l__keys_path_str { \l__keys_path_str . #1 }
20767     \__keys_property_find_auxi:w
20768 }
20769 \cs_new_protected:Npn \__keys_property_find_auxiv:w
20770     #1 \s__keys_nil \__keys_property_find_auxiii:w
20771     \s__keys_mark \s__keys_nil \__keys_property_find_auxiv:w
20772 {
20773     \cs_set_nopar:Npx \l__keys_property_str { . #1 }
20774     \cs_set_nopar:Npx \l__keys_path_str
20775     { \exp_after:wN \__keys_trim_spaces:n \exp_after:wN { \l__keys_path_str } }
20776     \tl_set_eq:NN \l__keys_path_tl \l__keys_path_str
20777 }
20778 \cs_new_protected:Npn \__keys_property_find_err:w
20779     #1 \s__keys_nil #2 \__keys_property_find_err:w
20780 {
20781     \str_clear:N \l__keys_property_str
20782     \msg_error:nnn { keys } { no-property } {#1}
20783 }

```

(End definition for __keys_property_find:n and others.)

__keys_define_code:n Two possible cases. If there is a value for the key, then just use the function. If not, then
 __keys_define_code:w a check to make sure there is no need for a value with the property. If there should be
 one then complain, otherwise execute it. There is no need to check for a : as if it was
 missing the earlier tests would have failed.

```

20784 \cs_new_protected:Npn \__keys_define_code:n #1
20785 {
20786     \bool_if:NTF \l__keys_no_value_bool
20787     {
20788         \exp_after:wN \__keys_define_code:w
20789         \l__keys_property_str \s__keys_stop
20790         { \use:c { \c__keys_props_root_str \l__keys_property_str } }
20791         {
20792             \msg_error:nnxx { keys } { property-requires-value }
20793             \l__keys_property_str \l__keys_path_str

```

```

20794     }
20795   }
20796   { \use:c { \c__keys_props_root_str \l__keys_property_str } {#1} }
20797 }
20798 \exp_last_unbraced:NNNN
20799   \cs_new:Npn \__keys_define_code:w #1 \c_colon_str #2 \s__keys_stop
20800     { \tl_if_empty:nTF {#2} }

```

(End definition for __keys_define_code:n and __keys_define_code:w.)

63.4 Turning properties into actions

Boolean keys are really just choices, but all done by hand. The second argument here is the scope: either empty or `g` for global.

```

\__keys_bool_set:Nn
\__keys_bool_set:cn
\__keys_bool_set_inverse:Nn
\__keys_bool_set_inverse:cn
\__keys_bool_set:Nnnn
20801 \cs_new_protected:Npn \__keys_bool_set:Nn #1#2
20802   { \__keys_bool_set:Nnnn #1 {#2} { true } { false } }
20803 \cs_generate_variant:Nn \__keys_bool_set:Nn { c }
20804 \cs_new_protected:Npn \__keys_bool_set_inverse:Nn #1#2
20805   { \__keys_bool_set:Nnnn #1 {#2} { false } { true } }
20806 \cs_generate_variant:Nn \__keys_bool_set_inverse:Nn { c }
20807 \cs_new_protected:Npn \__keys_bool_set:Nnnn #1#2#3#4
20808   {
20809     \bool_if_exist:NF #1 { \bool_new:N #1 }
20810     \__keys_choice_make:
20811     \__keys_cmd_set:nx { \l_keys_path_str / true }
20812       { \exp_not:c { bool_ #2 set_ #3 :N } \exp_not:N #1 }
20813     \__keys_cmd_set:nx { \l_keys_path_str / false }
20814       { \exp_not:c { bool_ #2 set_ #4 :N } \exp_not:N #1 }
20815     \__keys_cmd_set:nn { \l_keys_path_str / unknown }
20816       {
20817         \msg_error:nnx { keys } { boolean-values-only }
20818         \l_keys_key_str
20819       }
20820     \__keys_default_set:n { true }
20821   }
20822 \cs_generate_variant:Nn \__keys_bool_set:Nn { c }

```

(End definition for __keys_bool_set:Nn, __keys_bool_set_inverse:Nn, and __keys_bool_set:Nnnn.)

To make a choice from a key, two steps: set the code, and set the unknown key. As multichoice and choices are essentially the same bar one function, the code is given together.

```

\__keys_choice_make:
\__keys_multichoice_make:
\__keys_choice_make:N
\__keys_choice_make_aux:N
20823 \cs_new_protected:Npn \__keys_choice_make:
20824   { \__keys_choice_make:N \__keys_choice_find:n }
20825 \cs_new_protected:Npn \__keys_multichoice_make:
20826   { \__keys_choice_make:N \__keys_multichoice_find:n }
20827 \cs_new_protected:Npn \__keys_choice_make:N #1
20828   {
20829     \cs_if_exist:cTF
20830       { \c__keys_type_root_str \__keys_parent:o \l_keys_path_str }
20831       {
20832         \str_if_eq:vnTF
20833           { \c__keys_type_root_str \__keys_parent:o \l_keys_path_str }

```

```

20834         { choice }
20835     {
20836         \msg_error:nnxx { keys } { nested-choice-key }
20837         \l_keys_path_tl { \__keys_parent:o \l_keys_path_str }
20838     }
20839     { \__keys_choice_make_aux:N #1 }
20840 }
20841 { \__keys_choice_make_aux:N #1 }
20842 }
20843 \cs_new_protected:Npn \__keys_choice_make_aux:N #1
20844 {
20845     \cs_set_nopar:cpn { \c__keys_type_root_str \l_keys_path_str }
20846     { choice }
20847     \__keys_cmd_set:nn \l_keys_path_str { #1 {##1} }
20848     \__keys_cmd_set:nn { \l_keys_path_str / unknown }
20849     {
20850         \msg_error:nnxx { keys } { choice-unknown }
20851         \l_keys_path_str {##1}
20852     }
20853 }

```

(End definition for __keys_choice_make: and others.)

Auto-generating choices means setting up the root key as a choice, then defining each choice in turn.

```

\__keys_choices_make:nn
\__keys_multichoice_make:nn
\__keys_choices_make:Nnn

```

```

20854 \cs_new_protected:Npn \__keys_choices_make:nn
20855 { \__keys_choices_make:Nnn \__keys_choice_make: }
20856 \cs_new_protected:Npn \__keys_multichoice_make:nn
20857 { \__keys_choices_make:Nnn \__keys_multichoice_make: }
20858 \cs_new_protected:Npn \__keys_choices_make:Nnn #1#2#3
20859 {
20860     #1
20861     \int_zero:N \l_keys_choice_int
20862     \clist_map_inline:nn {#2}
20863     {
20864         \int_incr:N \l_keys_choice_int
20865         \__keys_cmd_set:nx
20866         { \l_keys_path_str / \__keys_trim_spaces:n {##1} }
20867         {
20868             \tl_set:Nn \exp_not:N \l_keys_choice_tl {##1}
20869             \int_set:Nn \exp_not:N \l_keys_choice_int
20870             { \int_use:N \l_keys_choice_int }
20871             \exp_not:n {#3}
20872         }
20873     }
20874 }

```

(End definition for __keys_choices_make:nn, __keys_multichoice_make:nn, and __keys_choices_make:Nnn.)

Setting the code for a key first logs if appropriate that we are defining a new key, then saves the code.

```

\__keys_cmd_set:nn
\__keys_cmd_set:nx
\__keys_cmd_set:Vn
\__keys_cmd_set:Vo

```

```

20875 \cs_new_protected:Npn \__keys_cmd_set:nn #1#2
20876 { \cs_set_protected:cpn { \c__keys_code_root_str #1 } ##1 {#2} }
20877 \cs_generate_variant:Nn \__keys_cmd_set:nn { nx , Vn , Vo }

```

(End definition for `_keys_cmd_set:nn`.)

`_keys_cs_set:NNpn` Creating control sequences is a bit more tricky than other cases as we need to pick up the `p` argument. To make the internals look clearer, the trailing `n` argument here is just for appearance.

`_keys_cs_set:Ncpn`

```

20878 \cs_new_protected:Npn \_keys_cs_set:NNpn #1#2#3#
20879 {
20880   \cs_set_protected:cpx { \c__keys_code_root_str \l_keys_path_str } ##1
20881   { #1 \exp_not:N #2 \exp_not:n {#3} {##1} }
20882   \use_none:n
20883 }
20884 \cs_generate_variant:Nn \_keys_cs_set:NNpn { Nc }

```

(End definition for `_keys_cs_set:NNpn`.)

`_keys_default_set:n` Setting a default value is easy. These are stored using `\cs_set_nopar:cpx` as this avoids any worries about whether a token list exists.

```

20885 \cs_new_protected:Npn \_keys_default_set:n #1
20886 {
20887   \tl_if_empty:nTF {#1}
20888   {
20889     \cs_set_eq:cN
20890     { \c__keys_default_root_str \l_keys_path_str }
20891     \tex_undefined:D
20892   }
20893   {
20894     \cs_set_nopar:cpx
20895     { \c__keys_default_root_str \l_keys_path_str }
20896     { \exp_not:n {#1} }
20897     \_keys_value_requirement:nn { required } { false }
20898   }
20899 }

```

(End definition for `_keys_default_set:n`.)

`_keys_groups_set:n` Assigning a key to one or more groups uses comma lists. As the list of groups only exists if there is anything to do, the setting is done using a scratch list. For the usual grouping reasons we use the low-level approach to undefining a list. We also use the low-level approach for the other case to avoid tripping up the `check-declarations` code.

```

20900 \cs_new_protected:Npn \_keys_groups_set:n #1
20901 {
20902   \clist_set:Nn \l__keys_groups_clist {#1}
20903   \clist_if_empty:NTF \l__keys_groups_clist
20904   {
20905     \cs_set_eq:cN { \c__keys_groups_root_str \l_keys_path_str }
20906     \tex_undefined:D
20907   }
20908   {
20909     \cs_set_eq:cN { \c__keys_groups_root_str \l_keys_path_str }
20910     \l__keys_groups_clist
20911   }
20912 }

```

(End definition for `_keys_groups_set:n`.)

`__keys_inherit:n` Inheritance means ignoring anything already said about the key: zap the lot and set up.

```
20913 \cs_new_protected:Npn \__keys_inherit:n #1
20914 {
20915     \__keys_undefine:
20916     \cs_set_nopar:cpn { \c__keys_inherit_root_str \l_keys_path_str } {#1}
20917 }
```

(End definition for __keys_inherit:n.)

`__keys_initialise:n` A set up for initialisation: just run the code if it exists.

```
20918 \cs_new_protected:Npn \__keys_initialise:n #1
20919 {
20920     \cs_if_exist:cTF
20921     { \c__keys_inherit_root_str \__keys_parent:o \l_keys_path_str }
20922     { \__keys_execute_inherit: }
20923     {
20924         \str_clear:N \l__keys_inherit_str
20925         \cs_if_exist:cT { \c__keys_code_root_str \l_keys_path_str }
20926         { \__keys_execute:nn \l_keys_path_str {#1} }
20927     }
20928 }
```

(End definition for __keys_initialise:n.)

`__keys_legacy_if_set:nn` Much the same as `expl3` booleans, except we assume that the switch exists.

```
\__keys_legacy_if_inverse:nn 20929 \cs_new_protected:Npn \__keys_legacy_if_set:nn #1#2
\__keys_legacy_if_inverse:nnn 20930 { \__keys_legacy_if_set:nnnn {#1} {#2} { true } { false } }
20931 \cs_new_protected:Npn \__keys_legacy_if_set_inverse:nn #1#2
20932 { \__keys_legacy_if_set:nnnn {#1} {#2} { false } { true } }
20933 \cs_new_protected:Npn \__keys_legacy_if_set:nnnn #1#2#3#4
20934 {
20935     \__keys_choice_make:
20936     \__keys_cmd_set:nx { \l_keys_path_str / true }
20937     { \exp_not:c { legacy_if_#2 set_ #3 :n } { \exp_not:n {#1} } }
20938     \__keys_cmd_set:nx { \l_keys_path_str / false }
20939     { \exp_not:c { legacy_if_#2 set_ #4 :n } { \exp_not:n {#1} } }
20940     \__keys_cmd_set:nn { \l_keys_path_str / unknown }
20941     {
20942         \msg_error:nnx { keys } { boolean-values-only }
20943         \l_keys_key_str
20944     }
20945     \__keys_default_set:n { true }
20946     \cs_if_exist:cF { if#1 }
20947     {
20948         \cs:w newif \exp_after:wN \cs_end:
20949         \cs:w if#1 \cs_end:
20950     }
20951 }
```

(End definition for __keys_legacy_if_set:nn, __keys_legacy_if_inverse:nn, and __keys_legacy_if_inverse:nnnn.)

`__keys_meta_make:n` To create a meta-key, simply set up to pass data through.

```
\__keys_meta_make:nn 20952 \cs_new_protected:Npn \__keys_meta_make:n #1
20953 {
```

```

20954 \__keys_cmd_set:Vo \l_keys_path_str
20955 {
20956   \exp_after:wN \keys_set:nn \exp_after:wN { \l__keys_module_str } {#1}
20957 }
20958 }
20959 \cs_new_protected:Npn \__keys_meta_make:nn #1#2
20960 { \__keys_cmd_set:Vn \l_keys_path_str { \keys_set:nn {#1} {#2} } }

```

(End definition for __keys_meta_make:n and __keys_meta_make:nn.)

__keys_prop_put:Nn Much the same as other variables, but needs a dedicated auxiliary.

```

\__keys_prop_put:cn
20961 \cs_new_protected:Npn \__keys_prop_put:Nn #1#2
20962 {
20963   \prop_if_exist:NF #1 { \prop_new:N #1 }
20964   \exp_after:wN \__keys_find_key_module:wNN \l_keys_path_str \s__keys_stop
20965   \l__keys_tmpa_tl \l__keys_tmpb_tl
20966   \__keys_cmd_set:nx \l_keys_path_str
20967   {
20968     \exp_not:c { prop_ #2 put:Nnn }
20969     \exp_not:N #1
20970     { \l__keys_tmpb_tl }
20971     \exp_not:n { {##1} }
20972   }
20973 }
20974 \cs_generate_variant:Nn \__keys_prop_put:Nn { c }

```

(End definition for __keys_prop_put:Nn.)

__keys_undefine: Undefined a key has to be done without \cs_undefine:c as that function acts globally.

```

20975 \cs_new_protected:Npn \__keys_undefine:
20976 {
20977   \clist_map_inline:nn
20978   { code , default , groups , inherit , type , check }
20979   {
20980     \cs_set_eq:cN
20981     { \tl_use:c { c__keys_ ##1 _root_str } \l_keys_path_str }
20982     \tex_undefined:D
20983   }
20984 }

```

(End definition for __keys_undefine:.)

__keys_value_requirement:nn Validating key input is done using a second function which runs before the main key code. Setting that up means setting it equal to a generic stub which does the check. This approach makes the lookup very fast at the cost of one additional csname per key that needs it. The cleanup here has to know the structure of the following code.

```

20985 \cs_new_protected:Npn \__keys_value_requirement:nn #1#2
20986 {
20987   \str_case:nnF {#2}
20988   {
20989     { true }
20990     {
20991       \cs_set_eq:cc
20992       { \c__keys_check_root_str \l_keys_path_str }

```

```

20993         { __keys_check_ #1 : }
20994     }
20995     { false }
20996     {
20997         \cs_if_eq:ccT
20998         { \c__keys_check_root_str \l_keys_path_str }
20999         { __keys_check_ #1 : }
21000         {
21001             \cs_set_eq:cN
21002             { \c__keys_check_root_str \l_keys_path_str }
21003             \tex_undefined:D
21004         }
21005     }
21006 }
21007 {
21008     \msg_error:nnx { keys }
21009     { boolean-values-only }
21010     { .value_ #1 :n }
21011 }
21012 }
21013 \cs_new_protected:Npn \__keys_check_forbidden:
21014 {
21015     \bool_if:NF \l__keys_no_value_bool
21016     {
21017         \msg_error:nnxx { keys } { value-forbidden }
21018         \l_keys_path_str \l_keys_value_tl
21019         \use_none:nnn
21020     }
21021 }
21022 \cs_new_protected:Npn \__keys_check_required:
21023 {
21024     \bool_if:NT \l__keys_no_value_bool
21025     {
21026         \msg_error:nnx { keys } { value-required }
21027         \l_keys_path_str
21028         \use_none:nnn
21029     }
21030 }

```

(End definition for `__keys_value_requirement:nn`, `__keys_check_forbidden:`, and `__keys_check_required:`.)

```

\__keys_usage:n Save the relevant data.
\__keys_usage:NN 21031 \cs_new_protected:Npn \__keys_usage:n #1
\__keys_usage:w 21032 {
21033     \str_case:nnF {#1}
21034     {
21035         { general }
21036         {
21037             \__keys_usage:NN \l_keys_usage_load_prop
21038             \c_false_bool
21039             \__keys_usage:NN \l_keys_usage_preamble_prop
21040             \c_false_bool
21041         }

```



```

21042     { load }
21043     {
21044         \__keys_usage:NN \l_keys_usage_load_prop
21045         \c_true_bool
21046         \__keys_usage:NN \l_keys_usage_preamble_prop
21047         \c_false_bool
21048     }
21049     { preamble }
21050     {
21051         \__keys_usage:NN \l_keys_usage_load_prop
21052         \c_false_bool
21053         \__keys_usage:NN \l_keys_usage_preamble_prop
21054         \c_true_bool
21055     }
21056 }
21057 {
21058     \msg_error:nnnn { keys }
21059     { choice-unknown }
21060     { .usage:n }
21061     {#1}
21062 }
21063 }
21064 \cs_new_protected:Npn \__keys_usage:NN #1#2
21065 {
21066     \prop_get:NVNF #1 \l__keys_module_str \l__keys_tmpa_tl
21067     { \tl_clear:N \l__keys_tmpa_tl }
21068     \tl_set:Nx \l__keys_tmpb_tl
21069     { \exp_after:wN \__keys_usage:w \l_keys_path_str \s__keys_stop }
21070     \bool_if:NTF #2
21071     { \clist_put_right:NV \l__keys_tmpa_tl \l__keys_tmpb_tl }
21072     { \clist_remove_all:NV \l__keys_tmpa_tl \l__keys_tmpb_tl }
21073     \prop_put:NVV #1 \l__keys_module_str
21074     \l__keys_tmpa_tl
21075 }
21076 \cs_new:Npn \__keys_usage:w #1 / #2 \s__keys_stop {#2}

```

(End definition for __keys_usage:n, __keys_usage:NN, and __keys_usage:w.)

__keys_variable_set:NnnN Setting a variable takes the type and scope separately so that it is easy to make a new
 __keys_variable_set:cnnN variable if needed.

```

\__keys_variable_set_required:NnnN
\__keys_variable_set_required:cnnN
21077 \cs_new_protected:Npn \__keys_variable_set:NnnN #1#2#3#4
21078 {
21079     \use:c { #2_if_exist:NF } #1 { \use:c { #2_new:N } #1 }
21080     \__keys_cmd_set:nx \l_keys_path_str
21081     {
21082         \exp_not:c { #2 _ #3 set:N #4 }
21083         \exp_not:N #1
21084         \exp_not:n { {#1} }
21085     }
21086 }
21087 \cs_generate_variant:Nn \__keys_variable_set:NnnN { c }
21088 \cs_new_protected:Npn \__keys_variable_set_required:NnnN #1#2#3#4
21089 {
21090     \__keys_variable_set:NnnN #1 {#2} {#3} #4

```

```

21091     \__keys_value_requirement:nn { required } { true }
21092   }
21093   \cs_generate_variant:Nn \__keys_variable_set_required:NnnN { c }

```

(End definition for __keys_variable_set:NnnN and __keys_variable_set_required:NnnN.)

63.5 Creating key properties

The key property functions are all wrappers for internal functions, meaning that things stay readable and can also be altered later on.

Importantly, while key properties have “normal” argument specs, the underlying code always supplies one braced argument to these. As such, argument expansion is handled by hand rather than using the standard tools. This shows up particularly for the two-argument properties, where things would otherwise go badly wrong.

```

. bool_set:N      One function for this.
. bool_set:c      21094 \cs_new_protected:cpn { \c__keys_props_root_str .bool_set:N } #1
. bool_gset:N      21095   { \__keys_bool_set:Nn #1 { } }
. bool_gset:c      21096 \cs_new_protected:cpn { \c__keys_props_root_str .bool_set:c } #1
                   21097   { \__keys_bool_set:cn {#1} { } }
                   21098 \cs_new_protected:cpn { \c__keys_props_root_str .bool_gset:N } #1
                   21099   { \__keys_bool_set:Nn #1 { g } }
                   21100 \cs_new_protected:cpn { \c__keys_props_root_str .bool_gset:c } #1
                   21101   { \__keys_bool_set:cn {#1} { g } }

```

(End definition for .bool_set:N and .bool_gset:N. These functions are documented on page 223.)

```

. bool_set_inverse:N One function for this.
. bool_set_inverse:c 21102 \cs_new_protected:cpn { \c__keys_props_root_str .bool_set_inverse:N } #1
. bool_gset_inverse:N 21103   { \__keys_bool_set_inverse:Nn #1 { } }
. bool_gset_inverse:c 21104 \cs_new_protected:cpn { \c__keys_props_root_str .bool_set_inverse:c } #1
                   21105   { \__keys_bool_set_inverse:cn {#1} { } }
                   21106 \cs_new_protected:cpn { \c__keys_props_root_str .bool_gset_inverse:N } #1
                   21107   { \__keys_bool_set_inverse:Nn #1 { g } }
                   21108 \cs_new_protected:cpn { \c__keys_props_root_str .bool_gset_inverse:c } #1
                   21109   { \__keys_bool_set_inverse:cn {#1} { g } }

```

(End definition for .bool_set_inverse:N and .bool_gset_inverse:N. These functions are documented on page 223.)

```

. choice: Making a choice is handled internally, as it is also needed by .generate_choices:n.
           21110 \cs_new_protected:cpn { \c__keys_props_root_str .choice: }
           21111   { \__keys_choice_make: }

```

(End definition for .choice:. This function is documented on page 223.)

```

. choices:nn For auto-generation of a series of mutually-exclusive choices. Here, #1 consists of two
. choices:Vn separate arguments, hence the slightly odd-looking implementation.
. choices:on 21112 \cs_new_protected:cpn { \c__keys_props_root_str .choices:nn } #1
. choices:xn 21113   { \__keys_choices_make:nn #1 }
           21114 \cs_new_protected:cpn { \c__keys_props_root_str .choices:Vn } #1
           21115   { \exp_args:NV \__keys_choices_make:nn #1 }
           21116 \cs_new_protected:cpn { \c__keys_props_root_str .choices:on } #1
           21117   { \exp_args:No \__keys_choices_make:nn #1 }

```

```

21118 \cs_new_protected:cpn { \c__keys_props_root_str .choices:xn } #1
21119 { \exp_args:Nx \__keys_choices_make:nn #1 }

```

(End definition for .choices:nn. This function is documented on page 223.)

.code:n Creating code is simply a case of passing through to the underlying set function.

```

21120 \cs_new_protected:cpn { \c__keys_props_root_str .code:n } #1
21121 { \__keys_cmd_set:nn \l_keys_path_str {#1} }

```

(End definition for .code:n. This function is documented on page 224.)

.clist_set:N

.clist_set:c

.clist_gset:N

.clist_gset:c

```

21122 \cs_new_protected:cpn { \c__keys_props_root_str .clist_set:N } #1
21123 { \__keys_variable_set:NnnN #1 { clist } { } n }
21124 \cs_new_protected:cpn { \c__keys_props_root_str .clist_set:c } #1
21125 { \__keys_variable_set:cnnN {#1} { clist } { } n }
21126 \cs_new_protected:cpn { \c__keys_props_root_str .clist_gset:N } #1
21127 { \__keys_variable_set:NnnN #1 { clist } { g } n }
21128 \cs_new_protected:cpn { \c__keys_props_root_str .clist_gset:c } #1
21129 { \__keys_variable_set:cnnN {#1} { clist } { g } n }

```

(End definition for .clist_set:N and .clist_gset:N. These functions are documented on page 223.)

.cs_set:Np

.cs_set:cp

.cs_set_protected:Np

.cs_set_protected:cp

.cs_gset:Np

.cs_gset:cp

.cs_gset_protected:Np

.cs_gset_protected:cp

```

21130 \cs_new_protected:cpn { \c__keys_props_root_str .cs_set:Np } #1
21131 { \__keys_cs_set:NNpn \cs_set:Npn #1 { } }
21132 \cs_new_protected:cpn { \c__keys_props_root_str .cs_set:cp } #1
21133 { \__keys_cs_set:Ncpn \cs_set:Npn #1 { } }
21134 \cs_new_protected:cpn { \c__keys_props_root_str .cs_set_protected:Np } #1
21135 { \__keys_cs_set:NNpn \cs_set_protected:Npn #1 { } }
21136 \cs_new_protected:cpn { \c__keys_props_root_str .cs_set_protected:cp } #1
21137 { \__keys_cs_set:Ncpn \cs_set_protected:Npn #1 { } }
21138 \cs_new_protected:cpn { \c__keys_props_root_str .cs_gset:Np } #1
21139 { \__keys_cs_set:NNpn \cs_gset:Npn #1 { } }
21140 \cs_new_protected:cpn { \c__keys_props_root_str .cs_gset:cp } #1
21141 { \__keys_cs_set:Ncpn \cs_gset:Npn #1 { } }
21142 \cs_new_protected:cpn { \c__keys_props_root_str .cs_gset_protected:Np } #1
21143 { \__keys_cs_set:NNpn \cs_gset_protected:Npn #1 { } }
21144 \cs_new_protected:cpn { \c__keys_props_root_str .cs_gset_protected:cp } #1
21145 { \__keys_cs_set:Ncpn \cs_gset_protected:Npn #1 { } }

```

(End definition for .cs_set:Np and others. These functions are documented on page 224.)

.default:n Expansion is left to the internal functions.

.default:V

.default:o

.default:x

```

21146 \cs_new_protected:cpn { \c__keys_props_root_str .default:n } #1
21147 { \__keys_default_set:n {#1} }
21148 \cs_new_protected:cpn { \c__keys_props_root_str .default:V } #1
21149 { \exp_args:NV \__keys_default_set:n #1 }
21150 \cs_new_protected:cpn { \c__keys_props_root_str .default:o } #1
21151 { \exp_args:No \__keys_default_set:n {#1} }
21152 \cs_new_protected:cpn { \c__keys_props_root_str .default:x } #1
21153 { \exp_args:Nx \__keys_default_set:n {#1} }

```

(End definition for .default:n. This function is documented on page 224.)

```

.dim_set:N Setting a variable is very easy: just pass the data along.
.dim_set:c 21154 \cs_new_protected:cpn { \c__keys_props_root_str .dim_set:N } #1
.dim_gset:N 21155 { \__keys_variable_set_required:NnnN #1 { dim } { } n }
.dim_gset:c 21156 \cs_new_protected:cpn { \c__keys_props_root_str .dim_set:c } #1
21157 { \__keys_variable_set_required:cnnN {#1} { dim } { } n }
21158 \cs_new_protected:cpn { \c__keys_props_root_str .dim_gset:N } #1
21159 { \__keys_variable_set_required:NnnN #1 { dim } { g } n }
21160 \cs_new_protected:cpn { \c__keys_props_root_str .dim_gset:c } #1
21161 { \__keys_variable_set_required:cnnN {#1} { dim } { g } n }

(End definition for .dim_set:N and .dim_gset:N. These functions are documented on page 224.)

.fp_set:N Setting a variable is very easy: just pass the data along.
.fp_set:c 21162 \cs_new_protected:cpn { \c__keys_props_root_str .fp_set:N } #1
.fp_gset:N 21163 { \__keys_variable_set_required:NnnN #1 { fp } { } n }
.fp_gset:c 21164 \cs_new_protected:cpn { \c__keys_props_root_str .fp_set:c } #1
21165 { \__keys_variable_set_required:cnnN {#1} { fp } { } n }
21166 \cs_new_protected:cpn { \c__keys_props_root_str .fp_gset:N } #1
21167 { \__keys_variable_set_required:NnnN #1 { fp } { g } n }
21168 \cs_new_protected:cpn { \c__keys_props_root_str .fp_gset:c } #1
21169 { \__keys_variable_set_required:cnnN {#1} { fp } { g } n }

(End definition for .fp_set:N and .fp_gset:N. These functions are documented on page 224.)

.groups:n A single property to create groups of keys.
21170 \cs_new_protected:cpn { \c__keys_props_root_str .groups:n } #1
21171 { \__keys_groups_set:n {#1} }

(End definition for .groups:n. This function is documented on page 224.)

.inherit:n Nothing complex: only one variant at the moment!
21172 \cs_new_protected:cpn { \c__keys_props_root_str .inherit:n } #1
21173 { \__keys_inherit:n {#1} }

(End definition for .inherit:n. This function is documented on page 225.)

.initial:n The standard hand-off approach.
.initial:V 21174 \cs_new_protected:cpn { \c__keys_props_root_str .initial:n } #1
.initial:o 21175 { \__keys_initialise:n {#1} }
.initial:x 21176 \cs_new_protected:cpn { \c__keys_props_root_str .initial:V } #1
21177 { \exp_args:NV \__keys_initialise:n #1 }
21178 \cs_new_protected:cpn { \c__keys_props_root_str .initial:o } #1
21179 { \exp_args:No \__keys_initialise:n {#1} }
21180 \cs_new_protected:cpn { \c__keys_props_root_str .initial:x } #1
21181 { \exp_args:Nx \__keys_initialise:n {#1} }

(End definition for .initial:n. This function is documented on page 225.)

.int_set:N Setting a variable is very easy: just pass the data along.
.int_set:c 21182 \cs_new_protected:cpn { \c__keys_props_root_str .int_set:N } #1
.int_gset:N 21183 { \__keys_variable_set_required:NnnN #1 { int } { } n }
.int_gset:c 21184 \cs_new_protected:cpn { \c__keys_props_root_str .int_set:c } #1
21185 { \__keys_variable_set_required:cnnN {#1} { int } { } n }
21186 \cs_new_protected:cpn { \c__keys_props_root_str .int_gset:N } #1
21187 { \__keys_variable_set_required:NnnN #1 { int } { g } n }
21188 \cs_new_protected:cpn { \c__keys_props_root_str .int_gset:c } #1
21189 { \__keys_variable_set_required:cnnN {#1} { int } { g } n }

```

(End definition for `.int_set:N` and `.int_gset:N`. These functions are documented on page 225.)

```
.legacy_if_set:n
.legacy_if_gset:n 21190 \cs_new_protected:cpn { \c__keys_props_root_str .legacy_if_set:n } #1
.legacy_if_set_inverse:n 21191 { \__keys_legacy_if_set:nn {#1} { } }
.legacy_if_gset_inverse:n 21192 \cs_new_protected:cpn { \c__keys_props_root_str .legacy_if_gset:n } #1
21193 { \__keys_legacy_if_set:nn {#1} { g } }
21194 \cs_new_protected:cpn { \c__keys_props_root_str .legacy_if_set_inverse:n } #1
21195 { \__keys_legacy_if_set_inverse:nn {#1} { } }
21196 \cs_new_protected:cpn { \c__keys_props_root_str .legacy_if_gset_inverse:n } #1
21197 { \__keys_legacy_if_set_inverse:nn {#1} { g } }
```

(End definition for `.legacy_if_set:n` and others. These functions are documented on page 225.)

.meta:n Making a meta is handled internally.

```
21198 \cs_new_protected:cpn { \c__keys_props_root_str .meta:n } #1
21199 { \__keys_meta_make:n {#1} }
```

(End definition for `.meta:n`. This function is documented on page 225.)

.meta:nn Meta with path: potentially lots of variants, but for the moment no so many defined.

```
21200 \cs_new_protected:cpn { \c__keys_props_root_str .meta:nn } #1
21201 { \__keys_meta_make:nn #1 }
```

(End definition for `.meta:nn`. This function is documented on page 225.)

.multichoice: The same idea as `.choice:` and `.choices:nn`, but where more than one choice is allowed.

```
.multichoices:nn 21202 \cs_new_protected:cpn { \c__keys_props_root_str .multichoice: }
21203 { \__keys_multichoice_make: }
.multichoices:Vn 21204 \cs_new_protected:cpn { \c__keys_props_root_str .multichoices:nn } #1
21205 { \__keys_multichoices_make:nn #1 }
.multichoices:on 21206 \cs_new_protected:cpn { \c__keys_props_root_str .multichoices:Vn } #1
21207 { \exp_args:NV \__keys_multichoices_make:nn #1 }
21208 \cs_new_protected:cpn { \c__keys_props_root_str .multichoices:on } #1
21209 { \exp_args:No \__keys_multichoices_make:nn #1 }
21210 \cs_new_protected:cpn { \c__keys_props_root_str .multichoices:xn } #1
21211 { \exp_args:Nx \__keys_multichoices_make:nn #1 }
```

(End definition for `.multichoice:` and `.multichoices:nn`. These functions are documented on page 226.)

.muskip_set:N Setting a variable is very easy: just pass the data along.

```
.muskip_set:c 21212 \cs_new_protected:cpn { \c__keys_props_root_str .muskip_set:N } #1
21213 { \__keys_variable_set_required:NnnN #1 { muskip } { } n }
.muskip_gset:N 21214 \cs_new_protected:cpn { \c__keys_props_root_str .muskip_set:c } #1
21215 { \__keys_variable_set_required:cnnN {#1} { muskip } { } n }
21216 \cs_new_protected:cpn { \c__keys_props_root_str .muskip_gset:N } #1
21217 { \__keys_variable_set_required:NnnN #1 { muskip } { g } n }
21218 \cs_new_protected:cpn { \c__keys_props_root_str .muskip_gset:c } #1
21219 { \__keys_variable_set_required:cnnN {#1} { muskip } { g } n }
```

(End definition for `.muskip_set:N` and `.muskip_gset:N`. These functions are documented on page 226.)

```

.prop_put:N Setting a variable is very easy: just pass the data along.
.prop_put:c 21220 \cs_new_protected:cpn { \c__keys_props_root_str .prop_put:N } #1
.prop_gput:N 21221 { \__keys_prop_put:Nn #1 { } }
.prop_gput:c 21222 \cs_new_protected:cpn { \c__keys_props_root_str .prop_put:c } #1
21223 { \__keys_prop_put:cn {#1} { } }
21224 \cs_new_protected:cpn { \c__keys_props_root_str .prop_gput:N } #1
21225 { \__keys_prop_put:Nn #1 { g } }
21226 \cs_new_protected:cpn { \c__keys_props_root_str .prop_gput:c } #1
21227 { \__keys_prop_put:cn {#1} { g } }

```

(End definition for .prop_put:N and .prop_gput:N. These functions are documented on page 226.)

```

.skip_set:N Setting a variable is very easy: just pass the data along.
.skip_set:c 21228 \cs_new_protected:cpn { \c__keys_props_root_str .skip_set:N } #1
.skip_gset:N 21229 { \__keys_variable_set_required:NnnN #1 { skip } { } n }
.skip_gset:c 21230 \cs_new_protected:cpn { \c__keys_props_root_str .skip_set:c } #1
21231 { \__keys_variable_set_required:cnnN {#1} { skip } { } n }
21232 \cs_new_protected:cpn { \c__keys_props_root_str .skip_gset:N } #1
21233 { \__keys_variable_set_required:NnnN #1 { skip } { g } n }
21234 \cs_new_protected:cpn { \c__keys_props_root_str .skip_gset:c } #1
21235 { \__keys_variable_set_required:cnnN {#1} { skip } { g } n }

```

(End definition for .skip_set:N and .skip_gset:N. These functions are documented on page 226.)

```

.str_set:N Setting a variable is very easy: just pass the data along.
.str_set:c 21236 \cs_new_protected:cpn { \c__keys_props_root_str .str_set:N } #1
.str_gset:N 21237 { \__keys_variable_set:NnnN #1 { str } { } n }
.str_gset:c 21238 \cs_new_protected:cpn { \c__keys_props_root_str .str_set:c } #1
.str_set_x:N 21239 { \__keys_variable_set:cnnN {#1} { str } { } n }
.str_set_x:c 21240 \cs_new_protected:cpn { \c__keys_props_root_str .str_set_x:N } #1
21241 { \__keys_variable_set:NnnN #1 { str } { } x }
.str_gset_x:N 21242 \cs_new_protected:cpn { \c__keys_props_root_str .str_set_x:c } #1
21243 { \__keys_variable_set:cnnN {#1} { str } { } x }
21244 \cs_new_protected:cpn { \c__keys_props_root_str .str_gset:N } #1
21245 { \__keys_variable_set:NnnN #1 { str } { g } n }
21246 \cs_new_protected:cpn { \c__keys_props_root_str .str_gset:c } #1
21247 { \__keys_variable_set:cnnN {#1} { str } { g } n }
21248 \cs_new_protected:cpn { \c__keys_props_root_str .str_gset_x:N } #1
21249 { \__keys_variable_set:NnnN #1 { str } { g } x }
21250 \cs_new_protected:cpn { \c__keys_props_root_str .str_gset_x:c } #1
21251 { \__keys_variable_set:cnnN {#1} { str } { g } x }

```

(End definition for .str_set:N and others. These functions are documented on page 226.)

```

.tl_set:N Setting a variable is very easy: just pass the data along.
.tl_set:c 21252 \cs_new_protected:cpn { \c__keys_props_root_str .tl_set:N } #1
.tl_gset:N 21253 { \__keys_variable_set:NnnN #1 { tl } { } n }
.tl_gset:c 21254 \cs_new_protected:cpn { \c__keys_props_root_str .tl_set:c } #1
21255 { \__keys_variable_set:cnnN {#1} { tl } { } n }
.tl_set_x:N 21256 \cs_new_protected:cpn { \c__keys_props_root_str .tl_set_x:N } #1
21257 { \__keys_variable_set:NnnN #1 { tl } { } x }
.tl_gset_x:N 21258 \cs_new_protected:cpn { \c__keys_props_root_str .tl_set_x:c } #1
21259 { \__keys_variable_set:cnnN {#1} { tl } { } x }
21260 \cs_new_protected:cpn { \c__keys_props_root_str .tl_gset:N } #1

```

```

21261 { \__keys_variable_set:NnnN #1 { t1 } { g } n }
21262 \cs_new_protected:cpn { \c__keys_props_root_str .tl_gset:c } #1
21263 { \__keys_variable_set:cnnN {#1} { t1 } { g } n }
21264 \cs_new_protected:cpn { \c__keys_props_root_str .tl_gset_x:N } #1
21265 { \__keys_variable_set:NnnN #1 { t1 } { g } x }
21266 \cs_new_protected:cpn { \c__keys_props_root_str .tl_gset_x:c } #1
21267 { \__keys_variable_set:cnnN {#1} { t1 } { g } x }

```

(End definition for .tl_set:N and others. These functions are documented on page 226.)

.undefine: Another simple wrapper.

```

21268 \cs_new_protected:cpn { \c__keys_props_root_str .undefine: }
21269 { \__keys_undefine: }

```

(End definition for .undefine:. This function is documented on page 227.)

.usage:n

```

21270 \cs_new_protected:cpn { \c__keys_props_root_str .usage:n } #1
21271 { \__keys_usage:n {#1} }

```

(End definition for .usage:n. This function is documented on page 230.)

.value_forbidden:n These are very similar, so both call the same function.

```

21272 \cs_new_protected:cpn { \c__keys_props_root_str .value_forbidden:n } #1
21273 { \__keys_value_requirement:nn { forbidden } {#1} }
21274 \cs_new_protected:cpn { \c__keys_props_root_str .value_required:n } #1
21275 { \__keys_value_requirement:nn { required } {#1} }

```

(End definition for .value_forbidden:n and .value_required:n. These functions are documented on page 227.)

63.6 Setting keys

\keys_set:nn A simple wrapper allowing for nesting.

```

\keys_set:nV 21276 \cs_new_protected:Npn \keys_set:nn #1#2
\keys_set:nv 21277 {
\keys_set:no 21278   \use:x
\__keys_set:nn 21279   {
\__keys_set:nnn 21280     \bool_set_false:N \exp_not:N \l__keys_only_known_bool
21281     \bool_set_false:N \exp_not:N \l__keys_filtered_bool
21282     \bool_set_false:N \exp_not:N \l__keys_selective_bool
21283     \tl_set:Nn \exp_not:N \l__keys_relative_tl
21284     { \exp_not:N \q__keys_no_value }
21285     \__keys_set:nn \exp_not:n { {#1} {#2} }
21286     \bool_if:NT \l__keys_only_known_bool
21287     { \bool_set_true:N \exp_not:N \l__keys_only_known_bool }
21288     \bool_if:NT \l__keys_filtered_bool
21289     { \bool_set_true:N \exp_not:N \l__keys_filtered_bool }
21290     \bool_if:NT \l__keys_selective_bool
21291     { \bool_set_true:N \exp_not:N \l__keys_selective_bool }
21292     \tl_set:Nn \exp_not:N \l__keys_relative_tl
21293     { \exp_not:o \l__keys_relative_tl }
21294   }
21295 }

```

```

21296 \cs_generate_variant:Nn \keys_set:nn { nV , nv , no }
21297 \cs_new_protected:Npn \__keys_set:nn #1#2
21298 { \exp_args:No \__keys_set:nnn \l__keys_module_str {#1} {#2} }
21299 \cs_new_protected:Npn \__keys_set:nnn #1#2#3
21300 {
21301   \str_set:Nx \l__keys_module_str { \__keys_trim_spaces:n {#2} }
21302   \keyval_parse:NNn \__keys_set_keyval:n \__keys_set_keyval:nn {#3}
21303   \str_set:Nn \l__keys_module_str {#1}
21304 }

```

(End definition for \keys_set:nn, __keys_set:nn, and __keys_set:nnn. This function is documented on page 230.)

```

\keys_set_known:nnN Setting known keys simply means setting the appropriate flag, then running the standard
\keys_set_known:nVN code. To allow for nested setting, any existing value of \l__keys_unused_clist is saved
\keys_set_known:nvN on the stack and reset afterwards. Note that for speed/simplicity reasons we use a tl
\keys_set_known:noN operation to set the clist here!
\keys_set_known:nnnN
\keys_set_known:nVnN
\keys_set_known:nvnN
\keys_set_known:nonN
\__keys_set_known:nnnnN
\keys_set_known:nn
\keys_set_known:nV
\keys_set_known:nv
\keys_set_known:no
\__keys_set_known:nnn
21305 \cs_new_protected:Npn \keys_set_known:nnN #1#2#3
21306 {
21307   \exp_args:No \__keys_set_known:nnnnN
21308   \l__keys_unused_clist \q__keys_no_value {#1} {#2} #3
21309 }
21310 \cs_generate_variant:Nn \keys_set_known:nnN { nV , nv , no }
21311 \cs_new_protected:Npn \keys_set_known:nnnN #1#2#3#4
21312 {
21313   \exp_args:No \__keys_set_known:nnnnN
21314   \l__keys_unused_clist {#3} {#1} {#2} #4
21315 }
21316 \cs_generate_variant:Nn \keys_set_known:nnnN { nV , nv , no }
21317 \cs_new_protected:Npn \__keys_set_known:nnnnN #1#2#3#4#5
21318 {
21319   \clist_clear:N \l__keys_unused_clist
21320   \__keys_set_known:nnn {#2} {#3} {#4}
21321   \__kernel_tl_set:Nx #5 { \exp_not:o \l__keys_unused_clist }
21322   \tl_set:Nn \l__keys_unused_clist {#1}
21323 }
21324 \cs_new_protected:Npn \keys_set_known:nn #1#2
21325 { \__keys_set_known:nnn \q__keys_no_value {#1} {#2} }
21326 \cs_generate_variant:Nn \keys_set_known:nn { nV , nv , no }
21327 \cs_new_protected:Npn \__keys_set_known:nnn #1#2#3
21328 {
21329   \use:x
21330   {
21331     \bool_set_true:N \exp_not:N \l__keys_only_known_bool
21332     \bool_set_false:N \exp_not:N \l__keys_filtered_bool
21333     \bool_set_false:N \exp_not:N \l__keys_selective_bool
21334     \tl_set:Nn \exp_not:N \l__keys_relative_tl { \exp_not:n {#1} }
21335     \__keys_set:nn \exp_not:n { {#2} {#3} }
21336     \bool_if:NF \l__keys_only_known_bool
21337     { \bool_set_false:N \exp_not:N \l__keys_only_known_bool }
21338     \bool_if:NT \l__keys_filtered_bool
21339     { \bool_set_true:N \exp_not:N \l__keys_filtered_bool }
21340     \bool_if:NT \l__keys_selective_bool
21341     { \bool_set_true:N \exp_not:N \l__keys_selective_bool }

```



```

21342         \tl_set:Nn \exp_not:N \l__keys_relative_tl
21343         { \exp_not:o \l__keys_relative_tl }
21344     }
21345 }

```

(End definition for `\keys_set_known:nnN` and others. These functions are documented on page 231.)

```

\keys_set_filter:nnnN The idea of setting keys in a selective manner again uses flags wrapped around the basic
\keys_set_filter:nnVN code. The comments on \keys_set_known:nnN also apply here. We have a bit more
\keys_set_filter:nnvN shuffling to do to keep everything nestable.
\keys_set_filter:nnoN
\keys_set_filter:nnnnN
\keys_set_filter:nnVnN
\keys_set_filter:nnvnN
\keys_set_filter:nnonN
\__keys_set_filter:nnnnnN
\keys_set_filter:nnn
\keys_set_filter:nnV
\keys_set_filter:nnv
\keys_set_filter:nno
\__keys_set_filter:nnnn
\keys_set_groups:nnn
\keys_set_groups:nnV
\keys_set_groups:nnv
\keys_set_groups:nno
\__keys_set_selective:nnn
\__keys_set_selective:nnnn
21346 \cs_new_protected:Npn \keys_set_filter:nnnN #1#2#3#4
21347 {
21348     \exp_args:No \__keys_set_filter:nnnnnN
21349     \l__keys_unused_clist
21350     \q__keys_no_value {#1} {#2} {#3} #4
21351 }
21352 \cs_generate_variant:Nn \keys_set_filter:nnnnN { nnV , nnv , nno }
21353 \cs_new_protected:Npn \keys_set_filter:nnnnN #1#2#3#4#5
21354 {
21355     \exp_args:No \__keys_set_filter:nnnnnN
21356     \l__keys_unused_clist {#4} {#1} {#2} {#3} #5
21357 }
21358 \cs_generate_variant:Nn \keys_set_filter:nnnnnN { nnV , nnv , nno }
21359 \cs_new_protected:Npn \__keys_set_filter:nnnnnN #1#2#3#4#5#6
21360 {
21361     \clist_clear:N \l__keys_unused_clist
21362     \__keys_set_filter:nnnn {#2} {#3} {#4} {#5}
21363     \__kernel_tl_set:Nx #6 { \exp_not:o \l__keys_unused_clist }
21364     \tl_set:Nn \l__keys_unused_clist {#1}
21365 }
21366 \cs_new_protected:Npn \keys_set_filter:nnn #1#2#3
21367 { \__keys_set_filter:nnnn \q__keys_no_value {#1} {#2} {#3} }
21368 \cs_generate_variant:Nn \keys_set_filter:nnn { nnV , nnv , nno }
21369 \cs_new_protected:Npn \__keys_set_filter:nnnn #1#2#3#4
21370 {
21371     \use:x
21372     {
21373         \bool_set_false:N \exp_not:N \l__keys_only_known_bool
21374         \bool_set_true:N \exp_not:N \l__keys_filtered_bool
21375         \bool_set_true:N \exp_not:N \l__keys_selective_bool
21376         \tl_set:Nn \exp_not:N \l__keys_relative_tl { \exp_not:n {#1} }
21377         \__keys_set_selective:nnn \exp_not:n { {#2} {#3} {#4} }
21378         \bool_if:NT \l__keys_only_known_bool
21379         { \bool_set_true:N \exp_not:N \l__keys_only_known_bool }
21380         \bool_if:NF \l__keys_filtered_bool
21381         { \bool_set_false:N \exp_not:N \l__keys_filtered_bool }
21382         \bool_if:NF \l__keys_selective_bool
21383         { \bool_set_false:N \exp_not:N \l__keys_selective_bool }
21384         \tl_set:Nn \exp_not:N \l__keys_relative_tl
21385         { \exp_not:o \l__keys_relative_tl }
21386     }
21387 }
21388 \cs_new_protected:Npn \keys_set_groups:nnn #1#2#3
21389 {

```

```

21390 \use:x
21391 {
21392     \bool_set_false:N \exp_not:N \l__keys_only_known_bool
21393     \bool_set_false:N \exp_not:N \l__keys_filtered_bool
21394     \bool_set_true:N \exp_not:N \l__keys_selective_bool
21395     \tl_set:Nn \exp_not:N \l__keys_relative_tl
21396         { \exp_not:N \q__keys_no_value }
21397     \__keys_set_selective:nnn \exp_not:n { {#1} {#2} {#3} }
21398     \bool_if:NT \l__keys_only_known_bool
21399         { \bool_set_true:N \exp_not:N \l__keys_only_known_bool }
21400     \bool_if:NF \l__keys_filtered_bool
21401         { \bool_set_true:N \exp_not:N \l__keys_filtered_bool }
21402     \bool_if:NF \l__keys_selective_bool
21403         { \bool_set_false:N \exp_not:N \l__keys_selective_bool }
21404     \tl_set:Nn \exp_not:N \l__keys_relative_tl
21405         { \exp_not:o \l__keys_relative_tl }
21406 }
21407 }
21408 \cs_generate_variant:Nn \keys_set_groups:nnn { nnV , nnv , nno }
21409 \cs_new_protected:Npn \__keys_set_selective:nnn
21410 { \exp_args:No \__keys_set_selective:nnnn \l__keys_selective_seq }
21411 \cs_new_protected:Npn \__keys_set_selective:nnnn #1#2#3#4
21412 {
21413     \seq_set_from_clist:Nn \l__keys_selective_seq {#3}
21414     \__keys_set:nn {#2} {#4}
21415     \tl_set:Nn \l__keys_selective_seq {#1}
21416 }

```

(End definition for `\keys_set_filter:nnnN` and others. These functions are documented on page 232.)

```

\__keys_set_keyval:n A shared system once again. First, set the current path and add a default if needed.
\__keys_set_keyval:nn There are then checks to see if the a value is required or forbidden. If everything passes,
\__keys_set_keyval:nnn move on to execute the code.
\__keys_set_keyval:onn
\__keys_find_key_module:wNN
\__keys_find_key_module_auxi:Nw
\__keys_find_key_module_auxii:Nw
\__keys_find_key_module_auxiii:Nn
\__keys_find_key_module_auxiv:Nw
\__keys_set_selective:
21417 \cs_new_protected:Npn \__keys_set_keyval:n #1
21418 {
21419     \bool_set_true:N \l__keys_no_value_bool
21420     \__keys_set_keyval:onn \l__keys_module_str {#1} { }
21421 }
21422 \cs_new_protected:Npn \__keys_set_keyval:nn #1#2
21423 {
21424     \bool_set_false:N \l__keys_no_value_bool
21425     \__keys_set_keyval:onn \l__keys_module_str {#1} {#2}
21426 }

```

The key path here can be fully defined, after which there is a search for the key and module names: the user may have passed them with part of what is actually the module (for our purposes) in the key name. As that happens on a per-key basis, we use the stack approach to restore the module name without a group.

```

21427 \cs_new_protected:Npn \__keys_set_keyval:nnn #1#2#3
21428 {
21429     \__kernel_tl_set:Nx \l__keys_path_str
21430     {
21431         \tl_if_blank:nF {#1}
21432         { #1 / }

```

```

21433         \__keys_trim_spaces:n {#2}
21434     }
21435     \str_clear:N \l__keys_module_str
21436     \str_clear:N \l__keys_inherit_str
21437     \exp_after:wN \__keys_find_key_module:wNN \l_keys_path_str \s__keys_stop
21438         \l__keys_module_str \l_keys_key_str
21439     \tl_set_eq:NN \l_keys_key_tl \l_keys_key_str
21440     \__keys_value_or_default:n {#3}
21441     \bool_if:NTF \l__keys_selective_bool
21442         \__keys_set_selective:
21443         \__keys_execute:
21444     \str_set:Nn \l__keys_module_str {#1}
21445 }
21446 \cs_generate_variant:Nn \__keys_set_keyval:nnn { o }

```

This function uses `\cs_set_nopar:Npx` internally for performance reasons, the argument #1 is already a string in every usage, so turning it into a string again seems unnecessary.

```

21447 \cs_new_protected:Npn \__keys_find_key_module:wNN #1 \s__keys_stop #2 #3
21448 {
21449     \__keys_find_key_module_auxi:Nw #2 #1 \s__keys_nil \__keys_find_key_module_auxii:Nw
21450     / \s__keys_nil \__keys_find_key_module_auxiv:Nw #3
21451 }
21452 \cs_new_protected:Npn \__keys_find_key_module_auxi:Nw #1 #2 / #3 \s__keys_nil #4
21453 {
21454     #4 #1 #2 \s__keys_mark #3 \s__keys_nil #4
21455 }
21456 \cs_new_protected:Npn \__keys_find_key_module_auxii:Nw
21457     #1 #2 \s__keys_mark #3 \s__keys_nil \__keys_find_key_module_auxiii:Nw
21458 {
21459     \cs_set_nopar:Npx #1 { \tl_if_empty:NF #1 { #1 / } #2 }
21460     \__keys_find_key_module_auxi:Nw #1 #3 \s__keys_nil \__keys_find_key_module_auxiii:Nw
21461 }
21462 \cs_new_protected:Npn \__keys_find_key_module_auxiii:Nw #1 #2 \s__keys_mark
21463 {
21464     \cs_set_nopar:Npx #1 { \tl_if_empty:NF #1 { #1 / } #2 }
21465     \__keys_find_key_module_auxi:Nw #1
21466 }
21467 \cs_new_protected:Npn \__keys_find_key_module_auxiv:Nw
21468     #1 #2 \s__keys_nil #3 \s__keys_mark
21469     \s__keys_nil \__keys_find_key_module_auxiv:Nw #4
21470 {
21471     \cs_set_nopar:Npn #4 { #2 }
21472 }

```

If selective setting is active, there are a number of possible sub-cases to consider. The key name may not be known at all or if it is, it may not have any groups assigned. There is then the question of whether the selection is opt-in or opt-out.

```

21473 \cs_new_protected:Npn \__keys_set_selective:
21474 {
21475     \cs_if_exist:cTF { \c__keys_groups_root_str \l_keys_path_str }
21476     {
21477         \clist_set_eq:Nc \l__keys_groups_clist
21478         { \c__keys_groups_root_str \l_keys_path_str }
21479         \__keys_check_groups:
21480     }

```

```

21481     {
21482         \bool_if:NTF \l__keys_filtered_bool
21483         \__keys_execute:
21484         \__keys_store_unused:
21485     }
21486 }

```

In the case where selective setting requires a comparison of the list of groups which apply to a key with the list of those which have been set active. That requires two mappings, and again a different outcome depending on whether opt-in or opt-out is set. We cannot replace the clist mapping by \clist_if_in:NnTF because catcodes may not be the same; they cannot be normalized easily in the clist because of the remote possibility that some items need braces if they involve commas or leading/trailing spaces.

```

21487 \cs_new_protected:Npn \__keys_check_groups:
21488 {
21489     \bool_set_false:N \l__keys_tmp_bool
21490     \seq_map_inline:Nn \l__keys_selective_seq
21491     {
21492         \clist_map_inline:Nn \l__keys_groups_clist
21493         {
21494             \str_if_eq:nnT {##1} {####1}
21495             {
21496                 \bool_set_true:N \l__keys_tmp_bool
21497                 \clist_map_break:n \seq_map_break:
21498             }
21499         }
21500     }
21501     \bool_if:NTF \l__keys_tmp_bool
21502     {
21503         \bool_if:NTF \l__keys_filtered_bool
21504         \__keys_store_unused:
21505         \__keys_execute:
21506     }
21507     {
21508         \bool_if:NTF \l__keys_filtered_bool
21509         \__keys_execute:
21510         \__keys_store_unused:
21511     }
21512 }

```

(End definition for __keys_set_keyval:n and others.)

__keys_value_or_default:n If a value is given, return it as #1, otherwise send a default if available.

```

\__keys_default_inherit:
21513 \cs_new_protected:Npn \__keys_value_or_default:n #1
21514 {
21515     \bool_if:NTF \l__keys_no_value_bool
21516     {
21517         \cs_if_exist:cTF { \c__keys_default_root_str \l_keys_path_str }
21518         {
21519             \tl_set_eq:Nc
21520             \l_keys_value_tl
21521             { \c__keys_default_root_str \l_keys_path_str }
21522         }
21523     }

```

```

21524         \tl_clear:N \l_keys_value_tl
21525         \cs_if_exist:cT
21526             { \c__keys_inherit_root_str \__keys_parent:o \l_keys_path_str }
21527             { \__keys_default_inherit: }
21528     }
21529 }
21530 { \tl_set:Nn \l_keys_value_tl {#1} }
21531 }
21532 \cs_new_protected:Npn \__keys_default_inherit:
21533 {
21534     \clist_map_inline:cn
21535         { \c__keys_inherit_root_str \__keys_parent:o \l_keys_path_str }
21536         {
21537             \cs_if_exist:cT
21538                 { \c__keys_default_root_str ##1 / \l_keys_key_str }
21539                 {
21540                     \tl_set_eq:Nc
21541                         \l_keys_value_tl
21542                         { \c__keys_default_root_str ##1 / \l_keys_key_str }
21543                     \clist_map_break:
21544                 }
21545             }
21546 }

```

(End definition for __keys_value_or_default:n and __keys_default_inherit:.)

__keys_execute: Actually executing a key is done in two parts. First, look for the key itself, then look for the **unknown** key with the same path. If both of these fail, complain. What exactly happens if a key is unknown depends on whether unknown keys are being skipped or if an error should be raised.

```

\__keys_execute:nn
\__keys_execute:no
\__keys_store_unused:
\__keys_store_unused_aux:
21547 \cs_new_protected:Npn \__keys_execute:
21548 {
21549     \cs_if_exist:cTF { \c__keys_code_root_str \l_keys_path_str }
21550     {
21551         \cs_if_exist_use:c { \c__keys_check_root_str \l_keys_path_str }
21552         \__keys_execute:no \l_keys_path_str \l_keys_value_tl
21553     }
21554     {
21555         \cs_if_exist:cTF
21556             { \c__keys_inherit_root_str \__keys_parent:o \l_keys_path_str }
21557             { \__keys_execute_inherit: }
21558             { \__keys_execute_unknown: }
21559     }
21560 }

```

To deal with the case where there is no hit, we leave __keys_execute_unknown: in the input stream and clean it up using the break function: that avoids needing a boolean.

```

21561 \cs_new_protected:Npn \__keys_execute_inherit:
21562 {
21563     \clist_map_inline:cn
21564         { \c__keys_inherit_root_str \__keys_parent:o \l_keys_path_str }
21565         {
21566             \cs_if_exist:cT
21567                 { \c__keys_code_root_str ##1 / \l_keys_key_str }

```

```

21568         {
21569             \str_set:Nn \l__keys_inherit_str {##1}
21570             \cs_if_exist_use:c { \c__keys_check_root_str ##1 / \l_keys_key_str }
21571             \__keys_execute:no { ##1 / \l_keys_key_str } \l_keys_value_tl
21572             \clist_map_break:n \use_none:n
21573         }
21574     }
21575     \__keys_execute_unknown:
21576 }
21577 \cs_new_protected:Npn \__keys_execute_unknown:
21578 {
21579     \bool_if:NTF \l__keys_only_known_bool
21580     { \__keys_store_unused: }
21581     {
21582         \cs_if_exist:cTF
21583         { \c__keys_code_root_str \l__keys_module_str / unknown }
21584         { \__keys_execute:no { \l__keys_module_str / unknown } \l_keys_value_tl }
21585         {
21586             \msg_error:nnxx { keys } { unknown }
21587             \l_keys_path_str \l__keys_module_str
21588         }
21589     }
21590 }

```

A key's code is in the control sequence with csname `\c__keys_code_root_str #1`. We expand it once to get the replacement text (with argument #2) and call `\use:n` with this replacement as its argument. This ensures that any undefined control sequence error in the key's code will lead to an error message of the form `<argument>...<control sequence>` in which one can read the (undefined) `<control sequence>` in full, rather than an error message that starts with the potentially very long key name, which would make the (undefined) `<control sequence>` be truncated or sometimes completely hidden. See <https://github.com/latex3/latex2e/issues/351>.

```

21591 \cs_new:Npn \__keys_execute:nn #1#2
21592 { \__keys_execute:no {#1} { \prg_do_nothing: #2 } }
21593 \cs_new:Npn \__keys_execute:no #1#2
21594 {
21595     \exp_args:NNo \exp_args:No \use:n
21596     {
21597         \cs:w \c__keys_code_root_str #1 \exp_after:wN \cs_end:
21598         \exp_after:wN {#2}
21599     }
21600 }

```

When there is no relative path, things here are easy: just save the key name and value. When we are working with a relative path, first we need to turn it into a string: that can't happen earlier as we need to store `\q__keys_no_value`. Then, use a standard delimited approach to fish out the partial path.

```

21601 \cs_new_protected:Npn \__keys_store_unused:
21602 {
21603     \__keys_quark_if_no_value:NTF \l__keys_relative_tl
21604     {
21605         \clist_put_right:Nx \l__keys_unused_clist
21606         {
21607             \l_keys_key_str

```

```

21608         \bool_if:NF \l__keys_no_value_bool
21609         { = { \exp_not:o \l_keys_value_tl } }
21610     }
21611 }
21612 {
21613     \tl_if_empty:NTF \l__keys_relative_tl
21614     {
21615         \clist_put_right:Nx \l__keys_unused_clist
21616         {
21617             \l_keys_path_str
21618             \bool_if:NF \l__keys_no_value_bool
21619             { = { \exp_not:o \l_keys_value_tl } }
21620         }
21621     }
21622     { \__keys_store_unused_aux: }
21623 }
21624 }
21625 \cs_new_protected:Npn \__keys_store_unused_aux:
21626 {
21627     \__kernel_tl_set:Nx \l__keys_relative_tl
21628     { \exp_args:No \__keys_trim_spaces:n \l__keys_relative_tl }
21629     \use:x
21630     {
21631         \cs_set_protected:Npn \__keys_store_unused:w
21632         #####1 \l__keys_relative_tl /
21633         #####2 \l__keys_relative_tl /
21634         #####3 \s__keys_stop
21635     }
21636     {
21637         \tl_if_blank:nF {##1}
21638         {
21639             \msg_error:nnxx { keys } { bad-relative-key-path }
21640             \l_keys_path_str
21641             \l__keys_relative_tl
21642         }
21643         \clist_put_right:Nx \l__keys_unused_clist
21644         {
21645             \exp_not:n {##2}
21646             \bool_if:NF \l__keys_no_value_bool
21647             { = { \exp_not:o \l_keys_value_tl } }
21648         }
21649     }
21650     \use:x
21651     {
21652         \__keys_store_unused:w \l_keys_path_str
21653         \l__keys_relative_tl / \l__keys_relative_tl /
21654         \s__keys_stop
21655     }
21656 }
21657 \cs_new_protected:Npn \__keys_store_unused:w { }

```

(End definition for __keys_execute: and others.)

__keys_choice_find:n Executing a choice has two parts. First, try the choice given, then if that fails call the
 __keys_choice_find:nn unknown key. That always exists, as it is created when a choice is first made. So there
 __keys_multichoice_find:n

is no need for any escape code. For multiple choices, the same code ends up used in a mapping.

```

21658 \cs_new:Npn \__keys_choice_find:n #1
21659 {
21660     \str_if_empty:NTF \l__keys_inherit_str
21661     { \__keys_choice_find:nn \l_keys_path_str {#1} }
21662     {
21663         \__keys_choice_find:nn
21664         { \l__keys_inherit_str / \l_keys_key_str } {#1}
21665     }
21666 }
21667 \cs_new:Npn \__keys_choice_find:nn #1#2
21668 {
21669     \cs_if_exist:cTF { \c__keys_code_root_str #1 / \__keys_trim_spaces:n {#2} }
21670     { \__keys_execute:nn { #1 / \__keys_trim_spaces:n {#2} } {#2} }
21671     { \__keys_execute:nn { #1 / unknown } {#2} }
21672 }
21673 \cs_new:Npn \__keys_multichoice_find:n #1
21674 { \clist_map_function:nN {#1} \__keys_choice_find:n }

```

(End definition for `__keys_choice_find:n`, `__keys_choice_find:nn`, and `__keys_multichoice_find:n`.)

63.7 Utilities

```

\__keys_parent:o      Used to strip off the ending part of the key path after the last /.
\__keys_parent_auxi:w 21675 \cs_new:Npn \__keys_parent:o #1
\__keys_parent_auxii:w 21676 {
\__keys_parent_auxiii:n 21677     \exp_after:wN \__keys_parent_auxi:w #1 \q_nil \__keys_parent_auxii:w
\__keys_parent_auxiv:w 21678     / \q_nil \__keys_parent_auxiv:w
21679 }
21680 \cs_new:Npn \__keys_parent_auxi:w #1 / #2 \q_nil #3
21681 {
21682     #3 { #1 } #2 \q_nil #3
21683 }
21684 \cs_new:Npn \__keys_parent_auxii:w #1 #2 \q_nil \__keys_parent_auxii:w
21685 {
21686     #1 \__keys_parent_auxi:w #2 \q_nil \__keys_parent_auxiii:n
21687 }
21688 \cs_new:Npn \__keys_parent_auxiii:n #1
21689 {
21690     / #1 \__keys_parent_auxi:w
21691 }
21692 \cs_new:Npn \__keys_parent_auxiv:w #1 \q_nil \__keys_parent_auxiv:w
21693 {
21694 }

```

(End definition for `__keys_parent:o` and others.)

```

\__keys_trim_spaces:n  Space stripping has to allow for the fact that the key here might have several parts, and
\__keys_trim_spaces_auxi:w spaces need to be stripped from each part. Since the key name is turned into a string
\__keys_trim_spaces_auxii:w groups can't be stripped accidentally and the precautions of \tl_trim_spaces:n aren't
\__keys_trim_spaces_auxiii:w necessary, in this case it is much faster to just directly strip spaces around /.

```



```

21695 \group_begin:
21696   \cs_set:Npn \__keys_tmp:w #1
21697     {
21698       \cs_new:Npn \__keys_trim_spaces:n ##1
21699         {
21700           \exp_after:wN \__keys_trim_spaces_auxi:w \tl_to_str:n { / ##1 } /
21701           \s__keys_nil \__keys_trim_spaces_auxi:w
21702           \s__keys_mark \__keys_trim_spaces_auxii:w
21703           #1 / #1
21704           \s__keys_nil \__keys_trim_spaces_auxii:w
21705           \s__keys_mark \__keys_trim_spaces_auxiii:w
21706         }
21707       }
21708   \__keys_tmp:w { ~ }
21709 \group_end:
21710 \cs_new:Npn \__keys_trim_spaces_auxi:w #1 ~ / #2 \s__keys_nil #3
21711   {
21712     #3 #1 / #2 \s__keys_nil #3
21713   }
21714 \cs_new:Npn \__keys_trim_spaces_auxii:w #1 / ~ #2 \s__keys_mark #3
21715   {
21716     #3 #1 / #2 \s__keys_mark #3
21717   }
21718 \cs_new:Npn \__keys_trim_spaces_auxiii:w
21719   / #1 /
21720   \s__keys_nil \__keys_trim_spaces_auxi:w
21721   \s__keys_mark \__keys_trim_spaces_auxii:w
21722   /
21723   \s__keys_nil \__keys_trim_spaces_auxii:w
21724   \s__keys_mark \__keys_trim_spaces_auxiii:w
21725   {
21726     #1
21727   }

```

(End definition for __keys_trim_spaces:n and others.)

\keys_if_exist_p:nn A utility for others to see if a key exists.

```

\keys_if_exist:nnTF 21728 \prg_new_conditional:Npnn \keys_if_exist:nn #1#2 { p , T , F , TF }
21729   {
21730     \cs_if_exist:cTF
21731       { \c__keys_code_root_str \__keys_trim_spaces:n { #1 / #2 } }
21732       { \prg_return_true: }
21733       { \prg_return_false: }
21734     }
21735 \prg_generate_conditional_variant:Nnn \keys_if_exist:nn { ne } { T , F , TF }

```

(End definition for \keys_if_exist:nnTF. This function is documented on page 233.)

\keys_if_choice_exist_p:nnn Just an alternative view on \keys_if_exist:nnTF.

```

\keys_if_choice_exist:nnnTF 21736 \prg_new_conditional:Npnn \keys_if_choice_exist:nnn #1#2#3
21737   { p , T , F , TF }
21738   {
21739     \cs_if_exist:cTF
21740       { \c__keys_code_root_str \__keys_trim_spaces:n { #1 / #2 / #3 } }
21741       { \prg_return_true: }

```

```

21742     { \prg_return_false: }
21743 }

```

(End definition for \keys_if_choice_exist:nnnTF. This function is documented on page 233.)

```

\keys_show:nn To show a key, show its code using a message.
\keys_log:nn
\__keys_show:Nnn
21744 \cs_new_protected:Npn \keys_show:nn
21745 { \__keys_show:Nnn \msg_show:nnxxxx }
21746 \cs_new_protected:Npn \keys_log:nn
21747 { \__keys_show:Nnn \msg_log:nnxxxx }
21748 \cs_new_protected:Npn \__keys_show:Nnn #1#2#3
21749 {
21750   #1 { keys } { show-key }
21751   { \__keys_trim_spaces:n { #2 / #3 } }
21752   {
21753     \keys_if_exist:nnT {#2} {#3}
21754     {
21755       \exp_args:Nnf \msg_show_item_unbraced:nn { code }
21756       {
21757         \exp_args:Nc \cs_replacement_spec:N
21758         {
21759           \c__keys_code_root_str
21760           \__keys_trim_spaces:n { #2 / #3 }
21761         }
21762       }
21763     }
21764   }
21765   { } { }
21766 }

```

(End definition for \keys_show:nn, \keys_log:nn, and __keys_show:Nnn. These functions are documented on page 233.)

63.8 Messages

For when there is a need to complain.

```

21767 \msg_new:nnnn { keys } { bad-relative-key-path }
21768 { The-key~'#1'-is-not-inside-the~'#2'-path. }
21769 { The-key~'#1'-cannot-be-expressed-relative-to-path~'#2'. }
21770 \msg_new:nnnn { keys } { boolean-values-only }
21771 { Key~'#1'~accepts-boolean-values-only. }
21772 { The-key~'#1'~only-accepts-the-values~'true'~and~'false'. }
21773 \msg_new:nnnn { keys } { choice-unknown }
21774 { Key~'#1'~accepts-only-a-fixed-set-of-choices. }
21775 {
21776   The-key~'#1'~only-accepts-predefined-values,~
21777   and~'#2'~is-not-one-of-these.
21778 }
21779 \msg_new:nnnn { keys } { unknown }
21780 { The-key~'#1'~is-unknown-and-is-being-ignored. }
21781 {
21782   The-module~'#2'~does-not-have-a-key-called~'#1'.\\
21783   Check-that-you-have-spelled-the-key-name-correctly.
21784 }

```

```

21785 \msg_new:nnnn { keys } { nested-choice-key }
21786 { Attempt-to-define-~'#1'~as-a-nested-choice-key. }
21787 {
21788   The-key~'#1'~cannot-be-defined-as-a-choice-as-the-parent-key~'#2'~is~
21789   itself-a-choice.
21790 }
21791 \msg_new:nnnn { keys } { value-forbidden }
21792 { The-key~'#1'~does-not-take-a-value. }
21793 {
21794   The-key~'#1'~should-be-given-without-a-value.\\
21795   The-value~'#2'~was-present:~the-key-will-be-ignored.
21796 }
21797 \msg_new:nnnn { keys } { value-required }
21798 { The-key~'#1'~requires-a-value. }
21799 {
21800   The-key~'#1'~must-have-a-value.\\
21801   No-value-was-present:~the-key-will-be-ignored.
21802 }
21803 \msg_new:nnn { keys } { show-key }
21804 {
21805   The-key~#1~
21806   \tl_if_empty:nTF {#2}
21807     { is-undefined. }
21808     { has-the-properties: #2 . }
21809 }
21810 \prop_gput:Nnn \g_msg_module_name_prop { keys } { LaTeX3 }
21811 \prop_gput:Nnn \g_msg_module_type_prop { keys } { }
21812 </package>

```

Chapter 64

l3intarray implementation

21813 $\langle *package \rangle$

21814 $\langle @@=intarray \rangle$

There are two implementations for this module: One `\fontdimen` based one for more traditional T_EX engines and a Lua based one for engines with Lua support.

Both versions do not allow negative array sizes.

21815 $\langle *tex \rangle$

21816 $\backslash msg_new:nnn \{ kernel \} \{ negative-array-size \}$

21817 $\{ Size-of-array-may-not-be-negative:~\#1 \}$

$\backslash l_intarray_loop_int$ A loop index.

21818 $\backslash int_new:N \backslash l_intarray_loop_int$

(End definition for $\backslash l_intarray_loop_int$.)

64.1 Lua implementation

First, let's look at the Lua variant:

We select the Lua version if the Lua helpers were defined. This can be detected by the presence of $\backslash_intarray_gset_count:Nw$.

21819 $\backslash cs_if_exist:NTF \backslash_intarray_gset_count:Nw$

21820 $\{$

64.1.1 Allocating arrays

$\backslash g_intarray_table_int$
 $\backslash l_intarray_bad_index_int$

Used to differentiate intarrays in Lua and to record an invalid index.

21821 $\backslash int_new:N \backslash g_intarray_table_int$

21822 $\backslash int_new:N \backslash l_intarray_bad_index_int$

21823 $\langle /tex \rangle$

(End definition for $\backslash g_intarray_table_int$ and $\backslash l_intarray_bad_index_int$.)

$\backslash_intarray:w$ Used as marker for intarrays in Lua. Followed by an unbraced number identifying the array and a single space. This format is used to make it easy to scan from Lua.

21824 $\langle *lua \rangle$

21825 $luacmd('_intarray:w', function()$

21826 $scan_int()$

```

21827 tex.error'LaTeX Error: Isolated intarray ignored'
21828 end, 'protected', 'global')
21829 </lua>

```

(End definition for `_intarray:w`.)

`\intarray_new:Nn` Declare #1 as a tokenlist with the scanmark and a unique number. Pass the array's size to the Lua helper. Every `intarray` must be global; it's enough to run this check in `\intarray_new:Nn`.

```

21830 (*tex)
21831 \cs_new_protected:Npn \_intarray_new:N #1
21832 {
21833   \_kernel_chk_if_free_cs:N #1
21834   \int_gincr:N \g\_intarray_table_int
21835   \cs_gset_nopar:Npx #1 { \_intarray:w \int_use:N \g\_intarray_table_int \c_space_tl
21836 }
21837 \cs_new_protected:Npn \intarray_new:Nn #1#2
21838 {
21839   \_intarray_new:N #1
21840   \_intarray_gset_count:Nw #1 \int_eval:n {#2} \scan_stop:
21841   \int_compare:nNnT { \intarray_count:N #1 } < 0
21842   {
21843     \msg_error:nxx { kernel } { negative-array-size }
21844     { \intarray_count:N #1 }
21845   }
21846 }
21847 \cs_generate_variant:Nn \intarray_new:Nn { c }
21848 </tex>

```

(End definition for `\intarray_new:Nn` and `_intarray_new:N`. This function is documented on page 236.)

Before we get to the first command implemented in Lua, we first need some definitions. Since `token.create` only works correctly if `TEX` has seen the tokens before, we first run a short `TEX` sequence to ensure that all relevant control sequences are known.

```

21849 (*lua)
21850
21851 local scan_token = token.scan_token
21852 local put_next = token.put_next
21853 local intarray_marker = token_create_safe'_intarray:w'
21854 local use_none = token_create_safe'use_none:n'
21855 local use_i = token_create_safe'use:n'
21856 local expand_after_scan_stop = {token_create_safe'exp_after:wN',
21857                                token_create_safe'scan_stop:'}
21858 local comma = token_create(string.byte,',')

```

`_intarray_table` Internal helper to scan an `intarray` token, extract the associated Lua table and return an error if the input is invalid.

```

21859 local \_intarray_table do
21860   local tables = get_lua_data and get_lua_data'_intarray' or {[0] = {}}
21861   function \_intarray_table()
21862     local t = scan_token()
21863     if t ~= intarray_marker then
21864       put_next(t)
21865       tex.error'LaTeX Error: intarray expected'

```

```

21866     return tables[0]
21867 end
21868 local i = scan_int()
21869 local current_table = tables[i]
21870 if current_table then return current_table end
21871 current_table = {}
21872 tables[i] = current_table
21873 return current_table
21874 end

```

Since in L^AT_EX this is loaded in the format, we want to preserve any intarrays which are created while format building for the actual run.

To do this, we use the `register_luadata` mechanism from l3luatex: Directly before the format get dumped, the following function gets invoked and serializes all existing tables into a string. This string gets compiled and dumped into the format and is made available at the beginning of regular runs as `get_luadata'@@'`.

```

21875 if register_luadata then
21876   register_luadata('__intarray', function()
21877     local t = "{[0]={},"
21878     for i=1, #tables do
21879       t = string.format("%s{%s},"", t, table.concat(tables[i], ', '))
21880     end
21881     return t .. "}"
21882   end)
21883 end
21884 end

```

(End definition for `__intarray_table`.)

`\intarray_count:N` Set and get the size of an array. “Setting the size” means in this context that we add
`\intarray_count:c` zeros until we reach the desired size.
`__intarray_gset_count:Nw`

```

21885
21886 local sprint = tex.sprint
21887
21888 luacmd('__intarray_gset_count:Nw', function()
21889   local t = __intarray_table()
21890   local n = scan_int()
21891   for i=#t+1, n do t[i] = 0 end
21892 end, 'protected', 'global')
21893
21894 luacmd('intarray_count:N', function()
21895   sprint(-2, #__intarray_table())
21896 end, 'global')
21897 </lua>
21898 <*tex>
21899   \cs_generate_variant:Nn \intarray_count:N { c }
21900 </tex>

```

(End definition for `\intarray_count:N` and `__intarray_gset_count:Nw`. This function is documented on page 236.)

64.1.2 Array items

`__intarray_gset:wF` The setter provided by Lua. The argument order somewhat emulates the `\fontdimen:`
`__intarray_gset:w` First the array index, followed by the intarray and then the new value. This has been
 chosen over a more conventional order to provide a delimiter for the numbers.

```

21901  $\langle$ *lua $\rangle$ 
21902 luacmd('__intarray_gset:wF', function()
21903   local i = scan_int()
21904   local t = __intarray_table()
21905   if t[i] then
21906     t[i] = scan_int()
21907     put_next(use_none)
21908   else
21909     tex.count.l__intarray_bad_index_int = i
21910     scan_int()
21911     put_next(use_i)
21912   end
21913 end, 'protected', 'global')
21914
21915 luacmd('__intarray_gset:w', function()
21916   local i = scan_int()
21917   local t = __intarray_table()
21918   t[i] = scan_int()
21919 end, 'protected', 'global')
21920  $\langle$ /lua $\rangle$ 

```

(End definition for `__intarray_gset:wF` and `__intarray_gset:w`.)

`\intarray_gset:Nnn` The `__kernel_intarray_gset:Nnn` function does not use `\int_eval:n`, namely its
`\intarray_gset:cnn` arguments must be suitable for `\int_value:w`. The user version checks the position and
`__kernel_intarray_gset:Nnn` value are within bounds.

```

21921  $\langle$ *tex $\rangle$ 
21922   \cs_new_protected:Npn \__kernel_intarray_gset:Nnn #1#2#3
21923     { \__intarray_gset:w #2 #1 #3 \scan_stop: }
21924   \cs_new_protected:Npn \intarray_gset:Nnn #1#2#3
21925     {
21926       \__intarray_gset:wF \int_eval:n {#2} #1 \int_eval:n{#3}
21927       {
21928         \msg_error:nnxxx { kernel } { out-of-bounds }
21929         { \token_to_str:N #1 } { \int_use:N \l__intarray_bad_index_int } { \intarray_
21930       }
21931     }
21932   \cs_generate_variant:Nn \intarray_gset:Nnn { c }
21933  $\langle$ /tex $\rangle$ 

```

(End definition for `\intarray_gset:Nnn` and `__kernel_intarray_gset:Nnn`. This function is documented on page 236.)

`\intarray_gzero:N` Set the appropriate array entry to zero. No bound checking needed.
`\intarray_gzero:c`

```

21934  $\langle$ *lua $\rangle$ 
21935 luacmd('intarray_gzero:N', function()
21936   local t = __intarray_table()
21937   for i=1, #t do
21938     t[i] = 0

```

```

21939     end
21940 end, 'global', 'protected')
21941  $\langle$ /lua $\rangle$ 
21942  $\langle$ *tex $\rangle$ 
21943     \cs_generate_variant:Nn \intarray_gzero:N { c }
21944  $\langle$ /tex $\rangle$ 

```

(End definition for `\intarray_gzero:N`. This function is documented on page 237.)

```

\intarray_item:Nn Get the appropriate entry and perform bound checks. The \__kernel_intarray_
\intarray_item:cn item:Nn function omits bound checks and omits \int_eval:n, namely its argument
\__kernel_intarray_item:Nn must be a TeX integer suitable for \int_value:w.
\__intarray_item:wF
\__intarray_item:w
21945  $\langle$ *lua $\rangle$ 
21946 luacmd('__intarray_item:wF', function()
21947     local i = scan_int()
21948     local t = __intarray_table()
21949     local item = t[i]
21950     if item then
21951         put_next(use_none)
21952     else
21953         tex.l__intarray_bad_index_int = i
21954         put_next(use_i)
21955     end
21956     put_next(expand_after_scan_stop)
21957     scan_token()
21958     if item then
21959         sprint(-2, item)
21960     end
21961 end, 'global')
21962
21963 luacmd('__intarray_item:w', function()
21964     local i = scan_int()
21965     local t = __intarray_table()
21966     sprint(-2, t[i])
21967 end, 'global')
21968  $\langle$ /lua $\rangle$ 
21969  $\langle$ *tex $\rangle$ 
21970     \cs_new:Npn \__kernel_intarray_item:Nn #1#2
21971         { \__intarray_item:w #2 #1 }
21972     \cs_new:Npn \intarray_item:Nn #1#2
21973         {
21974             \__intarray_item:wF \int_eval:n {#2} #1
21975             {
21976                 \msg_expandable_error:nnfff { kernel } { out-of-bounds }
21977                 { \token_to_str:N #1 } { \int_use:N \l__intarray_bad_index_int } { \intarray_
21978                     0
21979             }
21980         }
21981     \cs_generate_variant:Nn \intarray_item:Nn { c }

```

(End definition for `\intarray_item:Nn` and others. This function is documented on page 237.)

`\intarray_rand_item:N` Importantly, `\intarray_item:Nn` only evaluates its argument once.
`\intarray_rand_item:c`

```

21982     \cs_new:Npn \intarray_rand_item:N #1

```



```

21983     { \intarray_item:Nn #1 { \int_rand:n { \intarray_count:N #1 } } }
21984     \cs_generate_variant:Nn \intarray_rand_item:N { c }

```

(End definition for `\intarray_rand_item:N`. This function is documented on page 237.)

64.1.3 Working with contents of integer arrays

`\intarray_const_from_clist:Nn` We use the `__kernel_intarray_gset:Nnn` which does not do bounds checking and instead automatically resizes the array. This is not implemented in Lua to ensure that the clist parsing is consistent with the clist module.

```

21985     \cs_new_protected:Npn \intarray_const_from_clist:Nn #1#2
21986     {
21987         \__intarray_new:N #1
21988         \int_zero:N \l__intarray_loop_int
21989         \clist_map_inline:nn {#2}
21990         {
21991             \int_incr:N \l__intarray_loop_int
21992             \__kernel_intarray_gset:Nnn #1 \l__intarray_loop_int { \int_eval:n {##1} } }
21993     }
21994     \cs_generate_variant:Nn \intarray_const_from_clist:Nn { c }

```

(End definition for `\intarray_const_from_clist:Nn`. This function is documented on page 237.)

`\intarray_to_clist:N` The `__intarray_to_clist:Nn` auxiliary allows to choose the delimiter and is also used in `\intarray_show:N`. Here we just pass the information to Lua and let `table.concat` do the actual work. We discard the category codes of the passed delimiter but this is not an issue since the delimiter is always just a comma or a comma and a space. In both cases `sprint(2, ...)` provides the right catcodes.

```

21995     \cs_new:Npn \intarray_to_clist:N #1 { \__intarray_to_clist:Nn #1 { , } }
21996     \cs_generate_variant:Nn \intarray_to_clist:N { c }
21997     \</tex>
21998     \<lua>
21999     local concat = table.concat
22000     luacmd('\__intarray_to_clist:Nn', function()
22001         local t = __intarray_table()
22002         local sep = token.scan_string()
22003         sprint(-2, concat(t, sep))
22004     end, 'global')
22005     \</lua>

```

(End definition for `\intarray_to_clist:N`, `__intarray_to_clist:Nn`, and `__intarray_to_clist:w`. This function is documented on page 302.)

`__kernel_intarray_range_to_clist:Nnn` Loop through part of the array.

```

22006     \<tex>
22007     \cs_new:Npn \__kernel_intarray_range_to_clist:Nnn #1#2#3
22008     {
22009         \__intarray_range_to_clist:w #1
22010         \int_eval:n {#2} ~ \int_eval:n {#3} ~
22011     }
22012     \</tex>
22013     \<lua>
22014     luacmd('\__intarray_range_to_clist:w', function()
22015         local t = __intarray_table()

```

```

22016   local from = scan_int()
22017   local to = scan_int()
22018   sprint(-2, concat(t, ',', from, to))
22019 end, 'global')
22020  $\langle$ /lua $\rangle$ 

```

(End definition for `_kernel_intarray_range_to_clist:Nnn` and `_intarray_range_to_clist:w`.)

`_kernel_intarray_gset_range_from_clist:Nnn`
`_intarray_gset_range:nNw`

Loop through part of the array. We allow additional commas at the end.

```

22021  $\langle$ *tex $\rangle$ 
22022   \cs_new_protected:Npn \_kernel_intarray_gset_range_from_clist:Nnn #1#2#3
22023   {
22024     \_intarray_gset_range:w \int_eval:w #2 #1 #3 , , \scan_stop:
22025   }
22026  $\langle$ /tex $\rangle$ 
22027  $\langle$ *lua $\rangle$ 
22028   luacmd('\_intarray_gset_range:w', function()
22029     local from = scan_int()
22030     local t = \_intarray_table()
22031     while true do
22032       local tok = scan_token()
22033       if tok == comma then
22034         repeat
22035           tok = scan_token()
22036         until tok ~= comma
22037         break
22038       else
22039         put_next(tok)
22040       end
22041       t[from] = scan_int()
22042       scan_token()
22043       from = from + 1
22044     end
22045     end, 'global', 'protected')
22046  $\langle$ /lua $\rangle$ 

```

(End definition for `_kernel_intarray_gset_range_from_clist:Nnn` and `_intarray_gset_range:nNw`.)

`_intarray_gset_overflow_test:nw`

In order to allow some code sharing later we provide the `_intarray_gset_overflow_test:nw` name here. It doesn't actually test anything since the Lua implementation accepts all integers which could be tested with `\tex_ifabsnum:D`.

```

22047  $\langle$ *tex $\rangle$ 
22048   \cs_new_protected:Npn \_intarray_gset_overflow_test:nw #1
22049   {
22050   }

```

(End definition for `_intarray_gset_overflow_test:nw`.)

64.2 Font dimension based implementation

Go to the false branch of the conditional above.

```

22051   }
22052   {

```

64.2.1 Allocating arrays

<code>__intarray_entry:w</code>	We use these primitives quite a lot in this module.
<code>__intarray_count:w</code>	<pre> 22053 \cs_new_eq:NN __intarray_entry:w \tex_fontdimen:D 22054 \cs_new_eq:NN __intarray_count:w \tex_hyphenchar:D </pre> <p>(End definition for <code>__intarray_entry:w</code> and <code>__intarray_count:w</code>.)</p>
<code>\c__intarray_sp_dim</code>	Used to convert integers to dimensions fast. <pre> 22055 \dim_const:Nn \c__intarray_sp_dim { 1 sp } </pre> <p>(End definition for <code>\c__intarray_sp_dim</code>.)</p>
<code>\g__intarray_font_int</code>	Used to assign one font per array. <pre> 22056 \int_new:N \g__intarray_font_int </pre> <p>(End definition for <code>\g__intarray_font_int</code>.)</p>
<code>\intarray_new:Nn</code> <code>\intarray_new:cn</code> <code>__intarray_new:N</code>	<p>Declare <code>#1</code> to be a font (arbitrarily <code>cmr10</code> at a never-used size). Store the array's size as the <code>\hyphenchar</code> of that font and make sure enough <code>\fontdimen</code> are allocated, by setting the last one. Then clear any <code>\fontdimen</code> that <code>cmr10</code> starts with. It seems LuaTeX's <code>cmr10</code> has an extra <code>\fontdimen</code> parameter number 8 compared to other engines (for a math font we would replace 8 by 22 or some such). Every <code>intarray</code> must be global; it's enough to run this check in <code>\intarray_new:Nn</code>.</p> <pre> 22057 \cs_new_protected:Npn __intarray_new:N #1 22058 { 22059 __kernel_chk_if_free_cs:N #1 22060 \int_gincr:N \g__intarray_font_int 22061 \tex_global:D \tex_font:D #1 22062 = cmr10~at~ \g__intarray_font_int \c__intarray_sp_dim \scan_stop: 22063 \int_step_inline:nn { 8 } 22064 { __kernel_intarray_gset:Nnn #1 {##1} \c_zero_int } 22065 } 22066 \cs_new_protected:Npn \intarray_new:Nn #1#2 22067 { 22068 __intarray_new:N #1 22069 __intarray_count:w #1 = \int_eval:n {#2} \scan_stop: 22070 \int_compare:nNnT { \intarray_count:N #1 } < 0 22071 { 22072 \msg_error:nxx { kernel } { negative-array-size } 22073 { \intarray_count:N #1 } 22074 } 22075 \int_compare:nNnT { \intarray_count:N #1 } > 0 22076 { __kernel_intarray_gset:Nnn #1 { \intarray_count:N #1 } { 0 } } 22077 } 22078 \cs_generate_variant:Nn \intarray_new:Nn { c } </pre> <p>(End definition for <code>\intarray_new:Nn</code> and <code>__intarray_new:N</code>. This function is documented on page 236.)</p>
<code>\intarray_count:N</code> <code>\intarray_count:c</code>	<p>Size of an array.</p> <pre> 22079 \cs_new:Npn \intarray_count:N #1 { \int_value:w __intarray_count:w #1 } 22080 \cs_generate_variant:Nn \intarray_count:N { c } </pre> <p>(End definition for <code>\intarray_count:N</code>. This function is documented on page 236.)</p>

64.2.2 Array items

`__intarray_signed_max_dim:n` Used when an item to be stored is larger than `\c_max_dim` in absolute value; it is replaced by $\pm\c_max_dim$.

```
22081 \cs_new:Npn \__intarray_signed_max_dim:n #1
22082 { \int_value:w \int_compare:nNnT {#1} < 0 { - } \c_max_dim }
```

(End definition for `__intarray_signed_max_dim:n`.)

`__intarray_bounds:NNnTF` The functions `\intarray_gset:Nnn` and `\intarray_item:Nn` share bounds checking. `__intarray_bounds_error:NNnw` The T branch is used if #3 is within bounds of the array #2.

```
22083 \cs_new:Npn \__intarray_bounds:NNnTF #1#2#3
22084 {
22085   \if_int_compare:w 1 > #3 \exp_stop_f:
22086   \__intarray_bounds_error:NNnw #1 #2 {#3}
22087   \else:
22088     \if_int_compare:w #3 > \intarray_count:N #2 \exp_stop_f:
22089     \__intarray_bounds_error:NNnw #1 #2 {#3}
22090   \fi:
22091   \fi:
22092   \use_i:nn
22093 }
22094 \cs_new:Npn \__intarray_bounds_error:NNnw #1#2#3#4 \use_i:nn #5#6
22095 {
22096   #4
22097   #1 { kernel } { out-of-bounds }
22098   { \token_to_str:N #2 } {#3} { \intarray_count:N #2 }
22099   #6
22100 }
```

(End definition for `__intarray_bounds:NNnTF` and `__intarray_bounds_error:NNnw`.)

`\intarray_gset:Nnn`

`\intarray_gset:cnm`

`__kernel_intarray_gset:Nnn`

`__intarray_gset:Nnn`

`__intarray_gset_overflow:Nnn`

Set the appropriate `\fontdimen`. The `__kernel_intarray_gset:Nnn` function does not use `\int_eval:n`, namely its arguments must be suitable for `\int_value:w`. The user version checks the position and value are within bounds.

```
22101 \cs_new_protected:Npn \__kernel_intarray_gset:Nnn #1#2#3
22102 { \__intarray_entry:w #2 #1 #3 \c__intarray_sp_dim }
22103 \cs_new_protected:Npn \intarray_gset:Nnn #1#2#3
22104 {
22105   \exp_after:wN \__intarray_gset:Nww
22106   \exp_after:wN #1
22107   \int_value:w \int_eval:n {#2} \exp_after:wN ;
22108   \int_value:w \int_eval:n {#3} ;
22109 }
22110 \cs_generate_variant:Nn \intarray_gset:Nnn { c }
22111 \cs_new_protected:Npn \__intarray_gset:Nnw #1#2 ; #3 ;
22112 {
22113   \__intarray_bounds:NNnTF \msg_error:nnxxx #1 {#2}
22114   {
22115     \__intarray_gset_overflow_test:nw {#3}
22116     \__kernel_intarray_gset:Nnn #1 {#2} {#3}
22117   }
22118   { }
22119 }
```

```

22120 \cs_if_exist:NTF \tex_ifabsnum:D
22121 {
22122   \cs_new_protected:Npn \__intarray_gset_overflow_test:nw #1
22123   {
22124     \tex_ifabsnum:D #1 > \c_max_dim
22125     \exp_after:wN \__intarray_gset_overflow:NNnn
22126     \fi:
22127   }
22128 }
22129 {
22130   \cs_new_protected:Npn \__intarray_gset_overflow_test:nw #1
22131   {
22132     \if_int_compare:w \int_abs:n {#1} > \c_max_dim
22133     \exp_after:wN \__intarray_gset_overflow:NNnn
22134     \fi:
22135   }
22136 }
22137 \cs_new_protected:Npn \__intarray_gset_overflow:NNnn #1#2#3#4
22138 {
22139   \msg_error:nnxxxx { kernel } { overflow }
22140   { \token_to_str:N #2 } {#3} {#4} { \__intarray_signed_max_dim:n {#4} }
22141   #1 #2 {#3} { \__intarray_signed_max_dim:n {#4} }
22142 }

```

(End definition for `\intarray_gset:Nnn` and others. This function is documented on page 236.)

`\intarray_gzero:N` Set the appropriate `\fontdimen` to zero. No bound checking needed. The `\prg_replicate:nn` possibly uses quite a lot of memory, but this is somewhat comparable to the size of the array, and it is much faster than an `\int_step_inline:nn` loop.

`\intarray_gzero:c`

```

22143 \cs_new_protected:Npn \intarray_gzero:N #1
22144 {
22145   \int_zero:N \l__intarray_loop_int
22146   \prg_replicate:nn { \intarray_count:N #1 }
22147   {
22148     \int_incr:N \l__intarray_loop_int
22149     \__intarray_entry:w \l__intarray_loop_int #1 \c_zero_dim
22150   }
22151 }
22152 \cs_generate_variant:Nn \intarray_gzero:N { c }

```

(End definition for `\intarray_gzero:N`. This function is documented on page 237.)

`\intarray_item:Nn` Get the appropriate `\fontdimen` and perform bound checks. The `__kernel_intarray_item:Nn` function omits bound checks and omits `\int_eval:n`, namely its argument must be a TeX integer suitable for `\int_value:w`.

`\intarray_item:cn`

`__kernel_intarray_item:Nn`

`__intarray_item:Nn`

```

22153 \cs_new:Npn \__kernel_intarray_item:Nn #1#2
22154 { \int_value:w \__intarray_entry:w #2 #1 }
22155 \cs_new:Npn \intarray_item:Nn #1#2
22156 {
22157   \exp_after:wN \__intarray_item:Nw
22158   \exp_after:wN #1
22159   \int_value:w \int_eval:n {#2} ;
22160 }
22161 \cs_generate_variant:Nn \intarray_item:Nn { c }

```

```

22162 \cs_new:Npn \__intarray_item:Nw #1#2 ;
22163 {
22164     \__intarray_bounds:NNnTF \msg_expandable_error:nnfff #1 {#2}
22165     { \__kernel_intarray_item:Nn #1 {#2} }
22166     { 0 }
22167 }

```

(End definition for `\intarray_item:Nn`, `__kernel_intarray_item:Nn`, and `__intarray_item:Nn`. This function is documented on page 237.)

`\intarray_rand_item:N` Importantly, `\intarray_item:Nn` only evaluates its argument once.

```

\intarray_rand_item:c
22168 \cs_new:Npn \intarray_rand_item:N #1
22169 { \intarray_item:Nn #1 { \int_rand:n { \intarray_count:N #1 } } }
22170 \cs_generate_variant:Nn \intarray_rand_item:N { c }

```

(End definition for `\intarray_rand_item:N`. This function is documented on page 237.)

64.2.3 Working with contents of integer arrays

`\intarray_const_from_clist:Nn` Similar to `\intarray_new:Nn` (which we don't use because when debugging is enabled that function checks the variable name starts with `g_`). We make use of the fact that `TeX` allows allocation of successive `\fontdimen` as long as no other font has been declared: no need to count the comma list items first. We need the code in `\intarray_gset:Nnn` that checks the item value is not too big, namely `__intarray_gset_overflow_test:nw`, but not the code that checks bounds. At the end, set the size of the intarray.

```

22171 \cs_new_protected:Npn \intarray_const_from_clist:Nn #1#2
22172 {
22173     \__intarray_new:N #1
22174     \int_zero:N \l__intarray_loop_int
22175     \clist_map_inline:nn {#2}
22176     { \exp_args:Nf \__intarray_const_from_clist:nN { \int_eval:n {##1} } #1 }
22177     \__intarray_count:w #1 \l__intarray_loop_int
22178 }
22179 \cs_generate_variant:Nn \intarray_const_from_clist:Nn { c }
22180 \cs_new_protected:Npn \__intarray_const_from_clist:nN #1#2
22181 {
22182     \int_incr:N \l__intarray_loop_int
22183     \__intarray_gset_overflow_test:nw {#1}
22184     \__kernel_intarray_gset:Nnn #2 \l__intarray_loop_int {#1}
22185 }

```

(End definition for `\intarray_const_from_clist:Nn` and `__intarray_const_from_clist:nN`. This function is documented on page 237.)

`\intarray_to_clist:N` Loop through the array, putting a comma before each item. Remove the leading comma with `f`-expansion. We also use the auxiliary in `\intarray_show:N` with argument comma, space.

```

\intarray_to_clist:c
\__intarray_to_clist:Nn
\__intarray_to_clist:w
22186 \cs_new:Npn \intarray_to_clist:N #1 { \__intarray_to_clist:Nn #1 { , } }
22187 \cs_generate_variant:Nn \intarray_to_clist:N { c }
22188 \cs_new:Npn \__intarray_to_clist:Nn #1#2
22189 {
22190     \int_compare:nNf { \intarray_count:N #1 } = \c_zero_int
22191     {
22192         \exp_last_unbraced:Nf \use_none:n

```

```

22193         { \_intarray_to_clist:w 1 ; #1 {#2} \prg_break_point: }
22194     }
22195 }
22196 \cs_new:Npn \_intarray_to_clist:w #1 ; #2#3
22197 {
22198     \if_int_compare:w #1 > \_intarray_count:w #2
22199     \prg_break:n
22200     \fi:
22201     #3 \_kernel_intarray_item:Nn #2 {#1}
22202     \exp_after:wN \_intarray_to_clist:w
22203     \int_value:w \int_eval:w #1 + \c_one_int ; #2 {#3}
22204 }

```

(End definition for \intarray_to_clist:N, _intarray_to_clist:Nn, and _intarray_to_clist:w.
This function is documented on page 302.)

_kernel_intarray_range_to_clist:Nnn
_intarray_range_to_clist:ww

Loop through part of the array.

```

22205 \cs_new:Npn \_kernel_intarray_range_to_clist:Nnn #1#2#3
22206 {
22207     \exp_last_unbraced:Nf \use_none:n
22208     {
22209         \exp_after:wN \_intarray_range_to_clist:ww
22210         \int_value:w \int_eval:w #2 \exp_after:wN ;
22211         \int_value:w \int_eval:w #3 ;
22212         #1 \prg_break_point:
22213     }
22214 }
22215 \cs_new:Npn \_intarray_range_to_clist:ww #1 ; #2 ; #3
22216 {
22217     \if_int_compare:w #1 > #2 \exp_stop_f:
22218     \prg_break:n
22219     \fi:
22220     , \_kernel_intarray_item:Nn #3 {#1}
22221     \exp_after:wN \_intarray_range_to_clist:ww
22222     \int_value:w \int_eval:w #1 + \c_one_int ; #2 ; #3
22223 }

```

(End definition for _kernel_intarray_range_to_clist:Nnn and _intarray_range_to_clist:ww.)

_kernel_intarray_gset_range_from_clist:Nnn
_intarray_gset_range:Nw

Loop through part of the array.

```

22224 \cs_new_protected:Npn \_kernel_intarray_gset_range_from_clist:Nnn #1#2#3
22225 {
22226     \int_set:Nn \l__intarray_loop_int {#2}
22227     \_intarray_gset_range:Nw #1 #3 , , \prg_break_point:
22228 }
22229 \cs_new_protected:Npn \_intarray_gset_range:Nw #1 #2 ,
22230 {
22231     \if_catcode:w \scan_stop: \tl_to_str:n {#2} \scan_stop:
22232     \prg_break:n
22233     \fi:
22234     \_kernel_intarray_gset:Nnn #1 \l__intarray_loop_int {#2}
22235     \int_incr:N \l__intarray_loop_int
22236     \_intarray_gset_range:Nw #1
22237 }

```

(End definition for `__kernel_intarray_gset_range_from_clist:Nnn` and `__intarray_gset_range:Nw`.)

```
22238 }
```

64.3 Common parts

```
\intarray_show:N Convert the list to a comma list (with spaces after each comma)
\intarray_show:c 22239 \cs_new_protected:Npn \intarray_show:N { __intarray_show:NN \msg_show:nnxxxx }
\intarray_log:N 22240 \cs_generate_variant:Nn \intarray_show:N { c }
\intarray_log:c 22241 \cs_new_protected:Npn \intarray_log:N { __intarray_show:NN \msg_log:nnxxxx }
22242 \cs_generate_variant:Nn \intarray_log:N { c }
22243 \cs_new_protected:Npn \__intarray_show:NN #1#2
22244 {
22245     __kernel_chk_defined:NT #2
22246     {
22247         #1 { intarray } { show }
22248         { \token_to_str:N #2 }
22249         { \intarray_count:N #2 }
22250         { >~ __intarray_to_clist:Nn #2 { , ~ } }
22251         { }
22252     }
22253 }
```

(End definition for `\intarray_show:N` and `\intarray_log:N`. These functions are documented on page 237.)

64.3.1 Random arrays

```
\intarray_gset_rand:Nn We only perform the bounds checks once. This is done by two __intarray_gset_
\intarray_gset_rand:cn overflow_test:nw, with an appropriate empty argument to avoid a spurious “at posi-
\intarray_gset_rand:Nnn tion #1” part in the error message. Then calculate the number of choices: this is at most
\intarray_gset_rand:cnn (230 - 1) - (-(230 - 1)) + 1 = 231 - 1, which just barely does not overflow. For small
__intarray_gset_rand:Nnn ranges use __kernel_randint:n (making sure to subtract 1 before adding the random
__intarray_gset_rand:Nff number to the  $\langle min \rangle$ , to avoid overflow when  $\langle min \rangle$  or  $\langle max \rangle$  are  $\pm c\_max\_int$ ), other-
    \__intarray_gset_rand_auxi:Nnn wise __kernel_randint:nn. Finally, if there are no random numbers do not define any
    \__intarray_gset_rand_auxii:Nnn of the auxiliaries.
    \__intarray_gset_rand_auxiii:Nnn
__intarray_gset_all_same:Nn 22254 \cs_new_protected:Npn \intarray_gset_rand:Nn #1
22255 { \intarray_gset_rand:Nnn #1 { 1 } }
22256 \cs_generate_variant:Nn \intarray_gset_rand:Nn { c }
22257 \sys_if_rand_exist:TF
22258 {
22259     \cs_new_protected:Npn \intarray_gset_rand:Nnn #1#2#3
22260     {
22261         __intarray_gset_rand:Nff #1
22262         { \int_eval:n {#2} } { \int_eval:n {#3} }
22263     }
22264     \cs_new_protected:Npn __intarray_gset_rand:Nnn #1#2#3
22265     {
22266         \int_compare:nNnTF {#2} > {#3}
22267         {
22268             \msg_expandable_error:nnnn
22269             { kernel } { randint-backward-range } {#2} {#3}
22270             __intarray_gset_rand:Nnn #1 {#3} {#2}

```



```

22271     }
22272     {
22273         \__intarray_gset_overflow_test:nw {#2}
22274         \__intarray_gset_rand_auxi:Nnnn #1 { } {#2} {#3}
22275     }
22276 }
22277 \cs_generate_variant:Nn \__intarray_gset_rand:Nnn { Nff }
22278 \cs_new_protected:Npn \__intarray_gset_rand_auxi:Nnnn #1#2#3#4
22279 {
22280     \__intarray_gset_overflow_test:nw {#4}
22281     \__intarray_gset_rand_auxii:Nnnn #1 { } {#4} {#3}
22282 }
22283 \cs_new_protected:Npn \__intarray_gset_rand_auxii:Nnnn #1#2#3#4
22284 {
22285     \exp_args:NNf \__intarray_gset_rand_auxiii:Nnnn #1
22286     { \int_eval:n { #3 - #4 + 1 } } {#4} {#3}
22287 }
22288 \cs_new_protected:Npn \__intarray_gset_rand_auxiii:Nnnn #1#2#3#4
22289 {
22290     \exp_args:NNf \__intarray_gset_all_same:Nn #1
22291     {
22292         \int_compare:nNnTF {#2} > \c__kernel_randint_max_int
22293         {
22294             \exp_stop_f:
22295             \int_eval:n { \__kernel_randint:nn {#3} {#4} }
22296         }
22297         {
22298             \exp_stop_f:
22299             \int_eval:n { \__kernel_randint:n {#2} - 1 + #3 }
22300         }
22301     }
22302 }
22303 \cs_new_protected:Npn \__intarray_gset_all_same:Nn #1#2
22304 {
22305     \int_zero:N \l__intarray_loop_int
22306     \prg_replicate:nn { \intarray_count:N #1 }
22307     {
22308         \int_incr:N \l__intarray_loop_int
22309         \__kernel_intarray_gset:Nnn #1 \l__intarray_loop_int {#2}
22310     }
22311 }
22312 }
22313 {
22314     \cs_new_protected:Npn \intarray_gset_rand:Nnn #1#2#3
22315     {
22316         \msg_error:nnn { kernel } { fp-no-random }
22317         { \intarray_gset_rand:Nnn #1 {#2} {#3} }
22318     }
22319 }
22320 \cs_generate_variant:Nn \intarray_gset_rand:Nnn { c }

```

(End definition for \intarray_gset_rand:Nn and others. These functions are documented on page 302.)

```

22321 \end{tex}
22322 \end{package}

```

Chapter 65

l3fp implementation

Nothing to see here: everything is in the subfiles!

Chapter 66

l3fp-aux implementation

22323 $\langle *package \rangle$

22324 $\langle @@=fp \rangle$

66.1 Access to primitives

$\backslash_fp_int_eval:w$ Largely for performance reasons, we need to directly access primitives rather than use
 $\backslash_fp_int_eval_end:$ $\backslash int_eval:n$. This happens *a lot*, so we use private names. The same is true for
 $\backslash_fp_int_to_roman:w$ $\backslash romannumeral$, although it is used much less widely.

22325 $\backslash cs_new_eq:NN \backslash_fp_int_eval:w \backslash tex_numexpr:D$

22326 $\backslash cs_new_eq:NN \backslash_fp_int_eval_end: \backslash scan_stop:$

22327 $\backslash cs_new_eq:NN \backslash_fp_int_to_roman:w \backslash tex_romannumeral:D$

(End definition for $\backslash_fp_int_eval:w$, $\backslash_fp_int_eval_end:$, and $\backslash_fp_int_to_roman:w$.)

66.2 Internal representation

Internally, a floating point number $\langle X \rangle$ is a token list containing

$\backslash s_fp \backslash_fp_chk:w \langle case \rangle \langle sign \rangle \langle body \rangle ;$

Let us explain each piece separately.

Internal floating point numbers are used in expressions, and in this context are subject to **f**-expansion. They must leave a recognizable mark after **f**-expansion, to prevent the floating point number from being re-parsed. Thus, $\backslash s_fp$ is simply another name for $\backslash relax$.

When used directly without an accessor function, floating points should produce an error: this is the role of $\backslash_fp_chk:w$. We could make floating point variables be protected to prevent them from expanding under **x**-expansion, but it seems more convenient to treat them as a subcase of token list variables.

The (decimal part of the) IEEE-754-2008 standard requires the format to be able to represent special floating point numbers besides the usual positive and negative cases. We distinguish the various possibilities by their $\langle case \rangle$, which is a single digit:

0 zeros: $+0$ and -0 ,

1 “normal” numbers (positive and negative),

Table 3: Internal representation of floating point numbers.

Representation	Meaning
0 0 \s_fp_... ;	Positive zero.
0 2 \s_fp_... ;	Negative zero.
1 0 {<exponent>} {<X ₁ >} {<X ₂ >} {<X ₃ >} {<X ₄ >} ;	Positive floating point.
1 2 {<exponent>} {<X ₁ >} {<X ₂ >} {<X ₃ >} {<X ₄ >} ;	Negative floating point.
2 0 \s_fp_... ;	Positive infinity.
2 2 \s_fp_... ;	Negative infinity.
3 1 \s_fp_... ;	Quiet nan.
3 1 \s_fp_... ;	Signalling nan.

2 infinities: `+inf` and `-inf`,

3 quiet and signalling `nan`.

The $\langle sign \rangle$ is 0 (positive) or 2 (negative), except in the case of `nan`, which have $\langle sign \rangle = 1$. This ensures that changing the $\langle sign \rangle$ digit to $2 - \langle sign \rangle$ is exactly equivalent to changing the sign of the number.

Special floating point numbers have the form

`\s_fp _fp_chk:w <case> <sign> \s_fp_... ;`

where `\s_fp_...` is a scan mark carrying information about how the number was formed (useful for debugging).

Normal floating point numbers ($\langle case \rangle = 1$) have the form

`\s_fp _fp_chk:w 1 <sign> {<exponent>} {<X1>} {<X2>} {<X3>} {<X4>} ;`

Here, the $\langle exponent \rangle$ is an integer, between -10000 and 10000 . The body consists in four blocks of exactly 4 digits, $0000 \leq \langle X_i \rangle \leq 9999$, and the floating point is

$$(-1)^{\langle sign \rangle / 2} \langle X_1 \rangle \langle X_2 \rangle \langle X_3 \rangle \langle X_4 \rangle \cdot 10^{\langle exponent \rangle - 16}$$

where we have concatenated the 16 digits. Currently, floating point numbers are normalized such that the $\langle exponent \rangle$ is minimal, in other words, $1000 \leq \langle X_1 \rangle \leq 9999$.

Calculations are done in base 10000, *i.e.* one myriad.

66.3 Using arguments and semicolons

`_fp_use_none_stop_f:n` This function removes an argument (typically a digit) and replaces it by `\exp_stop_f:`, a marker which stops f-type expansion.

22328 \cs_new:Npn _fp_use_none_stop_f:n #1 { \exp_stop_f: }

(End definition for `_fp_use_none_stop_f:n`.)

`_fp_use_s:n` Those functions place a semicolon after one or two arguments (typically digits).

22329 \cs_new:Npn _fp_use_s:n #1 { #1; }
22330 \cs_new:Npn _fp_use_s:nn #1#2 { #1#2; }

(End definition for `_fp_use_s:n` and `_fp_use_s:nn`.)

`__fp_use_none_until_s:w` Those functions select specific arguments among a set of arguments delimited by a semicolon.

`__fp_use_i_until_s:nw`

`__fp_use_ii_until_s:nnw`

```

22331 \cs_new:Npn \__fp_use_none_until_s:w #1; { }
22332 \cs_new:Npn \__fp_use_i_until_s:nw #1#2; {#1}
22333 \cs_new:Npn \__fp_use_ii_until_s:nnw #1#2#3; {#2}

```

(End definition for `__fp_use_none_until_s:w`, `__fp_use_i_until_s:nw`, and `__fp_use_ii_until_s:nnw`.)

`__fp_reverse_args:Nww` Many internal functions take arguments delimited by semicolons, and it is occasionally useful to swap two such arguments.

```

22334 \cs_new:Npn \__fp_reverse_args:Nww #1 #2; #3; { #1 #3; #2; }

```

(End definition for `__fp_reverse_args:Nww`.)

`__fp_rrot:www` Rotate three arguments delimited by semicolons. This is the inverse (or the square) of the Forth primitive ROT, hence the name.

```

22335 \cs_new:Npn \__fp_rrot:www #1; #2; #3; { #2; #3; #1; }

```

(End definition for `__fp_rrot:www`.)

`__fp_use_i:ww` Many internal functions take arguments delimited by semicolons, and it is occasionally useful to remove one or two such arguments.

`__fp_use_i:www`

```

22336 \cs_new:Npn \__fp_use_i:ww #1; #2; { #1; }
22337 \cs_new:Npn \__fp_use_i:www #1; #2; #3; { #1; }

```

(End definition for `__fp_use_i:ww` and `__fp_use_i:www`.)

66.4 Constants, and structure of floating points

`__fp_misused:n` This receives a floating point object (floating point number or tuple) and generates an error stating that it was misused. This is called when for instance an `fp` variable is left in the input stream and its contents reach T_EX's stomach.

```

22338 \cs_new_protected:Npn \__fp_misused:n #1
22339 { \msg_error:nnx { fp } { misused } { \fp_to_tl:n {#1} } }

```

(End definition for `__fp_misused:n`.)

`\s__fp` Floating points numbers all start with `\s__fp __fp_chk:w`, where `\s__fp` is equal to the T_EX primitive `\relax`, and `__fp_chk:w` is protected. The rest of the floating point number is made of characters (or `\relax`). This ensures that nothing expands under f-expansion, nor under x-expansion. However, when typeset, `\s__fp` does nothing, and `__fp_chk:w` is expanded. We define `__fp_chk:w` to produce an error.

`__fp_chk:w`

```

22340 \scan_new:N \s__fp
22341 \cs_new_protected:Npn \__fp_chk:w #1 ;
22342 { \__fp_misused:n { \s__fp \__fp_chk:w #1 ; } }

```

(End definition for `\s__fp` and `__fp_chk:w`.)

`\s__fp_expr_mark` Aliases of `\tex_relax:D`, used to terminate expressions.

`\s__fp_expr_stop`

```

22343 \scan_new:N \s__fp_expr_mark
22344 \scan_new:N \s__fp_expr_stop

```

(End definition for `\s__fp_expr_mark` and `\s__fp_expr_stop`.)

`\s__fp_mark` Generic scan marks used throughout the module.

`\s__fp_stop` 22345 `\scan_new:N \s__fp_mark`
22346 `\scan_new:N \s__fp_stop`

(End definition for `\s__fp_mark` and `\s__fp_stop`.)

`_fp_use_i_delimit_by_s_stop:nw` Functions to gobble up to a scan mark.

22347 `\cs_new:Npn _fp_use_i_delimit_by_s_stop:nw #1 #2 \s__fp_stop {#1}`

(End definition for `_fp_use_i_delimit_by_s_stop:nw`.)

`\s__fp_invalid` A couple of scan marks used to indicate where special floating point numbers come from.

`\s__fp_underflow` 22348 `\scan_new:N \s__fp_invalid`
`\s__fp_overflow` 22349 `\scan_new:N \s__fp_underflow`
`\s__fp_division` 22350 `\scan_new:N \s__fp_overflow`
`\s__fp_exact` 22351 `\scan_new:N \s__fp_division`
22352 `\scan_new:N \s__fp_exact`

(End definition for `\s__fp_invalid` and others.)

`\c_zero_fp` The special floating points. We define the floating points here as “exact”.

`\c_minus_zero_fp` 22353 `\tl_const:Nn \c_zero_fp { \s__fp _fp_chk:w 0 0 \s__fp_exact ; }`
`\c_inf_fp` 22354 `\tl_const:Nn \c_minus_zero_fp { \s__fp _fp_chk:w 0 2 \s__fp_exact ; }`
`\c_minus_inf_fp` 22355 `\tl_const:Nn \c_inf_fp { \s__fp _fp_chk:w 2 0 \s__fp_exact ; }`
`\c_nan_fp` 22356 `\tl_const:Nn \c_minus_inf_fp { \s__fp _fp_chk:w 2 2 \s__fp_exact ; }`
22357 `\tl_const:Nn \c_nan_fp { \s__fp _fp_chk:w 3 1 \s__fp_exact ; }`

(End definition for `\c_zero_fp` and others. These variables are documented on page 246.)

`\c__fp_prec_int` The number of digits of floating points.

`\c__fp_half_prec_int` 22358 `\int_const:Nn \c__fp_prec_int { 16 }`
`\c__fp_block_int` 22359 `\int_const:Nn \c__fp_half_prec_int { 8 }`
22360 `\int_const:Nn \c__fp_block_int { 4 }`

(End definition for `\c__fp_prec_int`, `\c__fp_half_prec_int`, and `\c__fp_block_int`.)

`\c__fp_myriad_int` Blocks have 4 digits so this integer is useful.

22361 `\int_const:Nn \c__fp_myriad_int { 10000 }`

(End definition for `\c__fp_myriad_int`.)

`\c__fp_minus_min_exponent_int` Normal floating point numbers have an exponent between `– minus_min_exponent` and
`\c__fp_max_exponent_int` `max_exponent` inclusive. Larger numbers are rounded to $\pm\infty$. Smaller numbers are
rounded to ± 0 . It would be more natural to define a `min_exponent` with the opposite
sign but that would waste one T_EX count.

22362 `\int_const:Nn \c__fp_minus_min_exponent_int { 10000 }`
22363 `\int_const:Nn \c__fp_max_exponent_int { 10000 }`

(End definition for `\c__fp_minus_min_exponent_int` and `\c__fp_max_exponent_int`.)

`\c__fp_max_exp_exponent_int` If a number’s exponent is larger than that, its exponential overflows/underflows.

22364 `\int_const:Nn \c__fp_max_exp_exponent_int { 5 }`

(End definition for `\c__fp_max_exp_exponent_int`.)

`\c__fp_overflowing_fp` A floating point number that is bigger than all normal floating point numbers. This replaces infinities when converting to formats that do not support infinities.

```

22365 \tl_const:Nx \c__fp_overflowing_fp
22366 {
22367   \s__fp \__fp_chk:w 1 0
22368   { \int_eval:n { \c__fp_max_exponent_int + 1 } }
22369   {1000} {0000} {0000} {0000} ;
22370 }

```

(End definition for \c__fp_overflowing_fp.)

`__fp_zero_fp:N` In case of overflow or underflow, we have to output a zero or infinity with a given sign.
`__fp_inf_fp:N`

```

22371 \cs_new:Npn \__fp_zero_fp:N #1
22372 { \s__fp \__fp_chk:w 0 #1 \s__fp_underflow ; }
22373 \cs_new:Npn \__fp_inf_fp:N #1
22374 { \s__fp \__fp_chk:w 2 #1 \s__fp_overflow ; }

```

(End definition for __fp_zero_fp:N and __fp_inf_fp:N.)

`__fp_exponent:w` For normal numbers, the function expands to the exponent, otherwise to 0. This is used in l3str-format.

```

22375 \cs_new:Npn \__fp_exponent:w \s__fp \__fp_chk:w #1
22376 {
22377   \if_meaning:w 1 #1
22378     \exp_after:wN \__fp_use_ii_until_s:nnw
22379   \else:
22380     \exp_after:wN \__fp_use_i_until_s:nw
22381     \exp_after:wN 0
22382   \fi:
22383 }

```

(End definition for __fp_exponent:w.)

`__fp_neg_sign:N` When appearing in an integer expression or after `\int_value:w`, this expands to the sign opposite to #1, namely 0 (positive) is turned to 2 (negative), 1 (**nan**) to 1, and 2 to 0.

```

22384 \cs_new:Npn \__fp_neg_sign:N #1
22385 { \__fp_int_eval:w 2 - #1 \__fp_int_eval_end: }

```

(End definition for __fp_neg_sign:N.)

`__fp_kind:w` Expands to 0 for zeros, 1 for normal floating point numbers, 2 for infinities, 3 for **nan**, 4 for tuples.

```

22386 \cs_new:Npn \__fp_kind:w #1
22387 {
22388   \__fp_if_type_fp:NTwFw
22389   #1 \__fp_use_ii_until_s:nnw
22390   \s__fp { \__fp_use_i_until_s:nw 4 }
22391   \s__fp_stop
22392 }

```

(End definition for __fp_kind:w.)

66.5 Overflow, underflow, and exact zero

`__fp_sanitizew` Expects the sign and the exponent in some order, then the significand (which we don't touch). Outputs the corresponding floating point number, possibly underflowed to ± 0 or overflowed to $\pm\infty$. The functions `__fp_underfloww` and `__fp_overfloww` are defined in `l3fp-traps`.

```

22393 \cs_new:Npn \__fp_sanitizew #1 #2;
22394 {
22395   \if_case:w
22396     \if_int_compare:w #2 > \c__fp_max_exponent_int 1 ~ \else:
22397     \if_int_compare:w #2 < - \c__fp_minus_min_exponent_int 2 ~ \else:
22398     \if_meaning:w 1 #1 3 ~ \fi: \fi: \fi: 0 ~
22399     \or: \exp_after:wN \__fp_overflow:w
22400     \or: \exp_after:wN \__fp_underflow:w
22401     \or: \exp_after:wN \__fp_sanitizew
22402     \fi:
22403     \s__fp \__fp_chk:w 1 #1 {#2}
22404   }
22405 \cs_new:Npn \__fp_sanitizewN #1; #2 { \__fp_sanitizew #2 #1; }
22406 \cs_new:Npn \__fp_sanitizew_zero:w \s__fp \__fp_chk:w #1 #2 #3;
22407 { \c_zero_fp }

```

(End definition for `__fp_sanitizew`, `__fp_sanitizewN`, and `__fp_sanitizew_zero:w`.)

66.6 Expanding after a floating point number

`__fp_exp_after_ow`
`__fp_exp_after_fnw`

`__fp_exp_after_ow` *<floating point>*
`__fp_exp_after_fnw` *{<tokens>}* *<floating point>*

Places *<tokens>* (empty in the case of `__fp_exp_after_ow`) between the *<floating point>* and the following tokens, then hits those tokens with `o` or `f`-expansion, and leaves the floating point number unchanged.

We first distinguish normal floating points, which have a significand, from the much simpler special floating points.

```

22408 \cs_new:Npn \__fp_exp_after_ow \s__fp \__fp_chk:w #1
22409 {
22410   \if_meaning:w 1 #1
22411     \exp_after:wN \__fp_exp_after_normal:nNNw
22412   \else:
22413     \exp_after:wN \__fp_exp_after_special:nNNw
22414   \fi:
22415   { }
22416   #1
22417 }
22418 \cs_new:Npn \__fp_exp_after_fnw #1 \s__fp \__fp_chk:w #2
22419 {
22420   \if_meaning:w 1 #2
22421     \exp_after:wN \__fp_exp_after_normal:nNNw
22422   \else:
22423     \exp_after:wN \__fp_exp_after_special:nNNw
22424   \fi:
22425   { \exp:w \exp_end_continue_f:w #1 }
22426   #2

```


22427 }

(End definition for `_fp_exp_after_o:w` and `_fp_exp_after_f:nw`.)

`_fp_exp_after_special:nNw` `_fp_exp_after_special:nNw` {*after*} *case* *sign* *scan mark* ;
Special floating point numbers are easy to jump over since they contain few tokens.

```
22428 \cs_new:Npn \_fp_exp_after_special:nNw #1#2#3#4;
22429 {
22430   \exp_after:wN \_s\_fp
22431   \exp_after:wN \_fp_chk:w
22432   \exp_after:wN #2
22433   \exp_after:wN #3
22434   \exp_after:wN #4
22435   \exp_after:wN ;
22436   #1
22437 }
```

(End definition for `_fp_exp_after_special:nNw`.)

`_fp_exp_after_normal:nNw` For normal floating point numbers, life is slightly harder, since we have many tokens to jump over. Here it would be slightly better if the digits were not braced but instead were delimited arguments (for instance delimited by ,). That may be changed some day.

```
22438 \cs_new:Npn \_fp_exp_after_normal:nNw #1 1 #2 #3 #4#5#6#7;
22439 {
22440   \exp_after:wN \_fp_exp_after_normal:Nwwwww
22441   \exp_after:wN #2
22442   \int_value:w #3 \exp_after:wN ;
22443   \int_value:w 1 #4 \exp_after:wN ;
22444   \int_value:w 1 #5 \exp_after:wN ;
22445   \int_value:w 1 #6 \exp_after:wN ;
22446   \int_value:w 1 #7 \exp_after:wN ; #1
22447 }
22448 \cs_new:Npn \_fp_exp_after_normal:Nwwwww
22449   #1 #2; 1 #3 ; 1 #4 ; 1 #5 ; 1 #6 ;
22450   { \_fp \_fp_chk:w 1 #1 {#2} {#3} {#4} {#5} {#6} ; }
```

(End definition for `_fp_exp_after_normal:nNw`.)

66.7 Other floating point types

`\s_fp_tuple` Floating point tuples take the form `\s_fp_tuple _fp_tuple_chk:w { <fp 1> <fp 2> ... }` ; where each *<fp>* is a floating point number or tuple, hence ends with ; itself. When a tuple is typeset, `_fp_tuple_chk:w` produces an error, just like usual floating point numbers. Tuples may have zero or one element.

```
22451 \scan_new:N \s\_fp_tuple
22452 \cs_new_protected:Npn \_fp_tuple_chk:w #1 ;
22453   { \_fp_misused:n { \s\_fp_tuple \_fp_tuple_chk:w #1 ; } }
22454 \tl_const:Nn \c\_fp_empty_tuple_fp
22455   { \s\_fp_tuple \_fp_tuple_chk:w { } ; }
```

(End definition for `\s_fp_tuple`, `_fp_tuple_chk:w`, and `\c_fp_empty_tuple_fp`.)

`__fp_tuple_count:w` Count the number of items in a tuple of floating points by counting semicolons. The technique is very similar to `\tl_count:n`, but with the loop built-in. Checking for the end of the loop is done with the `\use_none:n #1` construction.

```

22456 \cs_new:Npn \__fp_array_count:n #1
22457 { \__fp_tuple_count:w \s__fp_tuple \__fp_tuple_chk:w {#1} ; }
22458 \cs_new:Npn \__fp_tuple_count:w \s__fp_tuple \__fp_tuple_chk:w #1 ;
22459 {
22460   \int_value:w \__fp_int_eval:w 0
22461   \__fp_tuple_count_loop:Nw #1 { ? \prg_break: } ;
22462   \prg_break_point:
22463   \__fp_int_eval_end:
22464 }
22465 \cs_new:Npn \__fp_tuple_count_loop:Nw #1#2;
22466 { \use_none:n #1 + 1 \__fp_tuple_count_loop:Nw }

```

(End definition for `__fp_tuple_count:w`, `__fp_array_count:n`, and `__fp_tuple_count_loop:Nw`.)

`__fp_if_type_fp:NTwFw` Used as `__fp_if_type_fp:NTwFw <marker> {<true code>} \s__fp {<false code>} \s__fp_stop`, this test whether the `<marker>` is `\s__fp` or not and runs the appropriate `<code>`. The very unusual syntax is for optimization purposes as that function is used for all floating point operations.

```

22467 \cs_new:Npn \__fp_if_type_fp:NTwFw #1 \s__fp #2 #3 \s__fp_stop {#2}

```

(End definition for `__fp_if_type_fp:NTwFw`.)

`__fp_array_if_all_fp:nTF` True if all items are floating point numbers. Used for min.
`__fp_array_if_all_fp_loop:w`

```

22468 \cs_new:Npn \__fp_array_if_all_fp:nTF #1
22469 {
22470   \__fp_array_if_all_fp_loop:w #1 { \s__fp \prg_break: } ;
22471   \prg_break_point: \use_i:nn
22472 }
22473 \cs_new:Npn \__fp_array_if_all_fp_loop:w #1#2 ;
22474 {
22475   \__fp_if_type_fp:NTwFw
22476   #1 \__fp_array_if_all_fp_loop:w
22477   \s__fp { \prg_break:n \use_iii:nnn }
22478   \s__fp_stop
22479 }

```

(End definition for `__fp_array_if_all_fp:nTF` and `__fp_array_if_all_fp_loop:w`.)

`__fp_type_from_scan:N` Used as `__fp_type_from_scan:N <token>`. Grabs the pieces of the stringified `<token>` which lies after the first `s__fp`. If the `<token>` does not contain that string, the result is `_?`.
`__fp_type_from_scan_other:N`
`__fp_type_from_scan:w`

```

22480 \cs_new:Npn \__fp_type_from_scan:N #1
22481 {
22482   \__fp_if_type_fp:NTwFw
22483   #1 { }
22484   \s__fp { \__fp_type_from_scan_other:N #1 }
22485   \s__fp_stop
22486 }
22487 \cs_new:Npx \__fp_type_from_scan_other:N #1
22488 {
22489   \exp_not:N \exp_after:wN \exp_not:N \__fp_type_from_scan:w

```

```

22490 \exp_not:N \token_to_str:N #1 \s__fp_mark
22491 \tl_to_str:n { s__fp _? } \s__fp_mark \s__fp_stop
22492 }
22493 \exp_last_unbraced:NNNNo
22494 \cs_new:Npn \__fp_type_from_scan:w #1
22495 { \tl_to_str:n { s__fp } } #2 \s__fp_mark #3 \s__fp_stop {#2}

```

(End definition for __fp_type_from_scan:N, __fp_type_from_scan_other:N, and __fp_type_from_scan:w.)

__fp_change_func_type:NNN Arguments are $\langle type\ marker \rangle$ $\langle function \rangle$ $\langle recovery \rangle$. This gives the function obtained by placing the type after @@. If the function is not defined then $\langle recovery \rangle$ $\langle function \rangle$ is used instead; however that test is not run when the $\langle type\ marker \rangle$ is \s__fp.

```

22496 \cs_new:Npn \__fp_change_func_type:NNN #1#2#3
22497 {
22498   \__fp_if_type_fp:NTwFw
22499   #1 #2
22500   \s__fp
22501   {
22502     \exp_after:wN \__fp_change_func_type_chk:NNN
22503     \cs:w
22504     __fp \__fp_type_from_scan_other:N #1
22505     \exp_after:wN \__fp_change_func_type_aux:w \token_to_str:N #2
22506     \cs_end:
22507     #2 #3
22508   }
22509   \s__fp_stop
22510 }
22511 \exp_last_unbraced:NNNNo
22512 \cs_new:Npn \__fp_change_func_type_aux:w #1 { \tl_to_str:n { __fp } } { }
22513 \cs_new:Npn \__fp_change_func_type_chk:NNN #1#2#3
22514 {
22515   \if_meaning:w \scan_stop: #1
22516   \exp_after:wN #3 \exp_after:wN #2
22517   \else:
22518   \exp_after:wN #1
22519   \fi:
22520 }

```

(End definition for __fp_change_func_type:NNN, __fp_change_func_type_aux:w, and __fp_change_func_type_chk:NNN.)

__fp_exp_after_any_f:Nnw The Nnw function simply dispatches to the appropriate __fp_exp_after..._f:nw with “...” (either empty or $\langle type \rangle$) extracted from #1, which should start with \s__fp. If __fp_exp_after_any_f:nw it doesn’t start with \s__fp the function __fp_exp_after_?_f:nw defined in l3fp-parse gives an error; another special $\langle type \rangle$ is stop, useful for loops, see below. The nw function has an important optimization for floating points numbers; it also fetches its type marker #2 from the floating point.

```

22521 \cs_new:Npn \__fp_exp_after_any_f:Nnw #1
22522 { \cs:w __fp_exp_after \__fp_type_from_scan_other:N #1 _f:nw \cs_end: }
22523 \cs_new:Npn \__fp_exp_after_any_f:nw #1#2
22524 {
22525   \__fp_if_type_fp:NTwFw
22526   #2 \__fp_exp_after_f:nw

```

```

22527     \s__fp { \__fp_exp_after_any_f:Nnw #2 }
22528     \s__fp_stop
22529     {#1} #2
22530   }
22531 \cs_new_eq:NN \__fp_exp_after_expr_stop_f:nw \use_none:nn

```

(End definition for `__fp_exp_after_any_f:Nnw`, `__fp_exp_after_any_f:nw`, and `__fp_exp_after_expr_stop_f:nw`.)

`__fp_exp_after_tuple_o:w` The loop works by using the `n` argument of `__fp_exp_after_any_f:nw` to place the
`__fp_exp_after_tuple_f:nw` loop macro after the next item in the tuple and expand it.
`__fp_exp_after_array_f:w`

```

\__fp_exp_after_array_f:w
<fp1> ;
...
<fpn> ;
\s__fp_expr_stop

22532 \cs_new:Npn \__fp_exp_after_tuple_o:w
22533   { \__fp_exp_after_tuple_f:nw { \exp_after:wN \exp_stop_f: } }
22534 \cs_new:Npn \__fp_exp_after_tuple_f:nw
22535   #1 \s__fp_tuple \__fp_tuple_chk:w #2 ;
22536   {
22537     \exp_after:wN \s__fp_tuple
22538     \exp_after:wN \__fp_tuple_chk:w
22539     \exp_after:wN {
22540       \exp:w \exp_end_continue_f:w
22541       \__fp_exp_after_array_f:w #2 \s__fp_expr_stop
22542     \exp_after:wN }
22543     \exp_after:wN ;
22544     \exp:w \exp_end_continue_f:w #1
22545   }
22546 \cs_new:Npn \__fp_exp_after_array_f:w
22547   { \__fp_exp_after_any_f:nw { \__fp_exp_after_array_f:w } }

```

(End definition for `__fp_exp_after_tuple_o:w`, `__fp_exp_after_tuple_f:nw`, and `__fp_exp_after_array_f:w`.)

66.8 Packing digits

When a positive integer `#1` is known to be less than 10^8 , the following trick splits it into two blocks of 4 digits, padding with zeros on the left.

```

\cs_new:Npn \pack:NNNNw #1 #2#3#4#5 #6; { {#2#3#4#5} {#6} }
\exp_after:wN \pack:NNNNw
\__fp_int_value:w \__fp_int_eval:w 1 0000 0000 + #1 ;

```

The idea is that adding 10^8 to the number ensures that it has exactly 9 digits, and can then easily find which digits correspond to what position in the number. Of course, this can be modified for any number of digits less or equal to 9 (we are limited by $\text{T}_{\text{E}}\text{X}$'s integers). This method is very heavily relied upon in `l3fp-basics`.

More specifically, the auxiliary inserts `+ #1#2#3#4#5 ; {#6}`, which allows us to compute several blocks of 4 digits in a nested manner, performing carries on the fly. Say we want to compute 12345×66778899 . With simplified names, we would do

```

\exp_after:wN \post_processing:w
\__fp_int_value:w \__fp_int_eval:w - 5 0000
\exp_after:wN \pack:NNNNNw
\__fp_int_value:w \__fp_int_eval:w 4 9995 0000
+ 12345 * 6677
\exp_after:wN \pack:NNNNNw
\__fp_int_value:w \__fp_int_eval:w 5 0000 0000
+ 12345 * 8899 ;

```

The `\exp_after:wN` triggers `\int_value:w __fp_int_eval:w`, which starts a first computation, whose initial value is $-5\,0000$ (the “leading shift”). In that computation appears an `\exp_after:wN`, which triggers the nested computation `\int_value:w __fp_int_eval:w` with starting value $4\,9995\,0000$ (the “middle shift”). That, in turn, expands `\exp_after:wN` which triggers the third computation. The third computation’s value is $5\,0000\,0000 + 12345 \times 8899$, which has 9 digits. Adding $5 \cdot 10^8$ to the product allowed us to know how many digits to expect as long as the numbers to multiply are not too big; it also works to some extent with negative results. The `pack` function puts the last 4 of those 9 digits into a brace group, moves the semi-colon delimiter, and inserts a `+`, which combines the carry with the previous computation. The shifts nicely combine into $5\,0000\,0000/10^4 + 4\,9995\,0000 = 5\,0000\,0000$. As long as the operands are in some range, the result of this second computation has 9 digits. The corresponding `pack` function, expanded after the result is computed, braces the last 4 digits, and leaves `+ <5 digits>` for the initial computation. The “leading shift” cancels the combination of the other shifts, and the `\post_processing:w` takes care of packing the last few digits.

Admittedly, this is quite intricate. It is probably the key in making `l3fp` as fast as other pure `TeX` floating point units despite its increased precision. In fact, this is used so much that we provide different sets of packing functions and shifts, depending on ranges of input.

```

\__fp_pack:NNNNNw
\c__fp_trailing_shift_int
\c__fp_middle_shift_int
\c__fp_leading_shift_int
22548 \int_const:Nn \c__fp_leading_shift_int { - 5 0000 }
22549 \int_const:Nn \c__fp_middle_shift_int { 5 0000 * 9999 }
22550 \int_const:Nn \c__fp_trailing_shift_int { 5 0000 * 10000 }
22551 \cs_new:Npn \__fp_pack:NNNNNw #1 #2#3#4#5 #6; { + #1#2#3#4#5 ; {#6} }

```

This set of shifts allows for computations involving results in the range $[-4 \cdot 10^8, 5 \cdot 10^8 - 1]$. Shifted values all have exactly 9 digits.

(End definition for `__fp_pack:NNNNNw` and others.)

```

\__fp_pack_big:NNNNNNw
\c__fp_big_trailing_shift_int
\c__fp_big_middle_shift_int
\c__fp_big_leading_shift_int
22552 \int_const:Nn \c__fp_big_leading_shift_int { - 15 2374 }
22553 \int_const:Nn \c__fp_big_middle_shift_int { 15 2374 * 9999 }
22554 \int_const:Nn \c__fp_big_trailing_shift_int { 15 2374 * 10000 }
22555 \cs_new:Npn \__fp_pack_big:NNNNNNw #1#2 #3#4#5#6 #7;
22556 { + #1#2#3#4#5#6 ; {#7} }

```

This set of shifts allows for computations involving results in the range $[-5 \cdot 10^8, 6 \cdot 10^8 - 1]$ (actually a bit more). Shifted values all have exactly 10 digits. Note that the upper bound is due to `TeX`’s limit of $2^{31} - 1$ on integers. The shifts are chosen to be roughly the mid-point of 10^9 and 2^{31} , the two bounds on 10-digit integers in `TeX`.

(End definition for `__fp_pack_big:NNNNNNw` and others.)

```

\__fp_pack_Bigg:NNNNNNw
\c_fp_Bigg_trailing_shift_int
\c__fp_Bigg_middle_shift_int
\c_fp_Bigg_leading_shift_int
22557 \int_const:Nn \c__fp_Bigg_leading_shift_int { - 20 0000 }
22558 \int_const:Nn \c__fp_Bigg_middle_shift_int { 20 0000 * 9999 }
22559 \int_const:Nn \c__fp_Bigg_trailing_shift_int { 20 0000 * 10000 }
22560 \cs_new:Npn \__fp_pack_Bigg:NNNNNNw #1#2 #3#4#5#6 #7;
22561 { + #1#2#3#4#5#6 ; {#7} }

(End definition for \__fp_pack_Bigg:NNNNNNw and others.)

\__fp_pack_twice_four:wNNNNNNNN
\__fp_pack_twice_four:wNNNNNNNN <tokens> ; <≥ 8 digits>
Grabs two sets of 4 digits and places them before the semi-colon delimiter. Putting
several copies of this function before a semicolon packs more digits since each takes the
digits packed by the others in its first argument.
22562 \cs_new:Npn \__fp_pack_twice_four:wNNNNNNNN #1; #2#3#4#5 #6#7#8#9
22563 { #1 {#2#3#4#5} {#6#7#8#9} ; }

(End definition for \__fp_pack_twice_four:wNNNNNNNN.)

\__fp_pack_eight:wNNNNNNNN
\__fp_pack_eight:wNNNNNNNN <tokens> ; <≥ 8 digits>
Grabs one set of 8 digits and places them before the semi-colon delimiter as a single
group. Putting several copies of this function before a semicolon packs more digits since
each takes the digits packed by the others in its first argument.
22564 \cs_new:Npn \__fp_pack_eight:wNNNNNNNN #1; #2#3#4#5 #6#7#8#9
22565 { #1 {#2#3#4#5#6#7#8#9} ; }

(End definition for \__fp_pack_eight:wNNNNNNNN.)

\__fp_basics_pack_low:NNNNNNw
\__fp_basics_pack_high:NNNNNNw
\__fp_basics_pack_high_carry:w
Addition and multiplication of significands are done in two steps: first compute a (more or
less) exact result, then round and pack digits in the final (braced) form. These functions
take care of the packing, with special attention given to the case where rounding has
caused a carry. Since rounding can only shift the final digit by 1, a carry always produces
an exact power of 10. Thus, \__fp_basics_pack_high_carry:w is always followed by
four times {0000}.
This is used in l3fp-basics and l3fp-extended.
22566 \cs_new:Npn \__fp_basics_pack_low:NNNNNNw #1 #2#3#4#5 #6;
22567 { + #1 - 1 ; {#2#3#4#5} {#6} ; }
22568 \cs_new:Npn \__fp_basics_pack_high:NNNNNNw #1 #2#3#4#5 #6;
22569 {
22570 \if_meaning:w 2 #1
22571 \__fp_basics_pack_high_carry:w
22572 \fi:
22573 ; {#2#3#4#5} {#6}
22574 }
22575 \cs_new:Npn \__fp_basics_pack_high_carry:w \fi: ; #1
22576 { \fi: + 1 ; {1000} }

(End definition for \__fp_basics_pack_low:NNNNNNw, \__fp_basics_pack_high:NNNNNNw, and \__fp_
basics_pack_high_carry:w.)

```

_fp_basics_pack_weird_low:NNNNw
_fp_basics_pack_weird_high:NNNNNNNNw

This is used in l3fp-basics for additions and divisions. Their syntax is confusing, hence the name.

```

22577 \cs_new:Npn \_fp\_basics\_pack\_weird\_low:NNNNw #1 #2#3#4 #5;
22578 {
22579   \if_meaning:w 2 #1
22580     + 1
22581   \fi:
22582   \_fp\_int\_eval\_end:
22583   #2#3#4; {#5} ;
22584 }
22585 \cs_new:Npn \_fp\_basics\_pack\_weird\_high:NNNNNNNNw
22586   1 #1#2#3#4 #5#6#7#8 #9; { ; {#1#2#3#4} {#5#6#7#8} {#9} }

```

(End definition for _fp_basics_pack_weird_low:NNNNw and _fp_basics_pack_weird_high:NNNNNNNNw.)

66.9 Decimate (dividing by a power of 10)

_fp_decimate:nNnnnn

_fp_decimate:nNnnnn {<shift>} {<f₁>}
{<X₁>} {<X₂>} {<X₃>} {<X₄>}

Each <X_i> consists in 4 digits exactly, and $1000 \leq \langle X_1 \rangle < 9999$. The first argument determines by how much we shift the digits. <f₁> is called as follows:

<f₁> <rounding> {<X'₁>} {<X'₂>} <extra-digits> ;

where $0 \leq \langle X'_i \rangle < 10^8 - 1$ are 8 digit integers, forming the truncation of our number. In other words,

$$\left(\sum_{i=1}^4 \langle X_i \rangle \cdot 10^{-4i} \cdot 10^{-\langle \text{shift} \rangle} \right) - (\langle X'_1 \rangle \cdot 10^{-8} + \langle X'_2 \rangle \cdot 10^{-16}) = 0. \langle \text{extra-digits} \rangle \cdot 10^{-16} \in [0, 10^{-16}).$$

To round properly later, we need to remember some information about the difference. The <rounding> digit is 0 if and only if the difference is exactly 0, and 5 if and only if the difference is exactly $0.5 \cdot 10^{-16}$. Otherwise, it is the (non-0, non-5) digit closest to 10^{17} times the difference. In particular, if the shift is 17 or more, all the digits are dropped, <rounding> is 1 (not 0), and <X'₁> and <X'₂> are both zero.

If the shift is 1, the <rounding> digit is simply the only digit that was pushed out of the brace groups (this is important for subtraction). It would be more natural for the <rounding> digit to be placed after the <X'_i>, but the choice we make involves less reshuffling.

Note that this function treats negative <shift> as 0.

```

22587 \cs_new:Npn \_fp\_decimate:nNnnnn #1
22588 {
22589   \cs:w
22590     \_fp\_decimate\_
22591     \if_int_compare:w \_fp\_int\_eval:w #1 > \c\_fp\_prec\_int
22592       tiny
22593     \else:
22594       \_fp\_int\_to\_roman:w \_fp\_int\_eval:w #1
22595     \fi:
22596     :Nnnnn
22597   \cs_end:
22598 }

```

Each of the auxiliaries see the function $\langle f_1 \rangle$, followed by 4 blocks of 4 digits.

(End definition for `_fp_decimate:nNnnnn`.)

If the $\langle shift \rangle$ is zero, or too big, life is very easy.

```
\_fp\_decimate_:Nnnnn
\_fp\_decimate\_tiny:Nnnnn
22599 \cs_new:Npn \_fp\_decimate_:Nnnnn #1 #2#3#4#5
22600 { #1 0 {#2#3} {#4#5} ; }
22601 \cs_new:Npn \_fp\_decimate\_tiny:Nnnnn #1 #2#3#4#5
22602 { #1 1 { 0000 0000 } { 0000 0000 } 0 #2#3#4#5 ; }
```

(End definition for `_fp_decimate_:Nnnnn` and `_fp_decimate_tiny:Nnnnn`.)

```
\_fp\_decimate\_auxi:Nnnnn      \_fp\_decimate\_auxi:Nnnnn  $\langle f_1 \rangle$  { $\langle X_1 \rangle$ } { $\langle X_2 \rangle$ } { $\langle X_3 \rangle$ } { $\langle X_4 \rangle$ }
\_fp\_decimate\_auxii:Nnnnn      Shifting happens in two steps: compute the  $\langle rounding \rangle$  digit, and repack digits into
\_fp\_decimate\_auxiii:Nnnnn      two blocks of 8. The sixteen functions are very similar, and defined through \_fp\_tmp:w. The arguments are as follows: #1 indicates which function is being defined;
\_fp\_decimate\_auxiv:Nnnnn      after one step of expansion, #2 yields the “extra digits” which are then converted by
\_fp\_decimate\_auxv:Nnnnn      \_fp\_round\_digit:Nw to the  $\langle rounding \rangle$  digit (note the + separating blocks of digits to
\_fp\_decimate\_auxvi:Nnnnn      avoid overflowing TeX’s integers). This triggers the f-expansion of \_fp\_decimate\_pack:nnnnnnnnnw,10 responsible for building two blocks of 8 digits, and removing the
\_fp\_decimate\_auxvii:Nnnnn      rest. For this to work, #3 alternates between braced and unbraced blocks of 4 digits, in
\_fp\_decimate\_auxviii:Nnnnn     such a way that the 5 first and 5 next token groups yield the correct blocks of 8 digits.
\_fp\_decimate\_auxix:Nnnnn
\_fp\_decimate\_auxxx:Nnnnn
\_fp\_decimate\_auxxi:Nnnnn
\_fp\_decimate\_auxxii:Nnnnn
\_fp\_decimate\_auxxiii:Nnnnn
\_fp\_decimate\_auxxiv:Nnnnn
\_fp\_decimate\_auxxv:Nnnnn
\_fp\_decimate\_auxxvi:Nnnnn
22603 \cs_new:Npn \_fp\_tmp:w #1 #2 #3
22604 {
22605   \cs_new:cpn { \_fp\_decimate_ #1 :Nnnnn } ##1 ##2##3##4##5
22606   {
22607     \exp_after:wN ##1
22608     \int_value:w
22609     \exp_after:wN \_fp\_round\_digit:Nw #2 ;
22610     \_fp\_decimate\_pack:nnnnnnnnnw #3 ;
22611   }
22612 }
22613 \_fp\_tmp:w {i}   {\use\_none:nnn   #50}{ 0{#2}#3{#4}#5           }
22614 \_fp\_tmp:w {ii}  {\use\_none:nn    #5 }{ 00{#2}#3{#4}#5           }
22615 \_fp\_tmp:w {iii} {\use\_none:n     #5 }{ 000{#2}#3{#4}#5           }
22616 \_fp\_tmp:w {iv}  {                #5 }{ {0000}#2{#3}#4 #5       }
22617 \_fp\_tmp:w {v}   {\use\_none:nnn   #4#5 }{ 0{0000}#2{#3}#4 #5       }
22618 \_fp\_tmp:w {vi}  {\use\_none:nn    #4#5 }{ 00{0000}#2{#3}#4 #5       }
22619 \_fp\_tmp:w {vii} {\use\_none:n     #4#5 }{ 000{0000}#2{#3}#4 #5       }
22620 \_fp\_tmp:w {viii}{                #4#5 }{ {0000}0000{#2}#3 #4 #5       }
22621 \_fp\_tmp:w {ix}   {\use\_none:nnn   #3#4+#5}{ 0{0000}0000{#2}#3 #4 #5       }
22622 \_fp\_tmp:w {x}    {\use\_none:nn    #3#4+#5}{ 00{0000}0000{#2}#3 #4 #5       }
22623 \_fp\_tmp:w {xi}   {\use\_none:n     #3#4+#5}{ 000{0000}0000{#2}#3 #4 #5       }
22624 \_fp\_tmp:w {xii} {                #3#4+#5}{ {0000}0000{0000}#2 #3 #4 #5       }
22625 \_fp\_tmp:w {xiii}{\use\_none:nnn#2#3+#4#5}{ 0{0000}0000{0000}#2 #3 #4 #5       }
22626 \_fp\_tmp:w {xiv}  {\use\_none:nn    #2#3+#4#5}{ 00{0000}0000{0000}#2 #3 #4 #5       }
22627 \_fp\_tmp:w {xv}   {\use\_none:n     #2#3+#4#5}{ 000{0000}0000{0000}#2 #3 #4 #5       }
22628 \_fp\_tmp:w {xvi} {                #2#3+#4#5}{ {0000}0000{0000}0000 #2 #3 #4 #5 }
```

(End definition for `_fp_decimate_auxi:Nnnnn` and others.)

¹⁰No, the argument spec is not a mistake: the function calls an auxiliary to do half of the job.

`_fp_decimate_pack:nnnnnnnnnw`

The computation of the *rounding* digit leaves an unfinished `\int_value:w`, which expands the following functions. This allows us to repack nicely the digits we keep. Those digits come as an alternation of unbraced and braced blocks of 4 digits, such that the first 5 groups of token consist in 4 single digits, and one brace group (in some order), and the next 5 have the same structure. This is followed by some digits and a semicolon.

```
22629 \cs_new:Npn \_fp_decimate_pack:nnnnnnnnnw #1#2#3#4#5
22630 { \_fp_decimate_pack:nnnnnw { #1#2#3#4#5 } }
22631 \cs_new:Npn \_fp_decimate_pack:nnnnnw #1 #2#3#4#5#6
22632 { {#1} {#2#3#4#5#6} }
```

(End definition for `_fp_decimate_pack:nnnnnnnnnw`.)

66.10 Functions for use within primitive conditional branches

The functions described in this section are not pretty and can easily be misused. When correctly used, each of them removes one `\fi:` as part of its parameter text, and puts one back as part of its replacement text.

Many computation functions in `l3fp` must perform tests on the type of floating points that they receive. This is often done in an `\if_case:w` statement or another conditional statement, and only a few cases lead to actual computations: most of the special cases are treated using a few standard functions which we define now. A typical use context for those functions would be

```
\if_case:w <integer> \exp_stop_f:
  \_fp_case_return_o:Nw <fp var>
\or: \_fp_case_use:nw {<some computation>}
\or: \_fp_case_return_same_o:w
\or: \_fp_case_return:nw {<something>}
\fi:
<junk>
<floating point>
```

In this example, the case 0 returns the floating point `<fp var>`, expanding once after that floating point. Case 1 does `<some computation>` using the `<floating point>` (presumably compute the operation requested by the user in that non-trivial case). Case 2 returns the `<floating point>` without modifying it, removing the `<junk>` and expanding once after. Case 3 closes the conditional, removes the `<junk>` and the `<floating point>`, and expands `<something>` next. In other cases, the “`<junk>`” is expanded, performing some other operation on the `<floating point>`. We provide similar functions with two trailing `<floating points>`.

`_fp_case_use:nw`

This function ends a TeX conditional, removes junk until the next floating point, and places its first argument before that floating point, to perform some operation on the floating point.

```
22633 \cs_new:Npn \_fp_case_use:nw #1#2 \fi: #3 \s_fp { \fi: #1 \s_fp }
```

(End definition for `_fp_case_use:nw`.)

`__fp_case_return:nw` This function ends a \TeX conditional, removes junk and a floating point, and places its first argument in the input stream. A quirk is that we don't define this function requiring a floating point to follow, simply anything ending in a semicolon. This, in turn, means that the *<junk>* may not contain semicolons.

```
22634 \cs_new:Npn \__fp_case_return:nw #1#2 \fi: #3 ; { \fi: #1 }
```

(End definition for `__fp_case_return:nw`.)

`__fp_case_return_o:Nw` This function ends a \TeX conditional, removes junk and a floating point, and returns its first argument (an *<fp var>*) then expands once after it.

```
22635 \cs_new:Npn \__fp_case_return_o:Nw #1#2 \fi: #3 \s__fp #4 ;
22636 { \fi: \exp_after:wN #1 }
```

(End definition for `__fp_case_return_o:Nw`.)

`__fp_case_return_same_o:w` This function ends a \TeX conditional, removes junk, and returns the following floating point, expanding once after it.

```
22637 \cs_new:Npn \__fp_case_return_same_o:w #1 \fi: #2 \s__fp
22638 { \fi: \__fp_exp_after_o:w \s__fp }
```

(End definition for `__fp_case_return_same_o:w`.)

`__fp_case_return_o:Nww` Same as `__fp_case_return_o:Nw` but with two trailing floating points.

```
22639 \cs_new:Npn \__fp_case_return_o:Nww #1#2 \fi: #3 \s__fp #4 ; #5 ;
22640 { \fi: \exp_after:wN #1 }
```

(End definition for `__fp_case_return_o:Nww`.)

`__fp_case_return_i_o:ww` Similar to `__fp_case_return_same_o:w`, but this returns the first or second of two trailing floating point numbers, expanding once after the result.

```
22641 \cs_new:Npn \__fp_case_return_i_o:ww #1 \fi: #2 \s__fp #3 ; \s__fp #4 ;
22642 { \fi: \__fp_exp_after_o:w \s__fp #3 ; }
22643 \cs_new:Npn \__fp_case_return_ii_o:ww #1 \fi: #2 \s__fp #3 ;
22644 { \fi: \__fp_exp_after_o:w }
```

(End definition for `__fp_case_return_i_o:ww` and `__fp_case_return_ii_o:ww`.)

66.11 Integer floating points

`__fp_int_p:w` Tests if the floating point argument is an integer. For normal floating point numbers, `__fp_int:w` **TF** this holds if the rounding digit resulting from `__fp_decimate:nNnnnn` is 0.

```
22645 \prg_new_conditional:Npnn \__fp_int:w \s__fp \__fp_chk:w #1 #2 #3 #4;
22646 { TF , T , F , p }
22647 {
22648   \if_case:w #1 \exp_stop_f:
22649     \prg_return_true:
22650   \or:
22651     \if_charcode:w 0
22652       \__fp_decimate:nNnnnn { \c__fp_prec_int - #3 }
22653       \__fp_use_i_until_s:nw #4
22654       \prg_return_true:
22655     \else:
22656       \prg_return_false:
```

```

22657     \fi:
22658   \else: \prg_return_false:
22659     \fi:
22660   }

```

(End definition for _fp_int:wTF.)

66.12 Small integer floating points

```

\_fp_small_int:wTF
\_fp_small_int_true:wTF
\_fp_small_int_normal:NnwTF
\_fp_small_int_test:NnnwNTF

```

Tests if the floating point argument is an integer or $\pm\infty$. If so, it is clipped to an integer in the range $[-10^8, 10^8]$ and fed as a braced argument to the *⟨true code⟩*. Otherwise, the *⟨false code⟩* is performed.

First filter special cases: zeros and infinities are integers, `nan` is not. For normal numbers, decimate. If the rounding digit is not 0 run the *⟨false code⟩*. If it is, then the integer is #2 #3; use #3 if #2 vanishes and otherwise 10^8 .

```

22661 \cs_new:Npn \_fp_small_int:wTF \s_fp \_fp_chk:w #1#2
22662 {
22663   \if_case:w #1 \exp_stop_f:
22664     \_fp_case_return:nw { \_fp_small_int_true:wTF 0 ; }
22665   \or:   \exp_after:wN \_fp_small_int_normal:NnwTF
22666   \or:
22667     \_fp_case_return:nw
22668     {
22669       \exp_after:wN \_fp_small_int_true:wTF \int_value:w
22670       \if_meaning:w 2 #2 - \fi: 1 0000 0000 ;
22671     }
22672   \else: \_fp_case_return:nw \use_ii:nn
22673   \fi:
22674   #2
22675 }
22676 \cs_new:Npn \_fp_small_int_true:wTF #1; #2#3 { #2 {#1} }
22677 \cs_new:Npn \_fp_small_int_normal:NnwTF #1#2#3;
22678 {
22679   \_fp_decimate:nNnnnn { \c_fp_prec_int - #2 }
22680   \_fp_small_int_test:NnnwNw
22681   #3 #1
22682 }
22683 \cs_new:Npn \_fp_small_int_test:NnnwNw #1#2#3#4; #5
22684 {
22685   \if_meaning:w 0 #1
22686     \exp_after:wN \_fp_small_int_true:wTF
22687     \int_value:w \if_meaning:w 2 #5 - \fi:
22688     \if_int_compare:w #2 > \c_zero_int
22689       1 0000 0000
22690     \else:
22691       #3
22692     \fi:
22693     \exp_after:wN ;
22694   \else:
22695     \exp_after:wN \use_ii:nn
22696   \fi:
22697 }

```

(End definition for _fp_small_int:wTF and others.)

66.13 Fast string comparison

`__fp_str_if_eq:nn` A private version of the low-level string comparison function.

```
22698 \cs_new_eq:NN \__fp_str_if_eq:nn \tex_strcmp:D
```

(End definition for __fp_str_if_eq:nn.)

66.14 Name of a function from its l3fp-parse name

`__fp_func_to_name:N` The goal is to convert for instance `__fp_sin_o:w` to `sin`. This is used in error messages hence does not need to be fast.

```
22699 \cs_new:Npn \__fp_func_to_name:N #1
22700 {
22701   \exp_last_unbraced:Nf
22702   \__fp_func_to_name_aux:w { \cs_to_str:N #1 } X
22703 }
22704 \cs_set_protected:Npn \__fp_tmp:w #1 #2
22705 { \cs_new:Npn \__fp_func_to_name_aux:w ##1 #1 ##2 #2 ##3 X {##2} }
22706 \exp_args:Nff \__fp_tmp:w { \tl_to_str:n { __fp_ } }
22707 { \tl_to_str:n { _o: } }
```

(End definition for __fp_func_to_name:N and __fp_func_to_name_aux:w.)

66.15 Messages

Using a floating point directly is an error.

```
22708 \msg_new:nnnn { fp } { misused }
22709 { A~floating~point~with~value~'#1'~was~misused. }
22710 {
22711   To~obtain~the~value~of~a~floating~point~variable,~use~
22712   '\token_to_str:N \fp_to_decimal:N',~
22713   '\token_to_str:N \fp_to_tl:N',~or~other~
22714   conversion~functions.
22715 }
22716 \prop_gput:Nnn \g_msg_module_name_prop { fp } { LaTeX3 }
22717 \prop_gput:Nnn \g_msg_module_type_prop { fp } { }
22718 </package>
```

Chapter 67

l3fp-traps Implementation

22719 `*package`

22720 `\@@=fp`

Exceptions should be accessed by an `n`-type argument, among

- `invalid_operation`
- `division_by_zero`
- `overflow`
- `underflow`
- `inexact` (actually never used).

67.1 Flags

Flags to denote exceptions.

`flag_fp_invalid_operation`
`flag_fp_division_by_zero`
`flag_fp_overflow`
`flag_fp_underflow`

22721 `\flag_new:n { fp_invalid_operation }`

22722 `\flag_new:n { fp_division_by_zero }`

22723 `\flag_new:n { fp_overflow }`

22724 `\flag_new:n { fp_underflow }`

(End definition for flag `fp_invalid_operation` and others. These variables are documented on page 248.)

67.2 Traps

Exceptions can be trapped to obtain custom behaviour. When an invalid operation or a division by zero is trapped, the trap receives as arguments the result as an `N`-type floating point number, the function name (multiple letters for prefix operations, or a single symbol for infix operations), and the operand(s). When an overflow or underflow is trapped, the trap receives the resulting overly large or small floating point number if it is not too big, otherwise it receives $+\infty$. Currently, the `inexact` exception is entirely ignored.

The behaviour when an exception occurs is controlled by the definitions of the functions

- `_fp_invalid_operation:nnw`,

- __fp_invalid_operation_o:Nww,
- __fp_invalid_operation_tl_o:ff,
- __fp_division_by_zero_o:Nnw,
- __fp_division_by_zero_o:NNww,
- __fp_overflow:w,
- __fp_underflow:w.

Rather than changing them directly, we provide a user interface as \fp_trap:nn {<exception>} {<way of trapping>}, where the <way of trapping> is one of **error**, **flag**, or **none**.

We also provide __fp_invalid_operation_o:nw, defined in terms of __fp_invalid_operation:nnw.

\fp_trap:nn

```

22725 \cs_new_protected:Npn \fp_trap:nn #1#2
22726   {
22727     \cs_if_exist_use:cF { __fp_trap_#1_set_#2: }
22728     {
22729       \clist_if_in:nnTF
22730       { invalid_operation , division_by_zero , overflow , underflow }
22731       {#1}
22732       {
22733         \msg_error:nnxx { fp }
22734         { unknown-fpu-trap-type } {#1} {#2}
22735       }
22736       {
22737         \msg_error:nnx
22738         { fp } { unknown-fpu-exception } {#1}
22739       }
22740     }
22741   }

```

(End definition for \fp_trap:nn. This function is documented on page 248.)

_fp_trap_invalid_operation_set_error: We provide three types of trapping for invalid operations: either produce an error and raise the relevant flag; or only raise the flag; or don't even raise the flag. In most cases, the function produces as a result its first argument, possibly with post-expansion.

_fp_trap_invalid_operation_set_flag:

_fp_trap_invalid_operation_set_none:

_fp_trap_invalid_operation_set:N

```

22742 \cs_new_protected:Npn \__fp_trap_invalid_operation_set_error:
22743   { \__fp_trap_invalid_operation_set:N \prg_do_nothing: }
22744 \cs_new_protected:Npn \__fp_trap_invalid_operation_set_flag:
22745   { \__fp_trap_invalid_operation_set:N \use_none:nnnnn }
22746 \cs_new_protected:Npn \__fp_trap_invalid_operation_set_none:
22747   { \__fp_trap_invalid_operation_set:N \use_none:nnnnnnn }
22748 \cs_new_protected:Npn \__fp_trap_invalid_operation_set:N #1
22749   {
22750     \exp_args:Nno \use:n
22751     { \cs_set:Npn \__fp_invalid_operation:nnw ##1##2##3; }
22752     {
22753       #1
22754       \__fp_error:nnfn { invalid } {##2} { \fp_to_tl:n { ##3; } } { }

```

```

22755     \flag_raise_if_clear:n { fp_invalid_operation }
22756     ##1
22757   }
22758   \exp_args:Nno \use:n
22759   { \cs_set:Npn \__fp_invalid_operation_o:Nww ##1##2; ##3; }
22760   {
22761     #1
22762     \__fp_error:nffn { invalid-ii }
22763     { \fp_to_tl:n { ##2; } } { \fp_to_tl:n { ##3; } } {##1}
22764     \flag_raise_if_clear:n { fp_invalid_operation }
22765     \exp_after:wN \c_nan_fp
22766   }
22767   \exp_args:Nno \use:n
22768   { \cs_set:Npn \__fp_invalid_operation_tl_o:ff ##1##2 }
22769   {
22770     #1
22771     \__fp_error:nffn { invalid } {##1} {##2} { }
22772     \flag_raise_if_clear:n { fp_invalid_operation }
22773     \exp_after:wN \c_nan_fp
22774   }
22775 }

```

(End definition for __fp_trap_invalid_operation_set_error: and others.)

__fp_trap_division_by_zero_set_error: We provide three types of trapping for invalid operations and division by zero: either
 __fp_trap_division_by_zero_set_flag: produce an error and raise the relevant flag; or only raise the flag; or don't even raise the
 __fp_trap_division_by_zero_set_none: flag. In all cases, the function must produce a result, namely its first argument, $\pm\infty$ or
 __fp_trap_division_by_zero_set:N nan.

```

22776 \cs_new_protected:Npn \__fp_trap_division_by_zero_set_error:
22777 { \__fp_trap_division_by_zero_set:N \prg_do_nothing: }
22778 \cs_new_protected:Npn \__fp_trap_division_by_zero_set_flag:
22779 { \__fp_trap_division_by_zero_set:N \use_none:nnnnn }
22780 \cs_new_protected:Npn \__fp_trap_division_by_zero_set_none:
22781 { \__fp_trap_division_by_zero_set:N \use_none:nnnnnnn }
22782 \cs_new_protected:Npn \__fp_trap_division_by_zero_set:N #1
22783 {
22784   \exp_args:Nno \use:n
22785   { \cs_set:Npn \__fp_division_by_zero_o:Nnw ##1##2##3; }
22786   {
22787     #1
22788     \__fp_error:nnfn { zero-div } {##2} { \fp_to_tl:n { ##3; } } { }
22789     \flag_raise_if_clear:n { fp_division_by_zero }
22790     \exp_after:wN ##1
22791   }
22792   \exp_args:Nno \use:n
22793   { \cs_set:Npn \__fp_division_by_zero_o:NNww ##1##2##3; ##4; }
22794   {
22795     #1
22796     \__fp_error:nffn { zero-div-ii }
22797     { \fp_to_tl:n { ##3; } } { \fp_to_tl:n { ##4; } } {##2}
22798     \flag_raise_if_clear:n { fp_division_by_zero }
22799     \exp_after:wN ##1
22800   }
22801 }

```

(End definition for `_fp_trap_division_by_zero_set_error:` and others.)

`_fp_trap_overflow_set_error:` Just as for invalid operations and division by zero, the three different behaviours are
`_fp_trap_overflow_set_flag:` obtained by feeding `\prg_do_nothing:`, `\use_none:nnnnn` or `\use_none:nnnnnnn` to an
`_fp_trap_overflow_set_none:` auxiliary, with a further auxiliary common to overflow and underflow functions. In most
`_fp_trap_overflow_set:N` cases, the argument of the `_fp_overflow:w` and `_fp_underflow:w` functions will
`_fp_trap_underflow_set_error:` be an (almost) normal number (with an exponent outside the allowed range), and the
`_fp_trap_underflow_set_flag:` error message thus displays that number together with the result to which it overflowed
`_fp_trap_underflow_set_none:` or underflowed. For extreme cases such as 10^{9999} , the exponent would be too
`_fp_trap_underflow_set:N` large for T_EX, and `_fp_overflow:w` receives $\pm\infty$ (`_fp_underflow:w` would receive
`_fp_trap_overflow_set:NnNn` ± 0); then we cannot do better than simply say an overflow or underflow occurred.

```

22802 \cs_new_protected:Npn \_fp_trap_overflow_set_error:
22803   { \_fp_trap_overflow_set:N \prg_do_nothing: }
22804 \cs_new_protected:Npn \_fp_trap_overflow_set_flag:
22805   { \_fp_trap_overflow_set:N \use_none:nnnnn }
22806 \cs_new_protected:Npn \_fp_trap_overflow_set_none:
22807   { \_fp_trap_overflow_set:N \use_none:nnnnnnn }
22808 \cs_new_protected:Npn \_fp_trap_overflow_set:N #1
22809   { \_fp_trap_overflow_set:NnNn #1 { overflow } \_fp_inf_fp:N { inf } }
22810 \cs_new_protected:Npn \_fp_trap_underflow_set_error:
22811   { \_fp_trap_underflow_set:N \prg_do_nothing: }
22812 \cs_new_protected:Npn \_fp_trap_underflow_set_flag:
22813   { \_fp_trap_underflow_set:N \use_none:nnnnn }
22814 \cs_new_protected:Npn \_fp_trap_underflow_set_none:
22815   { \_fp_trap_underflow_set:N \use_none:nnnnnnn }
22816 \cs_new_protected:Npn \_fp_trap_underflow_set:N #1
22817   { \_fp_trap_overflow_set:NnNn #1 { underflow } \_fp_zero_fp:N { 0 } }
22818 \cs_new_protected:Npn \_fp_trap_overflow_set:NnNn #1#2#3#4
22819   {
22820     \exp_args:Nno \use:n
22821     { \cs_set:cpn { \_fp_ #2 :w } \s_fp \_fp_chk:w ##1##2##3; }
22822     {
22823       #1
22824       \_fp_error:nffn
22825       { flow \if_meaning:w 1 ##1 -to \fi: }
22826       { \fp_to_tl:n { \s_fp \_fp_chk:w ##1##2##3; } }
22827       { \token_if_eq_meaning:NNF 0 ##2 { - } #4 }
22828       {#2}
22829       \flag_raise_if_clear:n { fp_#2 }
22830       #3 ##2
22831     }
22832   }

```

(End definition for `_fp_trap_overflow_set_error:` and others.)

`_fp_invalid_operation:nnw` Initialize the control sequences (to log properly their existence). Then set invalid opera-
`_fp_invalid_operation_o:Nnw` tions to trigger an error, and division by zero, overflow, and underflow to act silently on
`_fp_invalid_operation_tl_o:ff` their flag.

```

22833 \cs_new:Npn \_fp_invalid_operation:nnw #1#2#3; { }
22834 \cs_new:Npn \_fp_invalid_operation_o:Nnw #1#2; #3; { }
22835 \cs_new:Npn \_fp_invalid_operation_tl_o:ff #1 #2 { }
22836 \cs_new:Npn \_fp_division_by_zero_o:Nnw #1#2#3; { }
22837 \cs_new:Npn \_fp_division_by_zero_o:NNnw #1#2#3; #4; { }

```



```

22838 \cs_new:Npn \__fp_overflow:w { }
22839 \cs_new:Npn \__fp_underflow:w { }
22840 \fp_trap:nn { invalid_operation } { error }
22841 \fp_trap:nn { division_by_zero } { flag }
22842 \fp_trap:nn { overflow } { flag }
22843 \fp_trap:nn { underflow } { flag }

```

(End definition for __fp_invalid_operation:nnw and others.)

__fp_invalid_operation_o:nw Convenient short-hands for returning \c_nan_fp for a unary or binary operation, and
 __fp_invalid_operation_o:fw expanding after.

```

22844 \cs_new:Npn \__fp_invalid_operation_o:nw
22845 { \__fp_invalid_operation:nnw { \exp_after:wN \c_nan_fp } }
22846 \cs_generate_variant:Nn \__fp_invalid_operation_o:nw { f }

```

(End definition for __fp_invalid_operation_o:nw.)

67.3 Errors

```

\__fp_error:nnnn
\__fp_error:nmfn
\__fp_error:nffn
\__fp_error:nfff
22847 \cs_new:Npn \__fp_error:nnnn
22848 { \msg_expandable_error:nnnnn { fp } }
22849 \cs_generate_variant:Nn \__fp_error:nnnn { nnf, nff , nfff }

```

(End definition for __fp_error:nnnn.)

67.4 Messages

Some messages.

```

22850 \msg_new:nnnn { fp } { unknown-fpu-exception }
22851 {
22852   The-FPU-exception-~'#1'~is-not-known:~
22853   that-trap-will-never-be-triggered.
22854 }
22855 {
22856   The-only-exceptions-to-which-traps-can-be-attached-are \\
22857   \iow_indent:n
22858   {
22859     * ~ invalid_operation \\
22860     * ~ division_by_zero \\
22861     * ~ overflow \\
22862     * ~ underflow
22863   }
22864 }
22865 \msg_new:nnnn { fp } { unknown-fpu-trap-type }
22866 { The-FPU-trap-type-~'#2'~is-not-known. }
22867 {
22868   The-trap-type-must-be-one-of \\
22869   \iow_indent:n
22870   {
22871     * ~ error \\
22872     * ~ flag \\

```

```

22873         * ~ none
22874     }
22875 }
22876 \msg_new:nnn { fp } { flow }
22877 { An ~ #3 ~ occurred. }
22878 \msg_new:nnn { fp } { flow-to }
22879 { #1 ~ #3 ed ~ to ~ #2 . }
22880 \msg_new:nnn { fp } { zero-div }
22881 { Division-by-zero-in~ #1 (#2) }
22882 \msg_new:nnn { fp } { zero-div-ii }
22883 { Division-by-zero-in~ (#1) #3 (#2) }
22884 \msg_new:nnn { fp } { invalid }
22885 { Invalid-operation~ #1 (#2) }
22886 \msg_new:nnn { fp } { invalid-ii }
22887 { Invalid-operation~ (#1) #3 (#2) }
22888 \msg_new:nnn { fp } { unknown-type }
22889 { Unknown-type-for~'#1' }
22890 \end{package}

```

Chapter 68

l3fp-round implementation

```
22891 \package
22892 \@@=fp

\__fp_parse_word_trunc:N
\__fp_parse_word_floor:N
\__fp_parse_word_ceil:N
22893 \cs_new:Npn \__fp_parse_word_trunc:N
22894 { \__fp_parse_function:NNN \__fp_round_o:Nw \__fp_round_to_zero:NNN }
22895 \cs_new:Npn \__fp_parse_word_floor:N
22896 { \__fp_parse_function:NNN \__fp_round_o:Nw \__fp_round_to_ninf:NNN }
22897 \cs_new:Npn \__fp_parse_word_ceil:N
22898 { \__fp_parse_function:NNN \__fp_round_o:Nw \__fp_round_to_pinf:NNN }

(End definition for \__fp_parse_word_trunc:N, \__fp_parse_word_floor:N, and \__fp_parse_word_ceil:N.)

\__fp_parse_word_round:N
\__fp_parse_round:Nw
22899 \cs_new:Npn \__fp_parse_word_round:N #1#2
22900 {
22901   \__fp_parse_function:NNN
22902   \__fp_round_o:Nw \__fp_round_to_nearest:NNN #1
22903   #2
22904 }
22905 \cs_new:Npn \__fp_parse_round:Nw #1 #2 \__fp_round_to_nearest:NNN #3#4
22906 { #2 #1 #3 }
22907

(End definition for \__fp_parse_word_round:N and \__fp_parse_round:Nw.)
```

68.1 Rounding tools

\c__fp_five_int This is used as the half-point for which numbers are rounded up/down.

```
22908 \int_const:Nn \c__fp_five_int { 5 }
```

(End definition for \c__fp_five_int.)

Floating point operations often yield a result that cannot be exactly represented in a significand with 16 digits. In that case, we need to round the exact result to a representable number. The IEEE standard defines four rounding modes:

- Round to nearest: round to the representable floating point number whose absolute difference with the exact result is the smallest. If the exact result lies exactly at the mid-point between two consecutive representable floating point numbers, round to the floating point number whose last digit is even.
- Round towards negative infinity: round to the greatest floating point number not larger than the exact result.
- Round towards zero: round to a floating point number with the same sign as the exact result, with the largest absolute value not larger than the absolute value of the exact result.
- Round towards positive infinity: round to the least floating point number not smaller than the exact result.

This is not fully implemented in `l3fp` yet, and transcendental functions fall back on the “round to nearest” mode. All rounding for basic algebra is done through the functions defined in this module, which can be redefined to change their rounding behaviour (but there is not interface for that yet).

The rounding tools available in this module are many variations on a base function `__fp_round:NNN`, which expands to `0\exp_stop_f:` or `1\exp_stop_f:` depending on whether the final result should be rounded up or down.

- `__fp_round:NNN <sign> <digit1> <digit2>` can expand to `0\exp_stop_f:` or `1\exp_stop_f:`.
- `__fp_round_s:NNNw <sign> <digit1> <digit2> <more digits>`; can expand to `0\exp_stop_f:`; or `1\exp_stop_f:`;
- `__fp_round_neg:NNN <sign> <digit1> <digit2>` can expand to `0\exp_stop_f:` or `1\exp_stop_f:`.

See implementation comments for details on the syntax.

```
\__fp_round:NNN
\__fp_round_to_nearest:NNN
  \_fp_round_to_nearest_ninf:NNN
  \_fp_round_to_nearest_zero:NNN
  \_fp_round_to_nearest_pinf:NNN
\__fp_round_to_ninf:NNN
\__fp_round_to_zero:NNN
\__fp_round_to_pinf:NNN
```

```
\__fp_round:NNN <final sign> <digit1> <digit2>
```

If rounding the number `<final sign><digit1>.<digit2>` to an integer rounds it towards zero (truncates it), this function expands to `0\exp_stop_f:`, and otherwise to `1\exp_stop_f:`. Typically used within the scope of an `__fp_int_eval:w`, to add 1 if needed, and thereby round correctly. The result depends on the rounding mode.

It is very important that `<final sign>` be the final sign of the result. Otherwise, the result would be incorrect in the case of rounding towards $-\infty$ or towards $+\infty$. Also recall that `<final sign>` is 0 for positive, and 2 for negative.

By default, the functions below return `0\exp_stop_f:`, but this is superseded by `__fp_round_return_one:`, which instead returns `1\exp_stop_f:`, expanding everything and removing `0\exp_stop_f:` in the process. In the case of rounding towards $\pm\infty$ or towards 0, this is not really useful, but it prepares us for the “round to nearest, ties to even” mode.

The “round to nearest” mode is the default. If the `<digit2>` is larger than 5, then round up. If it is less than 5, round down. If it is exactly 5, then round such that `<digit1>` plus the result is even. In other words, round up if `<digit1>` is odd.

The “round to nearest” mode has three variants, which differ in how ties are rounded: down towards $-\infty$, truncated towards 0, or up towards $+\infty$.

```
22909 \cs_new:Npn \__fp_round_return_one:
22910 { \exp_after:wN 1 \exp_after:wN \exp_stop_f: \exp:w }
```

```

22911 \cs_new:Npn \__fp_round_to_ninf:NNN #1 #2 #3
22912 {
22913     \if_meaning:w 2 #1
22914         \if_int_compare:w #3 > \c_zero_int
22915             \__fp_round_return_one:
22916         \fi:
22917     \fi:
22918     \c_zero_int
22919 }
22920 \cs_new:Npn \__fp_round_to_zero:NNN #1 #2 #3 { \c_zero_int }
22921 \cs_new:Npn \__fp_round_to_pinf:NNN #1 #2 #3
22922 {
22923     \if_meaning:w 0 #1
22924         \if_int_compare:w #3 > \c_zero_int
22925             \__fp_round_return_one:
22926         \fi:
22927     \fi:
22928     \c_zero_int
22929 }
22930 \cs_new:Npn \__fp_round_to_nearest:NNN #1 #2 #3
22931 {
22932     \if_int_compare:w #3 > \c__fp_five_int
22933         \__fp_round_return_one:
22934     \else:
22935         \if_meaning:w 5 #3
22936             \if_int_odd:w #2 \exp_stop_f:
22937             \__fp_round_return_one:
22938         \fi:
22939     \fi:
22940     \c_zero_int
22941 }
22942 }
22943 \cs_new:Npn \__fp_round_to_nearest_ninf:NNN #1 #2 #3
22944 {
22945     \if_int_compare:w #3 > \c__fp_five_int
22946         \__fp_round_return_one:
22947     \else:
22948         \if_meaning:w 5 #3
22949             \if_meaning:w 2 #1
22950                 \__fp_round_return_one:
22951             \fi:
22952         \fi:
22953     \fi:
22954     \c_zero_int
22955 }
22956 \cs_new:Npn \__fp_round_to_nearest_zero:NNN #1 #2 #3
22957 {
22958     \if_int_compare:w #3 > \c__fp_five_int
22959         \__fp_round_return_one:
22960     \fi:
22961     \c_zero_int
22962 }
22963 \cs_new:Npn \__fp_round_to_nearest_pinf:NNN #1 #2 #3
22964 {

```

```

22965 \if_int_compare:w #3 > \c__fp_five_int
22966 \__fp_round_return_one:
22967 \else:
22968 \if_meaning:w 5 #3
22969 \if_meaning:w 0 #1
22970 \__fp_round_return_one:
22971 \fi:
22972 \fi:
22973 \fi:
22974 \c_zero_int
22975 }
22976 \cs_new_eq:NN \__fp_round:NNN \__fp_round_to_nearest:NNN

```

(End definition for __fp_round:NNN and others.)

__fp_round_s:NNNw __fp_round_s:NNNw *<final sign>* *<digit>* *<more digits>* ;

Similar to __fp_round:NNN, but with an extra semicolon, this function expands to 0\exp_stop_f:; if rounding *<final sign>**<digit>**<more digits>* to an integer truncates, and to 1\exp_stop_f:; otherwise. The *<more digits>* part must be a digit, followed by something that does not overflow a \int_use:N __fp_int_eval:w construction. The only relevant information about this piece is whether it is zero or not.

```

22977 \cs_new:Npn \__fp_round_s:NNNw #1 #2 #3 #4;
22978 {
22979 \exp_after:wN \__fp_round:NNN
22980 \exp_after:wN #1
22981 \exp_after:wN #2
22982 \int_value:w \__fp_int_eval:w
22983 \if_int_odd:w 0 \if_meaning:w 0 #3 1 \fi:
22984 \if_meaning:w 5 #3 1 \fi:
22985 \exp_stop_f:
22986 \if_int_compare:w \__fp_int_eval:w #4 > \c_zero_int
22987 1 +
22988 \fi:
22989 \fi:
22990 #3
22991 ;
22992 }

```

(End definition for __fp_round_s:NNNw.)

__fp_round_digit:Nw \int_value:w __fp_round_digit:Nw *<digit>* *<intexpr>* ;

This function should always be called within an \int_value:w or __fp_int_eval:w expansion; it may add an extra __fp_int_eval:w, which means that the integer or integer expression should not be ended with a synonym of \relax, but with a semi-colon for instance.

```

22993 \cs_new:Npn \__fp_round_digit:Nw #1 #2;
22994 {
22995 \if_int_odd:w \if_meaning:w 0 #1 1 \else:
22996 \if_meaning:w 5 #1 1 \else:
22997 0 \fi: \fi: \exp_stop_f:
22998 \if_int_compare:w \__fp_int_eval:w #2 > \c_zero_int
22999 \__fp_int_eval:w 1 +
23000 \fi:
23001 \fi:

```

```

23002     #1
23003 }

```

(End definition for `_fp_round_digit:Nw`.)

```

\_fp_round_neg:NNN
\_fp_round_to_nearest_neg:NNN
\_fp_round_to_nearest_ninf_neg:NNN
\_fp_round_to_nearest_zero_neg:NNN
\_fp_round_to_nearest_pinf_neg:NNN
\_fp_round_to_ninf_neg:NNN
\_fp_round_to_zero_neg:NNN
\_fp_round_to_pinf_neg:NNN

```

```

\_fp_round_neg:NNN <final sign> <digit1> <digit2>

```

This expands to `0\exp_stop_f:` or `1\exp_stop_f:` after doing the following test. Starting from a number of the form $\langle final\ sign \rangle 0.\langle 15\ digits \rangle \langle digit_1 \rangle$ with exactly 15 (non-all-zero) digits before $\langle digit_1 \rangle$, subtract from it $\langle final\ sign \rangle 0.0\dots 0 \langle digit_2 \rangle$, where there are 16 zeros. If in the current rounding mode the result should be rounded down, then this function returns `1\exp_stop_f:`. Otherwise, *i.e.*, if the result is rounded back to the first operand, then this function returns `0\exp_stop_f:`.

It turns out that this negative “round to nearest” is identical to the positive one. And this is the default mode.

```

23004 \cs_new_eq:NN \_fp_round_to_ninf_neg:NNN \_fp_round_to_pinf:NNN
23005 \cs_new:Npn \_fp_round_to_zero_neg:NNN #1 #2 #3
23006 {
23007     \if_int_compare:w #3 > \c_zero_int
23008         \_fp_round_return_one:
23009     \fi:
23010     \c_zero_int
23011 }
23012 \cs_new_eq:NN \_fp_round_to_pinf_neg:NNN \_fp_round_to_ninf:NNN
23013 \cs_new_eq:NN \_fp_round_to_nearest_neg:NNN \_fp_round_to_nearest:NNN
23014 \cs_new_eq:NN \_fp_round_to_nearest_ninf_neg:NNN
23015     \_fp_round_to_nearest_pinf:NNN
23016 \cs_new:Npn \_fp_round_to_nearest_zero_neg:NNN #1 #2 #3
23017 {
23018     \if_int_compare:w #3 < \c__fp_five_int \else:
23019         \_fp_round_return_one:
23020     \fi:
23021     \c_zero_int
23022 }
23023 \cs_new_eq:NN \_fp_round_to_nearest_pinf_neg:NNN
23024     \_fp_round_to_nearest_ninf:NNN
23025 \cs_new_eq:NN \_fp_round_neg:NNN \_fp_round_to_nearest_neg:NNN

```

(End definition for `_fp_round_neg:NNN` and others.)

68.2 The round function

```

\_fp_round_o:Nw
\_fp_round_aux_o:Nw

```

First check that all arguments are floating point numbers. The `trunc`, `ceil` and `floor` functions expect one or two arguments (the second is 0 by default), and the `round` function also accepts a third argument (`nan` by default), which changes `#1` from `_fp_round_to_nearest:NNN` to one of its analogues.

```

23026 \cs_new:Npn \_fp_round_o:Nw #1
23027 {
23028     \_fp_parse_function_all_fp_o:fnw
23029     { \_fp_round_name_from_cs:N #1 }
23030     { \_fp_round_aux_o:Nw #1 }
23031 }
23032 \cs_new:Npn \_fp_round_aux_o:Nw #1#2 @

```

```

23033 {
23034   \if_case:w
23035     \__fp_int_eval:w \__fp_array_count:n {#2} \__fp_int_eval_end:
23036     \__fp_round_no_arg_o:Nw #1 \exp:w
23037   \or: \__fp_round:Nwn #1 #2 {0} \exp:w
23038   \or: \__fp_round:Nww #1 #2 \exp:w
23039   \else: \__fp_round:Nwww #1 #2 @ \exp:w
23040   \fi:
23041   \exp_after:wN \exp_end:
23042 }

```

(End definition for __fp_round_o:Nw and __fp_round_aux_o:Nw.)

__fp_round_no_arg_o:Nw

```

23043 \cs_new:Npn \__fp_round_no_arg_o:Nw #1
23044 {
23045   \cs_if_eq:NNTF #1 \__fp_round_to_nearest:NNN
23046   { \__fp_error:nnnn { num-args } { round () } { 1 } { 3 } }
23047   {
23048     \__fp_error:nffn { num-args }
23049     { \__fp_round_name_from_cs:N #1 () } { 1 } { 2 }
23050   }
23051   \exp_after:wN \c_nan_fp
23052 }

```

(End definition for __fp_round_no_arg_o:Nw.)

__fp_round:Nwww Having three arguments is only allowed for round, not trunc, ceil, floor, so check for that case. If all is well, construct one of __fp_round_to_nearest:NNN, __fp_round_to_nearest_zero:NNN, __fp_round_to_nearest_ninf:NNN, __fp_round_to_nearest_pinf:NNN and act accordingly.

```

23053 \cs_new:Npn \__fp_round:Nwww #1#2 ; #3 ; \s__fp \__fp_chk:w #4#5#6 ; #7 @
23054 {
23055   \cs_if_eq:NNTF #1 \__fp_round_to_nearest:NNN
23056   {
23057     \tl_if_empty:nTF {#7}
23058     {
23059       \exp_args:Nc \__fp_round:Nww
23060       {
23061         \__fp_round_to_nearest
23062         \if_meaning:w 0 #4 _zero \else:
23063         \if_case:w #5 \exp_stop_f: _pinf \or: \else: _ninf \fi: \fi:
23064         :NNN
23065       }
23066       #2 ; #3 ;
23067     }
23068     {
23069       \__fp_error:nnnn { num-args } { round () } { 1 } { 3 }
23070       \exp_after:wN \c_nan_fp
23071     }
23072   }
23073   {
23074     \__fp_error:nffn { num-args }
23075     { \__fp_round_name_from_cs:N #1 () } { 1 } { 2 }

```



```

23076         \exp_after:wN \c_nan_fp
23077     }
23078 }

```

(End definition for _fp_round:Nwww.)

_fp_round_name_from_cs:N

```

23079 \cs_new:Npn \_fp_round_name_from_cs:N #1
23080 {
23081     \cs_if_eq:NNTF #1 \_fp_round_to_zero:NNN { trunc }
23082     {
23083         \cs_if_eq:NNTF #1 \_fp_round_to_ninf:NNN { floor }
23084         {
23085             \cs_if_eq:NNTF #1 \_fp_round_to_pinf:NNN { ceil }
23086             { round }
23087         }
23088     }
23089 }

```

(End definition for _fp_round_name_from_cs:N.)

_fp_round:Nww

_fp_round:Nwn

If the number of digits to round to is an integer or infinity all is good; if it is nan then just produce a nan; otherwise invalid as we have something like round(1,3.14) where the number of digits is not an integer.

_fp_round_normal:NwNNnw

_fp_round_normal:NnnwNNnn

_fp_round_pack:Nw

```

23090 \cs_new:Npn \_fp_round:Nww #1#2 ; #3 ;
23091 {
23092     \_fp_small_int:wTF #3; { \_fp_round:Nwn #1#2; }
23093     {
23094         \if:w 3 \_fp_kind:w #3 ;
23095         \exp_after:wN \use_i:nn
23096         \else:
23097             \exp_after:wN \use_ii:nn
23098         \fi:
23099         { \exp_after:wN \c_nan_fp }
23100         {
23101             \_fp_invalid_operation_tl_o:ff
23102             { \_fp_round_name_from_cs:N #1 }
23103             { \_fp_array_to_clist:n { #2; #3; } }
23104         }
23105     }
23106 }
23107 \cs_new:Npn \_fp_round:Nwn #1 \s_fp \_fp_chk:w #2#3#4; #5
23108 {
23109     \if_meaning:w 1 #2
23110     \exp_after:wN \_fp_round_normal:NwNNnw
23111     \exp_after:wN #1
23112     \int_value:w #5
23113     \else:
23114         \exp_after:wN \_fp_exp_after_o:w
23115     \fi:
23116     \s_fp \_fp_chk:w #2#3#4;
23117 }
23118 \cs_new:Npn \_fp_round_normal:NwNNnw #1#2 \s_fp \_fp_chk:w 1#3#4#5;
23119 {

```

```

23120     \__fp_decimate:nNnnnn { \c__fp_prec_int - #4 - #2 }
23121     \__fp_round_normal:NnnwNNnn #5 #1 #3 {#4} {#2}
23122 }
23123 \cs_new:Npn \__fp_round_normal:NnnwNNnn #1#2#3#4; #5#6
23124 {
23125     \exp_after:wN \__fp_round_normal:NNwNnn
23126     \int_value:w \__fp_int_eval:w
23127     \if_int_compare:w #2 > \c_zero_int
23128     1 \int_value:w #2
23129     \exp_after:wN \__fp_round_pack:Nw
23130     \int_value:w \__fp_int_eval:w 1#3 +
23131     \else:
23132     \if_int_compare:w #3 > \c_zero_int
23133     1 \int_value:w #3 +
23134     \fi:
23135     \fi:
23136     \exp_after:wN #5
23137     \exp_after:wN #6
23138     \use_none:nnnnnnn #3
23139     #1
23140     \__fp_int_eval_end:
23141     0000 0000 0000 0000 ; #6
23142 }
23143 \cs_new:Npn \__fp_round_pack:Nw #1
23144 { \if_meaning:w 2 #1 + 1 \fi: \__fp_int_eval_end: }
23145 \cs_new:Npn \__fp_round_normal:NNwNnn #1 #2
23146 {
23147     \if_meaning:w 0 #2
23148     \exp_after:wN \__fp_round_special:NwNnn
23149     \exp_after:wN #1
23150     \fi:
23151     \__fp_pack_twice_four:wNNNNNNNN
23152     \__fp_pack_twice_four:wNNNNNNNN
23153     \__fp_round_normal_end:wwNnn
23154     ; #2
23155 }
23156 \cs_new:Npn \__fp_round_normal_end:wwNnn #1;#2;#3#4#5
23157 {
23158     \exp_after:wN \__fp_exp_after_o:w \exp:w \exp_end_continue_f:w
23159     \__fp_sanitiz:Nw #3 #4 ; #1 ;
23160 }
23161 \cs_new:Npn \__fp_round_special:NwNnn #1#2;#3;#4#5#6
23162 {
23163     \if_meaning:w 0 #1
23164     \__fp_case_return:nw
23165     { \exp_after:wN \__fp_zero_fp:N \exp_after:wN #4 }
23166     \else:
23167     \exp_after:wN \__fp_round_special_aux:Nw
23168     \exp_after:wN #4
23169     \int_value:w \__fp_int_eval:w 1
23170     \if_meaning:w 1 #1 -#6 \else: +#5 \fi:
23171     \fi:
23172     ;
23173 }

```

```

23174 \cs_new:Npn \__fp_round_special_aux:Nw #1#2;
23175 {
23176   \exp_after:wN \__fp_exp_after_o:w \exp:w \exp_end_continue_f:w
23177   \__fp_sanitize:Nw #1#2; {1000}{0000}{0000}{0000};
23178 }

```

(End definition for __fp_round:Nww and others.)

```

23179 \end{package}

```

Chapter 69

l3fp-parse implementation

```
23180 <*package>
23181 <@@=fp>
```

69.1 Work plan

The task at hand is non-trivial, and some previous failed attempts show that the code leads to unreadable logs, so we had better get it (almost) right the first time. Let us first describe our goal, then discuss the design precisely before writing any code.

In this file at least, a *<floating point object>* is a floating point number or tuple. This can be extended to anything that starts with `\s__fp` or `\s__fp_<type>` and ends with `;` with some internal structure that depends on the *<type>*.

```
\__fp_parse:n      \__fp_parse:n {<fpexpr>}
```

Evaluates the *<floating point expression>* and leaves the result in the input stream as a floating point object. This function forms the basis of almost all public `l3fp` functions. During evaluation, each token is fully `f`-expanded.

`__fp_parse_o:n` does the same but expands once after its result.

T_EXhackers note: Registers (integers, toks, etc.) are automatically unpacked, without requiring a function such as `\int_use:N`. Invalid tokens remaining after `f`-expansion lead to unrecoverable low-level T_EX errors.

(End definition for `__fp_parse:n`.)

<code>\c__fp_prec_func_int</code>	Floating point expressions are composed of numbers, given in various forms, infix operators, such as <code>+</code> , <code>**</code> , or <code>,</code> (which joins two numbers into a list), and prefix operators, such as the unary <code>-</code> , functions, or opening parentheses. Here is a list of precedences which control the order of evaluation (some distinctions are irrelevant for the order of evaluation, but serve as signals), from the tightest binding to the loosest binding.
<code>\c__fp_prec_hatii_int</code>	
<code>\c__fp_prec_hat_int</code>	
<code>\c__fp_prec_not_int</code>	
<code>\c__fp_prec_juxt_int</code>	
<code>\c__fp_prec_times_int</code>	16 Function calls.
<code>\c__fp_prec_plus_int</code>	
<code>\c__fp_prec_comp_int</code>	13/14 Binary <code>**</code> and <code>^</code> (right to left).
<code>\c__fp_prec_and_int</code>	
<code>\c__fp_prec_or_int</code>	12 Unary <code>+</code> , <code>-</code> , <code>!</code> (right to left).
<code>\c__fp_prec_quest_int</code>	
<code>\c__fp_prec_colon_int</code>	11 Juxtaposition (implicit <code>*</code>) with no parenthesis.
<code>\c__fp_prec_comma_int</code>	
<code>\c__fp_prec_tuple_int</code>	
<code>\c__fp_prec_end_int</code>	

- 10 Binary `*` and `/`.
- 9 Binary `+` and `-`.
- 7 Comparisons.
- 6 Logical `and`, denoted by `&&`.
- 5 Logical `or`, denoted by `||`.
- 4 Ternary operator `?:`, piece `?`.
- 3 Ternary operator `?:`, piece `:`.
- 2 Commas.
- 1 Place where a comma is allowed and generates a tuple.
- 0 Start and end of the expression.

```

23182 \int_const:Nn \c__fp_prec_func_int { 16 }
23183 \int_const:Nn \c__fp_prec_hatii_int { 14 }
23184 \int_const:Nn \c__fp_prec_hat_int { 13 }
23185 \int_const:Nn \c__fp_prec_not_int { 12 }
23186 \int_const:Nn \c__fp_prec_juxt_int { 11 }
23187 \int_const:Nn \c__fp_prec_times_int { 10 }
23188 \int_const:Nn \c__fp_prec_plus_int { 9 }
23189 \int_const:Nn \c__fp_prec_comp_int { 7 }
23190 \int_const:Nn \c__fp_prec_and_int { 6 }
23191 \int_const:Nn \c__fp_prec_or_int { 5 }
23192 \int_const:Nn \c__fp_prec_quest_int { 4 }
23193 \int_const:Nn \c__fp_prec_colon_int { 3 }
23194 \int_const:Nn \c__fp_prec_comma_int { 2 }
23195 \int_const:Nn \c__fp_prec_tuple_int { 1 }
23196 \int_const:Nn \c__fp_prec_end_int { 0 }

```

(End definition for `\c__fp_prec_func_int` and others.)

69.1.1 Storing results

The main question in parsing expressions expandably is to decide where to put the intermediate results computed for various subexpressions.

One option is to store the values at the start of the expression, and carry them together as the first argument of each macro. However, we want to `f-expand` tokens one by one in the expression (as `\int_eval:n` does), and with this approach, expanding the next unread token forces us to jump with `\exp_after:wN` over every value computed earlier in the expression. With this approach, the run-time grows at least quadratically in the length of the expression, if not as its cube (inserting the `\exp_after:wN` is tricky and slow).

A second option is to place those values at the end of the expression. Then expanding the next unread token is straightforward, but this still hits a performance issue: for long expressions we would be reaching all the way to the end of the expression at every step of the calculation. The run-time is again quadratic.

A variation of the above attempts to place the intermediate results which appear when computing a parenthesized expression near the closing parenthesis. This still lets

us expand tokens as we go, and avoids performance problems as long as there are enough parentheses. However, it would be better to avoid requiring the closing parenthesis to be present as soon as the corresponding opening parenthesis is read: the closing parenthesis may still be hidden in a macro yet to be expanded.

Hence, we need to go for some fine expansion control: the result is stored *before* the start!

Let us illustrate this idea in a simple model: adding positive integers which may be resulting from the expansion of macros, or may be values of registers. Assume that one number, say, 12345, has already been found, and that we want to parse the next number. The current status of the code may look as follows.

```
\exp_after:wN \add:ww \int_value:w 12345 \exp_after:wN ;
\exp:w \operand:w (stuff)
```

One step of expansion expands `\exp_after:wN`, which triggers the primitive `\int_value:w`, which reads the five digits we have already found, 12345. This integer is unfinished, causing the second `\exp_after:wN` to expand, and to trigger the construction `\exp:w`, which expands `\operand:w`, defined to read what follows and make a number out of it, then leave `\exp_end:`, the number, and a semicolon in the input stream. Once `\operand:w` is done expanding, we obtain essentially

```
\exp_after:wN \add:ww \int_value:w 12345 ;
\exp:w \exp_end: 333444 ;
```

where in fact `\exp_after:wN` has already been expanded, `\int_value:w` has already seen 12345, and `\exp:w` is still looking for a number. It finds `\exp_end:`, hence expands to nothing. Now, `\int_value:w` sees the `;`, which cannot be part of a number. The expansion stops, and we are left with

```
\add:ww 12345 ; 333444 ;
```

which can safely perform the addition by grabbing two arguments delimited by `;`.

If we were to continue parsing the expression, then the following number should also be cleaned up before the next use of a binary operation such as `\add:ww`. Just like `\int_value:w 12345 \exp_after:wN ;` expanded what follows once, we need `\add:ww` to do the calculation, and in the process to expand the following once. This is also true in our real application: all the functions of the form `__fp_..._o:ww` expand what follows once. This comes at the cost of leaving tokens in the input stack, and we need to be careful not to waste this memory. All of our discussion above is nice but simplistic, as operations should not simply be performed in the order they appear.

69.1.2 Precedence and infix operators

The various operators we will encounter have different precedences, which influence the order of calculations: $1 + 2 \times 3 = 1 + (2 \times 3)$ because \times has a higher precedence than $+$. The true analog of our macro `\operand:w` must thus take care of that. When looking for an operand, it needs to perform calculations until reaching an operator which has lower precedence than the one which called `\operand:w`. This means that `\operand:w` must know what the previous binary operator is, or rather, its precedence: we thus rename it `\operand:Nw`. Let us describe as an example how we plan to do the calculation $41 - 2^3 * 4 + 5$. More precisely we describe how to perform the first operation in this expression. Here, we abuse notations: the first argument of `\operand:Nw` should be an integer

constant (`\c__fp_prec_plus_int, ...`) equal to the precedence of the given operator, not directly the operator itself.

- Clean up 41 and find `-`. We call `\operand:Nw -` to find the second operand.
- Clean up 2 and find `^`.
- Compare the precedences of `-` and `^`. Since the latter is higher, we need to compute the exponentiation. For this, find the second operand with a nested call to `\operand:Nw ^`.
- Clean up 3 and find `*`.
- Compare the precedences of `^` and `*`. Since the former is higher, `\operand:Nw ^` has found the second operand of the exponentiation, which is computed: $2^3 = 8$.
- We now have `41-8*4+5`, and `\operand:Nw -` is still looking for a second operand for the subtraction. Is it 8?
- Compare the precedences of `-` and `*`. Since the latter is higher, we are not done with 8. Call `\operand:Nw *` to find the second operand of the multiplication.
- Clean up 4, and find `+`.
- Compare the precedences of `*` and `+`. Since the former is higher, `\operand:Nw *` has found the second operand of the multiplication, which is computed: $8 * 4 = 32$.
- We now have `41-32+5`, and `\operand:Nw -` is still looking for a second operand for the subtraction. Is it 32?
- Compare the precedences of `-` and `+`. Since they are equal, `\operand:Nw -` has found the second operand for the subtraction, which is computed: $41 - 32 = 9$.
- We now have `9+5`.

The procedure above stops short of performing all computations, but adding a surrounding call to `\operand:Nw` with a very low precedence ensures that all computations are performed before `\operand:Nw` is done. Adding a trailing marker with the same very low precedence prevents the surrounding `\operand:Nw` from going beyond the marker.

The pattern above to find an operand for a given operator, is to find one number and the next operator, then compare precedences to know if the next computation should be done. If it should, then perform it after finding its second operand, and look at the next operator, then compare precedences to know if the next computation should be done. This continues until we find that the next computation should not be done. Then, we stop.

We are now ready to get a bit more technical and describe which of the `l3fp-parse` functions correspond to each step above.

First, `__fp_parse_operand:Nw` is the `\operand:Nw` function above, with small modifications due to expansion issues discussed later. We denote by $\langle precedence \rangle$ the argument of `__fp_parse_operand:Nw`, that is, the precedence of the binary operator whose operand we are trying to find. The basic action is to read numbers from the input stream. This is done by `__fp_parse_one:Nw`. A first approximation of this function is that it reads one $\langle number \rangle$, performing no computation, and finds the following binary $\langle operator \rangle$. Then it expands to

```

    <number>
    \__fp_parse_infix_<operator>:N <precedence>

```

expanding the `infix` auxiliary before leaving the above in the input stream.

We now explain the `infix` auxiliaries. We need some flexibility in how we treat the case of equal precedences: most often, the first operation encountered should be performed, such as `1-2-3` being computed as `(1-2)-3`, but `2^3^4` should be evaluated as `2^(3^4)` instead. For this reason, and to support the equivalence between `**` and `^` more easily, each binary operator is converted to a control sequence `__fp_parse_infix_<operator>:N` when it is encountered for the first time. Instead of passing both precedences to a test function to do the comparison steps above, we pass the `<precedence>` (of the earlier operator) to the `infix` auxiliary for the following `<operator>`, to know whether to perform the computation of the `<operator>`. If it should not be performed, the `infix` auxiliary expands to

```

@ \use_none:n \__fp_parse_infix_<operator>:N

```

and otherwise it calls `__fp_parse_operand:Nw` with the precedence of the `<operator>` to find its second operand `<number2>` and the next `<operator2>`, and expands to

```

@ \__fp_parse_apply_binary:NwNwN
  <operator> <number2>
@ \__fp_parse_infix_<operator2>:N

```

The `infix` function is responsible for comparing precedences, but cannot directly call the computation functions, because the first operand `<number>` is before the `infix` function in the input stream. This is why we stop the expansion here and give control to another function to close the loop.

A definition of `__fp_parse_operand:Nw <precedence>` with some of the expansion control removed is

```

\exp_after:wN \__fp_parse_continue:NwN
\exp_after:wN <precedence>
\exp:w \exp_end_continue_f:w
  \__fp_parse_one:Nw <precedence>

```

This expands `__fp_parse_one:Nw <precedence>` completely, which finds a number, wraps the next `<operator>` into an `infix` function, feeds this function the `<precedence>`, and expands it, yielding either

```

\__fp_parse_continue:NwN <precedence>
<number> @
\use_none:n \__fp_parse_infix_<operator>:N

```

or

```

\__fp_parse_continue:NwN <precedence>
<number> @
\__fp_parse_apply_binary:NwNwN
  <operator> <number2>
@ \__fp_parse_infix_<operator2>:N

```

The definition of `__fp_parse_continue:NwN` is then very simple:

```

\cs_new:Npn \__fp_parse_continue:NwN #1#2@#3 { #3 #1 #2 @ }

```


In the first case, #3 is `\use_none:n`, yielding

```
\use_none:n <precedence> <number> @
\__fp_parse_infix_<operator>:N
```

then `<number> @ __fp_parse_infix_<operator>:N`. In the second case, #3 is `__fp_parse_apply_binary:NwNwN`, whose role is to compute `<number> <operator> <number2>` and to prepare for the next comparison of precedences: first we get

```
\__fp_parse_apply_binary:NwNwN
<precedence> <number> @
<operator> <number2>
@ \__fp_parse_infix_<operator2>:N
```

then

```
\exp_after:wN \__fp_parse_continue:NwN
\exp_after:wN <precedence>
\exp:w \exp_end_continue_f:w
\__fp_<operator>_o:ww <number> <number2>
\exp:w \exp_end_continue_f:w
\__fp_parse_infix_<operator2>:N <precedence>
```

where `__fp_<operator>_o:ww` computes `<number> <operator> <number2>` and expands after the result, thus triggers the comparison of the precedence of the `<operator2>` and the `<precedence>`, continuing the loop.

We have introduced the most important functions here, and the next few paragraphs we describe various subtleties.

69.1.3 Prefix operators, parentheses, and functions

Prefix operators (unary `-`, `+`, `!`) and parentheses are taken care of by the same mechanism, and functions (`sin`, `exp`, etc.) as well. Finding the argument of the unary `-`, for instance, is very similar to grabbing the second operand of a binary infix operator, with a subtle precedence explained below. Once that operand is found, the operator can be applied to it (for the unary `-`, this simply flips the sign). A left parenthesis is just a prefix operator with a very low precedence equal to that of the closing parenthesis (which is treated as an infix operator, since it normally appears just after numbers), so that all computations are performed until the closing parenthesis. The prefix operator associated to the left parenthesis does not alter its argument, but it removes the closing parenthesis (with some checks).

Prefix operators are the reason why we only summarily described the function `__fp_parse_one:Nw` earlier. This function is responsible for reading in the input stream the first possible `<number>` and the next infix `<operator>`. If what follows `__fp_parse_one:Nw <precedence>` is a prefix operator, then we must find the operand of this prefix operator through a nested call to `__fp_parse_operand:Nw` with the appropriate precedence, then apply the operator to the operand found to yield the result of `__fp_parse_one:Nw`. So far, all is simple.

The unary operators `+`, `-`, `!` complicate things a little bit: `-3**2` should be $-(3^2) = -9$, and not $(-3)^2 = 9$. This would easily be done by giving `-` a lower precedence, equal to that of the infix `+` and `-`. Unfortunately, this fails in cases such as `3**-2*4`, yielding $3^{-2 \times 4}$ instead of the correct $3^{-2} \times 4$. A second attempt would be to call `__fp_parse_operand:Nw` with the `<precedence>` of the previous operator, but `0>-2+3` is then

parsed as $0 > -(2+3)$: the addition is performed because it binds more tightly than the comparison which precedes $-$. The correct approach is for a unary $-$ to perform operations whose precedence is greater than both that of the previous operation, and that of the unary $-$ itself. The unary $-$ is given a precedence higher than multiplication and division. This does not lead to any surprising result, since $-(x/y) = (-x)/y$ and similarly for multiplication, and it reduces the number of nested calls to `_fp_parse_operand:Nw`.

Functions are implemented as prefix operators with very high precedence, so that their argument is the first number that can possibly be built.

Note that contrarily to the `infix` functions discussed earlier, the `prefix` functions do perform tests on the previous *precedence* to decide whether to find an argument or not, since we know that we need a number, and must never stop there.

69.1.4 Numbers and reading tokens one by one

So far, we have glossed over one important point: what is a “number”? A number is typically given in the form $\langle \textit{significand} \rangle \mathbf{e} \langle \textit{exponent} \rangle$, where the $\langle \textit{significand} \rangle$ is any non-empty string composed of decimal digits and at most one decimal separator (a period), the exponent “ $\mathbf{e} \langle \textit{exponent} \rangle$ ” is optional and is composed of an exponent mark \mathbf{e} followed by a possibly empty string of signs $+$ or $-$ and a non-empty string of decimal digits. The $\langle \textit{significand} \rangle$ can also be an integer, dimension, skip, or muskip variable, in which case dimensions are converted from points (or mu units) to floating points, and the $\langle \textit{exponent} \rangle$ can also be an integer variable. Numbers can also be given as floating point variables, or as named constants such as `nan`, `inf` or `pi`. We may add more types in the future.

When `_fp_parse_one:Nw` is looking for a “number”, here is what happens.

- If the next token is a control sequence with the meaning of `\scan_stop:`, it can be: `\s_fp`, in which case our job is done, as what follows is an internal floating point number, or `\s_fp_expr_mark`, in which case the expression has come to an early end, as we are still looking for a number here, or something else, in which case we consider the control sequence to be a bad variable resulting from `c`-expansion.
- If the next token is a control sequence with a different meaning, we assume that it is a register, unpack it with `\tex_the:D`, and use its value (in `pt` for dimensions and skips, `mu` for muskips) as the $\langle \textit{significand} \rangle$ of a number: we look for an exponent.
- If the next token is a digit, we remove any leading zeros, then read a significand larger than 1 if the next character is a digit, read a significand smaller than 1 if the next character is a period, or we have found a significand equal to 0 otherwise, and look for an exponent.
- If the next token is a letter, we collect more letters until the first non-letter: the resulting word may denote a function such as `asin`, a constant such as `pi` or be unknown. In the first case, we call `_fp_parse_operand:Nw` to find the argument of the function, then apply the function, before declaring that we are done. Otherwise, we are done, either with the value of the constant, or with the value `nan` for unknown words.
- If the next token is anything else, we check whether it is a known prefix operator, in which case `_fp_parse_operand:Nw` finds its operand. If it is not known, then either a number is missing (if the token is a known infix operator) or the token is simply invalid in floating point expressions.

Once a number is found, `__fp_parse_one:Nw` also finds an infix operator. This goes as follows.

- If the next token is a control sequence, it could be the special marker `\s__fp_expr_mark`, and otherwise it is a case of juxtaposing numbers, such as `2\c_zero_int`, with an implied multiplication.
- If the next token is a letter, it is also a case of juxtaposition, as letters cannot be proper infix operators.
- Otherwise (including in the case of digits), if the token is a known infix operator, the appropriate `__fp_infix_⟨operator⟩:N` function is built, and if it does not exist, we complain. In particular, the juxtaposition `\c_zero_int 2` is disallowed.

In the above, we need to test whether a character token `#1` is a digit:

```
\if_int_compare:w 9 < 1 \token_to_str:N #1 \exp_stop_f:
  is a digit
\else:
  not a digit
\fi:
```

To exclude 0, replace 9 by 10. The use of `\token_to_str:N` ensures that a digit with any catcode is detected. To test if a character token is a letter, we need to work with its character code, testing if `'#1` lies in [65, 90] (uppercase letters) or [97, 112] (lowercase letters)

```
\if_int_compare:w \__fp_int_eval:w
  ( '#1 \if_int_compare:w '#1 > 'Z - 32 \fi: ) / 26 = 3 \exp_stop_f:
  is a letter
\else:
  not a letter
\fi:
```

At all steps, we try to accept all category codes: when `#1` is kept to be used later, it is almost always converted to category code other through `\token_to_str:N`. More precisely, catcodes {3, 6, 7, 8, 11, 12} should work without trouble, but not {1, 2, 4, 10, 13}, and of course {0, 5, 9} cannot become tokens.

Floating point expressions should behave as much as possible like ε -TeX-based integer expressions and dimension expressions. In particular, `f`-expansion should be performed as the expression is read, token by token, forcing the expansion of protected macros, and ignoring spaces. One advantage of expanding at every step is that restricted expandable functions can then be used in floating point expressions just as they can be in other kinds of expressions. Problematically, spaces stop `f`-expansion: for instance, the macro `\X` below would not be expanded if we simply performed `f`-expansion.

```
\DeclareDocumentCommand {\test} {m} { \fp_eval:n {#1} }
\ExplSyntaxOff
\test { 1 + \X }
```

Of course, spaces typically do not appear in a code setting, but may very easily come in document-level input, from which some expressions may come. To avoid this problem, at every step, we do essentially what `\use:f` would do: take an argument, put it back

in the input stream, then **f**-expand it. This is not a complete solution, since a macro's expansion could contain leading spaces which would stop the **f**-expansion before further macro calls are performed. However, in practice it should be enough: in particular, floating point numbers are correctly expanded to the underlying `\s__fp ...` structure. The **f**-expansion is performed by `__fp_parse_expand:w`.

69.2 Main auxiliary functions

`__fp_parse_operand:Nw` `\exp:w __fp_parse_operand:Nw <precedence> __fp_parse_expand:w`
 Reads the "...", performing every computation with a precedence higher than `<precedence>`, then expands to

`<result> @ __fp_parse_infix_<operation>:N ...`

where the `<operation>` is the first operation with a lower precedence, possibly `end`, and the "..." start just after the `<operation>`.

(End definition for `__fp_parse_operand:Nw`.)

`__fp_parse_infix_+:N` `__fp_parse_infix_+:N <precedence> ...`
 If `+` has a precedence higher than the `<precedence>`, cleans up a second `<operand>` and finds the `<operation2>` which follows, and expands to

`@ __fp_parse_apply_binary:NwNwN + <operand> @ __fp_parse_infix_<operation2>:N`
`...`

Otherwise expands to

`@ \use_none:n __fp_parse_infix_+:N ...`

A similar function exists for each infix operator.

(End definition for `__fp_parse_infix_+:N`.)

`__fp_parse_one:Nw` `__fp_parse_one:Nw <precedence> ...`
 Cleans up one or two operands depending on how the precedence of the next operation compares to the `<precedence>`. If the following `<operation>` has a precedence higher than `<precedence>`, expands to

`<operand1> @ __fp_parse_apply_binary:NwNwN <operation> <operand2> @`
`__fp_parse_infix_<operation2>:N ...`

and otherwise expands to

`<operand> @ \use_none:n __fp_parse_infix_<operation>:N ...`

(End definition for `__fp_parse_one:Nw`.)

69.3 Helpers

`__fp_parse_expand:w` `\exp:w __fp_parse_expand:w <tokens>`

This function must always come within a `\exp:w` expansion. The *<tokens>* should be the part of the expression that we have not yet read. This requires in particular closing all conditionals properly before expanding.

```
23197 \cs_new:Npn \__fp_parse_expand:w #1 { \exp_end_continue_f:w #1 }
```

(End definition for `__fp_parse_expand:w`.)

`__fp_parse_return_semicolon:w` This very odd function swaps its position with the following `\fi:` and removes `__fp_parse_expand:w` normally responsible for expansion. That turns out to be useful.

```
23198 \cs_new:Npn \__fp_parse_return_semicolon:w
23199     #1 \fi: \__fp_parse_expand:w { \fi: ; #1 }
```

(End definition for `__fp_parse_return_semicolon:w`.)

`__fp_parse_digits_vii:N` These functions must be called within an `\int_value:w` or `__fp_int_eval:w` construction. The first token which follows must be `f`-expanded prior to calling those functions. The functions read tokens one by one, and output digits into the input stream, until meeting a non-digit, or up to a number of digits equal to their index. The full expansion is

```
\__fp_parse_digits_vii:N    <digits> ; <filling 0> ; <length>
\__fp_parse_digits_vi:N
\__fp_parse_digits_v:N
\__fp_parse_digits_iv:N
\__fp_parse_digits_iii:N
\__fp_parse_digits_ii:N
\__fp_parse_digits_i:N
\__fp_parse_digits_:N
```

where *<filling 0>* is a string of zeros such that *<digits>* *<filling 0>* has the length given by the index of the function, and *<length>* is the number of zeros in the *<filling 0>* string. Each function puts a digit into the input stream and calls the next function, until we find a non-digit. We are careful to pass the tested tokens through `\token_to_str:N` to normalize their category code.

```
23200 \cs_set_protected:Npn \__fp_tmp:w #1 #2 #3
23201   {
23202     \cs_new:cpn { __fp_parse_digits_ #1 :N } ##1
23203     {
23204       \if_int_compare:w 9 < 1 \token_to_str:N ##1 \exp_stop_f:
23205       \token_to_str:N ##1 \exp_after:wN #2 \exp:w
23206       \else:
23207         \__fp_parse_return_semicolon:w #3 ##1
23208       \fi:
23209       \__fp_parse_expand:w
23210     }
23211   }
23212 \__fp_tmp:w {vii} \__fp_parse_digits_vi:N { 0000000 ; 7 }
23213 \__fp_tmp:w {vi} \__fp_parse_digits_v:N { 000000 ; 6 }
23214 \__fp_tmp:w {v} \__fp_parse_digits_iv:N { 00000 ; 5 }
23215 \__fp_tmp:w {iv} \__fp_parse_digits_iii:N { 0000 ; 4 }
23216 \__fp_tmp:w {iii} \__fp_parse_digits_ii:N { 000 ; 3 }
23217 \__fp_tmp:w {ii} \__fp_parse_digits_i:N { 00 ; 2 }
23218 \__fp_tmp:w {i} \__fp_parse_digits_:N { 0 ; 1 }
23219 \cs_new:Npn \__fp_parse_digits_:N { ; ; 0 }
```

(End definition for `__fp_parse_digits_vii:N` and others.)

69.4 Parsing one number

`__fp_parse_one:Nw` This function finds one number, and packs the symbol which follows in an `__fp_parse_infix_...` csname. #1 is the previous *<precedence>*, and #2 the first token of the operand. We distinguish four cases: #2 is equal to `\scan_stop:` in meaning, #2 is a different control sequence, #2 is a digit, and #2 is something else (this last case is split further later). Despite the earlier `f`-expansion, #2 may still be expandable if it was protected by `\exp_not:N`, as may happen with the \LaTeX 2_ϵ command `\protect`. Using a well placed `\reverse_if:N`, this case is sent to `__fp_parse_one_fp:NN` which deals with it robustly.

```

23220 \cs_new:Npn \__fp_parse_one:Nw #1 #2
23221 {
23222   \if_catcode:w \scan_stop: \exp_not:N #2
23223     \exp_after:wN \if_meaning:w \exp_not:N #2 #2 \else:
23224       \exp_after:wN \reverse_if:N
23225     \fi:
23226     \if_meaning:w \scan_stop: #2
23227       \exp_after:wN \exp_after:wN
23228       \exp_after:wN \__fp_parse_one_fp:NN
23229     \else:
23230       \exp_after:wN \exp_after:wN
23231       \exp_after:wN \__fp_parse_one_register:NN
23232     \fi:
23233   \else:
23234     \if_int_compare:w 9 < 1 \token_to_str:N #2 \exp_stop_f:
23235       \exp_after:wN \exp_after:wN
23236       \exp_after:wN \__fp_parse_one_digit:NN
23237     \else:
23238       \exp_after:wN \exp_after:wN
23239       \exp_after:wN \__fp_parse_one_other:NN
23240     \fi:
23241   \fi:
23242   #1 #2
23243 }
```

(End definition for `__fp_parse_one:Nw`.)

`__fp_parse_one_fp:NN` This function receives a *<precedence>* and a control sequence equal to `\scan_stop:` in meaning. There are three cases.

`_fp_exp_after_expr_mark_f:nw`
`__fp_exp_after_?_f:nw`

- `\s__fp` starts a floating point number, and we call `__fp_exp_after_f:nw`, which `f`-expands after the floating point.
- `\s__fp_expr_mark` is a premature end, we call `__fp_exp_after_expr_mark_f:nw`, which triggers an `fp-early-end` error.
- For a control sequence not containing `\s__fp`, we call `__fp_exp_after_?_f:nw`, causing a `bad-variable` error.

This scheme is extensible: additional types can be added by starting the variables with a scan mark of the form `\s__fp_<type>` and defining `__fp_exp_after_<type>_f:nw`. In all cases, we make sure that the second argument of `__fp_parse_infix:NN` is correctly expanded. A special case only enabled in \LaTeX 2_ϵ is that if `\protect` is encountered then

the error message mentions the control sequence which follows it rather than `\protect` itself. The test for $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}_{2\epsilon}$ uses `\@unexpandable@protect` rather than `\protect` because `\protect` is often `\scan_stop:` hence “does not exist”.

```

23244 \cs_new:Npn \__fp_parse_one_fp:NN #1
23245 {
23246   \__fp_exp_after_any_f:nw
23247   {
23248     \exp_after:wN \__fp_parse_infix:NN
23249     \exp_after:wN #1 \exp:w \__fp_parse_expand:w
23250   }
23251 }
23252 \cs_new:Npn \__fp_exp_after_expr_mark_f:nw #1
23253 {
23254   \int_case:nnF { \exp_after:wN \use_i:nnn \use_none:nnn #1 }
23255   {
23256     \c__fp_prec_comma_int { }
23257     \c__fp_prec_tuple_int { }
23258     \c__fp_prec_end_int
23259     {
23260       \exp_after:wN \c__fp_empty_tuple_fp
23261       \exp:w \exp_end_continue_f:w
23262     }
23263   }
23264   {
23265     \msg_expandable_error:nn { fp } { early-end }
23266     \exp_after:wN \c_nan_fp \exp:w \exp_end_continue_f:w
23267   }
23268   #1
23269 }
23270 \cs_new:cpn { __fp_exp_after_?_f:nw } #1#2
23271 {
23272   \msg_expandable_error:nnn { kernel } { bad-variable }
23273   {#2}
23274   \exp_after:wN \c_nan_fp \exp:w \exp_end_continue_f:w #1
23275 }
23276 \cs_set_protected:Npn \__fp_tmp:w #1
23277 {
23278   \cs_if_exist:NT #1
23279   {
23280     \cs_gset:cpn { __fp_exp_after_?_f:nw } ##1##2
23281     {
23282       \exp_after:wN \c_nan_fp \exp:w \exp_end_continue_f:w ##1
23283       \str_if_eq:nnTF {##2} { \protect }
23284       {
23285         \cs_if_eq:NNTF ##2 #1 { \use_i:nn } { \use:n }
23286         {
23287           \msg_expandable_error:nnn { fp }
23288           { robust-cmd }
23289         }
23290       }
23291     }
23292     \msg_expandable_error:nnn { kernel }
23293     { bad-variable } {##2}
23294   }

```

```

23295         }
23296     }
23297 }
23298 \exp_args:Nc \__fp_tmp:w { @unexpandable@protect }

```

(End definition for __fp_parse_one_fp:NN, __fp_exp_after_expr_mark_f:nw, and __fp_exp_after_?_f:nw.)

```

\__fp_parse_one_register:NN
  \__fp_parse_one_register_aux:Nw
  \__fp_parse_one_register_auxii:wwwNw
  \__fp_parse_one_register_int:www
  \__fp_parse_one_register_mu:www
  \__fp_parse_one_register_dim:ww

```

This is called whenever #2 is a control sequence other than \scan_stop: in meaning. We special-case \wd, \ht, \dp (see later) and otherwise assume that it is a register, but carefully unpack it with \tex_the:D within braces. First, we find the exponent following #2. Then we unpack #2 with \tex_the:D, and the auxii auxiliary distinguishes integer registers from dimensions/skips from muskips, according to the presence of a period and/or of pt. For integers, simply convert $\langle value \rangle e \langle exponent \rangle$ to a floating point number with __fp_parse:n (this is somewhat wasteful). For other registers, the decimal rounding provided by T_EX does not accurately represent the binary value that it manipulates, so we extract this binary value as a number of scaled points with \int_value:w \dim_to_decimal_in_sp:n { $\langle decimal value \rangle$ pt }, and use an auxiliary of \dim_to_fp:n, which performs the multiplication by 2^{-16} , correctly rounded.

```

23299 \cs_new:Npn \__fp_parse_one_register:NN #1#2
23300 {
23301   \exp_after:wN \__fp_parse_infix_after_operand:NwN
23302   \exp_after:wN #1
23303   \exp:w \exp_end_continue_f:w
23304   \__fp_parse_one_register_special:N #2
23305   \exp_after:wN \__fp_parse_one_register_aux:Nw
23306   \exp_after:wN #2
23307   \int_value:w
23308   \exp_after:wN \__fp_parse_exponent:N
23309   \exp:w \__fp_parse_expand:w
23310 }
23311 \cs_new:Npx \__fp_parse_one_register_aux:Nw #1
23312 {
23313   \exp_not:n
23314   {
23315     \exp_after:wN \use:nn
23316     \exp_after:wN \__fp_parse_one_register_auxii:wwwNw
23317   }
23318   \exp_not:N \exp_after:wN { \exp_not:N \tex_the:D #1 }
23319   ; \exp_not:N \__fp_parse_one_register_dim:ww
23320   \tl_to_str:n { pt } ; \exp_not:N \__fp_parse_one_register_mu:www
23321   . \tl_to_str:n { pt } ; \exp_not:N \__fp_parse_one_register_int:www
23322   \s__fp_stop
23323 }
23324 \exp_args:Nno \use:nn
23325 { \cs_new:Npn \__fp_parse_one_register_auxii:wwwNw #1 . #2 }
23326 { \tl_to_str:n { pt } #3 ; #4#5 \s__fp_stop }
23327 { #4 #1.#2; }
23328 \exp_args:Nno \use:nn
23329 { \cs_new:Npn \__fp_parse_one_register_mu:www #1 }
23330 { \tl_to_str:n { mu } ; #2 ; }
23331 { \__fp_parse_one_register_dim:ww #1 ; }
23332 \cs_new:Npn \__fp_parse_one_register_int:www #1; #2.; #3;
23333 { \__fp_parse:n { #1 e #3 } }

```



```

23334 \cs_new:Npn \__fp_parse_one_register_dim:ww #1; #2;
23335 {
23336   \exp_after:wN \__fp_from_dim_test:ww
23337   \int_value:w #2 \exp_after:wN ,
23338   \int_value:w \dim_to_decimal_in_sp:n { #1 pt } ;
23339 }

```

(End definition for __fp_parse_one_register:NN and others.)

__fp_parse_one_register_special:N
 __fp_parse_one_register_math:NNw
 __fp_parse_one_register_wd:w
 __fp_parse_one_register_wd:Nw

The \wd, \dp, \ht primitives expect an integer argument. We abuse the exponent parser to find the integer argument: simply include the exponent marker e. Once that “exponent” is found, use \tex_the:D to find the box dimension and then copy what we did for dimensions.

```

23340 \cs_new:Npn \__fp_parse_one_register_special:N #1
23341 {
23342   \if_meaning:w \box_wd:N #1 \__fp_parse_one_register_wd:w \fi:
23343   \if_meaning:w \box_ht:N #1 \__fp_parse_one_register_wd:w \fi:
23344   \if_meaning:w \box_dp:N #1 \__fp_parse_one_register_wd:w \fi:
23345   \if_meaning:w \infty #1
23346     \__fp_parse_one_register_math:NNw \infty #1
23347   \fi:
23348   \if_meaning:w \pi #1
23349     \__fp_parse_one_register_math:NNw \pi #1
23350   \fi:
23351 }
23352 \cs_new:Npn \__fp_parse_one_register_math:NNw
23353   #1#2#3#4 \__fp_parse_expand:w
23354 {
23355   #3
23356   \str_if_eq:nnTF {#1} {#2}
23357   {
23358     \msg_expandable_error:nnn
23359     { fp } { infinity-pi } {#1}
23360     \c_nan_fp
23361   }
23362   { #4 \__fp_parse_expand:w }
23363 }
23364 \cs_new:Npn \__fp_parse_one_register_wd:w
23365   #1#2 \exp_after:wN #3#4 \__fp_parse_expand:w
23366 {
23367   #1
23368   \exp_after:wN \__fp_parse_one_register_wd:Nw
23369   #4 \__fp_parse_expand:w e
23370 }
23371 \cs_new:Npn \__fp_parse_one_register_wd:Nw #1#2 ;
23372 {
23373   \exp_after:wN \__fp_from_dim_test:ww
23374   \exp_after:wN 0 \exp_after:wN ,
23375   \int_value:w \dim_to_decimal_in_sp:n { #1 #2 } ;
23376 }

```

(End definition for __fp_parse_one_register_special:N and others.)

__fp_parse_one_digit:NN

A digit marks the beginning of an explicit floating point number. Once the number is found, we catch the case of overflow and underflow with __fp_sanitize:wN,

then `__fp_parse_infix_after_operand:NwN` expands `__fp_parse_infix:NN` after the number we find, to wrap the following infix operator as required. Finding the number itself begins by removing leading zeros: further steps are described later.

```

23377 \cs_new:Npn \__fp_parse_one_digit:NN #1
23378 {
23379   \exp_after:wN \__fp_parse_infix_after_operand:NwN
23380   \exp_after:wN #1
23381   \exp:w \exp_end_continue_f:w
23382   \exp_after:wN \__fp_sanitize:wN
23383   \int_value:w \__fp_int_eval:w 0 \__fp_parse_trim_zeros:N
23384 }

```

(End definition for `__fp_parse_one_digit:NN`.)

`__fp_parse_one_other:NN` For this function, #2 is a character token which is not a digit. If it is an ASCII letter, `__fp_parse_letters:N` beyond this one and give the result to `__fp_parse_word:Nw`. Otherwise, the character is assumed to be a prefix operator, and we build `__fp_parse_prefix_{operator}:Nw`.

```

23385 \cs_new:Npn \__fp_parse_one_other:NN #1 #2
23386 {
23387   \if_int_compare:w
23388     \__fp_int_eval:w
23389     ( '#2 \if_int_compare:w '#2 > 'Z - 32 \fi: ) / 26
23390     = 3 \exp_stop_f:
23391     \exp_after:wN \__fp_parse_word:Nw
23392     \exp_after:wN #1
23393     \exp_after:wN #2
23394     \exp:w \exp_after:wN \__fp_parse_letters:N
23395     \exp:w
23396   \else:
23397     \exp_after:wN \__fp_parse_prefix:NNN
23398     \exp_after:wN #1
23399     \exp_after:wN #2
23400     \cs:w
23401     __fp_parse_prefix_ \token_to_str:N #2 :Nw
23402     \exp_after:wN
23403     \cs_end:
23404     \exp:w
23405   \fi:
23406   \__fp_parse_expand:w
23407 }

```

(End definition for `__fp_parse_one_other:NN`.)

`__fp_parse_word:Nw` Finding letters is a simple recursion. Once `__fp_parse_letters:N` has done its job, `__fp_parse_letters:N` we try to build a control sequence from the word #2. If it is a known word, then the corresponding action is taken, and otherwise, we complain about an unknown word, yield `\c_nan_fp`, and look for the following infix operator. Note that the unknown word could be a mistyped function as well as a mistyped constant, so there is no way to tell whether to look for arguments; we do not. The standard requires “inf” and “infinity” and “nan” to be recognized regardless of case, but we probably don’t want to allow every l3fp word to have an arbitrary mixture of lower and upper case, so we test and use a differently-named control sequence.

```

23408 \cs_new:Npn \__fp_parse_word:Nw #1#2;
23409 {
23410   \cs_if_exist_use:cF { __fp_parse_word_#2:N }
23411   {
23412     \cs_if_exist_use:cF
23413     { __fp_parse_caseless_ \str_foldcase:n {#2} :N }
23414     {
23415       \msg_expandable_error:nnn
23416       { fp } { unknown-fp-word } {#2}
23417       \exp_after:wN \c_nan_fp \exp:w \exp_end_continue_f:w
23418       \__fp_parse_infix:NN
23419     }
23420   }
23421   #1
23422 }
23423 \cs_new:Npn \__fp_parse_letters:N #1
23424 {
23425   \exp_end_continue_f:w
23426   \if_int_compare:w
23427   \if_catcode:w \scan_stop: \exp_not:N #1
23428   0
23429   \else:
23430     \__fp_int_eval:w
23431     ( '#1 \if_int_compare:w '#1 > 'Z - 32 \fi: ) / 26
23432     \fi:
23433     = 3 \exp_stop_f:
23434     \exp_after:wN #1
23435     \exp:w \exp_after:wN \__fp_parse_letters:N
23436     \exp:w
23437   \else:
23438     \__fp_parse_return_semicolon:w #1
23439     \fi:
23440     \__fp_parse_expand:w
23441 }

```

(End definition for __fp_parse_word:Nw and __fp_parse_letters:N.)

__fp_parse_prefix:NNN
 __fp_parse_prefix_unknown:NNN

For this function, #1 is the previous *precedence*, #2 is the operator just seen, and #3 is a control sequence which implements the operator if it is a known operator. If this control sequence is \scan_stop:, then the operator is in fact unknown. Either the expression is missing a number there (if the operator is valid as an infix operator), and we put `nan`, wrapping the infix operator in a csname as appropriate, or the character is simply invalid in floating point expressions, and we continue looking for a number, starting again from __fp_parse_one:Nw.

```

23442 \cs_new:Npn \__fp_parse_prefix:NNN #1#2#3
23443 {
23444   \if_meaning:w \scan_stop: #3
23445   \exp_after:wN \__fp_parse_prefix_unknown:NNN
23446   \exp_after:wN #2
23447   \fi:
23448   #3 #1
23449 }
23450 \cs_new:Npn \__fp_parse_prefix_unknown:NNN #1#2#3
23451 {

```

```

23452 \cs_if_exist:cTF { __fp_parse_infix_ \token_to_str:N #1 :N }
23453 {
23454   \msg_expandable_error:nnn
23455   { fp } { missing-number } {#1}
23456   \exp_after:wN \c_nan_fp \exp:w \exp_end_continue_f:w
23457   \__fp_parse_infix:NN #3 #1
23458 }
23459 {
23460   \msg_expandable_error:nnn
23461   { fp } { unknown-symbol } {#1}
23462   \__fp_parse_one:Nw #3
23463 }
23464 }

```

(End definition for `__fp_parse_prefix:NNN` and `__fp_parse_prefix_unknown:NNN`.)

69.4.1 Numbers: trimming leading zeros

Numbers are parsed as follows: first we trim leading zeros, then if the next character is a digit, start reading a significand ≥ 1 with the set of functions `__fp_parse_large...`; if it is a period, the significand is < 1 ; and otherwise it is zero. In the second case, trim additional zeros after the period, counting them for an exponent shift $\langle exp_1 \rangle < 0$, then read the significand with the set of functions `__fp_parse_small...`. Once the significand is read, read the exponent if `e` is present.

`__fp_parse_trim_zeros:N`
`__fp_parse_trim_end:w`

This function expects an already expanded token. It removes any leading zero, then distinguishes three cases: if the first non-zero token is a digit, then call `__fp_parse_large:N` (the significand is ≥ 1); if it is `.`, then continue trimming zeros with `__fp_parse_strim_zeros:N`; otherwise, our number is exactly zero, and we call `__fp_parse_zero:` to take care of that case.

```

23465 \cs_new:Npn \__fp_parse_trim_zeros:N #1
23466 {
23467   \if:w 0 \exp_not:N #1
23468     \exp_after:wN \__fp_parse_trim_zeros:N
23469     \exp:w
23470   \else:
23471     \if:w . \exp_not:N #1
23472       \exp_after:wN \__fp_parse_strim_zeros:N
23473       \exp:w
23474     \else:
23475       \__fp_parse_trim_end:w #1
23476     \fi:
23477   \fi:
23478   \__fp_parse_expand:w
23479 }
23480 \cs_new:Npn \__fp_parse_trim_end:w #1 \fi: \fi: \__fp_parse_expand:w
23481 {
23482   \fi:
23483   \fi:
23484   \if_int_compare:w 9 < 1 \token_to_str:N #1 \exp_stop_f:
23485     \exp_after:wN \__fp_parse_large:N
23486   \else:
23487     \exp_after:wN \__fp_parse_zero:

```

```

23488     \fi:
23489     #1
23490 }

```

(End definition for `_fp_parse_trim_zeros:N` and `_fp_parse_trim_end:w`.)

```

\_fp_parse_strim_zeros:N
\_fp_parse_strim_end:w

```

If we have removed all digits until a period (or if the body started with a period), then enter the “`small_trim`” loop which outputs `−1` for each removed 0. Those `−1` are added to an integer expression waiting for the exponent. If the first non-zero token is a digit, call `_fp_parse_small:N` (our significand is smaller than 1), and otherwise, the number is an exact zero. The name `strim` stands for “small trim”.

```

23491 \cs_new:Npn \_fp_parse_strim_zeros:N #1
23492 {
23493     \if:w 0 \exp_not:N #1
23494     - 1
23495     \exp_after:wN \_fp_parse_strim_zeros:N \exp:w
23496     \else:
23497         \_fp_parse_strim_end:w #1
23498     \fi:
23499     \_fp_parse_expand:w
23500 }
23501 \cs_new:Npn \_fp_parse_strim_end:w #1 \fi: \_fp_parse_expand:w
23502 {
23503     \fi:
23504     \if_int_compare:w 9 < 1 \token_to_str:N #1 \exp_stop_f:
23505     \exp_after:wN \_fp_parse_small:N
23506     \else:
23507     \exp_after:wN \_fp_parse_zero:
23508     \fi:
23509     #1
23510 }

```

(End definition for `_fp_parse_strim_zeros:N` and `_fp_parse_strim_end:w`.)

`_fp_parse_zero:` After reading a significand of 0, find any exponent, then put a sign of 1 for `_fp-sanitize:wN`, which removes everything and leaves an exact zero.

```

23511 \cs_new:Npn \_fp_parse_zero:
23512 {
23513     \exp_after:wN ; \exp_after:wN 1
23514     \int_value:w \_fp_parse_exponent:N
23515 }

```

(End definition for `_fp_parse_zero:.`)

69.4.2 Number: small significand

```

\_fp_parse_small:N

```

This function is called after we have passed the decimal separator and removed all leading zeros from the significand. It is followed by a non-zero digit (with any catcode). The goal is to read up to 16 digits. But we can’t do that all at once, because `\int_value:w` (which allows us to collect digits and continue expanding) can only go up to 9 digits. Hence we grab digits in two steps of 8 digits. Since `#1` is a digit, read seven more digits using `_fp_parse_digits_vii:N`. The `small_leading` auxiliary leaves those digits in the `\int_value:w`, and grabs some more, or stops if there are no more digits. Then the

`pack_leading` auxiliary puts the various parts in the appropriate order for the processing further up.

```

23516 \cs_new:Npn \__fp_parse_small:N #1
23517 {
23518   \exp_after:wN \__fp_parse_pack_leading:NNNNNww
23519   \int_value:w \__fp_int_eval:w 1 \token_to_str:N #1
23520   \exp_after:wN \__fp_parse_small_leading:wwNN
23521   \int_value:w 1
23522   \exp_after:wN \__fp_parse_digits_vii:N
23523   \exp:w \__fp_parse_expand:w
23524 }

```

(End definition for `__fp_parse_small:N`.)

```

\__fp_parse_small_leading:wwNN      \__fp_parse_small_leading:wwNN 1 <digits> ; <zeros> ; <number of zeros>

```

We leave *<digits>* *<zeros>* in the input stream: the functions used to grab digits are such that this constitutes digits 1 through 8 of the significand. Then prepare to pack 8 more digits, with an exponent shift of zero (this shift is used in the case of a large significand). If #4 is a digit, leave it behind for the packing function, and read 6 more digits to reach a total of 15 digits: further digits are involved in the rounding. Otherwise put 8 zeros in to complete the significand, then look for an exponent.

```

23525 \cs_new:Npn \__fp_parse_small_leading:wwNN 1 #1 ; #2; #3 #4
23526 {
23527   #1 #2
23528   \exp_after:wN \__fp_parse_pack_trailing:NNNNNNww
23529   \exp_after:wN 0
23530   \int_value:w \__fp_int_eval:w 1
23531   \if_int_compare:w 9 < 1 \token_to_str:N #4 \exp_stop_f:
23532     \token_to_str:N #4
23533     \exp_after:wN \__fp_parse_small_trailing:wwNN
23534     \int_value:w 1
23535     \exp_after:wN \__fp_parse_digits_vi:N
23536     \exp:w
23537   \else:
23538     0000 0000 \__fp_parse_exponent:Nw #4
23539   \fi:
23540   \__fp_parse_expand:w
23541 }

```

(End definition for `__fp_parse_small_leading:wwNN`.)

```

\__fp_parse_small_trailing:wwNN      \__fp_parse_small_trailing:wwNN 1 <digits> ; <zeros> ; <number of zeros>
                                     <next token>

```

Leave digits 10 to 15 (arguments #1 and #2) in the input stream. If the *<next token>* is a digit, it is the 16th digit, we keep it, then the `small_round` auxiliary considers this digit and all further digits to perform the rounding: the function expands to nothing, to +0 or to +1. Otherwise, there is no 16-th digit, so we put a 0, and look for an exponent.

```

23542 \cs_new:Npn \__fp_parse_small_trailing:wwNN 1 #1 ; #2; #3 #4
23543 {
23544   #1 #2
23545   \if_int_compare:w 9 < 1 \token_to_str:N #4 \exp_stop_f:
23546     \token_to_str:N #4
23547     \exp_after:wN \__fp_parse_small_round:NN

```

```

23548         \exp_after:wN #4
23549         \exp:w
23550     \else:
23551         0 \__fp_parse_exponent:Nw #4
23552     \fi:
23553     \__fp_parse_expand:w
23554 }

```

(End definition for `__fp_parse_small_trailing:wwNN`.)

```

\__fp_parse_pack_trailing:NNNNNww
\__fp_parse_pack_leading:NNNNNww
\__fp_parse_pack_carry:w

```

Those functions are expanded after all the digits are found, we took care of the rounding, as well as the exponent. The last argument is the exponent. The previous five arguments are 8 digits which we pack in groups of 4, and the argument before that is 1, except in the rare case where rounding lead to a carry, in which case the argument is 2. The `trailing` function has an exponent shift as its first argument, which we add to the exponent found in the `e...` syntax. If the trailing digits cause a carry, the integer expression for the leading digits is incremented (+1 in the code below). If the leading digits propagate this carry all the way up, the function `__fp_parse_pack_carry:w` increments the exponent, and changes the significand from 0000... to 1000...: this is simple because such a carry can only occur to give rise to a power of 10.

```

23555 \cs_new:Npn \__fp_parse_pack_trailing:NNNNNww #1 #2 #3#4#5#6 #7; #8 ;
23556 {
23557     \if_meaning:w 2 #2 + 1 \fi:
23558     ; #8 + #1 ; {#3#4#5#6} {#7};
23559 }
23560 \cs_new:Npn \__fp_parse_pack_leading:NNNNNww #1 #2#3#4#5 #6; #7;
23561 {
23562     + #7
23563     \if_meaning:w 2 #1 \__fp_parse_pack_carry:w \fi:
23564     ; 0 {#2#3#4#5} {#6}
23565 }
23566 \cs_new:Npn \__fp_parse_pack_carry:w \fi: ; 0 #1
23567 { \fi: + 1 ; 0 {1000} }

```

(End definition for `__fp_parse_pack_trailing:NNNNNww`, `__fp_parse_pack_leading:NNNNNww`, and `__fp_parse_pack_carry:w`.)

69.4.3 Number: large significand

Parsing a significand larger than 1 is a little bit more difficult than parsing small significands. We need to count the number of digits before the decimal separator, and add that to the final exponent. We also need to test for the presence of a dot each time we run out of digits, and branch to the appropriate `parse_small` function in those cases.

```

\__fp_parse_large:N

```

This function is followed by the first non-zero digit of a “large” significand (≥ 1). It is called within an integer expression for the exponent. Grab up to 7 more digits, for a total of 8 digits.

```

23568 \cs_new:Npn \__fp_parse_large:N #1
23569 {
23570     \exp_after:wN \__fp_parse_large_leading:wwNN
23571     \int_value:w 1 \token_to_str:N #1
23572     \exp_after:wN \__fp_parse_digits_vii:N
23573     \exp:w \__fp_parse_expand:w
23574 }

```

(End definition for `_fp_parse_large:N`.)

```
\_fp_parse_large_leading:wwNN
    \_fp_parse_large_leading:wwNN 1 <digits> ; <zeros> ; <number of zeros>
    <next token>
```

We shift the exponent by the number of digits in #1, namely the target number, 8, minus the *<number of zeros>* (number of digits missing). Then prepare to pack the 8 first digits. If the *<next token>* is a digit, read up to 6 more digits (digits 10 to 15). If it is a period, try to grab the end of our 8 first digits, branching to the `small` functions since the number of digit does not affect the exponent anymore. Finally, if this is the end of the significand, insert the *<zeros>* to complete the 8 first digits, insert 8 more, and look for an exponent.

```
23575 \cs_new:Npn \_fp_parse_large_leading:wwNN 1 #1 ; #2; #3 #4
23576 {
23577   + \c_fp_half_prec_int - #3
23578   \exp_after:wN \_fp_parse_pack_leading:NNNNNww
23579   \int_value:w \_fp_int_eval:w 1 #1
23580   \if_int_compare:w 9 < 1 \token_to_str:N #4 \exp_stop_f:
23581     \exp_after:wN \_fp_parse_large_trailing:wwNN
23582     \int_value:w 1 \token_to_str:N #4
23583     \exp_after:wN \_fp_parse_digits_vi:N
23584     \exp:w
23585   \else:
23586     \if:w . \exp_not:N #4
23587       \exp_after:wN \_fp_parse_small_leading:wwNN
23588       \int_value:w 1
23589       \cs:w
23590         \_fp_parse_digits_
23591         \_fp_int_to_roman:w #3
23592         :N \exp_after:wN
23593       \cs_end:
23594       \exp:w
23595     \else:
23596       #2
23597       \exp_after:wN \_fp_parse_pack_trailing:NNNNNNww
23598       \exp_after:wN 0
23599       \int_value:w 1 0000 0000
23600       \_fp_parse_exponent:Nw #4
23601     \fi:
23602   \fi:
23603   \_fp_parse_expand:w
23604 }
```

(End definition for `_fp_parse_large_leading:wwNN`.)

```
\_fp_parse_large_trailing:wwNN
    \_fp_parse_large_trailing:wwNN 1 <digits> ; <zeros> ; <number of zeros>
    <next token>
```

We have just read 15 digits. If the *<next token>* is a digit, then the exponent shift caused by this block of 8 digits is 8, first argument to the `pack_trailing` function. We keep the *<digits>* and this 16-th digit, and find how this should be rounded using `_fp_parse_large_round:NN`. Otherwise, the exponent shift is the number of *<digits>*, 7 minus the *<number of zeros>*, and we test for a decimal point. This case happens in 123451234512345.67 with exactly 15 digits before the decimal separator. Then branch to the appropriate `small` auxiliary, grabbing a few more digits to complement the digits

we already grabbed. Finally, if this is truly the end of the significand, look for an exponent after using the *<zeros>* and providing a 16-th digit of 0.

```

23605 \cs_new:Npn \__fp_parse_large_trailing:wwNN 1 #1 ; #2; #3 #4
23606 {
23607   \if_int_compare:w 9 < 1 \token_to_str:N #4 \exp_stop_f:
23608     \exp_after:wN \__fp_parse_pack_trailing:NNNNNNww
23609     \exp_after:wN \c__fp_half_prec_int
23610     \int_value:w \__fp_int_eval:w 1 #1 \token_to_str:N #4
23611     \exp_after:wN \__fp_parse_large_round:NN
23612     \exp_after:wN #4
23613     \exp:w
23614   \else:
23615     \exp_after:wN \__fp_parse_pack_trailing:NNNNNNww
23616     \int_value:w \__fp_int_eval:w 7 - #3 \exp_stop_f:
23617     \int_value:w \__fp_int_eval:w 1 #1
23618     \if:w . \exp_not:N #4
23619       \exp_after:wN \__fp_parse_small_trailing:wwNN
23620       \int_value:w 1
23621       \cs:w
23622         __fp_parse_digits_
23623         \__fp_int_to_roman:w #3
23624         :N \exp_after:wN
23625       \cs_end:
23626       \exp:w
23627     \else:
23628       #2 0 \__fp_parse_exponent:Nw #4
23629     \fi:
23630   \fi:
23631   \__fp_parse_expand:w
23632 }

```

(End definition for *__fp_parse_large_trailing:wwNN*.)

69.4.4 Number: beyond 16 digits, rounding

__fp_parse_round_loop:N This loop is called when rounding a number (whether the mantissa is small or large).
__fp_parse_round_up:N It should appear in an integer expression. This function reads digits one by one, until reaching a non-digit, and adds 1 to the integer expression for each digit. If all digits found are 0, the function ends the expression by ;0, otherwise by ;1. This is done by switching the loop to *round_up* at the first non-zero digit, thus we avoid to test whether digits are 0 or not once we see a first non-zero digit.

```

23633 \cs_new:Npn \__fp_parse_round_loop:N #1
23634 {
23635   \if_int_compare:w 9 < 1 \token_to_str:N #1 \exp_stop_f:
23636     + 1
23637     \if:w 0 \token_to_str:N #1
23638       \exp_after:wN \__fp_parse_round_loop:N
23639       \exp:w
23640     \else:
23641       \exp_after:wN \__fp_parse_round_up:N
23642       \exp:w
23643     \fi:
23644   \else:

```

```

23645     \__fp_parse_return_semicolon:w 0 #1
23646     \fi:
23647     \__fp_parse_expand:w
23648 }
23649 \cs_new:Npn \__fp_parse_round_up:N #1
23650 {
23651     \if_int_compare:w 9 < 1 \token_to_str:N #1 \exp_stop_f:
23652     + 1
23653     \exp_after:wN \__fp_parse_round_up:N
23654     \exp:w
23655 \else:
23656     \__fp_parse_return_semicolon:w 1 #1
23657     \fi:
23658     \__fp_parse_expand:w
23659 }

```

(End definition for __fp_parse_round_loop:N and __fp_parse_round_up:N.)

__fp_parse_round_after:wN After the loop __fp_parse_round_loop:N, this function fetches an exponent with __fp_parse_exponent:N, and combines it with the number of digits counted by __fp_parse_round_loop:N. At the same time, the result 0 or 1 is added to the surrounding integer expression.

```

23660 \cs_new:Npn \__fp_parse_round_after:wN #1; #2
23661 {
23662     + #2 \exp_after:wN ;
23663     \int_value:w \__fp_int_eval:w #1 + \__fp_parse_exponent:N
23664 }

```

(End definition for __fp_parse_round_after:wN.)

__fp_parse_small_round:NN Here, #1 is the digit that we are currently rounding (we only care whether it is even or odd). If #2 is not a digit, then fetch an exponent and expand to ;*<exponent>* only. Otherwise, we expand to +0 or +1, then ;*<exponent>*. To decide which, call __fp_round_s:NNNw to know whether to round up, giving it as arguments a sign 0 (all explicit numbers are positive), the digit #1 to round, the first following digit #2, and either +0 or +1 depending on whether the following digits are all zero or not. This last argument is obtained by __fp_parse_round_loop:N, whose number of digits we discard by multiplying it by 0. The exponent which follows the number is also fetched by __fp_parse_round_after:wN.

```

23665 \cs_new:Npn \__fp_parse_small_round:NN #1#2
23666 {
23667     \if_int_compare:w 9 < 1 \token_to_str:N #2 \exp_stop_f:
23668     +
23669     \exp_after:wN \__fp_round_s:NNNw
23670     \exp_after:wN 0
23671     \exp_after:wN #1
23672     \exp_after:wN #2
23673     \int_value:w \__fp_int_eval:w
23674     \exp_after:wN \__fp_parse_round_after:wN
23675     \int_value:w \__fp_int_eval:w 0 * \__fp_int_eval:w 0
23676     \exp_after:wN \__fp_parse_round_loop:N
23677     \exp:w
23678 \else:
23679     \__fp_parse_exponent:Nw #2

```

```

23680     \fi:
23681     \__fp_parse_expand:w
23682 }

```

(End definition for __fp_parse_small_round:NN and __fp_parse_round_after:wN.)

```

\__fp_parse_large_round:NN
  \__fp_parse_large_round_test:NN
  \__fp_parse_large_round_aux:wNN

```

Large numbers are harder to round, as there may be a period in the way. Again, #1 is the digit that we are currently rounding (we only care whether it is even or odd). If there are no more digits (#2 is not a digit), then we must test for a period: if there is one, then switch to the rounding function for small significands, otherwise fetch an exponent. If there are more digits (#2 is a digit), then round, checking with __fp_parse_round_loop:N if all further digits vanish, or some are non-zero. This loop is not enough, as it is stopped by a period. After the loop, the aux function tests for a period: if it is present, then we must continue looking for digits, this time discarding the number of digits we find.

```

23683 \cs_new:Npn \__fp_parse_large_round:NN #1#2
23684 {
23685   \if_int_compare:w 9 < 1 \token_to_str:N #2 \exp_stop_f:
23686   +
23687   \exp_after:wN \__fp_round_s:NNNw
23688   \exp_after:wN 0
23689   \exp_after:wN #1
23690   \exp_after:wN #2
23691   \int_value:w \__fp_int_eval:w
23692   \exp_after:wN \__fp_parse_large_round_aux:wNN
23693   \int_value:w \__fp_int_eval:w 1
23694   \exp_after:wN \__fp_parse_round_loop:N
23695   \else: %^^A could be dot, or e, or other
23696   \exp_after:wN \__fp_parse_large_round_test:NN
23697   \exp_after:wN #1
23698   \exp_after:wN #2
23699   \fi:
23700 }
23701 \cs_new:Npn \__fp_parse_large_round_test:NN #1#2
23702 {
23703   \if:w . \exp_not:N #2
23704   \exp_after:wN \__fp_parse_small_round:NN
23705   \exp_after:wN #1
23706   \exp:w
23707   \else:
23708   \__fp_parse_exponent:Nw #2
23709   \fi:
23710   \__fp_parse_expand:w
23711 }
23712 \cs_new:Npn \__fp_parse_large_round_aux:wNN #1 ; #2 #3
23713 {
23714   + #2
23715   \exp_after:wN \__fp_parse_round_after:wN
23716   \int_value:w \__fp_int_eval:w #1
23717   \if:w . \exp_not:N #3
23718   + 0 * \__fp_int_eval:w 0
23719   \exp_after:wN \__fp_parse_round_loop:N
23720   \exp:w \exp_after:wN \__fp_parse_expand:w
23721   \else:

```

```

23722         \exp_after:wN ;
23723         \exp_after:wN 0
23724         \exp_after:wN #3
23725     \fi:
23726 }

```

(End definition for `_fp_parse_large_round:NN`, `_fp_parse_large_round_test:NN`, and `_fp_parse_large_round_aux:wNN`.)

69.4.5 Number: finding the exponent

Expansion is a little bit tricky here, in part because we accept input where multiplication is implicit.

```

\_fp_parse:n { 3.2 erf(0.1) }
\_fp_parse:n { 3.2 e\l_my_int }
\_fp_parse:n { 3.2 \c_pi_fp }

```

The first case indicates that just looking one character ahead for an “e” is not enough, since we would mistake the function `erf` for an exponent of “rf”. An alternative would be to look two tokens ahead and check if what follows is a sign or a digit, considering in that case that we must be finding an exponent. But taking care of the second case requires that we unpack registers after `e`. However, blindly expanding the two tokens ahead completely would break the third example (unpacking is even worse). Indeed, in the course of reading `3.2`, `\c_pi_fp` is expanded to `\s__fp _fp_chk:w 1 0 {-1} {3141} ...`; and `\s__fp` stops the expansion. Expanding two tokens ahead would then force the expansion of `_fp_chk:w` (despite it being protected), and that function tries to produce an error.

What can we do? Really, the reason why this last case breaks is that just as \TeX does, we should read ahead as little as possible. Here, the only case where there may be an exponent is if the first token ahead is `e`. Then we expand (and possibly unpack) the second token.

`_fp_parse_exponent:Nw` This auxiliary is convenient to smuggle some material through `\fi:` ending conditional processing. We place those `\fi:` (argument #2) at a very odd place because this allows us to insert `_fp_int_eval:w ...` there if needed.

```

23727 \cs_new:Npn \_fp_parse_exponent:Nw #1 #2 \_fp_parse_expand:w
23728 {
23729     \exp_after:wN ;
23730     \int_value:w #2 \_fp_parse_exponent:N #1
23731 }

```

(End definition for `_fp_parse_exponent:Nw`.)

`_fp_parse_exponent:N`
`_fp_parse_exponent_aux:NN` This function should be called within an `\int_value:w` expansion (or within an integer expression). It leaves digits of the exponent behind it in the input stream, and terminates the expansion with a semicolon. If there is no `e` (or `E`), leave an exponent of 0. If there is an `e` or `E`, expand the next token to run some tests on it. The first rough test is that if the character code of #1 is greater than that of 9 (largest code valid for an exponent, less than any code valid for an identifier), there was in fact no exponent; otherwise, we search for the sign of the exponent.

```

23732 \cs_new:Npn \_fp_parse_exponent:N #1
23733 {

```

```

23734 \if:w e \if:w E \exp_not:N #1 e \else: \exp_not:N #1 \fi:
23735 \exp_after:wN \__fp_parse_exponent_aux:NN
23736 \exp_after:wN #1
23737 \exp:w
23738 \else:
23739 0 \__fp_parse_return_semicolon:w #1
23740 \fi:
23741 \__fp_parse_expand:w
23742 }
23743 \cs_new:Npn \__fp_parse_exponent_aux:NN #1#2
23744 {
23745 \if_int_compare:w \if_catcode:w \scan_stop: \exp_not:N #2
23746 0 \else: '#2 \fi: > '9 \exp_stop_f:
23747 0 \exp_after:wN ; \exp_after:wN #1
23748 \else:
23749 \exp_after:wN \__fp_parse_exponent_sign:N
23750 \fi:
23751 #2
23752 }

```

(End definition for __fp_parse_exponent:N and __fp_parse_exponent_aux:NN.)

__fp_parse_exponent_sign:N Read signs one by one (if there is any).

```

23753 \cs_new:Npn \__fp_parse_exponent_sign:N #1
23754 {
23755 \if:w + \if:w - \exp_not:N #1 + \fi: \token_to_str:N #1
23756 \exp_after:wN \__fp_parse_exponent_sign:N
23757 \exp:w \exp_after:wN \__fp_parse_expand:w
23758 \else:
23759 \exp_after:wN \__fp_parse_exponent_body:N
23760 \exp_after:wN #1
23761 \fi:
23762 }

```

(End definition for __fp_parse_exponent_sign:N.)

__fp_parse_exponent_body:N An exponent can be an explicit integer (most common case), or various other things (most of which are invalid).

```

23763 \cs_new:Npn \__fp_parse_exponent_body:N #1
23764 {
23765 \if_int_compare:w 9 < 1 \token_to_str:N #1 \exp_stop_f:
23766 \token_to_str:N #1
23767 \exp_after:wN \__fp_parse_exponent_digits:N
23768 \exp:w
23769 \else:
23770 \__fp_parse_exponent_keep:NTF #1
23771 { \__fp_parse_return_semicolon:w #1 }
23772 {
23773 \exp_after:wN ;
23774 \exp:w
23775 }
23776 \fi:
23777 \__fp_parse_expand:w
23778 }

```

(End definition for `_fp_parse_exponent_body:N`.)

`_fp_parse_exponent_digits:N` Read digits one by one, and leave them behind in the input stream. When finding a non-digit, stop, and insert a semicolon. Note that we do not check for overflow of the exponent, hence there can be a `TEX` error. It is mostly harmless, except when parsing `0e9876543210`, which should be a valid representation of 0, but is not.

```

23779 \cs_new:Npn \_fp_parse_exponent_digits:N #1
23780 {
23781   \if_int_compare:w 9 < 1 \token_to_str:N #1 \exp_stop_f:
23782   \token_to_str:N #1
23783   \exp_after:wN \_fp_parse_exponent_digits:N
23784   \exp:w
23785   \else:
23786     \_fp_parse_return_semicolon:w #1
23787   \fi:
23788   \_fp_parse_expand:w
23789 }

```

(End definition for `_fp_parse_exponent_digits:N`.)

`_fp_parse_exponent_keep:NTF` This is the last building block for parsing exponents. The argument `#1` is already fully expanded, and neither `+` nor `-` nor a digit. It can be:

- `\s_fp`, marking the start of an internal floating point, invalid here;
- another control sequence equal to `\relax`, probably a bad variable;
- a register: in this case we make sure that it is an integer register, not a dimension;
- a character other than `+`, `-` or digits, again, an error.

```

23790 \prg_new_conditional:Npnn \_fp_parse_exponent_keep:N #1 { TF }
23791 {
23792   \if_catcode:w \scan_stop: \exp_not:N #1
23793   \if_meaning:w \scan_stop: #1
23794     \if:w 0 \_fp_str_if_eq:nn { \s_fp } { \exp_not:N #1 }
23795     0
23796     \msg_expandable_error:nnn
23797       { fp } { after-e } { floating-point~ }
23798     \prg_return_true:
23799   \else:
23800     0
23801     \msg_expandable_error:nnn
23802       { kernel } { bad-variable } { #1 }
23803     \prg_return_false:
23804   \fi:
23805   \else:
23806     \if:w 0 \_fp_str_if_eq:nn { \int_value:w #1 } { \tex_the:D #1 }
23807     \int_value:w #1
23808   \else:
23809     0
23810     \msg_expandable_error:nnn
23811       { fp } { after-e } { dimension~#1 }
23812   \fi:
23813   \prg_return_false:

```

```

23814      \fi:
23815    \else:
23816      0
23817      \msg_expandable_error:nnn
23818        { fp } { missing } { exponent }
23819      \prg_return_true:
23820    \fi:
23821  }

```

(End definition for _fp_parse_exponent_keep:NTF.)

69.5 Constants, functions and prefix operators

69.5.1 Prefix operators

_fp_parse_prefix+:Nw A unary + does nothing: we should continue looking for a number.

```

23822 \cs_new_eq:cN { \_fp_parse_prefix+:Nw } \_fp_parse_one:Nw

```

(End definition for _fp_parse_prefix+:Nw.)

_fp_parse_apply_function:NNNwN Here, #1 is a precedence, #2 is some extra data used by some functions, #3 is *e.g.*, _fp_sin_o:w, and expands once after the calculation, #4 is the operand, and #5 is a _fp_parse_infix_...:N function. We feed the data #2, and the argument #4, to the function #3, which expands \exp:w thus the infix function #5.

```

23823 \cs_new:Npn \_fp_parse_apply_function:NNNwN #1#2#3#4#5
23824 {
23825   #3 #2 #4 @
23826   \exp:w \exp_end_continue_f:w #5 #1
23827 }

```

(End definition for _fp_parse_apply_function:NNNwN.)

_fp_parse_apply_unary:NNNwN In contrast to _fp_parse_apply_function:NNNwN, this checks that the operand #4 is a single argument (namely there is a single ;). We use the fact that any floating point starts with a “safe” token like \s__fp. If there is no argument produce the `fp-no-arg` error; if there are at least two produce `fp-multi-arg`. For the error message extract the mathematical function name (such as `sin`) from the `expl3` function that computes it, such as _fp_sin_o:w.

In addition, since there is a single argument we can dispatch on type and check that the resulting function exists. This catches things like `sin((1,2))` where it does not make sense to take the sine of a tuple.

```

23828 \cs_new:Npn \_fp_parse_apply_unary:NNNwN #1#2#3#4#5
23829 {
23830   \_fp_parse_apply_unary_chk:NwNw #4 @ ; . \s__fp_stop
23831   \_fp_parse_apply_unary_type:NNN
23832   #3 #2 #4 @
23833   \exp:w \exp_end_continue_f:w #5 #1
23834 }
23835 \cs_new:Npn \_fp_parse_apply_unary_chk:NwNw #1#2 ; #3#4 \s__fp_stop
23836 {
23837   \if_meaning:w @ #3 \else:
23838     \token_if_eq_meaning:NNTF . #3
23839     { \_fp_parse_apply_unary_chk:nNNNNw { no } }

```

```

23840         { \__fp_parse_apply_unary_chk:nNNNNw { multi } }
23841     \fi:
23842 }
23843 \cs_new:Npn \__fp_parse_apply_unary_chk:nNNNNw #1#2#3#4#5#6 @
23844 {
23845     #2
23846     \__fp_error:nffn { #1-arg } { \__fp_func_to_name:N #4 } { } { }
23847     \exp_after:wN #4 \exp_after:wN #5 \c_nan_fp @
23848 }
23849 \cs_new:Npn \__fp_parse_apply_unary_type:NNN #1#2#3
23850 {
23851     \__fp_change_func_type:NNN #3 #1 \__fp_parse_apply_unary_error:NNw
23852     #2 #3
23853 }
23854 \cs_new:Npn \__fp_parse_apply_unary_error:NNw #1#2#3 @
23855 { \__fp_invalid_operation_o:fw { \__fp_func_to_name:N #1 } #3 }

```

(End definition for __fp_parse_apply_unary:NNNwN and others.)

__fp_parse_prefix -:Nw
__fp_parse_prefix !:Nw

The unary - and boolean not are harder: we parse the operand using a precedence equal to the maximum of the previous precedence ##1 and the precedence \c__fp_prec_not_int of the unary operator, then call the appropriate __fp_⟨operation⟩_o:w function, where the ⟨operation⟩ is set_sign or not.

```

23856 \cs_set_protected:Npn \__fp_tmp:w #1#2#3#4
23857 {
23858     \cs_new:cpn { __fp_parse_prefix_ #1 :Nw } ##1
23859     {
23860         \exp_after:wN \__fp_parse_apply_unary:NNNwN
23861         \exp_after:wN ##1
23862         \exp_after:wN #4
23863         \exp_after:wN #3
23864         \exp:w
23865         \if_int_compare:w #2 < ##1
23866             \__fp_parse_operand:Nw ##1
23867         \else:
23868             \__fp_parse_operand:Nw #2
23869         \fi:
23870         \__fp_parse_expand:w
23871     }
23872 }
23873 \__fp_tmp:w - \c__fp_prec_not_int \__fp_set_sign_o:w 2
23874 \__fp_tmp:w ! \c__fp_prec_not_int \__fp_not_o:w ?

```

(End definition for __fp_parse_prefix -:Nw and __fp_parse_prefix !:Nw.)

__fp_parse_prefix .:Nw

Numbers which start with a decimal separator (a period) end up here. Of course, we do not look for an operand, but for the rest of the number. This function is very similar to __fp_parse_one_digit:NN but calls __fp_parse_trim_zeros:N to trim zeros after the decimal point, rather than the trim_zeros function for zeros before the decimal point.

```

23875 \cs_new:cpn { __fp_parse_prefix .:Nw } #1
23876 {
23877     \exp_after:wN \__fp_parse_infix_after_operand:NwN
23878     \exp_after:wN #1

```



```

23879 \exp:w \exp_end_continue_f:w
23880 \exp_after:wN \__fp_sanitize:wN
23881 \int_value:w \__fp_int_eval:w 0 \__fp_parse_strim_zeros:N
23882 }

```

(End definition for __fp_parse_prefix_.:Nw.)

```

\__fp_parse_prefix_(:Nw
\__fp_parse_lparen_after:NwN

```

The left parenthesis is treated as a unary prefix operator because it appears in exactly the same settings. If the previous precedence is \c__fp_prec_func_int we are parsing arguments of a function and commas should not build tuples; otherwise commas should build tuples. We distinguish these cases by precedence: \c__fp_prec_comma_int for the case of arguments, \c__fp_prec_tuple_int for the case of tuples. Once the operand is found, the lparen_after auxiliary makes sure that there was a closing parenthesis (otherwise it complains), and leaves in the input stream an operand, fetching the following infix operator.

```

23883 \cs_new:cpn { __fp_parse_prefix_(:Nw } #1
23884 {
23885   \exp_after:wN \__fp_parse_lparen_after:NwN
23886   \exp_after:wN #1
23887   \exp:w
23888   \if_int_compare:w #1 = \c__fp_prec_func_int
23889     \__fp_parse_operand:Nw \c__fp_prec_comma_int
23890   \else:
23891     \__fp_parse_operand:Nw \c__fp_prec_tuple_int
23892   \fi:
23893   \__fp_parse_expand:w
23894 }
23895 \cs_new:Npx \__fp_parse_lparen_after:NwN #1#2 @ #3
23896 {
23897   \exp_not:N \token_if_eq_meaning:NNTF #3
23898   \exp_not:c { __fp_parse_infix_):N }
23899   {
23900     \exp_not:N \__fp_exp_after_array_f:w #2 \s__fp_expr_stop
23901     \exp_not:N \exp_after:wN
23902     \exp_not:N \__fp_parse_infix_after_paren:NN
23903     \exp_not:N \exp_after:wN #1
23904     \exp_not:N \exp:w
23905     \exp_not:N \__fp_parse_expand:w
23906   }
23907   {
23908     \exp_not:N \msg_expandable_error:nnn
23909     { fp } { missing } { ) }
23910     \exp_not:N \tl_if_empty:nT {#2} \exp_not:N \c__fp_empty_tuple_fp
23911     #2 @
23912     \exp_not:N \use_none:n #3
23913   }
23914 }

```

(End definition for __fp_parse_prefix_(:Nw and __fp_parse_lparen_after:NwN.)

```

\__fp_parse_prefix_):Nw

```

The right parenthesis can appear as a prefix in two similar cases: in an empty tuple or tuple ending with a comma, or in an empty argument list or argument list ending with a comma, such as in max(1,2,) or in rand().

```

23915 \cs_new:cpn { __fp_parse_prefix_):Nw } #1

```

```

23916 {
23917   \if_int_compare:w #1 = \c__fp_prec_comma_int
23918   \else:
23919     \if_int_compare:w #1 = \c__fp_prec_tuple_int
23920     \exp_after:wN \c__fp_empty_tuple_fp \exp:w
23921     \else:
23922       \msg_expandable_error:nnn
23923       { fp } { missing-number } { ) }
23924       \exp_after:wN \c_nan_fp \exp:w
23925     \fi:
23926     \exp_end_continue_f:w
23927   \fi:
23928   \__fp_parse_infix_after_paren:NN #1 )
23929 }

```

(End definition for __fp_parse_prefix_):Nw.)

69.5.2 Constants

Some words correspond to constant floating points. The floating point constant is left as a result of __fp_parse_one:Nw after expanding __fp_parse_infix:NN.

```

\__fp_parse_word_inf:N
\__fp_parse_word_nan:N
\__fp_parse_word_pi:N
\__fp_parse_word_deg:N
\__fp_parse_word_true:N
\__fp_parse_word_false:N
23930 \cs_set_protected:Npn \__fp_tmp:w #1 #2
23931 {
23932   \cs_new:cpn { __fp_parse_word_#1:N }
23933   { \exp_after:wN #2 \exp:w \exp_end_continue_f:w \__fp_parse_infix:NN }
23934 }

```

```

23935 \__fp_tmp:w { inf } \c_inf_fp
23936 \__fp_tmp:w { nan } \c_nan_fp
23937 \__fp_tmp:w { pi } \c_pi_fp
23938 \__fp_tmp:w { deg } \c_one_degree_fp
23939 \__fp_tmp:w { true } \c_one_fp
23940 \__fp_tmp:w { false } \c_zero_fp

```

(End definition for __fp_parse_word_inf:N and others.)

Copies of __fp_parse_word_...:N commands, to allow arbitrary case as mandated by the standard.

```

\__fp_parse_caseless_inf:N
\__fp_parse_caseless_infinity:N
\__fp_parse_caseless_nan:N
23941 \cs_new_eq:NN \__fp_parse_caseless_inf:N \__fp_parse_word_inf:N
23942 \cs_new_eq:NN \__fp_parse_caseless_infinity:N \__fp_parse_word_inf:N
23943 \cs_new_eq:NN \__fp_parse_caseless_nan:N \__fp_parse_word_nan:N

```

(End definition for __fp_parse_caseless_inf:N, __fp_parse_caseless_infinity:N, and __fp_parse_caseless_nan:N.)

Dimension units are also floating point constants but their value is not stored as a floating point constant. We give the values explicitly here.

```

\__fp_parse_word_pt:N
\__fp_parse_word_in:N
\__fp_parse_word_pc:N
\__fp_parse_word_cm:N
\__fp_parse_word_mm:N
\__fp_parse_word_dd:N
\__fp_parse_word_cc:N
\__fp_parse_word_nd:N
\__fp_parse_word_nc:N
\__fp_parse_word_bp:N
\__fp_parse_word_sp:N
23944 \cs_set_protected:Npn \__fp_tmp:w #1 #2
23945 {
23946   \cs_new:cpn { __fp_parse_word_#1:N }
23947   {
23948     \__fp_exp_after_f:nw { \__fp_parse_infix:NN }
23949     \s__fp \__fp_chk:w 10 #2 ;
23950   }
23951 }

```

```

23952 \__fp_tmp:w {pt} { {1} {1000} {0000} {0000} {0000} }
23953 \__fp_tmp:w {in} { {2} {7227} {0000} {0000} {0000} }
23954 \__fp_tmp:w {pc} { {2} {1200} {0000} {0000} {0000} }
23955 \__fp_tmp:w {cm} { {2} {2845} {2755} {9055} {1181} }
23956 \__fp_tmp:w {mm} { {1} {2845} {2755} {9055} {1181} }
23957 \__fp_tmp:w {dd} { {1} {1070} {0085} {6496} {0630} }
23958 \__fp_tmp:w {cc} { {2} {1284} {0102} {7795} {2756} }
23959 \__fp_tmp:w {nd} { {1} {1066} {9783} {4645} {6693} }
23960 \__fp_tmp:w {nc} { {2} {1280} {3740} {1574} {8031} }
23961 \__fp_tmp:w {bp} { {1} {1003} {7500} {0000} {0000} }
23962 \__fp_tmp:w {sp} { {-4} {1525} {8789} {0625} {0000} }

```

(End definition for __fp_parse_word_pt:N and others.)

__fp_parse_word_em:N The font-dependent units em and ex must be evaluated on the fly. We reuse an auxiliary of \dim_to_fp:n.

```

23963 \tl_map_inline:nn { {em} {ex} }
23964 {
23965   \cs_new:cpn { __fp_parse_word_#1:N }
23966   {
23967     \exp_after:wN \__fp_from_dim_test:ww
23968     \exp_after:wN 0 \exp_after:wN ,
23969     \int_value:w \dim_to_decimal_in_sp:n { 1 #1 } \exp_after:wN ;
23970     \exp:w \exp_end_continue_f:w \__fp_parse_infix:NN
23971   }
23972 }

```

(End definition for __fp_parse_word_em:N and __fp_parse_word_ex:N.)

69.5.3 Functions

```

\__fp_parse_unary_function:NNN
\__fp_parse_function:NNN
23973 \cs_new:Npn \__fp_parse_unary_function:NNN #1#2#3
23974 {
23975   \exp_after:wN \__fp_parse_apply_unary:NNNwN
23976   \exp_after:wN #3
23977   \exp_after:wN #2
23978   \exp_after:wN #1
23979   \exp:w
23980   \__fp_parse_operand:Nw \c__fp_prec_func_int \__fp_parse_expand:w
23981 }
23982 \cs_new:Npn \__fp_parse_function:NNN #1#2#3
23983 {
23984   \exp_after:wN \__fp_parse_apply_function:NNNwN
23985   \exp_after:wN #3
23986   \exp_after:wN #2
23987   \exp_after:wN #1
23988   \exp:w
23989   \__fp_parse_operand:Nw \c__fp_prec_func_int \__fp_parse_expand:w
23990 }

```

(End definition for __fp_parse_unary_function:NNN and __fp_parse_function:NNN.)

69.6 Main functions

`__fp_parse:n` Start an `\exp:w` expansion so that `__fp_parse:n` expands in two steps. The `__fp_parse_operand:Nw` function performs computations until reaching an operation with precedence `\c__fp_prec_end_int` or less, namely, the end of the expression. The marker `\s__fp_expr_mark` indicates that the next token is an already parsed version of an infix operator, and `__fp_parse_infix_end:N` has infinitely negative precedence. Finally, clean up a (well-defined) set of extra tokens and stop the initial expansion with `\exp_end:.`

```

23991 \cs_new:Npn \__fp_parse:n #1
23992 {
23993   \exp:w
23994   \exp_after:wN \__fp_parse_after:ww
23995   \exp:w
23996   \__fp_parse_operand:Nw \c__fp_prec_end_int
23997   \__fp_parse_expand:w #1
23998   \s__fp_expr_mark \__fp_parse_infix_end:N
23999   \s__fp_expr_stop
24000   \exp_end:
24001 }
24002 \cs_new:Npn \__fp_parse_after:ww
24003   #1@ \__fp_parse_infix_end:N \s__fp_expr_stop #2 { #2 #1 }
24004 \cs_new:Npn \__fp_parse_o:n #1
24005 {
24006   \exp:w
24007   \exp_after:wN \__fp_parse_after:ww
24008   \exp:w
24009   \__fp_parse_operand:Nw \c__fp_prec_end_int
24010   \__fp_parse_expand:w #1
24011   \s__fp_expr_mark \__fp_parse_infix_end:N
24012   \s__fp_expr_stop
24013   {
24014     \exp_end_continue_f:w
24015     \__fp_exp_after_any_f:nw { \exp_after:wN \exp_stop_f: }
24016   }
24017 }

```

(End definition for `__fp_parse:n`, `__fp_parse_o:n`, and `__fp_parse_after:ww`.)

`__fp_parse_operand:Nw` This is just a shorthand which sets up both `__fp_parse_continue:NwN` and `__fp_parse_one:Nw` with the same precedence. Note the trailing `\exp:w`.

```

24018 \cs_new:Npn \__fp_parse_operand:Nw #1
24019 {
24020   \exp_end_continue_f:w
24021   \exp_after:wN \__fp_parse_continue:NwN
24022   \exp_after:wN #1
24023   \exp:w \exp_end_continue_f:w
24024   \exp_after:wN \__fp_parse_one:Nw
24025   \exp_after:wN #1
24026   \exp:w
24027 }
24028 \cs_new:Npn \__fp_parse_continue:NwN #1 #2 @ #3 { #3 #1 #2 @ }

```

(End definition for `__fp_parse_operand:Nw` and `__fp_parse_continue:NwN`.)

_fp_parse_apply_binary:NwNwN
 _fp_parse_apply_binary_chk:NN
 _fp_parse_apply_binary_error:NNN

Receives $\langle precedence \rangle \langle operand_1 \rangle @ \langle operation \rangle \langle operand_2 \rangle @ \langle infix command \rangle$. Builds the appropriate call to the $\langle operation \rangle$ #3, dispatching on both types. If the resulting control sequence does not exist, the operation is not allowed.

This is redefined in l3fp-extras.

```

24029 \cs_new:Npn \_fp_parse_apply_binary:NwNwN #1 #2#3@ #4 #5#6@ #7
24030 {
24031   \exp_after:wN \_fp_parse_continue:NwN
24032   \exp_after:wN #1
24033   \exp:w \exp_end_continue_f:w
24034   \exp_after:wN \_fp_parse_apply_binary_chk:NN
24035   \cs:w
24036     __fp
24037     \_fp_type_from_scan:N #2
24038     _#4
24039     \_fp_type_from_scan:N #5
24040     _o:ww
24041     \cs_end:
24042     #4
24043     #2#3 #5#6
24044     \exp:w \exp_end_continue_f:w #7 #1
24045 }
24046 \cs_new:Npn \_fp_parse_apply_binary_chk:NN #1#2
24047 {
24048   \if_meaning:w \scan_stop: #1
24049     \_fp_parse_apply_binary_error:NNN #2
24050   \fi:
24051   #1
24052 }
24053 \cs_new:Npn \_fp_parse_apply_binary_error:NNN #1#2#3
24054 {
24055   #2
24056   \_fp_invalid_operation_o:Nww #1
24057 }
```

(End definition for _fp_parse_apply_binary:NwNwN, _fp_parse_apply_binary_chk:NN, and _fp_parse_apply_binary_error:NNN.)

_fp_binary_type_o:Nww
 _fp_binary_rev_type_o:Nww

Applies the operator #1 to its two arguments, dispatching according to their types, and expands once after the result. The rev version swaps its arguments before doing this.

```

24058 \cs_new:Npn \_fp_binary_type_o:Nww #1 #2#3 ; #4
24059 {
24060   \exp_after:wN \_fp_parse_apply_binary_chk:NN
24061   \cs:w
24062     __fp
24063     \_fp_type_from_scan:N #2
24064     _#1
24065     \_fp_type_from_scan:N #4
24066     _o:ww
24067     \cs_end:
24068     #1
24069     #2 #3 ; #4
24070 }
24071 \cs_new:Npn \_fp_binary_rev_type_o:Nww #1 #2#3 ; #4#5 ;
24072 {
```

```

24073 \exp_after:wN \_fp_parse_apply_binary_chk:NN
24074 \cs:w
24075 \_fp
24076 \_fp_type_from_scan:N #4
24077 _ #1
24078 \_fp_type_from_scan:N #2
24079 _o:ww
24080 \cs_end:
24081 #1
24082 #4 #5 ; #2 #3 ;
24083 }

```

(End definition for _fp_binary_type_o:Nww and _fp_binary_rev_type_o:Nww.)

69.7 Infix operators

_fp_parse_infix_after_operand:NwN

```

24084 \cs_new:Npn \_fp_parse_infix_after_operand:NwN #1 #2;
24085 {
24086 \_fp_exp_after_f:nw { \_fp_parse_infix:NN #1 }
24087 #2;
24088 }
24089 \cs_new:Npn \_fp_parse_infix:NN #1 #2
24090 {
24091 \if_catcode:w \scan_stop: \exp_not:N #2
24092 \if:w 0 \_fp_str_if_eq:nn { \s_fp_expr_mark } { \exp_not:N #2 }
24093 \exp_after:wN \exp_after:wN
24094 \exp_after:wN \_fp_parse_infix_mark:NNN
24095 \else:
24096 \exp_after:wN \exp_after:wN
24097 \exp_after:wN \_fp_parse_infix_juxt:N
24098 \fi:
24099 \else:
24100 \if_int_compare:w
24101 \_fp_int_eval:w
24102 ( '#2 \if_int_compare:w '#2 > 'Z - 32 \fi: ) / 26
24103 = 3 \exp_stop_f:
24104 \exp_after:wN \exp_after:wN
24105 \exp_after:wN \_fp_parse_infix_juxt:N
24106 \else:
24107 \exp_after:wN \_fp_parse_infix_check:NNN
24108 \cs:w
24109 \_fp_parse_infix_ \token_to_str:N #2 :N
24110 \exp_after:wN \exp_after:wN \exp_after:wN
24111 \cs_end:
24112 \fi:
24113 \fi:
24114 #1
24115 #2
24116 }
24117 \cs_new:Npn \_fp_parse_infix_check:NNN #1#2#3
24118 {
24119 \if_meaning:w \scan_stop: #1

```

```

24120     \msg_expandable_error:nnn
24121     { fp } { missing } { * }
24122     \exp_after:wN \__fp_parse_infix_mul:N
24123     \exp_after:wN #2
24124     \exp_after:wN #3
24125     \else:
24126     \exp_after:wN #1
24127     \exp_after:wN #2
24128     \exp:w \exp_after:wN \__fp_parse_expand:w
24129     \fi:
24130 }

```

(End definition for __fp_parse_infix_after_operand:NwN.)

__fp_parse_infix_after_paren:NN Variant of __fp_parse_infix:NN for use after a closing parenthesis. The only difference is that __fp_parse_infix_juxt:N is replaced by __fp_parse_infix_mul:N.

```

24131 \cs_new:Npn \__fp_parse_infix_after_paren:NN #1 #2
24132 {
24133   \if_catcode:w \scan_stop: \exp_not:N #2
24134   \if:w 0 \__fp_str_if_eq:nn { \s__fp_expr_mark } { \exp_not:N #2 }
24135   \exp_after:wN \exp_after:wN
24136   \exp_after:wN \__fp_parse_infix_mark:NNN
24137   \else:
24138   \exp_after:wN \exp_after:wN
24139   \exp_after:wN \__fp_parse_infix_mul:N
24140   \fi:
24141   \else:
24142   \if_int_compare:w
24143     \__fp_int_eval:w
24144     ( '#2 \if_int_compare:w '#2 > 'Z - 32 \fi: ) / 26
24145     = 3 \exp_stop_f:
24146   \exp_after:wN \exp_after:wN
24147   \exp_after:wN \__fp_parse_infix_mul:N
24148   \else:
24149   \exp_after:wN \__fp_parse_infix_check:NNN
24150   \cs:w
24151     __fp_parse_infix_ \token_to_str:N #2 :N
24152   \exp_after:wN \exp_after:wN \exp_after:wN
24153   \cs_end:
24154   \fi:
24155   \fi:
24156   #1
24157   #2
24158 }

```

(End definition for __fp_parse_infix_after_paren:NN.)

69.7.1 Closing parentheses and commas

__fp_parse_infix_mark:NNN As an infix operator, \s__fp_expr_mark means that the next token (#3) has already gone through __fp_parse_infix:NN and should be provided the precedence #1. The scan mark #2 is discarded.

```

24159 \cs_new:Npn \__fp_parse_infix_mark:NNN #1#2#3 { #3 #1 }

```

(End definition for _fp_parse_infix_mark:NNN.)

_fp_parse_infix_end:N This one is a little bit odd: force every previous operator to end, regardless of the precedence.

```
24160 \cs_new:Npn \_fp_parse_infix_end:N #1
24161 { @ \use_none:n \_fp_parse_infix_end:N }
```

(End definition for _fp_parse_infix_end:N.)

_fp_parse_infix_):N This is very similar to _fp_parse_infix_end:N, complaining about an extra closing parenthesis if the previous operator was the beginning of the expression, with precedence \c_fp_prec_end_int.

```
24162 \cs_set_protected:Npn \_fp_tmp:w #1
24163 {
24164   \cs_new:Npn #1 ##1
24165   {
24166     \if_int_compare:w ##1 > \c_fp_prec_end_int
24167       \exp_after:wN @
24168       \exp_after:wN \use_none:n
24169       \exp_after:wN #1
24170     \else:
24171       \msg_expandable_error:nnn { fp } { extra } { } }
24172     \exp_after:wN \_fp_parse_infix:NN
24173     \exp_after:wN ##1
24174     \exp:w \exp_after:wN \_fp_parse_expand:w
24175   \fi:
24176 }
24177 }
24178 \exp_args:Nc \_fp_tmp:w { \_fp_parse_infix_):N }
```

(End definition for _fp_parse_infix_):N.)

_fp_parse_infix_,:N As for other infix operations, if the previous operations has higher precedence the comma waits. Otherwise we call _fp_parse_operand:Nw to read more comma-delimited arguments that _fp_parse_infix_comma:w simply concatenates into a @-delimited array. The first comma in a tuple that is not a function argument is distinguished: in that case call _fp_parse_apply_comma:NwNwN whose job is to convert the first item of the tuple and an array of the remaining items into a tuple. In contrast to _fp_parse_apply_binary:NwNwN this function's operands are not single-object arrays.

```
24179 \cs_set_protected:Npn \_fp_tmp:w #1
24180 {
24181   \cs_new:Npn #1 ##1
24182   {
24183     \if_int_compare:w ##1 > \c_fp_prec_comma_int
24184       \exp_after:wN @
24185       \exp_after:wN \use_none:n
24186       \exp_after:wN #1
24187     \else:
24188       \if_int_compare:w ##1 < \c_fp_prec_comma_int
24189         \exp_after:wN @
24190         \exp_after:wN \_fp_parse_apply_comma:NwNwN
24191         \exp_after:wN ,
24192         \exp:w
24193       \else:
```



```

24194         \exp_after:wN \__fp_parse_infix_comma:w
24195         \exp:w
24196         \fi:
24197         \__fp_parse_operand:Nw \c__fp_prec_comma_int
24198         \exp_after:wN \__fp_parse_expand:w
24199         \fi:
24200     }
24201 }
24202 \exp_args:Nc \__fp_tmp:w { __fp_parse_infix_,:N }
24203 \cs_new:Npn \__fp_parse_infix_comma:w #1 @
24204 { #1 @ \use_none:n }
24205 \cs_new:Npn \__fp_parse_apply_comma:NwNwN #1 #2@ #3 #4@ #5
24206 {
24207     \exp_after:wN \__fp_parse_continue:NwN
24208     \exp_after:wN #1
24209     \exp:w \exp_end_continue_f:w
24210     \__fp_exp_after_tuple_f:nw { }
24211     \s__fp_tuple \__fp_tuple_chk:w { #2 #4 } ;
24212     #5 #1
24213 }

```

(End definition for `__fp_parse_infix_,:N`, `__fp_parse_infix_comma:w`, and `__fp_parse_apply_comma:NwNwN`.)

69.7.2 Usual infix operators

As described in the “work plan”, each infix operator has an associated `\...infix...` function, a computing function, and precedence, given as arguments to `__fp_tmp:w`. Using the general mechanism for arithmetic operations. The power operation must be associative in the opposite order from all others. For this, we use two distinct precedences.

```

\__fp_parse_infix_+:N
\__fp_parse_infix_-:N
\__fp_parse_infix_juxt:N
\__fp_parse_infix_/:N
\__fp_parse_infix_mul:N
\__fp_parse_infix_and:N
\__fp_parse_infix_or:N
\__fp_parse_infix_^:N
24214 \cs_set_protected:Npn \__fp_tmp:w #1#2#3#4
24215 {
24216     \cs_new:Npn #1 ##1
24217     {
24218         \if_int_compare:w ##1 < #3
24219         \exp_after:wN @
24220         \exp_after:wN \__fp_parse_apply_binary:NwNwN
24221         \exp_after:wN #2
24222         \exp:w
24223         \__fp_parse_operand:Nw #4
24224         \exp_after:wN \__fp_parse_expand:w
24225     \else:
24226         \exp_after:wN @
24227         \exp_after:wN \use_none:n
24228         \exp_after:wN #1
24229     \fi:
24230 }
24231 }
24232 \exp_args:Nc \__fp_tmp:w { __fp_parse_infix_^:N } ^
24233 \c__fp_prec_hatii_int \c__fp_prec_hat_int
24234 \exp_args:Nc \__fp_tmp:w { __fp_parse_infix_juxt:N } *
24235 \c__fp_prec_juxt_int \c__fp_prec_juxt_int
24236 \exp_args:Nc \__fp_tmp:w { __fp_parse_infix_/:N } /
24237 \c__fp_prec_times_int \c__fp_prec_times_int

```

```

24238 \exp_args:Nc \__fp_tmp:w { __fp_parse_infix_mul:N } *
24239 \c__fp_prec_times_int \c__fp_prec_times_int
24240 \exp_args:Nc \__fp_tmp:w { __fp_parse_infix -:N } -
24241 \c__fp_prec_plus_int \c__fp_prec_plus_int
24242 \exp_args:Nc \__fp_tmp:w { __fp_parse_infix +:N } +
24243 \c__fp_prec_plus_int \c__fp_prec_plus_int
24244 \exp_args:Nc \__fp_tmp:w { __fp_parse_infix_and:N } &
24245 \c__fp_prec_and_int \c__fp_prec_and_int
24246 \exp_args:Nc \__fp_tmp:w { __fp_parse_infix_or:N } |
24247 \c__fp_prec_or_int \c__fp_prec_or_int

```

(End definition for __fp_parse_infix_+:N and others.)

69.7.3 Juxtaposition

__fp_parse_infix_(:N When an opening parenthesis appears where we expect an infix operator, we compute the product of the previous operand and the contents of the parentheses using __fp_parse_infix_mul:N.

```

24248 \cs_new:cpn { __fp_parse_infix_(:N } #1
24249 { \__fp_parse_infix_mul:N #1 ( }

```

(End definition for __fp_parse_infix_(:N.)

69.7.4 Multi-character cases

__fp_parse_infix_*:N

```

24250 \cs_set_protected:Npn \__fp_tmp:w #1
24251 {
24252   \cs_new:cpn { __fp_parse_infix_*:N } ##1##2
24253   {
24254     \if:w * \exp_not:N ##2
24255       \exp_after:wN #1
24256       \exp_after:wN ##1
24257     \else:
24258       \exp_after:wN \__fp_parse_infix_mul:N
24259       \exp_after:wN ##1
24260       \exp_after:wN ##2
24261     \fi:
24262   }
24263 }
24264 \exp_args:Nc \__fp_tmp:w { __fp_parse_infix^:N }

```

(End definition for __fp_parse_infix_*:N.)

__fp_parse_infix_|:Nw

__fp_parse_infix_&:Nw

```

24265 \cs_set_protected:Npn \__fp_tmp:w #1#2#3
24266 {
24267   \cs_new:Npn #1 ##1##2
24268   {
24269     \if:w #2 \exp_not:N ##2
24270       \exp_after:wN #1
24271       \exp_after:wN ##1
24272       \exp:w \exp_after:wN \__fp_parse_expand:w
24273     \else:

```

```

24274         \exp_after:wN #3
24275         \exp_after:wN ##1
24276         \exp_after:wN ##2
24277     \fi:
24278 }
24279 }
24280 \exp_args:Nc \__fp_tmp:w { __fp_parse_infix_|:N } | \__fp_parse_infix_or:N
24281 \exp_args:Nc \__fp_tmp:w { __fp_parse_infix_&:N } & \__fp_parse_infix_and:N

```

(End definition for __fp_parse_infix_|:Nw and __fp_parse_infix_&:Nw.)

69.7.5 Ternary operator

```

\__fp_parse_infix_?:N
\__fp_parse_infix_:N
24282 \cs_set_protected:Npn \__fp_tmp:w #1#2#3#4
24283 {
24284     \cs_new:Npn #1 ##1
24285     {
24286         \if_int_compare:w ##1 < \c__fp_prec_quest_int
24287         #4
24288         \exp_after:wN @
24289         \exp_after:wN #2
24290         \exp:w
24291         \__fp_parse_operand:Nw #3
24292         \exp_after:wN \__fp_parse_expand:w
24293     \else:
24294         \exp_after:wN @
24295         \exp_after:wN \use_none:n
24296         \exp_after:wN #1
24297     \fi:
24298 }
24299 }
24300 \exp_args:Nc \__fp_tmp:w { __fp_parse_infix_?:N }
24301 \__fp_ternary:NwwN \c__fp_prec_quest_int { }
24302 \exp_args:Nc \__fp_tmp:w { __fp_parse_infix_:N }
24303 \__fp_ternary_auxii:NwwN \c__fp_prec_colon_int
24304 {
24305     \msg_expandable_error:nnnn
24306     { fp } { missing } { ? } { ~for~?: }
24307 }

```

(End definition for __fp_parse_infix_?:N and __fp_parse_infix_:N.)

69.7.6 Comparisons

```

\__fp_parse_infix_<:N
\__fp_parse_infix_=:N
\__fp_parse_infix_>:N
\__fp_parse_infix_!:N
\__fp_parse_excl_error:
\__fp_parse_compare:NNNNNNN
\__fp_parse_compare_auxi:NNNNNN
\__fp_parse_compare_auxii:NNNNN
\__fp_parse_compare_end:NNNNw
\__fp_compare:wNNNNw
24308 \cs_new:cpn { __fp_parse_infix_<:N } #1
24309 { \__fp_parse_compare:NNNNNNN #1 1 0 0 0 0 < }
24310 \cs_new:cpn { __fp_parse_infix_=:N } #1
24311 { \__fp_parse_compare:NNNNNNN #1 1 0 0 0 0 = }
24312 \cs_new:cpn { __fp_parse_infix_>:N } #1
24313 { \__fp_parse_compare:NNNNNNN #1 1 0 0 0 0 > }
24314 \cs_new:cpn { __fp_parse_infix_!:N } #1
24315 {

```

```

24316     \exp_after:wN \__fp_parse_compare:NNNNNNN
24317     \exp_after:wN #1
24318     \exp_after:wN 0
24319     \exp_after:wN 1
24320     \exp_after:wN 1
24321     \exp_after:wN 1
24322     \exp_after:wN 1
24323 }
24324 \cs_new:Npn \__fp_parse_excl_error:
24325 {
24326     \msg_expandable_error:nnnn
24327     { fp } { missing } { = } { { ~after~!. }
24328 }
24329 \cs_new:Npn \__fp_parse_compare:NNNNNNN #1
24330 {
24331     \if_int_compare:w #1 < \c__fp_prec_comp_int
24332     \exp_after:wN \__fp_parse_compare_auxi:NNNNNNN
24333     \exp_after:wN \__fp_parse_excl_error:
24334     \else:
24335     \exp_after:wN @
24336     \exp_after:wN \use_none:n
24337     \exp_after:wN \__fp_parse_compare:NNNNNNN
24338     \fi:
24339 }
24340 \cs_new:Npn \__fp_parse_compare_auxi:NNNNNNN #1#2#3#4#5#6#7
24341 {
24342     \if_case:w
24343     \__fp_int_eval:w \exp_after:wN ' \token_to_str:N #7 - '<
24344     \__fp_int_eval_end:
24345     \__fp_parse_compare_auxii:NNNNN #2#2#4#5#6
24346     \or: \__fp_parse_compare_auxii:NNNNN #2#3#2#5#6
24347     \or: \__fp_parse_compare_auxii:NNNNN #2#3#4#2#6
24348     \or: \__fp_parse_compare_auxii:NNNNN #2#3#4#5#2
24349     \else: #1 \__fp_parse_compare_end:NNNNw #3#4#5#6#7
24350     \fi:
24351 }
24352 \cs_new:Npn \__fp_parse_compare_auxii:NNNNN #1#2#3#4#5
24353 {
24354     \exp_after:wN \__fp_parse_compare_auxi:NNNNNNN
24355     \exp_after:wN \prg_do_nothing:
24356     \exp_after:wN #1
24357     \exp_after:wN #2
24358     \exp_after:wN #3
24359     \exp_after:wN #4
24360     \exp_after:wN #5
24361     \exp:w \exp_after:wN \__fp_parse_expand:w
24362 }
24363 \cs_new:Npn \__fp_parse_compare_end:NNNNw #1#2#3#4#5 \fi:
24364 {
24365     \fi:
24366     \exp_after:wN @
24367     \exp_after:wN \__fp_parse_apply_compare:NwNNNNNNwN
24368     \exp_after:wN \c_one_fp
24369     \exp_after:wN #1

```

```

24370     \exp_after:wN #2
24371     \exp_after:wN #3
24372     \exp_after:wN #4
24373     \exp:w
24374     \__fp_parse_operand:Nw \c__fp_prec_comp_int \__fp_parse_expand:w #5
24375 }
24376 \cs_new:Npn \__fp_parse_apply_compare:NwNNNNNwN
24377   #1 #2@ #3 #4#5#6#7 #8@ #9
24378 {
24379   \if_int_odd:w
24380     \if_meaning:w \c_zero_fp #3
24381     0
24382   \else:
24383     \if_case:w \__fp_compare_back_any:ww #8 #2 \exp_stop_f:
24384     #5 \or: #6 \or: #7 \else: #4
24385     \fi:
24386   \fi:
24387   \exp_stop_f:
24388   \exp_after:wN \__fp_parse_apply_compare_aux:NNwN
24389   \exp_after:wN \c_one_fp
24390 \else:
24391   \exp_after:wN \__fp_parse_apply_compare_aux:NNwN
24392   \exp_after:wN \c_zero_fp
24393 \fi:
24394 #1 #8 #9
24395 }
24396 \cs_new:Npn \__fp_parse_apply_compare_aux:NNwN #1 #2 #3; #4
24397 {
24398   \if_meaning:w \__fp_parse_compare:NNNNNNN #4
24399   \exp_after:wN \__fp_parse_continue_compare:NNwNN
24400   \exp_after:wN #1
24401   \exp_after:wN #2
24402   \exp:w \exp_end_continue_f:w
24403   \__fp_exp_after_o:w #3;
24404   \exp:w \exp_end_continue_f:w
24405 \else:
24406   \exp_after:wN \__fp_parse_continue:NwN
24407   \exp_after:wN #2
24408   \exp:w \exp_end_continue_f:w
24409   \exp_after:wN #1
24410   \exp:w \exp_end_continue_f:w
24411 \fi:
24412 #4 #2
24413 }
24414 \cs_new:Npn \__fp_parse_continue_compare:NNwNN #1#2 #3@ #4#5
24415 { #4 #2 #3@ #1 }

```

(End definition for __fp_parse_infix_<:N and others.)

69.8 Tools for functions

_fp_parse_function_all_fp_o:fnw Followed by $\{\langle function\ name \rangle\} \{\langle code \rangle\} \langle float\ array \rangle$ @ this checks all floats are floating point numbers (no tuples).

```

24416 \cs_new:Npn \__fp_parse_function_all_fp_o:fnw #1#2#3 @
24417 {
24418     \__fp_array_if_all_fp:nTF {#3}
24419     { #2 #3 @ }
24420     {
24421         \__fp_error:nffn { bad-args }
24422         {#1}
24423         { \fp_to_tl:n { \s__fp_tuple \__fp_tuple_chk:w {#3} ; } }
24424         { }
24425         \exp_after:wN \c_nan_fp
24426     }
24427 }

```

(End definition for __fp_parse_function_all_fp_o:fnw.)

__fp_parse_function_one_two:nnw
 __fp_parse_function_one_two_error_o:w
 __fp_parse_function_one_two_aux:nnw
 __fp_parse_function_one_two_auxii:nnw

This is followed by $\{(function\ name)\} \{(code)\} \langle float\ array \rangle @$. It checks that the $\langle float\ array \rangle$ consists of one or two floating point numbers (not tuples), then leaves the $\langle code \rangle$ (if there is one float) or its tail (if there are two floats) followed by the $\langle float\ array \rangle$. The $\langle code \rangle$ should start with a single token such as `__fp_atan_default:w` that deals with the single-float case.

The first `__fp_if_type_fp:NTwFw` test catches the case of no argument and the case of a tuple argument. The next one distinguishes the case of a single argument (no error, just add `\c_one_fp`) from a tuple second argument. Finally check there is no further argument.

```

24428 \cs_new:Npn \__fp_parse_function_one_two:nnw #1#2#3
24429 {
24430     \__fp_if_type_fp:NTwFw
24431     #3 { } \s__fp \__fp_parse_function_one_two_error_o:w \s__fp_stop
24432     \__fp_parse_function_one_two_aux:nnw {#1} {#2} #3
24433 }
24434 \cs_new:Npn \__fp_parse_function_one_two_error_o:w #1#2#3#4 @
24435 {
24436     \__fp_error:nffn { bad-args }
24437     {#2}
24438     { \fp_to_tl:n { \s__fp_tuple \__fp_tuple_chk:w {#4} ; } }
24439     { }
24440     \exp_after:wN \c_nan_fp
24441 }
24442 \cs_new:Npn \__fp_parse_function_one_two_aux:nnw #1#2 #3; #4
24443 {
24444     \__fp_if_type_fp:NTwFw
24445     #4 { }
24446     \s__fp
24447     {
24448         \if_meaning:w @ #4
24449         \exp_after:wN \use_iv:nnnn
24450         \fi:
24451         \__fp_parse_function_one_two_error_o:w
24452     }
24453     \s__fp_stop
24454     \__fp_parse_function_one_two_auxii:nnw {#1} {#2} #3; #4
24455 }
24456 \cs_new:Npn \__fp_parse_function_one_two_auxii:nnw #1#2#3; #4; #5
24457 {

```

```

24458     \if_meaning:w @ #5 \else:
24459         \exp_after:wN \__fp_parse_function_one_two_error_o:w
24460     \fi:
24461     \use_ii:nn {#1} { \use_none:n #2 } #3; #4; #5
24462 }

```

(End definition for __fp_parse_function_one_two:nnw and others.)

__fp_tuple_map_o:nw Apply #1 to all items in the following tuple and expand once afterwards. The code #1
 __fp_tuple_map_loop_o:nw should itself expand once after its result.

```

24463 \cs_new:Npn \__fp_tuple_map_o:nw #1 \s__fp_tuple \__fp_tuple_chk:w #2 ;
24464 {
24465     \exp_after:wN \s__fp_tuple
24466     \exp_after:wN \__fp_tuple_chk:w
24467     \exp_after:wN {
24468         \exp:w \exp_end_continue_f:w
24469         \__fp_tuple_map_loop_o:nw {#1} #2
24470         { \s__fp \prg_break: } ;
24471         \prg_break_point:
24472     \exp_after:wN } \exp_after:wN ;
24473 }
24474 \cs_new:Npn \__fp_tuple_map_loop_o:nw #1#2#3 ;
24475 {
24476     \use_none:n #2
24477     #1 #2 #3 ;
24478     \exp:w \exp_end_continue_f:w
24479     \__fp_tuple_map_loop_o:nw {#1}
24480 }

```

(End definition for __fp_tuple_map_o:nw and __fp_tuple_map_loop_o:nw.)

__fp_tuple_mapthread_o:nww Apply #1 to pairs of items in the two following tuples and expand once afterwards.

```

\__fp_tuple_mapthread_loop_o:nw
24481 \cs_new:Npn \__fp_tuple_mapthread_o:nww #1
24482     \s__fp_tuple \__fp_tuple_chk:w #2 ;
24483     \s__fp_tuple \__fp_tuple_chk:w #3 ;
24484 {
24485     \exp_after:wN \s__fp_tuple
24486     \exp_after:wN \__fp_tuple_chk:w
24487     \exp_after:wN {
24488         \exp:w \exp_end_continue_f:w
24489         \__fp_tuple_mapthread_loop_o:nw {#1}
24490         #2 { \s__fp \prg_break: } ; @
24491         #3 { \s__fp \prg_break: } ;
24492         \prg_break_point:
24493     \exp_after:wN } \exp_after:wN ;
24494 }
24495 \cs_new:Npn \__fp_tuple_mapthread_loop_o:nw #1#2#3 ; #4 @ #5#6 ;
24496 {
24497     \use_none:n #2
24498     \use_none:n #5
24499     #1 #2 #3 ; #5 #6 ;
24500     \exp:w \exp_end_continue_f:w
24501     \__fp_tuple_mapthread_loop_o:nw {#1} #4 @
24502 }

```

(End definition for __fp_tuple_mapthread_o:nww and __fp_tuple_mapthread_loop_o:nw.)

69.9 Messages

```
24503 \msg_new:nnn { fp } { deprecated }
24504 { '#1'~deprecated;~use~'#2' }
24505 \msg_new:nnn { fp } { unknown-fp-word }
24506 { Unknown~fp~word~#1. }
24507 \msg_new:nnn { fp } { missing }
24508 { Missing~#1~inserted #2. }
24509 \msg_new:nnn { fp } { extra }
24510 { Extra~#1~ignored. }
24511 \msg_new:nnn { fp } { early-end }
24512 { Premature~end~in~fp~expression. }
24513 \msg_new:nnn { fp } { after-e }
24514 { Cannot~use~#1 after~'e'. }
24515 \msg_new:nnn { fp } { missing-number }
24516 { Missing~number~before~'#1'. }
24517 \msg_new:nnn { fp } { unknown-symbol }
24518 { Unknown~symbol~#1~ignored. }
24519 \msg_new:nnn { fp } { extra-comma }
24520 { Unexpected~comma~turned~to~nan~result.}
24521 \msg_new:nnn { fp } { no-arg }
24522 { #1~got~no~argument;~used~nan. }
24523 \msg_new:nnn { fp } { multi-arg }
24524 { #1~got~more~than~one~argument;~used~nan. }
24525 \msg_new:nnn { fp } { num-args }
24526 { #1~expects~between~#2~and~#3~arguments. }
24527 \msg_new:nnn { fp } { bad-args }
24528 { Arguments~in~#1#2~are~invalid. }
24529 \msg_new:nnn { fp } { infty-pi }
24530 { Math~command~#1 is~not~an~fp }
24531 \cs_if_exist:cT { @unexpandable@protect }
24532 {
24533   \msg_new:nnn { fp } { robust-cmd }
24534   { Robust~command~#1 invalid~in~fp~expression! }
24535 }
24536 </package>
```


Chapter 70

l3fp-assign implementation

```
24537 ⟨*package⟩
24538 ⟨@@=fp⟩
```

70.1 Assigning values

\fp_new:N Floating point variables are initialized to be +0.

```
24539 \cs_new_protected:Npn \fp_new:N #1
24540   { \cs_new_eq:NN #1 \c_zero_fp }
24541 \cs_generate_variant:Nn \fp_new:N {c}
```

(End definition for \fp_new:N. This function is documented on page 240.)

\fp_set:Nn Simply use __fp_parse:n within various f-expanding assignments.

```
\fp_set:cn 24542 \cs_new_protected:Npn \fp_set:Nn #1#2
\fp_gset:Nn 24543   { \__kernel_tl_set:Nx #1 { \exp_not:f { \__fp_parse:n {#2} } } }
\fp_gset:cn 24544 \cs_new_protected:Npn \fp_gset:Nn #1#2
\fp_const:Nn 24545   { \__kernel_tl_gset:Nx #1 { \exp_not:f { \__fp_parse:n {#2} } } }
\fp_const:cn 24546 \cs_new_protected:Npn \fp_const:Nn #1#2
24547   { \tl_const:Nx #1 { \exp_not:f { \__fp_parse:n {#2} } } }
24548 \cs_generate_variant:Nn \fp_set:Nn {c}
24549 \cs_generate_variant:Nn \fp_gset:Nn {c}
24550 \cs_generate_variant:Nn \fp_const:Nn {c}
```

(End definition for \fp_set:Nn, \fp_gset:Nn, and \fp_const:Nn. These functions are documented on page 240.)

\fp_set_eq:NN Copying a floating point is the same as copying the underlying token list.

```
\fp_set_eq:cn 24551 \cs_new_eq:NN \fp_set_eq:NN \tl_set_eq:NN
\fp_set_eq:Nc 24552 \cs_new_eq:NN \fp_gset_eq:NN \tl_gset_eq:NN
\fp_set_eq:cc 24553 \cs_generate_variant:Nn \fp_set_eq:NN { c , Nc , cc }
\fp_gset_eq:NN 24554 \cs_generate_variant:Nn \fp_gset_eq:NN { c , Nc , cc }
```

(End definition for \fp_set_eq:NN and \fp_gset_eq:NN. These functions are documented on page 240.)

\fp_gset_zero:cn Setting a floating point to zero: copy \c_zero_fp.

```
\fp_zero:c 24555 \cs_new_protected:Npn \fp_zero:N #1 { \fp_set_eq:NN #1 \c_zero_fp }
\fp_gzero:N 24556 \cs_new_protected:Npn \fp_gzero:N #1 { \fp_gset_eq:NN #1 \c_zero_fp }
\fp_gzero:c 24557 \cs_generate_variant:Nn \fp_zero:N { c }
24558 \cs_generate_variant:Nn \fp_gzero:N { c }
```

(End definition for `\fp_zero:N` and `\fp_gzero:N`. These functions are documented on page 240.)

```

\fp_zero_new:N Set the floating point to zero, or define it if needed.
\fp_zero_new:c 24559 \cs_new_protected:Npn \fp_zero_new:N #1
\fp_gzero_new:N 24560 { \fp_if_exist:NTF #1 { \fp_zero:N #1 } { \fp_new:N #1 } }
\fp_gzero_new:c 24561 \cs_new_protected:Npn \fp_gzero_new:N #1
24562 { \fp_if_exist:NTF #1 { \fp_gzero:N #1 } { \fp_new:N #1 } }
24563 \cs_generate_variant:Nn \fp_zero_new:N { c }
24564 \cs_generate_variant:Nn \fp_gzero_new:N { c }

```

(End definition for `\fp_zero_new:N` and `\fp_gzero_new:N`. These functions are documented on page 240.)

70.2 Updating values

These match the equivalent functions in `l3int` and `l3skip`.

```

\fp_add:Nn For the sake of error recovery we should not simply set #1 to #1 ± (#2): for instance, if #2
\fp_add:cn is 0)+2, the parsing error would be raised at the last closing parenthesis rather than at
\fp_gadd:Nn the closing parenthesis in the user argument. Thus we evaluate #2 instead of just putting
\fp_gadd:cn parentheses. As an optimization we use __fp_parse:n rather than \fp_eval:n, which
\fp_sub:Nn would convert the result away from the internal representation and back.
\fp_sub:cn 24565 \cs_new_protected:Npn \fp_add:Nn { __fp_add:NNNn \fp_set:Nn + }
\fp_gsub:Nn 24566 \cs_new_protected:Npn \fp_gadd:Nn { __fp_add:NNNn \fp_gset:Nn + }
\fp_gsub:cn 24567 \cs_new_protected:Npn \fp_sub:Nn { __fp_add:NNNn \fp_set:Nn - }
__fp_add:NNNn 24568 \cs_new_protected:Npn \fp_gsub:Nn { __fp_add:NNNn \fp_gset:Nn - }
24569 \cs_new_protected:Npn __fp_add:NNNn #1#2#3#4
24570 { #1 #3 { #3 #2 __fp_parse:n {#4} } }
24571 \cs_generate_variant:Nn \fp_add:Nn { c }
24572 \cs_generate_variant:Nn \fp_gadd:Nn { c }
24573 \cs_generate_variant:Nn \fp_sub:Nn { c }
24574 \cs_generate_variant:Nn \fp_gsub:Nn { c }

```

(End definition for `\fp_add:Nn` and others. These functions are documented on page 240.)

70.3 Showing values

```

\fp_show:N This shows the result of computing its argument by passing the right data to \tl_show:n
\fp_show:c or \tl_log:n.
\fp_log:N 24575 \cs_new_protected:Npn \fp_show:N { __fp_show:NN \tl_show:n }
\fp_log:c 24576 \cs_generate_variant:Nn \fp_show:N { c }
__fp_show:NN 24577 \cs_new_protected:Npn \fp_log:N { __fp_show:NN \tl_log:n }
__fp_show_validate:w 24578 \cs_generate_variant:Nn \fp_log:N { c }
24579 \cs_new_protected:Npn __fp_show:NN #1#2
24580 {
24581   \__kernel_chk_tl_type:NnnT #2 { fp }
24582   {
24583     \str_if_eq:eeTF { \tl_head:N #2 } { \s__fp_tuple } { \exp_not:o #2 }
24584     {
24585       \exp_after:wN __fp_show_validate:w #2
24586       \s__fp __fp_chk:w ??? ; \s__fp_stop
24587     }

```

```

24588     }
24589     { \exp_args:Nx #1 { \token_to_str:N #2 = \fp_to_tl:N #2 } }
24590   }
24591   \cs_new:Npn \__fp_show_validate:w
24592     #1 \s__fp \__fp_chk:w #2#3#4#5 ; #6 \s__fp_stop
24593   {
24594     \token_if_eq_meaning:NNTF #2 1
24595     { \s__fp \__fp_chk:w #2 #3 {#4} #5 ; }
24596     { \s__fp \__fp_chk:w #2 #3 #4 #5 ; }
24597   }

```

(End definition for `\fp_show:N` and others. These functions are documented on page 248.)

`\fp_show:n` Use general tools.

```

\fp_log:n
24598 \cs_new_protected:Npn \fp_show:n
24599   { \msg_show_eval:Nn \fp_to_tl:n }
24600 \cs_new_protected:Npn \fp_log:n
24601   { \msg_log_eval:Nn \fp_to_tl:n }

```

(End definition for `\fp_show:n` and `\fp_log:n`. These functions are documented on page 248.)

70.4 Some useful constants and scratch variables

`\c_one_fp` Some constants.

```

\c_e_fp
24602 \fp_const:Nn \c_e_fp { 2.718 2818 2845 9045 }
24603 \fp_const:Nn \c_one_fp { 1 }

```

(End definition for `\c_one_fp` and `\c_e_fp`. These variables are documented on page 246.)

`\c_pi_fp` We simply round π to and $\pi/180$ to 16 significant digits.

```

\c_one_degree_fp
24604 \fp_const:Nn \c_pi_fp { 3.141 5926 5358 9793 }
24605 \fp_const:Nn \c_one_degree_fp { 0.0 1745 3292 5199 4330 }

```

(End definition for `\c_pi_fp` and `\c_one_degree_fp`. These variables are documented on page 247.)

`\l_tmpa_fp` Scratch variables are simply initialized there.

```

\l_tmpb_fp
24606 \fp_new:N \l_tmpa_fp
\g_tmpa_fp
24607 \fp_new:N \l_tmpb_fp
\g_tmpb_fp
24608 \fp_new:N \g_tmpa_fp
24609 \fp_new:N \g_tmpb_fp

```

(End definition for `\l_tmpa_fp` and others. These variables are documented on page 247.)

```

24610 \endpackage

```

Chapter 71

l3fp-logic Implementation

```

24611 <*package>
24612 <@@=fp>

\__fp_parse_word_max:N Those functions may receive a variable number of arguments.
\__fp_parse_word_min:N
24613 \cs_new:Npn \__fp_parse_word_max:N
24614 { \__fp_parse_function:NNN \__fp_minmax_o:Nw 2 }
24615 \cs_new:Npn \__fp_parse_word_min:N
24616 { \__fp_parse_function:NNN \__fp_minmax_o:Nw 0 }

(End definition for \__fp_parse_word_max:N and \__fp_parse_word_min:N.)

```

71.1 Syntax of internal functions

- `__fp_compare_npos:nwnw {<exp1>} <body1> ; {<exp2>} <body2> ;`
- `__fp_minmax_o:Nw <sign> <floating point array>`
- `__fp_not_o:w ? <floating point array>` (with one floating point number only)
- `__fp_&_o:ww <floating point> <floating point>`
- `__fp_|_o:ww <floating point> <floating point>`
- `__fp_ternary:NwwN, __fp_ternary_auxi:NwwN, __fp_ternary_auxii:NwwN` have to be understood.

71.2 Tests

```

\fp_if_exist_p:N Copies of the cs functions defined in l3basics.
\fp_if_exist_p:c
\fp_if_exist:NTF 24617 \prg_new_eq_conditional:NNn \fp_if_exist:N \cs_if_exist:N { TF , T , F , p }
\fp_if_exist:cTF 24618 \prg_new_eq_conditional:NNn \fp_if_exist:c \cs_if_exist:c { TF , T , F , p }
\fp_if_exist:cTF
(End definition for \fp_if_exist:NTF. This function is documented on page 242.)

```

\fp_if_nan_p:n Evaluate and check if the result is a floating point of the same kind as nan.
\fp_if_nan:nTF

```

24619 \prg_new_conditional:Npnn \fp_if_nan:n #1 { TF , T , F , p }
24620 {
24621   \if:w 3 \exp_last_unbraced:Nf \__fp_kind:w { \__fp_parse:n {#1} }
24622   \prg_return_true:
24623   \else:
24624     \prg_return_false:
24625   \fi:
24626 }

```

(End definition for \fp_if_nan:nTF. This function is documented on page 301.)

71.3 Comparison

\fp_compare_p:n Within floating point expressions, comparison operators are treated as operations, so we
\fp_compare:nTF evaluate #1, then compare with ± 0 . Tuples are true.

```

\__fp_compare_return:w 24627 \prg_new_conditional:Npnn \fp_compare:n #1 { p , T , F , TF }
24628 {
24629   \exp_after:wN \__fp_compare_return:w
24630   \exp:w \exp_end_continue_f:w \__fp_parse:n {#1}
24631 }
24632 \cs_new:Npn \__fp_compare_return:w #1#2#3;
24633 {
24634   \if_charcode:w 0
24635     \__fp_if_type_fp:NTwFw
24636     #1 { \__fp_use_i_delimit_by_s_stop:nw #3 \s__fp_stop }
24637     \s__fp 1 \s__fp_stop
24638     \prg_return_false:
24639   \else:
24640     \prg_return_true:
24641   \fi:
24642 }

```

(End definition for \fp_compare:nTF and __fp_compare_return:w. This function is documented on page 244.)

\fp_compare_p:nNn Evaluate #1 and #3, using an auxiliary to expand both, and feed the two floating point
\fp_compare:nNnTF numbers swapped to __fp_compare_back_any:ww, defined below. Compare the result
 __fp_compare_aux:wn with ‘#2-’, which is -1 for $<$, 0 for $=$, 1 for $>$ and 2 for $?$.

```

24643 \prg_new_conditional:Npnn \fp_compare:nNn #1#2#3 { p , T , F , TF }
24644 {
24645   \if_int_compare:w
24646     \exp_after:wN \__fp_compare_aux:wn
24647     \exp:w \exp_end_continue_f:w \__fp_parse:n {#1} {#3}
24648     = \__fp_int_eval:w ‘#2 - ‘= \__fp_int_eval_end:
24649     \prg_return_true:
24650   \else:
24651     \prg_return_false:
24652   \fi:
24653 }
24654 \cs_new:Npn \__fp_compare_aux:wn #1; #2
24655 {
24656   \exp_after:wN \__fp_compare_back_any:ww

```

```

24657     \exp:w \exp_end_continue_f:w \__fp_parse:n {#2} #1;
24658 }

```

(End definition for \fp_compare:nNnTF and __fp_compare_aux:wn. This function is documented on page 243.)

```

\__fp_compare_back_any:ww \__fp_compare_back_any:ww <y> ; <x> ;
\__fp_compare_back:ww
\__fp_compare_nan:w

```

Expands (in the same way as \int_eval:n) to -1 if $x < y$, 0 if $x = y$, 1 if $x > y$, and 2 otherwise (denoted as $x?y$). If either operand is `nan`, stop the comparison with __fp_compare_nan:w returning 2 . If x is negative, swap the outputs 1 and -1 (i.e., $>$ and $<$); we can henceforth assume that $x \geq 0$. If $y \geq 0$, and they have the same type, either they are normal and we compare them with __fp_compare_npos:nwnw, or they are equal. If $y \geq 0$, but of a different type, the highest type is a larger number. Finally, if $y \leq 0$, then $x > y$, unless both are zero.

```

24659 \cs_new:Npn \__fp_compare_back_any:ww #1#2; #3
24660 {
24661     \__fp_if_type_fp:NTwFw
24662     #1 { \__fp_if_type_fp:NTwFw #3 \use_i:nn \s__fp \use_ii:nn \s__fp_stop }
24663     \s__fp \use_ii:nn \s__fp_stop
24664     \__fp_compare_back:ww
24665     {
24666         \cs:w
24667         __fp
24668         \__fp_type_from_scan:N #1
24669         _compare_back
24670         \__fp_type_from_scan:N #3
24671         :ww
24672         \cs_end:
24673     }
24674     #1#2 ; #3
24675 }
24676 \cs_new:Npn \__fp_compare_back:ww
24677 \s__fp \__fp_chk:w #1 #2 #3;
24678 \s__fp \__fp_chk:w #4 #5 #6;
24679 {
24680     \int_value:w
24681     \if_meaning:w 3 #1 \exp_after:wN \__fp_compare_nan:w \fi:
24682     \if_meaning:w 3 #4 \exp_after:wN \__fp_compare_nan:w \fi:
24683     \if_meaning:w 2 #5 - \fi:
24684     \if_meaning:w #2 #5
24685     \if_meaning:w #1 #4
24686     \if_meaning:w 1 #1
24687     \__fp_compare_npos:nwnw #6; #3;
24688     \else:
24689         0
24690     \fi:
24691     \else:
24692     \if_int_compare:w #4 < #1 - \fi: 1
24693     \fi:
24694     \else:
24695     \if_int_compare:w #1#4 = \c_zero_int
24696         0
24697     \else:
24698         1

```

```

24699         \fi:
24700     \fi:
24701     \exp_stop_f:
24702 }
24703 \cs_new:Npn \__fp_compare_nan:w #1 \fi: \exp_stop_f: { 2 \exp_stop_f: }

(End definition for \__fp_compare_back_any:ww, \__fp_compare_back:ww, and \__fp_compare_nan:w.)

```

__fp_compare_back_tuple:ww Tuple and floating point numbers are not comparable so return 2 in mixed cases or
 __fp_tuple_compare_back:ww when tuples have a different number of items. Otherwise compare pairs of items with
 __fp_tuple_compare_back_tuple:ww __fp_compare_back_any:ww and if any don't match return 2 (as \int_value:w 02
 __fp_tuple_compare_back_loop:w \exp_stop_f:).

```

24704 \cs_new:Npn \__fp_compare_back_tuple:ww #1; #2; { 2 }
24705 \cs_new:Npn \__fp_tuple_compare_back:ww #1; #2; { 2 }
24706 \cs_new:Npn \__fp_tuple_compare_back_tuple:ww
24707   \s__fp_tuple \__fp_tuple_chk:w #1;
24708   \s__fp_tuple \__fp_tuple_chk:w #2;
24709   {
24710     \int_compare:nNnTF { \__fp_array_count:n {#1} } =
24711       { \__fp_array_count:n {#2} }
24712       {
24713         \int_value:w 0
24714         \__fp_tuple_compare_back_loop:w
24715           #1 { \s__fp \prg_break: } ; @
24716           #2 { \s__fp \prg_break: } ;
24717         \prg_break_point:
24718         \exp_stop_f:
24719       }
24720       { 2 }
24721   }
24722 \cs_new:Npn \__fp_tuple_compare_back_loop:w #1#2 ; #3 @ #4#5 ;
24723   {
24724     \use_none:n #1
24725     \use_none:n #4
24726     \if_int_compare:w
24727       \__fp_compare_back_any:ww #1 #2 ; #4 #5 ; = \c_zero_int
24728     \else:
24729       2 \exp_after:wN \prg_break:
24730     \fi:
24731     \__fp_tuple_compare_back_loop:w #3 @
24732   }

```

(End definition for __fp_compare_back_tuple:ww and others.)

__fp_compare_npos:nwnw __fp_compare_npos:nwnw {<exp₀₁>} <body₁> ; {<exp₀₂>} <body₂> ;
 __fp_compare_significand:nnnnnnnn Within an \int_value:w ... \exp_stop_f: construction, this expands to 0 if the

two numbers are equal, -1 if the first is smaller, and 1 if the first is bigger. First compare the exponents: the larger one denotes the larger number. If they are equal, we must compare significands. If both the first 8 digits and the next 8 digits coincide, the numbers are equal. If only the first 8 digits coincide, the next 8 decide. Otherwise, the first 8 digits are compared.

```

24733 \cs_new:Npn \__fp_compare_npos:nwnw #1#2; #3#4;
24734   {
24735     \if_int_compare:w #1 = #3 \exp_stop_f:

```

```

24736     \__fp_compare_significand:nnnnnnnn #2 #4
24737 \else:
24738     \if_int_compare:w #1 < #3 - \fi: 1
24739 \fi:
24740 }
24741 \cs_new:Npn \__fp_compare_significand:nnnnnnnn #1#2#3#4#5#6#7#8
24742 {
24743     \if_int_compare:w #1#2 = #5#6 \exp_stop_f:
24744     \if_int_compare:w #3#4 = #7#8 \exp_stop_f:
24745     0
24746 \else:
24747     \if_int_compare:w #3#4 < #7#8 - \fi: 1
24748 \fi:
24749 \else:
24750     \if_int_compare:w #1#2 < #5#6 - \fi: 1
24751 \fi:
24752 }

```

(End definition for __fp_compare_npos:nwnw and __fp_compare_significand:nnnnnnnn.)

71.4 Floating point expression loops

\fp_do_until:nn These are quite easy given the above functions. The `do_until` and `do_while` versions execute the body, then test. The `until_do` and `while_do` do it the other way round.

```

\fp_do_while:nn
\fp_until_do:nn
\fp_while_do:nn
24753 \cs_new:Npn \fp_do_until:nn #1#2
24754 {
24755     #2
24756     \fp_compare:nF {#1}
24757     { \fp_do_until:nn {#1} {#2} }
24758 }
24759 \cs_new:Npn \fp_do_while:nn #1#2
24760 {
24761     #2
24762     \fp_compare:nT {#1}
24763     { \fp_do_while:nn {#1} {#2} }
24764 }
24765 \cs_new:Npn \fp_until_do:nn #1#2
24766 {
24767     \fp_compare:nF {#1}
24768     {
24769         #2
24770         \fp_until_do:nn {#1} {#2}
24771     }
24772 }
24773 \cs_new:Npn \fp_while_do:nn #1#2
24774 {
24775     \fp_compare:nT {#1}
24776     {
24777         #2
24778         \fp_while_do:nn {#1} {#2}
24779     }
24780 }

```

(End definition for \fp_do_until:nn and others. These functions are documented on page 245.)

`\fp_do_until:nNnn` As above but not using the `nNn` syntax.

```

\fp_do_while:nNnn
\fp_until_do:nNnn
\fp_while_do:nNnn
24781 \cs_new:Npn \fp_do_until:nNnn #1#2#3#4
24782 {
24783     #4
24784     \fp_compare:nNnF {#1} #2 {#3}
24785     { \fp_do_until:nNnn {#1} #2 {#3} {#4} }
24786 }
24787 \cs_new:Npn \fp_do_while:nNnn #1#2#3#4
24788 {
24789     #4
24790     \fp_compare:nNnT {#1} #2 {#3}
24791     { \fp_do_while:nNnn {#1} #2 {#3} {#4} }
24792 }
24793 \cs_new:Npn \fp_until_do:nNnn #1#2#3#4
24794 {
24795     \fp_compare:nNnF {#1} #2 {#3}
24796     {
24797         #4
24798         \fp_until_do:nNnn {#1} #2 {#3} {#4}
24799     }
24800 }
24801 \cs_new:Npn \fp_while_do:nNnn #1#2#3#4
24802 {
24803     \fp_compare:nNnT {#1} #2 {#3}
24804     {
24805         #4
24806         \fp_while_do:nNnn {#1} #2 {#3} {#4}
24807     }
24808 }

```

(End definition for `\fp_do_until:nNnn` and others. These functions are documented on page 244.)

`\fp_step_function:nnnN` The approach here is somewhat similar to `\int_step_function:nnnN`. There are two subtleties: we use the internal parser `__fp_parse:n` to avoid converting back and forth from the internal representation; and (due to rounding) even a non-zero step does not guarantee that the loop counter increases.

```

\fp_step_function:nnnc
  \__fp_step:wwwN
  \__fp_step_fp:wwwN
  \__fp_step:NnnnnN
  \__fp_step:NfnnnN
24809 \cs_new:Npn \fp_step_function:nnnN #1#2#3
24810 {
24811     \exp_after:wN \__fp_step:wwwN
24812     \exp:w \exp_end_continue_f:w \__fp_parse_o:n {#1}
24813     \exp:w \exp_end_continue_f:w \__fp_parse_o:n {#2}
24814     \exp:w \exp_end_continue_f:w \__fp_parse:n {#3}
24815 }
24816 \cs_generate_variant:Nn \fp_step_function:nnnN { nnc }
24817 % \end{macrocode}
24818 % Only floating point numbers (not tuples) are allowed arguments.
24819 % Only \enquote{normal} floating points (not $\pm 0$,
24820 % $\pm\texttt{inf}$, $\texttt{nan}$) can be used as step; if positive,
24821 % call \cs{__fp_step:NnnnnN} with argument |>| otherwise~|<|. This
24822 % function has one more argument than its integer counterpart, namely
24823 % the previous value, to catch the case where the loop has made no
24824 % progress. Conversion to decimal is done just before calling the
24825 % user's function.
24826 % \begin{macrocode}

```

```

24827 \cs_new:Npn \__fp_step:wwwN #1#2; #3#4; #5#6; #7
24828 {
24829   \__fp_if_type_fp:NTwFw #1 { } \s__fp \prg_break: \s__fp_stop
24830   \__fp_if_type_fp:NTwFw #3 { } \s__fp \prg_break: \s__fp_stop
24831   \__fp_if_type_fp:NTwFw #5 { } \s__fp \prg_break: \s__fp_stop
24832   \use_i:nnnn { \__fp_step_fp:wwwN #1#2; #3#4; #5#6; #7 }
24833   \prg_break_point:
24834   \use:n
24835     {
24836       \__fp_error:nfff { step-tuple } { \fp_to_tl:n { #1#2 ; } }
24837       { \fp_to_tl:n { #3#4 ; } } { \fp_to_tl:n { #5#6 ; } }
24838     }
24839 }
24840 \cs_new:Npn \__fp_step_fp:wwwN #1 ; \s__fp \__fp_chk:w #2#3#4 ; #5; #6
24841 {
24842   \token_if_eq_meaning:NNTF #2 1
24843   {
24844     \token_if_eq_meaning:NNTF #3 0
24845     { \__fp_step:NnnnnN > }
24846     { \__fp_step:NnnnnN < }
24847   }
24848   {
24849     \token_if_eq_meaning:NNTF #2 0
24850     {
24851       \msg_expandable_error:nnn { kernel }
24852       { zero-step } {#6}
24853     }
24854     {
24855       \__fp_error:nnfn { bad-step } { }
24856       { \fp_to_tl:n { \s__fp \__fp_chk:w #2#3#4 ; } } {#6}
24857     }
24858     \use_none:nnnnn
24859   }
24860   { #1 ; } { \c_nan_fp } { \s__fp \__fp_chk:w #2#3#4 ; } { #5 ; } #6
24861 }
24862 \cs_new:Npn \__fp_step:NnnnnN #1#2#3#4#5#6
24863 {
24864   \fp_compare:nNnTF {#2} = {#3}
24865   {
24866     \__fp_error:nffn { tiny-step }
24867     { \fp_to_tl:n {#3} } { \fp_to_tl:n {#4} } {#6}
24868   }
24869   {
24870     \fp_compare:nNnF {#2} #1 {#5}
24871     {
24872       \exp_args:Nf #6 { \__fp_to_decimal_dispatch:w #2 }
24873       \__fp_step:NfnnnnN
24874       #1 { \__fp_parse:n { #2 + #4 } } {#2} {#4} {#5} #6
24875     }
24876   }
24877 }
24878 \cs_generate_variant:Nn \__fp_step:NnnnnN { Nf }

```

(End definition for \fp_step_function:nnnN and others. This function is documented on page 246.)

```

\fp_step_inline:nnnn
\fp_step_variable:nnnNn
  \__fp_step:NNnnnn

```

As for `\int_step_inline:nnnn`, create a global function and apply it, following up with a break point.

```

24879 \cs_new_protected:Npn \fp_step_inline:nnnn
24880 {
24881   \int_gincr:N \g__kernel_prg_map_int
24882   \exp_args:NNc \__fp_step:NNnnnn
24883   \cs_gset_protected:Npn
24884     { \__fp_map_ \int_use:N \g__kernel_prg_map_int :w }
24885 }
24886 \cs_new_protected:Npn \fp_step_variable:nnnNn #1#2#3#4#5
24887 {
24888   \int_gincr:N \g__kernel_prg_map_int
24889   \exp_args:NNc \__fp_step:NNnnnn
24890   \cs_gset_protected:Npx
24891     { \__fp_map_ \int_use:N \g__kernel_prg_map_int :w }
24892     {#1} {#2} {#3}
24893     {
24894       \tl_set:Nn \exp_not:N #4 {##1}
24895       \exp_not:n {#5}
24896     }
24897 }
24898 \cs_new_protected:Npn \__fp_step:NNnnnn #1#2#3#4#5#6
24899 {
24900   #1 #2 ##1 {#6}
24901   \fp_step_function:nnnN {#3} {#4} {#5} #2
24902   \prg_break_point:Nn \scan_stop: { \int_gdecr:N \g__kernel_prg_map_int }
24903 }

```

(End definition for `\fp_step_inline:nnnn`, `\fp_step_variable:nnnNn`, and `__fp_step:NNnnnn`. These functions are documented on page 246.)

```

24904 \msg_new:nnn { fp } { step-tuple }
24905   { Tuple~argument~in~fp_step~...~{#1}{#2}{#3}. }
24906 \msg_new:nnn { fp } { bad-step }
24907   { Invalid~step~size~#2~for~function~#3. }
24908 \msg_new:nnn { fp } { tiny-step }
24909   { Tiny~step~size~(#{1}+#{2}=#{1})~for~function~#3. }

```

71.5 Extrema

```

\__fp_minmax_o:Nw
\__fp_minmax_aux_o:Nw

```

First check all operands are floating point numbers. The argument `#1` is 2 to find the maximum of an array `#2` of floating point numbers, and 0 to find the minimum. We read numbers sequentially, keeping track of the largest (smallest) number found so far. If numbers are equal (for instance ± 0), the first is kept. We append $-\infty$ (∞), for the case of an empty array. Since no number is smaller (larger) than that, this additional item only affects the maximum (minimum) in the case of `max()` and `min()` with no argument. The weird fp-like trailing marker breaks the loop correctly: see the precise definition of `__fp_minmax_loop:Nww`.

```

24910 \cs_new:Npn \__fp_minmax_o:Nw #1
24911 {
24912   \__fp_parse_function_all_fp_o:fnw
24913   { \token_if_eq_meaning:NTTF 0 #1 { min } { max } }
24914   { \__fp_minmax_aux_o:Nw #1 }

```

```

24915 }
24916 \cs_new:Npn \__fp_minmax_aux_o:Nw #1#2 @
24917 {
24918   \if_meaning:w 0 #1
24919     \exp_after:wN \__fp_minmax_loop:Nww \exp_after:wN +
24920   \else:
24921     \exp_after:wN \__fp_minmax_loop:Nww \exp_after:wN -
24922   \fi:
24923   #2
24924   \s__fp \__fp_chk:w 2 #1 \s__fp_exact ;
24925   \s__fp \__fp_chk:w { 3 \__fp_minmax_break_o:w } ;
24926 }

```

(End definition for __fp_minmax_o:Nw and __fp_minmax_aux_o:Nw.)

__fp_minmax_loop:Nww The first argument is $-$ or $+$ to denote the case where the currently largest (smallest) number found (first floating point argument) should be replaced by the new number (second floating point argument). If the new number is `nan`, keep that as the extremum, unless that extremum is already a `nan`. Otherwise, compare the two numbers. If the new number is larger (in the case of `max`) or smaller (in the case of `min`), the test yields `true`, and we keep the second number as a new maximum; otherwise we keep the first number. Then loop.

```

24927 \cs_new:Npn \__fp_minmax_loop:Nww
24928   #1 \s__fp \__fp_chk:w #2#3; \s__fp \__fp_chk:w #4#5;
24929 {
24930   \if_meaning:w 3 #4
24931     \if_meaning:w 3 #2
24932       \__fp_minmax_auxi:ww
24933     \else:
24934       \__fp_minmax_auxii:ww
24935     \fi:
24936   \else:
24937     \if_int_compare:w
24938       \__fp_compare_back:ww
24939       \s__fp \__fp_chk:w #4#5;
24940       \s__fp \__fp_chk:w #2#3;
24941       = #1 1 \exp_stop_f:
24942     \__fp_minmax_auxii:ww
24943   \else:
24944     \__fp_minmax_auxi:ww
24945   \fi:
24946 \fi:
24947 \__fp_minmax_loop:Nww #1
24948   \s__fp \__fp_chk:w #2#3;
24949   \s__fp \__fp_chk:w #4#5;
24950 }

```

(End definition for __fp_minmax_loop:Nww.)

__fp_minmax_auxi:ww Keep the first/second number, and remove the other.
 __fp_minmax_auxii:ww

```

24951 \cs_new:Npn \__fp_minmax_auxi:ww #1 \fi: \fi: #2 \s__fp #3 ; \s__fp #4;
24952 { \fi: \fi: #2 \s__fp #3 ; }
24953 \cs_new:Npn \__fp_minmax_auxii:ww #1 \fi: \fi: #2 \s__fp #3 ;
24954 { \fi: \fi: #2 }

```

(End definition for _fp_minmax_auxi:ww and _fp_minmax_auxii:ww.)

_fp_minmax_break_o:w This function is called from within an \if_meaning:w test. Skip to the end of the tests, close the current test with \fi:, clean up, and return the appropriate number with one post-expansion.

```
24955 \cs_new:Npn \_fp_minmax_break_o:w #1 \fi: \fi: #2 \s__fp #3; #4;
24956 { \fi: \_fp_exp_after_o:w \s__fp #3; }
```

(End definition for _fp_minmax_break_o:w.)

71.6 Boolean operations

_fp_not_o:w Return true or false, with two expansions, one to exit the conditional, and one to please l3fp-parse. The first argument is provided by l3fp-parse and is ignored.

```
24957 \cs_new:Npn \_fp_not_o:w #1 \s__fp \_fp_chk:w #2#3; @
24958 {
24959   \if_meaning:w 0 #2
24960   \exp_after:wN \exp_after:wN \exp_after:wN \c_one_fp
24961   \else:
24962   \exp_after:wN \exp_after:wN \exp_after:wN \c_zero_fp
24963   \fi:
24964 }
24965 \cs_new:Npn \_fp_tuple_not_o:w #1 @ { \exp_after:wN \c_zero_fp }
```

(End definition for _fp_not_o:w and _fp_tuple_not_o:w.)

fp&_o:ww For and, if the first number is zero, return it (with the same sign). Otherwise, return the second one. For or, the logic is reversed: if the first number is non-zero, return it, otherwise return the second number: we achieve that by hi-jacking _fp_&_o:ww, inserting an extra argument, \else:, before \s__fp. In all cases, expand after the floating point number.

```
24966 \group_begin:
24967 \char_set_catcode_letter:N &
24968 \char_set_catcode_letter:N |
24969 \cs_new:Npn \_fp_&_o:ww #1 \s__fp \_fp_chk:w #2#3;
24970 {
24971   \if_meaning:w 0 #2 #1
24972   \_fp_and_return:wNw \s__fp \_fp_chk:w #2#3;
24973   \fi:
24974   \_fp_exp_after_o:w
24975 }
24976 \cs_new:Npn \_fp_&_tuple_o:ww #1 \s__fp \_fp_chk:w #2#3;
24977 {
24978   \if_meaning:w 0 #2 #1
24979   \_fp_and_return:wNw \s__fp \_fp_chk:w #2#3;
24980   \fi:
24981   \_fp_exp_after_tuple_o:w
24982 }
24983 \cs_new:Npn \_fp_tuple_&_o:ww #1; { \_fp_exp_after_o:w }
24984 \cs_new:Npn \_fp_tuple_&_tuple_o:ww #1; { \_fp_exp_after_tuple_o:w }
24985 \cs_new:Npn \_fp_|_o:ww { \_fp_&_o:ww \else: }
24986 \cs_new:Npn \_fp_|_tuple_o:ww { \_fp_&_tuple_o:ww \else: }
24987 \cs_new:Npn \_fp_tuple_|_o:ww #1; #2; { \_fp_exp_after_tuple_o:w #1; }
```

```

24988 \cs_new:Npn \__fp_tuple_|_tuple_o:ww #1; #2;
24989 { \__fp_exp_after_tuple_o:w #1; }
24990 \group_end:
24991 \cs_new:Npn \__fp_and_return:wNw #1; \fi: #2;
24992 { \fi: \__fp_exp_after_o:w #1; }

```

(End definition for __fp_&_o:ww and others.)

71.7 Ternary operator

__fp_ternary:NwwN The first function receives the test and the true branch of the ?: ternary operator. It calls __fp_ternary_auxii:NwwN if the test branch is a floating point number ± 0 , and otherwise calls __fp_ternary_auxi:NwwN. These functions select one of their two arguments.

```

24993 \cs_new:Npn \__fp_ternary:NwwN #1 #2#3@ #4@ #5
24994 {
24995   \if_meaning:w \__fp_parse_infix_: :N #5
24996   \if_charcode:w 0
24997     \__fp_if_type_fp:NTwFw
24998     #2 { \use_i:nn \__fp_use_i_delimit_by_s_stop:nw #3 \s__fp_stop }
24999     \s__fp 1 \s__fp_stop
25000     \exp_after:wN \exp_after:wN \exp_after:wN \__fp_ternary_auxii:NwwN
25001   \else:
25002     \exp_after:wN \exp_after:wN \exp_after:wN \__fp_ternary_auxi:NwwN
25003   \fi:
25004   \exp_after:wN #1
25005   \exp:w \exp_end_continue_f:w
25006   \__fp_exp_after_array_f:w #4 \s__fp_expr_stop
25007   \exp_after:wN @
25008   \exp:w
25009   \__fp_parse_operand:Nw \c__fp_prec_colon_int
25010   \__fp_parse_expand:w
25011 \else:
25012   \msg_expandable_error:nnnn
25013   { fp } { missing } { : } { ~for~?: }
25014   \exp_after:wN \__fp_parse_continue:NwN
25015   \exp_after:wN #1
25016   \exp:w \exp_end_continue_f:w
25017   \__fp_exp_after_array_f:w #4 \s__fp_expr_stop
25018   \exp_after:wN #5
25019   \exp_after:wN #1
25020 \fi:
25021 }
25022 \cs_new:Npn \__fp_ternary_auxi:NwwN #1#2@#3@#4
25023 {
25024   \exp_after:wN \__fp_parse_continue:NwN
25025   \exp_after:wN #1
25026   \exp:w \exp_end_continue_f:w
25027   \__fp_exp_after_array_f:w #2 \s__fp_expr_stop
25028   #4 #1
25029 }
25030 \cs_new:Npn \__fp_ternary_auxii:NwwN #1#2@#3@#4
25031 {

```

```

25032     \exp_after:wN \_fp_parse_continue:NwN
25033     \exp_after:wN #1
25034     \exp:w \exp_end_continue_f:w
25035     \_fp_exp_after_array_f:w #3 \s_fp_expr_stop
25036     #4 #1
25037 }

```

(End definition for _fp_ternary:NwwN, _fp_ternary_auxi:NwwN, and _fp_ternary_auxii:NwwN.)

```

25038 \</package>

```

Chapter 72

l3fp-basics Implementation

```
25039 ⟨*package⟩
25040 ⟨@@=fp⟩
```

The `l3fp-basics` module implements addition, subtraction, multiplication, and division of two floating points, and the absolute value and sign-changing operations on one floating point. All operations implemented in this module yield the outcome of rounding the infinitely precise result of the operation to the nearest floating point.

Some algorithms used below end up being quite similar to some described in “What Every Computer Scientist Should Know About Floating Point Arithmetic”, by David Goldberg, which can be found at <http://cr.yp.to/2005-590/goldberg.pdf>.

Unary functions.

```

__fp_parse_word_abs:N
__fp_parse_word_logb:N
__fp_parse_word_sign:N
__fp_parse_word_sqrt:N
25041 \cs_new:Npn __fp_parse_word_abs:N
25042   { __fp_parse_unary_function:NNN __fp_set_sign_o:w 0 }
25043 \cs_new:Npn __fp_parse_word_logb:N
25044   { __fp_parse_unary_function:NNN __fp_logb_o:w ? }
25045 \cs_new:Npn __fp_parse_word_sign:N
25046   { __fp_parse_unary_function:NNN __fp_sign_o:w ? }
25047 \cs_new:Npn __fp_parse_word_sqrt:N
25048   { __fp_parse_unary_function:NNN __fp_sqrt_o:w ? }
```

(End definition for `__fp_parse_word_abs:N` and others.)

72.1 Addition and subtraction

We define here two functions, `__fp_-_o:ww` and `__fp+_o:ww`, which perform the subtraction and addition of their two floating point operands, and expand the tokens following the result once.

A more obscure function, `__fp_add_big_i_o:wNww`, is used in `l3fp-expo`.

The logic goes as follows:

- `__fp_-_o:ww` calls `__fp+_o:ww` to do the work, with the sign of the second operand flipped;
- `__fp+_o:ww` dispatches depending on the type of floating point, calling specialized auxiliaries;

- in all cases except summing two normal floating point numbers, we return one or the other operands depending on the signs, or detect an invalid operation in the case of $\infty - \infty$;
- for normal floating point numbers, compare the signs;
- to add two floating point numbers of the same sign or of opposite signs, shift the significand of the smaller one to match the bigger one, perform the addition or subtraction of significands, check for a carry, round, and pack using the `__fp-basics_pack...` functions.

The trickiest part is to round correctly when adding or subtracting normal floating point numbers.

72.1.1 Sign, exponent, and special numbers

`__fp_-_o:ww` The `__fp+_o:ww` auxiliary has a hook: it takes one argument between the first `\s__-fp` and `__fp_chk:w`, which is applied to the sign of the second operand. Positioning the hook there means that `__fp+_o:ww` can still perform the sanity check that it was followed by `\s__fp`.

```

25049 \cs_new:cpx { __fp_-_o:ww } \s__fp
25050 {
25051   \exp_not:c { __fp+_o:ww }
25052   \exp_not:n { \s__fp \__fp_neg_sign:N }
25053 }

```

(End definition for `__fp_-_o:ww`.)

`__fp+_o:ww` This function is either called directly with an empty #1 to compute an addition, or it is called by `__fp_-_o:ww` with `__fp_neg_sign:N` as #1 to compute a subtraction, in which case the second operand's sign should be changed. If the *<types>* #2 and #4 are the same, dispatch to case #2 (0, 1, 2, or 3), where we call specialized functions: thanks to `\int_value:w`, those receive the tweaked *<sign₂>* (expansion of #1#5) as an argument. If the *<types>* are distinct, the result is simply the floating point number with the highest *<type>*. Since case 3 (used for two nan) also picks the first operand, we can also use it when *<type₁>* is greater than *<type₂>*. Also note that we don't need to worry about *<sign₂>* in that case since the second operand is discarded.

```

25054 \cs_new:cpn { __fp+_o:ww }
25055   \s__fp #1 \__fp_chk:w #2 #3 ; \s__fp \__fp_chk:w #4 #5
25056 {
25057   \if_case:w
25058     \if_meaning:w #2 #4
25059     #2
25060   \else:
25061     \if_int_compare:w #2 > #4 \exp_stop_f:
25062     3
25063   \else:
25064     4
25065   \fi:
25066   \fi:
25067   \exp_stop_f:
25068     \exp_after:wN \__fp_add_zeros_o:Nww \int_value:w
25069   \or:   \exp_after:wN \__fp_add_normal_o:Nww \int_value:w

```

```

25070 \or: \exp_after:wN \__fp_add_inf_o:Nww \int_value:w
25071 \or: \__fp_case_return_i_o:ww
25072 \else: \exp_after:wN \__fp_add_return_ii_o:Nww \int_value:w
25073 \fi:
25074 #1 #5
25075 \s__fp \__fp_chk:w #2 #3 ;
25076 \s__fp \__fp_chk:w #4 #5
25077 }

```

(End definition for __fp_+_o:ww.)

__fp_add_return_ii_o:Nww Ignore the first operand, and return the second, but using the sign #1 rather than #4. As usual, expand after the floating point.

```

25078 \cs_new:Npn \__fp_add_return_ii_o:Nww #1 #2 ; \s__fp \__fp_chk:w #3 #4
25079 { \__fp_exp_after_o:w \s__fp \__fp_chk:w #3 #1 }

```

(End definition for __fp_add_return_ii_o:Nww.)

__fp_add_zeros_o:Nww Adding two zeros yields \c_zero_fp, except if both zeros were -0 .

```

25080 \cs_new:Npn \__fp_add_zeros_o:Nww #1 \s__fp \__fp_chk:w 0 #2
25081 {
25082   \if_int_compare:w #2 #1 = 20 \exp_stop_f:
25083   \exp_after:wN \__fp_add_return_ii_o:Nww
25084   \else:
25085     \__fp_case_return_i_o:ww
25086   \fi:
25087   #1
25088   \s__fp \__fp_chk:w 0 #2
25089 }

```

(End definition for __fp_add_zeros_o:Nww.)

__fp_add_inf_o:Nww If both infinities have the same sign, just return that infinity, otherwise, it is an invalid operation. We find out if that invalid operation is an addition or a subtraction by testing whether the tweaked $\langle sign_2 \rangle$ (#1) and the $\langle sign_2 \rangle$ (#4) are identical.

```

25090 \cs_new:Npn \__fp_add_inf_o:Nww
25091   #1 \s__fp \__fp_chk:w 2 #2 #3; \s__fp \__fp_chk:w 2 #4
25092 {
25093   \if_meaning:w #1 #2
25094     \__fp_case_return_i_o:ww
25095   \else:
25096     \__fp_case_use:nw
25097     {
25098       \exp_last_unbraced:Nf \__fp_invalid_operation_o:Nww
25099       { \token_if_eq_meaning:NNTF #1 #4 + - }
25100     }
25101   \fi:
25102   \s__fp \__fp_chk:w 2 #2 #3;
25103   \s__fp \__fp_chk:w 2 #4
25104 }

```

(End definition for __fp_add_inf_o:Nww.)

```

\__fp_add_normal_o:Nww \__fp_add_normal_o:Nww <sign2> \s__fp \__fp_chk:w 1 <sign1> <exp1>
<body1> ; \s__fp \__fp_chk:w 1 <initial sign2> <exp2> <body2> ;

```

We now have two normal numbers to add, and we have to check signs and exponents more carefully before performing the addition.

```

25105 \cs_new:Npn \__fp_add_normal_o:Nww #1 \s__fp \__fp_chk:w 1 #2
25106 {
25107   \if_meaning:w #1#2
25108     \exp_after:wN \__fp_add_npos_o:NnwNnw
25109   \else:
25110     \exp_after:wN \__fp_sub_npos_o:NnwNnw
25111   \fi:
25112   #2
25113 }

```

(End definition for __fp_add_normal_o:Nww.)

72.1.2 Absolute addition

In this subsection, we perform the addition of two positive normal numbers.

```

\__fp_add_npos_o:NnwNnw \__fp_add_npos_o:NnwNnw <sign1> <exp1> <body1> ; \s__fp \__fp_chk:w 1
<initial sign2> <exp2> <body2> ;

```

Since we are doing an addition, the final sign is $\langle sign_1 \rangle$. Start an __fp_int_eval:w, responsible for computing the exponent: the result, and the $\langle final\ sign \rangle$ are then given to __fp_sanitize:Nw which checks for overflow. The exponent is computed as the largest exponent #2 or #5, incremented if there is a carry. To add the significands, we decimate the smaller number by the difference between the exponents. This is done by __fp_add_big_i:wNww or __fp_add_big_ii:wNww. We need to bring the final sign with us in the midst of the calculation to round properly at the end.

```

25114 \cs_new:Npn \__fp_add_npos_o:NnwNnw #1#2#3 ; \s__fp \__fp_chk:w 1 #4 #5
25115 {
25116   \exp_after:wN \__fp_sanitize:Nw
25117   \exp_after:wN #1
25118   \int_value:w \__fp_int_eval:w
25119   \if_int_compare:w #2 > #5 \exp_stop_f:
25120     #2
25121   \exp_after:wN \__fp_add_big_i_o:wNww \int_value:w -
25122   \else:
25123     #5
25124   \exp_after:wN \__fp_add_big_ii_o:wNww \int_value:w
25125   \fi:
25126   \__fp_int_eval:w #5 - #2 ; #1 #3;
25127 }

```

(End definition for __fp_add_npos_o:NnwNnw.)

```

\__fp_add_big_i_o:wNww \__fp_add_big_i_o:wNww <shift> ; <final sign> <body1> ; <body2> ;

```

__fp_add_big_ii_o:wNww Used in l3fp-expo. Shift the significand of the small number, then add with __fp_add_significand_o:NnnwnnnnN.

```

25128 \cs_new:Npn \__fp_add_big_i_o:wNww #1; #2 #3; #4;
25129 {
25130   \__fp_decimate:nNnnnn {#1}
25131   \__fp_add_significand_o:NnnwnnnnN

```

```

25132     #4
25133     #3
25134     #2
25135   }
25136 \cs_new:Npn \__fp_add_big_ii_o:wNww #1; #2 #3; #4;
25137 {
25138   \__fp_decimate:nNnnnn {#1}
25139   \__fp_add_significand_o:NnnwnnnnN
25140   #3
25141   #4
25142   #2
25143 }

```

(End definition for __fp_add_big_i_o:wNww and __fp_add_big_ii_o:wNww.)

```

\__fp_add_significand_o:NnnwnnnnN   \__fp_add_significand_o:NnnwnnnnN <rounding digit> {\langle Y_1 \rangle} {\langle Y_2 \rangle}
\__fp_add_significand_pack:NNNNNNN   <extra-digits> ; {\langle X_1 \rangle} {\langle X_2 \rangle} {\langle X_3 \rangle} {\langle X_4 \rangle} <final sign>
\__fp_add_significand_test_o:N

```

To round properly, we must know at which digit the rounding should occur. This requires to know whether the addition produces an overall carry or not. Thus, we do the computation now and check for a carry, then go back and do the rounding. The rounding may cause a carry in very rare cases such as $0.99 \dots 95 \rightarrow 1.00 \dots 0$, but this situation always give an exact power of 10, for which it is easy to correct the result at the end.

```

25144 \cs_new:Npn \__fp_add_significand_o:NnnwnnnnN #1 #2#3 #4; #5#6#7#8
25145 {
25146   \exp_after:wN \__fp_add_significand_test_o:N
25147   \int_value:w \__fp_int_eval:w 1#5#6 + #2
25148   \exp_after:wN \__fp_add_significand_pack:NNNNNNN
25149   \int_value:w \__fp_int_eval:w 1#7#8 + #3 ; #1
25150 }
25151 \cs_new:Npn \__fp_add_significand_pack:NNNNNNN #1 #2#3#4#5#6#7
25152 {
25153   \if_meaning:w 2 #1
25154     + 1
25155   \fi:
25156   ; #2 #3 #4 #5 #6 #7 ;
25157 }
25158 \cs_new:Npn \__fp_add_significand_test_o:N #1
25159 {
25160   \if_meaning:w 2 #1
25161     \exp_after:wN \__fp_add_significand_carry_o:wwwNN
25162   \else:
25163     \exp_after:wN \__fp_add_significand_no_carry_o:wwwNN
25164   \fi:
25165 }

```

(End definition for __fp_add_significand_o:NnnwnnnnN, __fp_add_significand_pack:NNNNNNN, and __fp_add_significand_test_o:N.)

```

\__fp_add_significand_no_carry_o:wwwNN   \__fp_add_significand_no_carry_o:wwwNN <8d> ; <6d> ; <2d> ; <rounding
digit> <sign>

```

If there's no carry, grab all the digits again and round. The packing function __fp-basics_pack_high:NNNNNw takes care of the case where rounding brings a carry.

```

25166 \cs_new:Npn \__fp_add_significand_no_carry_o:wwwNN
25167   #1; #2; #3#4 ; #5#6

```

```

25168 {
25169   \exp_after:wN \_fp_basics_pack_high:NNNNw
25170   \int_value:w \_fp_int_eval:w 1 #1
25171   \exp_after:wN \_fp_basics_pack_low:NNNNw
25172   \int_value:w \_fp_int_eval:w 1 #2 #3#4
25173   + \_fp_round:NNN #6 #4 #5
25174   \exp_after:wN ;
25175 }

```

(End definition for _fp_add_significand_no_carry_o:wwwNN.)

```

\_fp_add_significand_carry_o:wwwNN \_fp_add_significand_carry_o:wwwNN <8d> ; <6d> ; <2d> ; <rounding
digit> <sign>

```

The case where there is a carry is very similar. Rounding can even raise the first digit from 1 to 2, but we don't care.

```

25176 \cs_new:Npn \_fp_add_significand_carry_o:wwwNN
25177   #1; #2; #3#4; #5#6
25178 {
25179   + 1
25180   \exp_after:wN \_fp_basics_pack_weird_high:NNNNNNNw
25181   \int_value:w \_fp_int_eval:w 1 1 #1
25182   \exp_after:wN \_fp_basics_pack_weird_low:NNNNw
25183   \int_value:w \_fp_int_eval:w 1 #2#3 +
25184   \exp_after:wN \_fp_round:NNN
25185   \exp_after:wN #6
25186   \exp_after:wN #3
25187   \int_value:w \_fp_round_digit:Nw #4 #5 ;
25188   \exp_after:wN ;
25189 }

```

(End definition for _fp_add_significand_carry_o:wwwNN.)

72.1.3 Absolute subtraction

```

\_fp_sub_npos_o:NnwNnw \_fp_sub_npos_o:NnwNnw <sign1> <exp1> <body1> ; \s_fp \_fp_chk:w 1
\_fp_sub_eq_o:Nnwnw <initial sign2> <exp2> <body2> ;
\_fp_sub_npos_ii_o:Nnwnw

```

Rounding properly in some modes requires to know what the sign of the result will be. Thus, we start by comparing the exponents and significands. If the numbers coincide, return zero. If the second number is larger, swap the numbers and call _fp_sub_npos_i_o:Nnwnw with the opposite of $\langle sign_1 \rangle$.

```

25190 \cs_new:Npn \_fp_sub_npos_o:NnwNnw #1#2#3; \s_fp \_fp_chk:w 1 #4#5#6;
25191 {
25192   \if_case:w \_fp_compare_npos:nwnw {#2} #3; {#5} #6; \exp_stop_f:
25193   \exp_after:wN \_fp_sub_eq_o:Nnwnw
25194   \or:
25195   \exp_after:wN \_fp_sub_npos_i_o:Nnwnw
25196   \else:
25197   \exp_after:wN \_fp_sub_npos_ii_o:Nnwnw
25198   \fi:
25199   #1 {#2} #3; {#5} #6;
25200 }
25201 \cs_new:Npn \_fp_sub_eq_o:Nnwnw #1#2; #3; { \exp_after:wN \c_zero_fp }
25202 \cs_new:Npn \_fp_sub_npos_ii_o:Nnwnw #1 #2; #3;

```

```

25203 {
25204   \exp_after:wN \_fp_sub_npos_i_o:Nnwnw
25205   \int_value:w \_fp_neg_sign:N #1
25206   #3; #2;
25207 }

```

(End definition for _fp_sub_npos_o:Nnwnw, _fp_sub_eq_o:Nnwnw, and _fp_sub_npos_ii_o:Nnwnw.)

_fp_sub_npos_i_o:Nnwnw

After the computation is done, _fp_sanitize:Nw checks for overflow/underflow. It expects the $\langle final\ sign \rangle$ and the $\langle exponent \rangle$ (delimited by ;). Start an integer expression for the exponent, which starts with the exponent of the largest number, and may be decreased if the two numbers are very close. If the two numbers have the same exponent, call the **near** auxiliary. Otherwise, decimate y , then call the **far** auxiliary to evaluate the difference between the two significands. Note that we decimate by 1 less than one could expect.

```

25208 \cs_new:Npn \_fp_sub_npos_i_o:Nnwnw #1 #2#3; #4#5;
25209 {
25210   \exp_after:wN \_fp_sanitize:Nw
25211   \exp_after:wN #1
25212   \int_value:w \_fp_int_eval:w
25213   #2
25214   \if_int_compare:w #2 = #4 \exp_stop_f:
25215     \exp_after:wN \_fp_sub_back_near_o:nnnnnnnnN
25216   \else:
25217     \exp_after:wN \_fp_decimate:nNnnnn \exp_after:wN
25218     { \int_value:w \_fp_int_eval:w #2 - #4 - 1 \exp_after:wN }
25219     \exp_after:wN \_fp_sub_back_far_o:NnwnnnnnN
25220   \fi:
25221   #5
25222   #3
25223   #1
25224 }

```

(End definition for _fp_sub_npos_i_o:Nnwnw.)

_fp_sub_back_near_o:nnnnnnnnN
_fp_sub_back_near_pack:NNNNNNw
_fp_sub_back_near_after:wNNNNw

_fp_sub_back_near_o:nnnnnnnnN { $\langle Y_1 \rangle$ } { $\langle Y_2 \rangle$ } { $\langle Y_3 \rangle$ } { $\langle Y_4 \rangle$ } { $\langle X_1 \rangle$ }
{ $\langle X_2 \rangle$ } { $\langle X_3 \rangle$ } { $\langle X_4 \rangle$ } $\langle final\ sign \rangle$

In this case, the subtraction is exact, so we discard the $\langle final\ sign \rangle$ #9. The very large shifts of 10^9 and $1.1 \cdot 10^9$ are unnecessary here, but allow the auxiliaries to be reused later. Each integer expression produces a 10 digit result. If the resulting 16 digits start with a 0, then we need to shift the group, padding with trailing zeros.

```

25225 \cs_new:Npn \_fp_sub_back_near_o:nnnnnnnnN #1#2#3#4 #5#6#7#8 #9
25226 {
25227   \exp_after:wN \_fp_sub_back_near_after:wNNNNw
25228   \int_value:w \_fp_int_eval:w 10#5#6 - #1#2 - 11
25229   \exp_after:wN \_fp_sub_back_near_pack:NNNNNNw
25230   \int_value:w \_fp_int_eval:w 11#7#8 - #3#4 \exp_after:wN ;
25231 }
25232 \cs_new:Npn \_fp_sub_back_near_pack:NNNNNNw #1#2#3#4#5#6#7 ;
25233 { + #1#2 ; {#3#4#5#6} {#7} ; }
25234 \cs_new:Npn \_fp_sub_back_near_after:wNNNNw 10 #1#2#3#4 #5 ;
25235 {
25236   \if_meaning:w 0 #1
25237     \exp_after:wN \_fp_sub_back_shift:wnnnn

```

```

25238 \fi:
25239 ; {#1#2#3#4} {#5}
25240 }

```

(End definition for _fp_sub_back_near_o:nnnnnnnnN, _fp_sub_back_near_pack:NNNNNNw, and _fp_sub_back_near_after:wNNNNw.)

```

\_fp_sub_back_shift:wnnnn
\_fp_sub_back_shift_ii:ww
\_fp_sub_back_shift_iii:NNNNNNNNw
\_fp_sub_back_shift_iv:nnnnw

```

```

\_fp_sub_back_shift:wnnnn ; {⟨Z1⟩} {⟨Z2⟩} {⟨Z3⟩} {⟨Z4⟩} ;

```

This function is called with $\langle Z_1 \rangle \leq 999$. Act with `\number` to trim leading zeros from $\langle Z_1 \rangle \langle Z_2 \rangle$ (we don't do all four blocks at once, since non-zero blocks would then overflow \TeX 's integers). If the first two blocks are zero, the auxiliary receives an empty `#1` and trims `#2#30` from leading zeros, yielding a total shift between 7 and 16 to the exponent. Otherwise we get the shift from `#1` alone, yielding a result between 1 and 6. Once the exponent is taken care of, trim leading zeros from `#1#2#3` (when `#1` is empty, the space before `#2#3` is ignored), get four blocks of 4 digits and finally clean up. Trailing zeros are added so that digits can be grabbed safely.

```

25241 \cs_new:Npn \_fp_sub_back_shift:wnnnn ; #1#2
25242 {
25243   \exp_after:wN \_fp_sub_back_shift_ii:ww
25244   \int_value:w #1 #2 0 ;
25245 }
25246 \cs_new:Npn \_fp_sub_back_shift_ii:ww #1 0 ; #2#3 ;
25247 {
25248   \if_meaning:w @ #1 @
25249   - 7
25250   - \exp_after:wN \use_i:nnn
25251   \exp_after:wN \_fp_sub_back_shift_iii:NNNNNNNNw
25252   \int_value:w #2#3 0 ~ 123456789;
25253   \else:
25254   - \_fp_sub_back_shift_iii:NNNNNNNNw #1 123456789;
25255   \fi:
25256   \exp_after:wN \_fp_pack_twice_four:wNNNNNNNN
25257   \exp_after:wN \_fp_pack_twice_four:wNNNNNNNN
25258   \exp_after:wN \_fp_sub_back_shift_iv:nnnnw
25259   \exp_after:wN ;
25260   \int_value:w
25261   #1 ~ #2#3 0 ~ 0000 0000 0000 000 ;
25262 }
25263 \cs_new:Npn \_fp_sub_back_shift_iii:NNNNNNNNw #1#2#3#4#5#6#7#8#9; {#8}
25264 \cs_new:Npn \_fp_sub_back_shift_iv:nnnnw #1 ; #2 ; { ; #1 ; }

```

(End definition for _fp_sub_back_shift:wnnnn and others.)

```

\_fp_sub_back_far_o:NnnwnnnnN

```

```

\_fp_sub_back_far_o:NnnwnnnnN ⟨rounding⟩ {⟨Y1'⟩} {⟨Y2'⟩}
⟨extra-digits⟩ ; {⟨X1⟩} {⟨X2⟩} {⟨X3⟩} {⟨X4⟩} ⟨final sign⟩

```

If the difference is greater than $10^{\langle expo_x \rangle}$, call the `very_far` auxiliary. If the result is less than $10^{\langle expo_x \rangle}$, call the `not_far` auxiliary. If it is too close a call to know yet, namely if $1\langle Y_1' \rangle \langle Y_2' \rangle = \langle X_1 \rangle \langle X_2 \rangle \langle X_3 \rangle \langle X_4 \rangle 0$, then call the `quite_far` auxiliary. We use the odd combination of space and semi-colon delimiters to allow the `not_far` auxiliary to grab each piece individually, the `very_far` auxiliary to use `_fp_pack_eight:wNNNNNNNN`, and the `quite_far` to ignore the significands easily (using the `; delimiter`).

```

25265 \cs_new:Npn \_fp_sub_back_far_o:NnnwnnnnN #1 #2#3 #4; #5#6#7#8
25266 {

```

```

25267 \if_case:w
25268 \if_int_compare:w 1 #2 = #5#6 \use_i:nnnn #7 \exp_stop_f:
25269 \if_int_compare:w #3 = \use_none:n #7#8 0 \exp_stop_f:
25270 0
25271 \else:
25272 \if_int_compare:w #3 > \use_none:n #7#8 0 - \fi: 1
25273 \fi:
25274 \else:
25275 \if_int_compare:w 1 #2 > #5#6 \use_i:nnnn #7 - \fi: 1
25276 \fi:
25277 \exp_stop_f:
25278 \exp_after:wN \__fp_sub_back_quite_far_o:wwNN
25279 \or: \exp_after:wN \__fp_sub_back_very_far_o:wwwNN
25280 \else: \exp_after:wN \__fp_sub_back_not_far_o:wwwNN
25281 \fi:
25282 #2 ~ #3 ; #5 #6 ~ #7 #8 ; #1
25283 }

```

(End definition for __fp_sub_back_far_o:NnnwnnnN.)

__fp_sub_back_quite_far_o:wwNN
__fp_sub_back_quite_far_ii:NN

The easiest case is when $x - y$ is extremely close to a power of 10, namely the first digit of x is 1, and all others vanish when subtracting y . Then the *rounding* #3 and the *final sign* #4 control whether we get 1 or 0.9999999999999999. In the usual round-to-nearest mode, we get 1 whenever the *rounding* digit is less than or equal to 5 (remember that the *rounding* digit is only equal to 5 if there was no further non-zero digit).

```

25284 \cs_new:Npn \__fp_sub_back_quite_far_o:wwNN #1; #2; #3#4
25285 {
25286 \exp_after:wN \__fp_sub_back_quite_far_ii:NN
25287 \exp_after:wN #3
25288 \exp_after:wN #4
25289 }
25290 \cs_new:Npn \__fp_sub_back_quite_far_ii:NN #1#2
25291 {
25292 \if_case:w \__fp_round_neg:NNN #2 0 #1
25293 \exp_after:wN \use_i:nn
25294 \else:
25295 \exp_after:wN \use_ii:nn
25296 \fi:
25297 { ; {1000} {0000} {0000} {0000} ; }
25298 { - 1 ; {9999} {9999} {9999} {9999} ; }
25299 }

```

(End definition for __fp_sub_back_quite_far_o:wwNN and __fp_sub_back_quite_far_ii:NN.)

__fp_sub_back_not_far_o:wwwNN

In the present case, x and y have different exponents, but y is large enough that $x - y$ has a smaller exponent than x . Decrement the exponent (with -1). Then proceed in a way similar to the **near** auxiliaries seen earlier, but multiplying x by 10 (#30 and #40 below), and with the added quirk that the *rounding* digit has to be taken into account. Namely, we may have to decrease the result by one unit if __fp_round_neg:NNN returns 1. This function expects the *final sign* #6, the last digit of 1100000000+#40-#2, and the *rounding* digit. Instead of redoing the computation for the second argument, we note that __fp_round_neg:NNN only cares about its parity, which is identical to that of the last digit of #2.


```

25300 \cs_new:Npn \__fp_sub_back_not_far_o:wwwNN #1 ~ #2; #3 ~ #4; #5#6
25301 {
25302   - 1
25303   \exp_after:wN \__fp_sub_back_near_after:wNNNNw
25304   \int_value:w \__fp_int_eval:w 1#30 - #1 - 11
25305   \exp_after:wN \__fp_sub_back_near_pack:NNNNNNw
25306   \int_value:w \__fp_int_eval:w 11 0000 0000 + #40 - #2
25307   - \exp_after:wN \__fp_round_neg:NNN
25308   \exp_after:wN #6
25309   \use_none:nnnnnn #2 #5
25310   \exp_after:wN ;
25311 }

```

(End definition for __fp_sub_back_not_far_o:wwwNN.)

__fp_sub_back_very_far_o:wwwNN
__fp_sub_back_very_far_ii_o:nnNwwNN

The case where $x - y$ and x have the same exponent is a bit more tricky, mostly because it cannot reuse the same auxiliaries. Shift the y significand by adding a leading 0. Then the logic is similar to the `not_far` functions above. Rounding is a bit more complicated: we have two *rounding* digits `#3` and `#6` (from the decimation, and from the new shift) to take into account, and getting the parity of the main result requires a computation. The first `\int_value:w` triggers the second one because the number is unfinished; we can thus not use 0 in place of 2 there.

```

25312 \cs_new:Npn \__fp_sub_back_very_far_o:wwwNN #1#2#3#4#5#6#7
25313 {
25314   \__fp_pack_eight:wNNNNNNNN
25315   \__fp_sub_back_very_far_ii_o:nnNwwNN
25316   { 0 #1#2#3 #4#5#6#7 }
25317   ;
25318 }
25319 \cs_new:Npn \__fp_sub_back_very_far_ii_o:nnNwwNN #1#2 ; #3 ; #4 ~ #5; #6#7
25320 {
25321   \exp_after:wN \__fp_basics_pack_high:NNNNNw
25322   \int_value:w \__fp_int_eval:w 1#4 - #1 - 1
25323   \exp_after:wN \__fp_basics_pack_low:NNNNNw
25324   \int_value:w \__fp_int_eval:w 2#5 - #2
25325   - \exp_after:wN \__fp_round_neg:NNN
25326   \exp_after:wN #7
25327   \int_value:w
25328   \if_int_odd:w \__fp_int_eval:w #5 - #2 \__fp_int_eval_end:
25329   1 \else: 2 \fi:
25330   \int_value:w \__fp_round_digit:Nw #3 #6 ;
25331   \exp_after:wN ;
25332 }

```

(End definition for __fp_sub_back_very_far_o:wwwNN and __fp_sub_back_very_far_ii_o:nnNwwNN.)

72.2 Multiplication

72.2.1 Signs, and special numbers

__fp*_o:ww

We go through an auxiliary, which is common with `__fp/_o:ww`. The first argument is the operation, used for the invalid operation exception. The second is inserted in a formula to dispatch cases slightly differently between multiplication and division. The

third is the operation for normal floating points. The fourth is there for extra cases needed in `__fp_/_o:ww`.

```

25333 \cs_new:cpn { __fp*_o:ww }
25334 {
25335   \__fp_mul_cases_o:NnNnww
25336   *
25337   { - 2 + }
25338   \__fp_mul_npos_o:Nww
25339   { }
25340 }

```

(End definition for `__fp*_o:ww`.)

`__fp_mul_cases_o:nNnNnww` Split into 10 cases (12 for division). If both numbers are normal, go to case 0 (same sign) or case 1 (opposite signs): in both cases, call `__fp_mul_npos_o:Nww` to do the work. If the first operand is `nan`, go to case 2, in which the second operand is discarded; if the second operand is `nan`, go to case 3, in which the first operand is discarded (note the weird interaction with the final test on signs). Then we separate the case where the first number is normal and the second is zero: this goes to cases 4 and 5 for multiplication, 10 and 11 for division. Otherwise, we do a computation which dispatches the products $0 \times 0 = 0 \times 1 = 1 \times 0 = 0$ to case 4 or 5 depending on the combined sign, the products $0 \times \infty$ and $\infty \times 0$ to case 6 or 7 (invalid operation), and the products $1 \times \infty = \infty \times 1 = \infty \times \infty = \infty$ to cases 8 and 9. Note that the code for these two cases (which return $\pm\infty$) is inserted as argument #4, because it differs in the case of divisions.

```

25341 \cs_new:Npn \__fp_mul_cases_o:NnNnww
25342   #1#2#3#4 \s__fp \__fp_chk:w #5#6#7; \s__fp \__fp_chk:w #8#9
25343 {
25344   \if_case:w \__fp_int_eval:w
25345     \if_int_compare:w #5 #8 = 11 ~
25346     1
25347   \else:
25348     \if_meaning:w 3 #8
25349     3
25350   \else:
25351     \if_meaning:w 3 #5
25352     2
25353   \else:
25354     \if_int_compare:w #5 #8 = 10 ~
25355     9 #2 - 2
25356   \else:
25357     (#5 #2 #8) / 2 * 2 + 7
25358   \fi:
25359   \fi:
25360   \fi:
25361   \fi:
25362   \if_meaning:w #6 #9 - 1 \fi:
25363   \__fp_int_eval_end:
25364   \__fp_case_use:nw { #3 0 }
25365 \or: \__fp_case_use:nw { #3 2 }
25366 \or: \__fp_case_return_i_o:ww
25367 \or: \__fp_case_return_ii_o:ww
25368 \or: \__fp_case_return_o:Nww \c_zero_fp
25369 \or: \__fp_case_return_o:Nww \c_minus_zero_fp

```

```

25370 \or: \__fp_case_use:nw { \__fp_invalid_operation_o:Nww #1 }
25371 \or: \__fp_case_use:nw { \__fp_invalid_operation_o:Nww #1 }
25372 \or: \__fp_case_return_o:Nww \c_inf_fp
25373 \or: \__fp_case_return_o:Nww \c_minus_inf_fp
25374 #4
25375 \fi:
25376 \s__fp \__fp_chk:w #5 #6 #7;
25377 \s__fp \__fp_chk:w #8 #9
25378 }

```

(End definition for __fp_mul_cases_o:nNnnww.)

72.2.2 Absolute multiplication

In this subsection, we perform the multiplication of two positive normal numbers.

```

\__fp_mul_npos_o:Nww \__fp_mul_npos_o:Nww <final sign> \s__fp \__fp_chk:w 1 <sign1> {<exp1>}
<body1> ; \s__fp \__fp_chk:w 1 <sign2> {<exp2>} <body2> ;

```

After the computation, __fp_sanitize:Nw checks for overflow or underflow. As we did for addition, __fp_int_eval:w computes the exponent, catching any shift coming from the computation in the significand. The <final sign> is needed to do the rounding properly in the significand computation. We setup the post-expansion here, triggered by __fp_mul_significand_o:nnnnNnnnn.

This is also used in l3fp-convert.

```

25379 \cs_new:Npn \__fp_mul_npos_o:Nww
25380 #1 \s__fp \__fp_chk:w #2 #3 #4 #5 ; \s__fp \__fp_chk:w #6 #7 #8 #9 ;
25381 {
25382 \exp_after:wN \__fp_sanitize:Nw
25383 \exp_after:wN #1
25384 \int_value:w \__fp_int_eval:w
25385 #4 + #8
25386 \__fp_mul_significand_o:nnnnNnnnn #5 #1 #9
25387 }

```

(End definition for __fp_mul_npos_o:Nww.)

```

\__fp_mul_significand_o:nnnnNnnnn \__fp_mul_significand_o:nnnnNnnnn {<X1>} {<X2>} {<X3>} {<X4>} <sign>
{<Y1>} {<Y2>} {<Y3>} {<Y4>}
\__fp_mul_significand_drop:NNNNNw
\__fp_mul_significand_keep:NNNNNw

```

Note the three semicolons at the end of the definition. One is for the last __fp_mul_significand_drop:NNNNNw; one is for __fp_round_digit:Nw later on; and one, preceded by \exp_after:wN, which is correctly expanded (within an __fp_int_eval:w), is used by __fp_basics_pack_low:NNNNNw.

The product of two 16 digit integers has 31 or 32 digits, but it is impossible to know which one before computing. The place where we round depends on that number of digits, and may depend on all digits until the last in some rare cases. The approach is thus to compute the 5 first blocks of 4 digits (the first one is between 100 and 9999 inclusive), and a compact version of the remaining 3 blocks. Afterwards, the number of digits is known, and we can do the rounding within yet another set of __fp_int_eval:w.

```

25388 \cs_new:Npn \__fp_mul_significand_o:nnnnNnnnn #1#2#3#4 #5 #6#7#8#9
25389 {
25390 \exp_after:wN \__fp_mul_significand_test_f:NNN
25391 \exp_after:wN #5
25392 \int_value:w \__fp_int_eval:w 99990000 + #1#6 +

```

```

25393 \exp_after:wN \_fp_mul_significand_keep:NNNNNw
25394 \int_value:w \_fp_int_eval:w 99990000 + #1*#7 + #2*#6 +
25395 \exp_after:wN \_fp_mul_significand_keep:NNNNNw
25396 \int_value:w \_fp_int_eval:w 99990000 + #1*#8 + #2*#7 + #3*#6 +
25397 \exp_after:wN \_fp_mul_significand_drop:NNNNNw
25398 \int_value:w \_fp_int_eval:w 99990000 + #1*#9 + #2*#8 +
25399 #3*#7 + #4*#6 +
25400 \exp_after:wN \_fp_mul_significand_drop:NNNNNw
25401 \int_value:w \_fp_int_eval:w 99990000 + #2*#9 + #3*#8 +
25402 #4*#7 +
25403 \exp_after:wN \_fp_mul_significand_drop:NNNNNw
25404 \int_value:w \_fp_int_eval:w 99990000 + #3*#9 + #4*#8 +
25405 \exp_after:wN \_fp_mul_significand_drop:NNNNNw
25406 \int_value:w \_fp_int_eval:w 100000000 + #4*#9 ;
25407 ; \exp_after:wN ;
25408 }
25409 \cs_new:Npn \_fp_mul_significand_drop:NNNNNw #1#2#3#4#5 #6;
25410 { #1#2#3#4#5 ; + #6 }
25411 \cs_new:Npn \_fp_mul_significand_keep:NNNNNw #1#2#3#4#5 #6;
25412 { #1#2#3#4#5 ; #6 ; }

```

(End definition for `_fp_mul_significand_o:nnnnNnnnn`, `_fp_mul_significand_drop:NNNNNw`, and `_fp_mul_significand_keep:NNNNNw`.)

```

\_fp_mul_significand_test_f:NNN \_fp_mul_significand_test_f:NNN <sign> 1 <digits 1–8> ; <digits 9–12> ;
<digits 13–16> ; + <digits 17–20> + <digits 21–24> + <digits 25–28> + <digits
29–32> ; \exp_after:wN ;

```

If the *<digit 1>* is non-zero, then for rounding we only care about the digits 16 and 17, and whether further digits are zero or not (check for exact ties). On the other hand, if *<digit 1>* is zero, we care about digits 17 and 18, and whether further digits are zero.

```

25413 \cs_new:Npn \_fp_mul_significand_test_f:NNN #1 #2 #3
25414 {
25415   \if_meaning:w 0 #3
25416     \exp_after:wN \_fp_mul_significand_small_f:NNwwwN
25417   \else:
25418     \exp_after:wN \_fp_mul_significand_large_f:NwwNNNN
25419   \fi:
25420   #1 #3
25421 }

```

(End definition for `_fp_mul_significand_test_f:NNN`.)

`_fp_mul_significand_large_f:NwwNNNN`

In this branch, *<digit 1>* is non-zero. The result is thus *<digits 1–16>*, plus some rounding which depends on the digits 16, 17, and whether all subsequent digits are zero or not. Here, `_fp_round_digit:Nw` takes digits 17 and further (as an integer expression), and replaces it by a *<rounding digit>*, suitable for `_fp_round:NNN`.

```

25422 \cs_new:Npn \_fp_mul_significand_large_f:NwwNNNN #1 #2; #3; #4#5#6#7; +
25423 {
25424   \exp_after:wN \_fp_basics_pack_high:NNNNNw
25425   \int_value:w \_fp_int_eval:w 1#2
25426   \exp_after:wN \_fp_basics_pack_low:NNNNNw
25427   \int_value:w \_fp_int_eval:w 1#3#4#5#6#7
25428   + \exp_after:wN \_fp_round:NNN
25429   \exp_after:wN #1

```

```

25430         \exp_after:wN #7
25431         \int_value:w \__fp_round_digit:Nw
25432     }

```

(End definition for __fp_mul_significand_large_f:NwwNNNN.)

__fp_mul_significand_small_f:NNwwN

In this branch, $\langle digit\ 1 \rangle$ is zero. Our result is thus $\langle digits\ 2-17 \rangle$, plus some rounding which depends on the digits 17, 18, and whether all subsequent digits are zero or not. The 8 digits 1#3 are followed, after expansion of the `small_pack` auxiliary, by the next digit, to form a 9 digit number.

```

25433 \cs_new:Npn \__fp_mul_significand_small_f:NNwwN #1 #2#3; #4#5; #6; + #7
25434 {
25435     - 1
25436     \exp_after:wN \__fp_basics_pack_high:NNNNNw
25437     \int_value:w \__fp_int_eval:w 1#3#4
25438     \exp_after:wN \__fp_basics_pack_low:NNNNNw
25439     \int_value:w \__fp_int_eval:w 1#5#6#7
25440     + \exp_after:wN \__fp_round:NNN
25441     \exp_after:wN #1
25442     \exp_after:wN #7
25443     \int_value:w \__fp_round_digit:Nw
25444 }

```

(End definition for __fp_mul_significand_small_f:NNwwN.)

72.3 Division

72.3.1 Signs, and special numbers

Time is now ripe to tackle the hardest of the four elementary operations: division.

__fp_/_o:ww

Filtering special floating point is very similar to what we did for multiplications, with a few variations. Invalid operation exceptions display `/` rather than `*`. In the formula for dispatch, we replace `- 2 +` by `-`. The case of normal numbers is treated using `__fp_div_npos_o:Nww` rather than `__fp_mul_npos_o:Nww`. There are two additional cases: if the first operand is normal and the second is a zero, then the division by zero exception is raised: cases 10 and 11 of the `\if_case:w` construction in `__fp_mul_cases_o:NnNww` are provided as the fourth argument here.

```

25445 \cs_new:cpn { \__fp_/_o:ww }
25446 {
25447     \__fp_mul_cases_o:NnNww
25448     /
25449     { - }
25450     \__fp_div_npos_o:Nww
25451     {
25452         \or:
25453         \__fp_case_use:nw
25454         { \__fp_division_by_zero_o:NNww \c_inf_fp / }
25455         \or:
25456         \__fp_case_use:nw
25457         { \__fp_division_by_zero_o:NNww \c_minus_inf_fp / }
25458     }
25459 }

```

(End definition for `_fp_/_o:ww`.)

```
\_fp_div_npos_o:Nww \_fp_div_npos_o:Nww <final sign> \s\_fp \_fp_chk:w 1 <sign_A> {\exp A}\{A_1}\{A_2}\{A_3}\{A_4}\ ; \s\_fp \_fp_chk:w 1 <sign_Z> {\exp Z}\{Z_1}\{Z_2}\{Z_3}\{Z_4}\ ;
```

We want to compute A/Z . As for multiplication, `_fp_sanitiz:Nw` checks for overflow or underflow; we provide it with the *final sign*, and an integer expression in which we compute the exponent. We set up the arguments of `_fp_div_significand_i_o:wnnw`, namely an integer $\langle y \rangle$ obtained by adding 1 to the first 5 digits of Z (explanation given soon below), then the four $\{A_i\}$, then the four $\{Z_i\}$, a semi-colon, and the *final sign*, used for rounding at the end.

```
25460 \cs_new:Npn \_fp_div_npos_o:Nww
25461   #1 \s\_fp \_fp_chk:w 1 #2 #3 #4 ; \s\_fp \_fp_chk:w 1 #5 #6 #7#8#9;
25462   {
25463     \exp_after:wN \_fp_sanitiz:Nw
25464     \exp_after:wN #1
25465     \int_value:w \_fp_int_eval:w
25466     #3 - #6
25467     \exp_after:wN \_fp_div_significand_i_o:wnnw
25468     \int_value:w \_fp_int_eval:w #7 \use_i:nnnn #8 + 1 ;
25469     #4
25470     {\#7}{\#8}\#9 ;
25471     #1
25472   }
```

(End definition for `_fp_div_npos_o:Nww`.)

72.3.2 Work plan

In this subsection, we explain how to avoid overflowing $\text{T}_{\text{E}}\text{X}$'s integers when performing the division of two positive normal numbers.

We are given two numbers, $A = 0.A_1A_2A_3A_4$ and $Z = 0.Z_1Z_2Z_3Z_4$, in blocks of 4 digits, and we know that the first digits of A_1 and of Z_1 are non-zero. To compute A/Z , we proceed as follows.

- Find an integer $Q_A \simeq 10^4 A/Z$.
- Replace A by $B = 10^4 A - Q_A Z$.
- Find an integer $Q_B \simeq 10^4 B/Z$.
- Replace B by $C = 10^4 B - Q_B Z$.
- Find an integer $Q_C \simeq 10^4 C/Z$.
- Replace C by $D = 10^4 C - Q_C Z$.
- Find an integer $Q_D \simeq 10^4 D/Z$.
- Consider $E = 10^4 D - Q_D Z$, and ensure correct rounding.

The result is then $Q = 10^{-4}Q_A + 10^{-8}Q_B + 10^{-12}Q_C + 10^{-16}Q_D + \text{rounding}$. Since the Q_i are integers, B , C , D , and E are all exact multiples of 10^{-16} , in other words, computing with 16 digits after the decimal separator yields exact results. The problem is the risk of overflow: in general B , C , D , and E may be greater than 1.

Unfortunately, things are not as easy as they seem. In particular, we want all intermediate steps to be positive, since negative results would require extra calculations at the end. This requires that $Q_A \leq 10^4 A/Z$ etc. A reasonable attempt would be to define Q_A as

$$\backslash\text{int_eval:n}\left\{\frac{A_1 A_2}{Z_1 + 1} - 1\right\} \leq 10^4 \frac{A}{Z}$$

Subtracting 1 at the end takes care of the fact that $\varepsilon\text{-TeX}$'s $\backslash_fp_int_eval:w$ rounds divisions instead of truncating (really, $1/2$ would be sufficient, but we work with integers). We add 1 to Z_1 because $Z_1 \leq 10^4 Z < Z_1 + 1$ and we need Q_A to be an underestimate. However, we are now underestimating Q_A too much: it can be wrong by up to 100, for instance when $Z = 0.1$ and $A \simeq 1$. Then B could take values up to 10 (maybe more), and a few steps down the line, we would run into arithmetic overflow, since TeX can only handle integers less than roughly $2 \cdot 10^9$.

A better formula is to take

$$Q_A = \backslash\text{int_eval:n}\left\{\frac{10 \cdot A_1 A_2}{\lfloor 10^{-3} \cdot Z_1 Z_2 \rfloor + 1} - 1\right\}.$$

This is always less than $10^9 A/(10^5 Z)$, as we wanted. In words, we take the 5 first digits of Z into account, and the 8 first digits of A , using 0 as a 9-th digit rather than the true digit for efficiency reasons. We shall prove that using this formula to define all the Q_i avoids any overflow. For convenience, let us denote

$$y = \lfloor 10^{-3} \cdot Z_1 Z_2 \rfloor + 1,$$

so that, taking into account the fact that $\varepsilon\text{-TeX}$ rounds ties away from zero,

$$\begin{aligned} Q_A &= \left\lfloor \frac{A_1 A_2 0}{y} - \frac{1}{2} \right\rfloor \\ &> \frac{A_1 A_2 0}{y} - \frac{3}{2}. \end{aligned}$$

Note that $10^4 < y \leq 10^5$, and $999 \leq Q_A \leq 99989$. Also note that this formula does not cause an overflow as long as $A < (2^{31} - 1)/10^9 \simeq 2.147 \dots$, since the numerator involves an integer slightly smaller than $10^9 A$.

Let us bound B :

$$\begin{aligned} 10^5 B &= A_1 A_2 0 + 10 \cdot 0.A_3 A_4 - 10 \cdot Z_1.Z_2 Z_3 Z_4 \cdot Q_A \\ &< A_1 A_2 0 \cdot \left(1 - 10 \cdot \frac{Z_1.Z_2 Z_3 Z_4}{y}\right) + \frac{3}{2} \cdot 10 \cdot Z_1.Z_2 Z_3 Z_4 + 10 \\ &\leq \frac{A_1 A_2 0 \cdot (y - 10 \cdot Z_1.Z_2 Z_3 Z_4)}{y} + \frac{3}{2}y + 10 \\ &\leq \frac{A_1 A_2 0 \cdot 1}{y} + \frac{3}{2}y + 10 \leq \frac{10^9 A}{y} + 1.6 \cdot y. \end{aligned}$$

At the last step, we hide 10 into the second term for later convenience. The same reasoning yields

$$\begin{aligned} 10^5 B &< 10^9 A/y + 1.6y, \\ 10^5 C &< 10^9 B/y + 1.6y, \\ 10^5 D &< 10^9 C/y + 1.6y, \\ 10^5 E &< 10^9 D/y + 1.6y. \end{aligned}$$

The goal is now to prove that none of B , C , D , and E can go beyond $(2^{31} - 1)/10^9 = 2.147 \dots$.

Combining the various inequalities together with $A < 1$, we get

$$\begin{aligned} 10^5 B &< 10^9/y + 1.6y, \\ 10^5 C &< 10^{13}/y^2 + 1.6(y + 10^4), \\ 10^5 D &< 10^{17}/y^3 + 1.6(y + 10^4 + 10^8/y), \\ 10^5 E &< 10^{21}/y^4 + 1.6(y + 10^4 + 10^8/y + 10^{12}/y^2). \end{aligned}$$

All of those bounds are convex functions of y (since every power of y involved is convex, and the coefficients are positive), and thus maximal at one of the end-points of the allowed range $10^4 < y \leq 10^5$. Thus,

$$\begin{aligned} 10^5 B &< \max(1.16 \cdot 10^5, 1.7 \cdot 10^5), \\ 10^5 C &< \max(1.32 \cdot 10^5, 1.77 \cdot 10^5), \\ 10^5 D &< \max(1.48 \cdot 10^5, 1.777 \cdot 10^5), \\ 10^5 E &< \max(1.64 \cdot 10^5, 1.7777 \cdot 10^5). \end{aligned}$$

All of those bounds are less than $2.147 \cdot 10^5$, and we are thus within \TeX 's bounds in all cases!

We later need to have a bound on the Q_i . Their definitions imply that $Q_A < 10^9 A/y - 1/2 < 10^5 A$ and similarly for the other Q_i . Thus, all of them are less than 177770.

The last step is to ensure correct rounding. We have

$$A/Z = \sum_{i=1}^4 (10^{-4i} Q_i) + 10^{-16} E/Z$$

exactly. Furthermore, we know that the result is in $[0.1, 10)$, hence will be rounded to a multiple of 10^{-16} or of 10^{-15} , so we only need to know the integer part of E/Z , and a “rounding” digit encoding the rest. Equivalently, we need to find the integer part of $2E/Z$, and determine whether it was an exact integer or not (this serves to detect ties). Since

$$\frac{2E}{Z} = 2 \frac{10^5 E}{10^5 Z} \leq 2 \frac{10^5 E}{10^4} < 36,$$

this integer part is between 0 and 35 inclusive. We let ε -TeX round

$$P = \backslash\mathrm{int_eval:n} \left\{ \frac{2 \cdot E_1 E_2}{Z_1 Z_2} \right\},$$

which differs from $2E/Z$ by at most

$$\frac{1}{2} + 2 \left| \frac{E}{Z} - \frac{E}{10^{-8} Z_1 Z_2} \right| + 2 \left| \frac{10^8 E - E_1 E_2}{Z_1 Z_2} \right| < 1,$$

($1/2$ comes from ε -TeX's rounding) because each absolute value is less than 10^{-7} . Thus P is either the correct integer part, or is off by 1; furthermore, if $2E/Z$ is an integer, $P = 2E/Z$. We will check the sign of $2E - PZ$. If it is negative, then $E/Z \in ((P-1)/2, P/2)$. If it is zero, then $E/Z = P/2$. If it is positive, then $E/Z \in (P/2, (P+1)/2)$. In each case, we know how to round to an integer, depending on the parity of P , and the rounding mode.

72.3.3 Implementing the significand division

`_fp_div_significand_i_o:wnnw`

`_fp_div_significand_i_o:wnnw` $\langle y \rangle$; $\{\langle A_1 \rangle\}$ $\{\langle A_2 \rangle\}$ $\{\langle A_3 \rangle\}$ $\{\langle A_4 \rangle\}$
 $\{\langle Z_1 \rangle\}$ $\{\langle Z_2 \rangle\}$ $\{\langle Z_3 \rangle\}$ $\{\langle Z_4 \rangle\}$; $\langle sign \rangle$

Compute $10^6 + Q_A$ (a 7 digit number thanks to the shift), unbrace $\langle A_1 \rangle$ and $\langle A_2 \rangle$, and prepare the $\langle continuation \rangle$ arguments for 4 consecutive calls to `_fp_div_significand_calc:wnnnnnnnn`. Each of these calls needs $\langle y \rangle$ (#1), and it turns out that we need post-expansion there, hence the `\int_value:w`. Here, #4 is six brace groups, which give the six first n-type arguments of the `calc` function.

```

25473 \cs_new:Npn \_fp\_div\_significand\_i\_o:wnnw #1 ; #2#3 #4 ;
25474 {
25475   \exp\_after:wN \_fp\_div\_significand\_test\_o:w
25476   \int\_value:w \_fp\_int\_eval:w
25477   \exp\_after:wN \_fp\_div\_significand\_calc:wnnnnnnnn
25478   \int\_value:w \_fp\_int\_eval:w 999999 + #2 #3 0 / #1 ;
25479   #2 #3 ;
25480   #4
25481   { \exp\_after:wN \_fp\_div\_significand\_ii:wN \int\_value:w #1 }
25482   { \exp\_after:wN \_fp\_div\_significand\_ii:wN \int\_value:w #1 }
25483   { \exp\_after:wN \_fp\_div\_significand\_ii:wN \int\_value:w #1 }
25484   { \exp\_after:wN \_fp\_div\_significand\_iii:wnnnnnn \int\_value:w #1 }
25485 }

```

(End definition for `_fp_div_significand_i_o:wnnw`.)

`_fp_div_significand_calc:wnnnnnnnn`
`_fp_div_significand_calc_i:wnnnnnnnn`
`_fp_div_significand_calc_ii:wnnnnnnnn`

`_fp_div_significand_calc:wnnnnnnnn` $\langle 10^6 + Q_A \rangle$; $\langle A_1 \rangle$ $\langle A_2 \rangle$; $\{\langle A_3 \rangle\}$
 $\{\langle A_4 \rangle\}$ $\{\langle Z_1 \rangle\}$ $\{\langle Z_2 \rangle\}$ $\{\langle Z_3 \rangle\}$ $\{\langle Z_4 \rangle\}$ $\{\langle continuation \rangle\}$

expands to

$\langle 10^6 + Q_A \rangle$ $\langle continuation \rangle$; $\langle B_1 \rangle$ $\langle B_2 \rangle$; $\{\langle B_3 \rangle\}$ $\{\langle B_4 \rangle\}$ $\{\langle Z_1 \rangle\}$ $\{\langle Z_2 \rangle\}$ $\{\langle Z_3 \rangle\}$
 $\{\langle Z_4 \rangle\}$

where $B = 10^4 A - Q_A \cdot Z$. This function is also used to compute C , D , E (with the input shifted accordingly), and is used in `l3fp-expo`.

We know that $0 < Q_A < 1.8 \cdot 10^5$, so the product of Q_A with each Z_i is within TeX's bounds. However, it is a little bit too large for our purposes: we would not be able to

use the usual trick of adding a large power of 10 to ensure that the number of digits is fixed.

The bound on Q_A , implies that $10^6 + Q_A$ starts with the digit 1, followed by 0 or 1. We test, and call different auxiliaries for the two cases. An earlier implementation did the tests within the computation, but since we added a *continuation*, this is not possible because the macro has 9 parameters.

The result we want is then (the overall power of 10 is arbitrary):

$$10^{-4}(\#2 - \#1 \cdot \#5 - 10 \cdot \langle i \rangle \cdot \#5\#6) + 10^{-8}(\#3 - \#1 \cdot \#6 - 10 \cdot \langle i \rangle \cdot \#7) \\ + 10^{-12}(\#4 - \#1 \cdot \#7 - 10 \cdot \langle i \rangle \cdot \#8) + 10^{-16}(-\#1 \cdot \#8),$$

where $\langle i \rangle$ stands for the 10^5 digit of Q_A , which is 0 or 1, and $\#1$, $\#2$, *etc.* are the parameters of either auxiliary. The factors of 10 come from the fact that $Q_A = 10 \cdot 10^4 \cdot \langle i \rangle + \#1$. As usual, to combine all the terms, we need to choose some shifts which must ensure that the number of digits of the second, third, and fourth terms are each fixed. Here, the positive contributions are at most 10^8 and the negative contributions can go up to 10^9 . Indeed, for the auxiliary with $\langle i \rangle = 1$, $\#1$ is at most 80000, leading to contributions of at worse $-8 \cdot 10^8 4$, while the other negative term is very small $< 10^6$ (except in the first expression, where we don't care about the number of digits); for the auxiliary with $\langle i \rangle = 0$, $\#1$ can go up to 99999, but there is no other negative term. Hence, a good choice is $2 \cdot 10^9$, which produces totals in the range $[10^9, 2.1 \cdot 10^9]$. We are flirting with TeX's limits once more.

```

25486 \cs_new:Npn \__fp_div_significand_calc:wwnnnnnnn 1#1
25487 {
25488   \if_meaning:w 1 #1
25489     \exp_after:wN \__fp_div_significand_calc_i:wwnnnnnnn
25490   \else:
25491     \exp_after:wN \__fp_div_significand_calc_ii:wwnnnnnnn
25492   \fi:
25493 }
25494 \cs_new:Npn \__fp_div_significand_calc_i:wwnnnnnnn
25495   #1; #2; #3#4 #5#6#7#8 #9
25496 {
25497   1 1 #1
25498   #9 \exp_after:wN ;
25499   \int_value:w \__fp_int_eval:w \c__fp_Bigg_leading_shift_int
25500     + #2 - #1 * #5 - #5#60
25501   \exp_after:wN \__fp_pack_Bigg:NNNNNNw
25502   \int_value:w \__fp_int_eval:w \c__fp_Bigg_middle_shift_int
25503     + #3 - #1 * #6 - #70
25504   \exp_after:wN \__fp_pack_Bigg:NNNNNNw
25505   \int_value:w \__fp_int_eval:w \c__fp_Bigg_middle_shift_int
25506     + #4 - #1 * #7 - #80
25507   \exp_after:wN \__fp_pack_Bigg:NNNNNNw
25508   \int_value:w \__fp_int_eval:w \c__fp_Bigg_trailing_shift_int
25509     - #1 * #8 ;
25510   {#5}{#6}{#7}{#8}
25511 }
25512 \cs_new:Npn \__fp_div_significand_calc_ii:wwnnnnnnn
25513   #1; #2; #3#4 #5#6#7#8 #9
25514 {
25515   1 0 #1

```

```

25516      #9 \exp_after:wN ;
25517      \int_value:w \_fp_int_eval:w \c\_fp_Bigg_leading_shift_int
25518      + #2 - #1 * #5
25519      \exp_after:wN \_fp_pack_Bigg:NNNNNNw
25520      \int_value:w \_fp_int_eval:w \c\_fp_Bigg_middle_shift_int
25521      + #3 - #1 * #6
25522      \exp_after:wN \_fp_pack_Bigg:NNNNNNw
25523      \int_value:w \_fp_int_eval:w \c\_fp_Bigg_middle_shift_int
25524      + #4 - #1 * #7
25525      \exp_after:wN \_fp_pack_Bigg:NNNNNNw
25526      \int_value:w \_fp_int_eval:w \c\_fp_Bigg_trailing_shift_int
25527      - #1 * #8 ;
25528      {#5}{#6}{#7}{#8}
25529    }

```

(End definition for _fp_div_significand_calc:wwnnnnnn, _fp_div_significand_calc_i:wwnnnnnn, and _fp_div_significand_calc_ii:wwnnnnnn.)

```

\_fp_div_significand_ii:wnn      \_fp_div_significand_ii:wnn <y> ; <B1> ; {<B2>} {<B3>} {<B4>} {<Z1>}
                                {<Z2>} {<Z3>} {<Z4>} <continuations> <sign>

```

Compute Q_B by evaluating $\langle B_1 \rangle \langle B_2 \rangle 0 / y - 1$. The result is output to the left, in an `_fp_int_eval:w` which we start now. Once that is evaluated (and the other Q_i also, since later expansions are triggered by this one), a packing auxiliary takes care of placing the digits of Q_B in an appropriate way for the final addition to obtain Q . This auxiliary is also used to compute Q_C and Q_D with the inputs C and D instead of B .

```

25530 \cs_new:Npn \_fp_div_significand_ii:wnn #1; #2;#3
25531 {
25532   \exp_after:wN \_fp_div_significand_pack:NNN
25533   \int_value:w \_fp_int_eval:w
25534   \exp_after:wN \_fp_div_significand_calc:wwnnnnnn
25535   \int_value:w \_fp_int_eval:w 999999 + #2 #3 0 / #1 ; #2 #3 ;
25536 }

```

(End definition for _fp_div_significand_ii:wnn.)

```

\_fp_div_significand_iii:wwnnnnn \_fp_div_significand_iii:wwnnnnn <y> ; <E1> ; {<E2>} {<E3>} {<E4>}
                                {<Z1>} {<Z2>} {<Z3>} {<Z4>} <sign>

```

We compute $P \simeq 2E/Z$ by rounding $2E_1E_2/Z_1Z_2$. Note the first 0, which multiplies Q_D by 10: we later add (roughly) $5 \cdot P$, which amounts to adding $P/2 \simeq E/Z$ to Q_D , the appropriate correction from a hypothetical Q_E .

```

25537 \cs_new:Npn \_fp_div_significand_iii:wwnnnnn #1; #2;#3#4#5 #6#7
25538 {
25539   0
25540   \exp_after:wN \_fp_div_significand_iv:wwnnnnnnn
25541   \int_value:w \_fp_int_eval:w ( 2 * #2 #3 ) / #6 #7 ; % <- P
25542   #2 ; {#3} {#4} {#5}
25543   {#6} {#7}
25544 }

```

(End definition for _fp_div_significand_iii:wwnnnnn.)

```

\_fp_div_significand_iv:wwnnnnnnn \_fp_div_significand_iv:wwnnnnnnn <P> ; <E1> ; {<E2>} {<E3>} {<E4>}
\_fp_div_significand_v:NNw      {<Z1>} {<Z2>} {<Z3>} {<Z4>} <sign>
\_fp_div_significand_vi:Nw

```

This adds to the current expression $(10^7 + 10 \cdot Q_D)$ a contribution of $5 \cdot P + \text{sign}(T)$ with $T = 2E - PZ$. This amounts to adding $P/2$ to Q_D , with an extra *rounding* digit. This *rounding* digit is 0 or 5 if T does not contribute, *i.e.*, if $0 = T = 2E - PZ$, in other words if $10^{16}A/Z$ is an integer or half-integer. Otherwise it is in the appropriate range, $[1, 4]$ or $[6, 9]$. This is precise enough for rounding purposes (in any mode).

It seems an overkill to compute T exactly as I do here, but I see no faster way right now.

Once more, we need to be careful and show that the calculation $\#1 \cdot \#6\#7$ below does not cause an overflow: naively, P can be up to 35, and $\#6\#7$ up to 10^8 , but both cannot happen simultaneously. To show that things are fine, we split in two (non-disjoint) cases.

- For $P < 10$, the product obeys $P \cdot \#6\#7 < 10^8 \cdot P < 10^9$.
- For large $P \geq 3$, the rounding error on P , which is at most 1, is less than a factor of 2, hence $P \leq 4E/Z$. Also, $\#6\#7 \leq 10^8 \cdot Z$, hence $P \cdot \#6\#7 \leq 4E \cdot 10^8 < 10^9$.

Both inequalities could be made tighter if needed.

Note however that $P \cdot \#8\#9$ may overflow, since the two factors are now independent, and the result may reach $3.5 \cdot 10^9$. Thus we compute the two lower levels separately. The rest is standard, except that we use $+$ as a separator (ending integer expressions explicitly). T is negative if the first character is $-$, it is positive if the first character is neither 0 nor $-$. It is also positive if the first character is 0 and second argument of `__fp_div_significand_vi:Nw`, a sum of several terms, is also zero. Otherwise, there was an exact agreement: $T = 0$.

```

25545 \cs_new:Npn \__fp_div_significand_iv:wwnnnnnnn #1; #2;#3#4#5 #6#7#8#9
25546 {
25547   + 5 * #1
25548   \exp_after:wN \__fp_div_significand_vi:Nw
25549   \int_value:w \__fp_int_eval:w -20 + 2*#2#3 - #1*#6#7 +
25550   \exp_after:wN \__fp_div_significand_v:NN
25551   \int_value:w \__fp_int_eval:w 199980 + 2*#4 - #1*#8 +
25552   \exp_after:wN \__fp_div_significand_v:NN
25553   \int_value:w \__fp_int_eval:w 200000 + 2*#5 - #1*#9 ;
25554 }
25555 \cs_new:Npn \__fp_div_significand_v:NN #1#2 { #1#2 \__fp_int_eval_end: + }
25556 \cs_new:Npn \__fp_div_significand_vi:Nw #1#2;
25557 {
25558   \if_meaning:w 0 #1
25559   \if_int_compare:w \__fp_int_eval:w #2 > 0 + 1 \fi:
25560   \else:
25561   \if_meaning:w - #1 - \else: + \fi: 1
25562   \fi:
25563   ;
25564 }

```

(End definition for `__fp_div_significand_iv:wwnnnnnnn`, `__fp_div_significand_v:NNw`, and `__fp_div_significand_vi:Nw`.)

`__fp_div_significand_pack:NNN`

At this stage, we are in the following situation: \TeX is in the process of expanding several integer expressions, thus functions at the bottom expand before those above.

```

    \_fp_div_significand_test_o:w 106 + QA \_fp_div_significand_-
    pack:NNN 106 + QB \_fp_div_significand_pack:NNN 106 + QC \_fp_-
    div_significand_pack:NNN 107 + 10 · QD + 5 · P + ε ; ⟨sign⟩

```

Here, $\varepsilon = \text{sign}(T)$ is 0 in case $2E = PZ$, 1 in case $2E > PZ$, which means that P was the correct value, but not with an exact quotient, and -1 if $2E < PZ$, i.e., P was an overestimate. The packing function we define now does nothing special: it removes the 10^6 and carries two digits (for the 10^5 's and the 10^4 's).

```

25565 \cs_new:Npn \_fp_div_significand_pack:NNN 1 #1 #2 { + #1 #2 ; }

```

(End definition for `_fp_div_significand_pack:NNN`.)

```

\_fp_div_significand_test_o:w    \_fp_div_significand_test_o:w 1 0 ⟨5d⟩ ;  ⟨4d⟩ ; ⟨4d⟩ ; ⟨5d⟩ ; ⟨sign⟩

```

The reason we know that the first two digits are 1 and 0 is that the final result is known to be between 0.1 (inclusive) and 10, hence \widetilde{Q}_A (the tilde denoting the contribution from the other Q_i) is at most 99999, and $10^6 + \widetilde{Q}_A = 10 \dots$.

It is now time to round. This depends on how many digits the final result will have.

```

25566 \cs_new:Npn \_fp_div_significand_test_o:w 10 #1
25567 {
25568   \if_meaning:w 0 #1
25569     \exp_after:wN \_fp_div_significand_small_o:wwwNNNNwN
25570   \else:
25571     \exp_after:wN \_fp_div_significand_large_o:wwwNNNNwN
25572   \fi:
25573   #1
25574 }

```

(End definition for `_fp_div_significand_test_o:w`.)

```

\_fp_div_significand_small_o:wwwNNNNwN    \_fp_div_significand_small_o:wwwNNNNwN 0 ⟨4d⟩ ;  ⟨4d⟩ ; ⟨4d⟩ ; ⟨5d⟩
; ⟨final sign⟩

```

Standard use of the functions `_fp_basics_pack_low:NNNNw` and `_fp_basics_pack_high:NNNNw`. We finally get to use the *⟨final sign⟩* which has been sitting there for a while.

```

25575 \cs_new:Npn \_fp_div_significand_small_o:wwwNNNNwN
25576   0 #1; #2; #3; #4#5#6#7#8; #9
25577 {
25578   \exp_after:wN \_fp_basics_pack_high:NNNNw
25579   \int_value:w \_fp_int_eval:w 1 #1#2
25580   \exp_after:wN \_fp_basics_pack_low:NNNNw
25581   \int_value:w \_fp_int_eval:w 1 #3#4#5#6#7
25582   + \_fp_round:NNN #9 #7 #8
25583   \exp_after:wN ;
25584 }

```

(End definition for `_fp_div_significand_small_o:wwwNNNNwN`.)

```

\_fp_div_significand_large_o:wwwNNNNwN    \_fp_div_significand_large_o:wwwNNNNwN ⟨5d⟩ ;  ⟨4d⟩ ; ⟨4d⟩ ; ⟨5d⟩ ;
⟨sign⟩

```

We know that the final result cannot reach 10, hence `1#1#2`, together with contributions from the level below, cannot reach $2 \cdot 10^9$. For rounding, we build the *⟨rounding digit⟩* from the last two of our 18 digits.

```

25585 \cs_new:Npn \_fp_div_significand_large_o:wwwNNNNwN

```

```

25586     #1; #2; #3; #4#5#6#7#8; #9
25587     {
25588     + 1
25589     \exp_after:wN \_fp_basics_pack_weird_high:NNNNNNNNw
25590     \int_value:w \_fp_int_eval:w 1 #1 #2
25591     \exp_after:wN \_fp_basics_pack_weird_low:NNNNw
25592     \int_value:w \_fp_int_eval:w 1 #3 #4 #5 #6 +
25593     \exp_after:wN \_fp_round:NNN
25594     \exp_after:wN #9
25595     \exp_after:wN #6
25596     \int_value:w \_fp_round_digit:Nw #7 #8 ;
25597     \exp_after:wN ;
25598     }

```

(End definition for _fp_div_significand_large_o:wwwNNNNwN.)

72.4 Square root

_fp_sqrt_o:w Zeros are unchanged: $\sqrt{-0} = -0$ and $\sqrt{+0} = +0$. Negative numbers (other than -0) have no real square root. Positive infinity, and nan, are unchanged. Finally, for normal positive numbers, there is some work to do.

```

25599 \cs_new:Npn \_fp_sqrt_o:w #1 \s_fp \_fp_chk:w #2#3#4; @
25600 {
25601     \if_meaning:w 0 #2 \_fp_case_return_same_o:w \fi:
25602     \if_meaning:w 2 #3
25603         \_fp_case_use:nw { \_fp_invalid_operation_o:nw { sqrt } }
25604     \fi:
25605     \if_meaning:w 1 #2 \else: \_fp_case_return_same_o:w \fi:
25606     \_fp_sqrt_npos_o:w
25607     \s_fp \_fp_chk:w #2 #3 #4;
25608 }

```

(End definition for _fp_sqrt_o:w.)

_fp_sqrt_npos_o:w Prepare _fp_sanitize:Nw to receive the final sign 0 (the result is always positive) and the exponent, equal to half of the exponent #1 of the argument. If the exponent #1 is even, find a first approximation of the square root of the significand $10^8 a_1 + a_2 = 10^8 \#2\#3 + \#4\#5$ through Newton's method, starting at $x = 57234133 \simeq 10^{7.75}$. Otherwise, first shift the significand of the argument by one digit, getting $a'_1 \in [10^6, 10^7)$ instead of $[10^7, 10^8)$, then use Newton's method starting at $17782794 \simeq 10^{7.25}$.

```

25609 \cs_new:Npn \_fp_sqrt_npos_o:w \s_fp \_fp_chk:w 1 0 #1#2#3#4#5;
25610 {
25611     \exp_after:wN \_fp_sanitize:Nw
25612     \exp_after:wN 0
25613     \int_value:w \_fp_int_eval:w
25614     \if_int_odd:w #1 \exp_stop_f:
25615         \exp_after:wN \_fp_sqrt_npos_auxi_o:wwnnN
25616     \fi:
25617     #1 / 2
25618     \_fp_sqrt_Newton_o:wnn 56234133; 0; {#2#3} {#4#5} 0
25619 }
25620 \cs_new:Npn \_fp_sqrt_npos_auxi_o:wwnnN #1 / 2 #2; 0; #3#4#5
25621 {

```

```

25622      ( #1 + 1 ) / 2
25623      \__fp_pack_eight:wNNNNNNNN
25624      \__fp_sqrt_npos_auxii_o:wNNNNNNNN
25625      ;
25626      0 #3 #4
25627    }
25628 \cs_new:Npn \__fp_sqrt_npos_auxii_o:wNNNNNNNN #1; #2#3#4#5#6#7#8#9
25629 { \__fp_sqrt_Newton_o:wnn 17782794; 0; {#1} {#2#3#4#5#6#7#8#9} }

```

(End definition for __fp_sqrt_npos_o:w, __fp_sqrt_npos_auxi_o:wnnnN, and __fp_sqrt_npos_auxii_o:wNNNNNNNN.)

__fp_sqrt_Newton_o:wnn

Newton's method maps $x \mapsto [(x + [10^8 a_1/x])/2]$ in each iteration, where $[b/c]$ denotes ε -TeX's division. This division rounds the real number b/c to the closest integer, rounding ties away from zero, hence when c is even, $b/c - 1/2 + 1/c \leq [b/c] \leq b/c + 1/2$ and when c is odd, $b/c - 1/2 + 1/(2c) \leq [b/c] \leq b/c + 1/2 - 1/(2c)$. For all c , $b/c - 1/2 + 1/(2c) \leq [b/c] \leq b/c + 1/2$.

Let us prove that the method converges when implemented with ε -TeX integer division, for any $10^6 \leq a_1 < 10^8$ and starting value $10^6 \leq x < 10^8$. Using the inequalities above and the arithmetic–geometric inequality $(x + t)/2 \geq \sqrt{xt}$ for $t = 10^8 a_1/x$, we find

$$x' = \left\lceil \frac{x + [10^8 a_1/x]}{2} \right\rceil \geq \frac{x + 10^8 a_1/x - 1/2 + 1/(2x)}{2} \geq \sqrt{10^8 a_1} - \frac{1}{4} + \frac{1}{4x}.$$

After any step of iteration, we thus have $\delta = x - \sqrt{10^8 a_1} \geq -0.25 + 0.25 \cdot 10^{-8}$. The new difference $\delta' = x' - \sqrt{10^8 a_1}$ after one step is bounded above as

$$x' - \sqrt{10^8 a_1} \leq \frac{x + 10^8 a_1/x + 1/2}{2} + \frac{1}{2} - \sqrt{10^8 a_1} \leq \frac{\delta}{2} \frac{\delta}{\sqrt{10^8 a_1} + \delta} + \frac{3}{4}.$$

For $\delta > 3/2$, this last expression is $\leq \delta/2 + 3/4 < \delta$, hence δ decreases at each step: since all x are integers, δ must reach a value $-1/4 < \delta \leq 3/2$. In this range of values, we get $\delta' \leq \frac{3}{4} \frac{3}{2\sqrt{10^8 a_1}} + \frac{3}{4} \leq 0.75 + 1.125 \cdot 10^{-7}$. We deduce that the difference $\delta = x - \sqrt{10^8 a_1}$ eventually reaches a value in the interval $[-0.25 + 0.25 \cdot 10^{-8}, 0.75 + 11.25 \cdot 10^{-8}]$, whose width is $1 + 11 \cdot 10^{-8}$. The corresponding interval for x may contain two integers, hence x might oscillate between those two values.

However, the fact that $x \mapsto x - 1$ and $x - 1 \mapsto x$ puts stronger constraints, which are not compatible: the first implies

$$x + [10^8 a_1/x] \leq 2x - 2$$

hence $10^8 a_1/x \leq x - 3/2$, while the second implies

$$x - 1 + [10^8 a_1/(x - 1)] \geq 2x - 1$$

hence $10^8 a_1/(x - 1) \geq x - 1/2$. Combining the two inequalities yields $x^2 - 3x/2 \geq 10^8 a_1 \geq x - 3x/2 + 1/2$, which cannot hold. Therefore, the iteration always converges to a single integer x . To stop the iteration when two consecutive results are equal, the function __fp_sqrt_Newton_o:wnn receives the newly computed result as #1, the previous result as #2, and a_1 as #3. Note that ε -TeX combines the computation of a multiplication and a following division, thus avoiding overflow in $\#3 * 100000000 / \#1$. In any case, the result is within $[10^7, 10^8]$.

```

25630 \cs_new:Npn \__fp_sqrt_Newton_o:wnn #1; #2; #3
25631 {
25632   \if_int_compare:w #1 = #2 \exp_stop_f:
25633     \exp_after:wN \__fp_sqrt_auxi_o:NNNNwnnnN
25634     \int_value:w \__fp_int_eval:w 9999 9999 +
25635     \exp_after:wN \__fp_use_none_until_s:w
25636   \fi:
25637   \exp_after:wN \__fp_sqrt_Newton_o:wnn
25638   \int_value:w \__fp_int_eval:w (#1 + #3 * 1 0000 0000 / #1) / 2 ;
25639   #1; {#3}
25640 }

```

(End definition for __fp_sqrt_Newton_o:wnn.)

__fp_sqrt_auxi_o:NNNNwnnnN

This function is followed by $10^8 + x - 1$, which has 9 digits starting with 1, then ; $\{\langle a_1 \rangle\} \{\langle a_2 \rangle\} \langle a' \rangle$. Here, $x \simeq \sqrt{10^8 a_1}$ and we want to estimate the square root of $a = 10^{-8}a_1 + 10^{-16}a_2 + 10^{-17}a'$. We set up an initial underestimate

$$y = (x - 1)10^{-8} + 0.2499998875 \cdot 10^{-8} \lesssim \sqrt{a}.$$

From the inequalities shown earlier, we know that $y \leq \sqrt{10^{-8}a_1} \leq \sqrt{a}$ and that $\sqrt{10^{-8}a_1} \leq y + 10^{-8} + 11 \cdot 10^{-16}$ hence (using $0.1 \leq y \leq \sqrt{a} \leq 1$)

$$a - y^2 \leq 10^{-8}a_1 + 10^{-8} - y^2 \leq (y + 10^{-8} + 11 \cdot 10^{-16})^2 - y^2 + 10^{-8} < 3.2 \cdot 10^{-8},$$

and $\sqrt{a} - y = (a - y^2)/(\sqrt{a} + y) \leq 16 \cdot 10^{-8}$. Next, __fp_sqrt_auxii_o:NnnnnnnnnN is called several times to get closer and closer underestimates of \sqrt{a} . By construction, the underestimates y are always increasing, $a - y^2 < 3.2 \cdot 10^{-8}$ for all. Also, $y < 1$.

```

25641 \cs_new:Npn \__fp_sqrt_auxi_o:NNNNwnnnN 1 #1#2#3#4#5;
25642 {
25643   \__fp_sqrt_auxii_o:NnnnnnnnnN
25644   \__fp_sqrt_auxiii_o:wnnnnnnnnn
25645   {#1#2#3#4} {#5} {2499} {9988} {7500}
25646 }

```

(End definition for __fp_sqrt_auxi_o:NNNNwnnnN.)

__fp_sqrt_auxii_o:NnnnnnnnnN

This receives a continuation function #1, then five blocks of 4 digits for y , then two 8-digit blocks and a single digit for a . A common estimate of $\sqrt{a} - y = (a - y^2)/(\sqrt{a} + y)$ is $(a - y^2)/(2y)$, which leads to alternating overestimates and underestimates. We tweak this, to only work with underestimates (no need then to worry about signs in the computation). Each step finds the largest integer $j \leq 6$ such that $10^{4j}(a - y^2) < 2 \cdot 10^8$, then computes the integer (with ε -TeX's rounding division)

$$10^{4j}z = \left[([10^{4j}(a - y^2)] - 257) \cdot (0.5 \cdot 10^8) / [10^8y + 1] \right].$$

The choice of j ensures that $10^{4j}z < 2 \cdot 10^8 \cdot 0.5 \cdot 10^8 / 10^7 = 10^9$, thus $10^9 + 10^{4j}z$ has exactly 10 digits, does not overflow TeX's integer range, and starts with 1. Incidentally, since all $a - y^2 \leq 3.2 \cdot 10^{-8}$, we know that $j \geq 3$.

Let us show that z is an underestimate of $\sqrt{a} - y$. On the one hand, $\sqrt{a} - y \leq 16 \cdot 10^{-8}$ because this holds for the initial y and values of y can only increase. On the other hand, the choice of j implies that $\sqrt{a} - y \leq 5(\sqrt{a} + y)(\sqrt{a} - y) = 5(a - y^2) < 10^{9-4j}$. For $j = 3$,

the first bound is better, while for larger j , the second bound is better. For all $j \in [3, 6]$, we find $\sqrt{a} - y < 16 \cdot 10^{-2j}$. From this, we deduce that

$$10^{4j}(\sqrt{a} - y) = \frac{10^{4j}(a - y^2 - (\sqrt{a} - y)^2)}{2y} \geq \frac{\lfloor 10^{4j}(a - y^2) \rfloor - 257}{2 \cdot 10^{-8} \lfloor 10^8 y + 1 \rfloor} + \frac{1}{2}$$

where we have replaced the bound $10^{4j}(16 \cdot 10^{-2j}) = 256$ by 257 and extracted the corresponding term $1/(2 \cdot 10^{-8} \lfloor 10^8 y + 1 \rfloor) \geq 1/2$. Given that ε -TeX's integer division obeys $\lfloor b/c \rfloor \leq b/c + 1/2$, we deduce that $10^{4j}z \leq 10^{4j}(\sqrt{a} - y)$, hence $y + z \leq \sqrt{a}$ is an underestimate of \sqrt{a} , as claimed. One implementation detail: because the computation involves $-4 \cdot 4 - 2 \cdot 3 \cdot 5 - 2 \cdot 2 \cdot 6$ which may be as low as $-5 \cdot 10^8$, we need to use the `pack_big` functions, and the big shifts.

```

25647 \cs_new:Npn \__fp_sqrt_auxii_o:NnnnnnnN #1 #2#3#4#5#6 #7#8#9
25648 {
25649   \exp_after:wN #1
25650   \int_value:w \__fp_int_eval:w \c__fp_big_leading_shift_int
25651     + #7 - #2 * #2
25652   \exp_after:wN \__fp_pack_big:NNNNNNw
25653   \int_value:w \__fp_int_eval:w \c__fp_big_middle_shift_int
25654     - 2 * #2 * #3
25655   \exp_after:wN \__fp_pack_big:NNNNNNw
25656   \int_value:w \__fp_int_eval:w \c__fp_big_middle_shift_int
25657     + #8 - #3 * #3 - 2 * #2 * #4
25658   \exp_after:wN \__fp_pack_big:NNNNNNw
25659   \int_value:w \__fp_int_eval:w \c__fp_big_middle_shift_int
25660     - 2 * #3 * #4 - 2 * #2 * #5
25661   \exp_after:wN \__fp_pack_big:NNNNNNw
25662   \int_value:w \__fp_int_eval:w \c__fp_big_middle_shift_int
25663     + #9 000 0000 - #4 * #4 - 2 * #3 * #5 - 2 * #2 * #6
25664   \exp_after:wN \__fp_pack_big:NNNNNNw
25665   \int_value:w \__fp_int_eval:w \c__fp_big_middle_shift_int
25666     - 2 * #4 * #5 - 2 * #3 * #6
25667   \exp_after:wN \__fp_pack_big:NNNNNNw
25668   \int_value:w \__fp_int_eval:w \c__fp_big_middle_shift_int
25669     - #5 * #5 - 2 * #4 * #6
25670   \exp_after:wN \__fp_pack_big:NNNNNNw
25671   \int_value:w \__fp_int_eval:w
25672     \c__fp_big_middle_shift_int
25673     - 2 * #5 * #6
25674   \exp_after:wN \__fp_pack_big:NNNNNNw
25675   \int_value:w \__fp_int_eval:w
25676     \c__fp_big_trailing_shift_int
25677     - #6 * #6 ;
25678   % (
25679   - 257 ) * 5000 0000 / (#2#3 + 1) + 10 0000 0000 ;
25680   {#2}{#3}{#4}{#5}{#6} {#7}{#8}{#9}
25681 }

```

(End definition for `__fp_sqrt_auxii_o:NnnnnnnN`.)

```

\__fp_sqrt_auxiii_o:wnnnnnnn
\__fp_sqrt_auxiv_o:NNNNNw
\__fp_sqrt_auxv_o:NNNNNw
\__fp_sqrt_auxvi_o:NNNNNw
\__fp_sqrt_auxvii_o:NNNNNw

```

We receive here the difference $a - y^2 = d = \sum_i d_i \cdot 10^{-4i}$, as $\langle d_2 \rangle$; $\{\langle d_3 \rangle\} \dots \{\langle d_{10} \rangle\}$, where each block has 4 digits, except $\langle d_2 \rangle$. This function finds the largest $j \leq 6$ such that $10^{4j}(a - y^2) < 2 \cdot 10^8$, then leaves an open parenthesis and the integer $\lfloor 10^{4j}(a - y^2) \rfloor$

in an integer expression. The closing parenthesis is provided by the caller `__fp_sqrt_auxii_o:NnnnnnnnN`, which completes the expression

$$10^{4j}z = \left[([10^{4j}(a - y^2)] - 257) \cdot (0.5 \cdot 10^8) / [10^8y + 1] \right]$$

for an estimate of $10^{4j}(\sqrt{a} - y)$. If $d_2 \geq 2$, $j = 3$ and the `auxiv` auxiliary receives $10^{12}z$. If $d_2 \leq 1$ but $10^4d_2 + d_3 \geq 2$, $j = 4$ and the `auxv` auxiliary is called, and receives $10^{16}z$, and so on. In all those cases, the `auxviii` auxiliary is set up to add z to y , then go back to the `auxii` step with continuation `auxiii` (the function we are currently describing). The maximum value of j is 6, regardless of whether $10^{12}d_2 + 10^8d_3 + 10^4d_4 + d_5 \geq 1$. In this last case, we detect when $10^{24}z < 10^7$, which essentially means $\sqrt{a} - y \lesssim 10^{-17}$: once this threshold is reached, there is enough information to find the correctly rounded \sqrt{a} with only one more call to `__fp_sqrt_auxii_o:NnnnnnnnN`. Note that the iteration cannot be stuck before reaching $j = 6$, because for $j < 6$, one has $2 \cdot 10^8 \leq 10^{4(j+1)}(a - y^2)$, hence

$$10^{4j}z \geq \frac{(20000 - 257)(0.5 \cdot 10^8)}{[10^8y + 1]} \geq (20000 - 257) \cdot 0.5 > 0.$$

```

25682 \cs_new:Npn \__fp_sqrt_auxiii_o:wnnnnnnnn
25683   #1; #2#3#4#5#6#7#8#9
25684   {
25685     \if_int_compare:w #1 > \c_one_int
25686       \exp_after:wN \__fp_sqrt_auxiv_o:NNNNNw
25687       \int_value:w \__fp_int_eval:w (#1#2 %)
25688     \else:
25689       \if_int_compare:w #1#2 > \c_one_int
25690         \exp_after:wN \__fp_sqrt_auxv_o:NNNNNw
25691         \int_value:w \__fp_int_eval:w (#1#2#3 %)
25692       \else:
25693         \if_int_compare:w #1#2#3 > \c_one_int
25694           \exp_after:wN \__fp_sqrt_auxvi_o:NNNNNw
25695           \int_value:w \__fp_int_eval:w (#1#2#3#4 %)
25696         \else:
25697           \exp_after:wN \__fp_sqrt_auxvii_o:NNNNNw
25698           \int_value:w \__fp_int_eval:w (#1#2#3#4#5 %)
25699         \fi:
25700       \fi:
25701     \fi:
25702   }
25703 \cs_new:Npn \__fp_sqrt_auxiv_o:NNNNNw 1#1#2#3#4#5#6;
25704   { \__fp_sqrt_auxviii_o:nnnnnnn {#1#2#3#4#5#6} {00000000} }
25705 \cs_new:Npn \__fp_sqrt_auxv_o:NNNNNw 1#1#2#3#4#5#6;
25706   { \__fp_sqrt_auxviii_o:nnnnnnn {000#1#2#3#4#5} {#60000} }
25707 \cs_new:Npn \__fp_sqrt_auxvi_o:NNNNNw 1#1#2#3#4#5#6;
25708   { \__fp_sqrt_auxviii_o:nnnnnnn {0000000#1} {#2#3#4#5#6} }
25709 \cs_new:Npn \__fp_sqrt_auxvii_o:NNNNNw 1#1#2#3#4#5#6;
25710   {
25711     \if_int_compare:w #1#2 = \c_zero_int
25712       \exp_after:wN \__fp_sqrt_auxx_o:Nnnnnnnn
25713     \fi:
25714     \__fp_sqrt_auxviii_o:nnnnnnn {00000000} {000#1#2#3#4#5}
25715   }

```

(End definition for `__fp_sqrt_auxiii_o:wnnnnnnnn` and others.)

`__fp_sqrt_auxviii_o:nnnnnnn` Simply add the two 8-digit blocks of z , aligned to the last four of the five 4-digit blocks of y , then call the `auxii` auxiliary to evaluate $y'^2 = (y + z)^2$.

```

25716 \cs_new:Npn \__fp_sqrt_auxviii_o:nnnnnnn #1#2 #3#4#5#6#7
25717 {
25718   \exp_after:wN \__fp_sqrt_auxix_o:wnwnw
25719   \int_value:w \__fp_int_eval:w #3
25720   \exp_after:wN \__fp_basics_pack_low:NNNNNw
25721   \int_value:w \__fp_int_eval:w #1 + 1#4#5
25722   \exp_after:wN \__fp_basics_pack_low:NNNNNw
25723   \int_value:w \__fp_int_eval:w #2 + 1#6#7 ;
25724 }
25725 \cs_new:Npn \__fp_sqrt_auxix_o:wnwnw #1; #2#3; #4#5;
25726 {
25727   \__fp_sqrt_auxii_o:NnnnnnnnN
25728   \__fp_sqrt_auxiii_o:wnnnnnnnn {#1}{#2}{#3}{#4}{#5}
25729 }

```

(End definition for `__fp_sqrt_auxviii_o:nnnnnnn` and `__fp_sqrt_auxix_o:wnwnw`.)

`__fp_sqrt_auxx_o:Nnnnnnnn` At this stage, $j = 6$ and $10^{24}z < 10^7$, hence
`__fp_sqrt_auxxi_o:wnnnN`

$$10^7 + 1/2 > 10^{24}z + 1/2 \geq (10^{24}(a - y^2) - 258) \cdot (0.5 \cdot 10^8) / (10^8y + 1),$$

then $10^{24}(a - y^2) - 258 < 2(10^7 + 1/2)(y + 10^{-8})$, and

$$10^{24}(a - y^2) < (10^7 + 1290.5)(1 + 10^{-8}/y)(2y) < (10^7 + 1290.5)(1 + 10^{-7})(y + \sqrt{a}),$$

which finally implies $0 \leq \sqrt{a} - y < 0.2 \cdot 10^{-16}$. In particular, y is an underestimate of \sqrt{a} and $y + 0.5 \cdot 10^{-16}$ is a (strict) overestimate. There is at exactly one multiple m of $0.5 \cdot 10^{-16}$ in the interval $[y, y + 0.5 \cdot 10^{-16})$. If $m^2 > a$, then the square root is inexact and is obtained by rounding $m - \epsilon$ to a multiple of 10^{-16} (the precise shift $0 < \epsilon < 0.5 \cdot 10^{-16}$ is irrelevant for rounding). If $m^2 = a$ then the square root is exactly m , and there is no rounding. If $m^2 < a$ then we round $m + \epsilon$. For now, discard a few irrelevant arguments `#1`, `#2`, `#3`, and find the multiple of $0.5 \cdot 10^{-16}$ within $[y, y + 0.5 \cdot 10^{-16})$; rather, only the last 4 digits `#8` of y are considered, and we do not perform any carry yet. The `auxxi` auxiliary sets up `auxii` with a continuation function `auxxii` instead of `auxiii` as before. To prevent `auxii` from giving a negative results $a - m^2$, we compute $a + 10^{-16} - m^2$ instead, always positive since $m < \sqrt{a} + 0.5 \cdot 10^{-16}$ and $a \leq 1 - 10^{-16}$.

```

25730 \cs_new:Npn \__fp_sqrt_auxx_o:Nnnnnnnn #1#2#3 #4#5#6#7#8
25731 {
25732   \exp_after:wN \__fp_sqrt_auxxi_o:wnnnN
25733   \int_value:w \__fp_int_eval:w
25734   (#8 + 2499) / 5000 * 5000 ;
25735   {#4} {#5} {#6} {#7} ;
25736 }
25737 \cs_new:Npn \__fp_sqrt_auxxi_o:wnnnN #1; #2; #3#4#5
25738 {
25739   \__fp_sqrt_auxii_o:NnnnnnnnN
25740   \__fp_sqrt_auxxii_o:nnnnnnnnw
25741   #2 {#1}
25742   {#3} { #4 + 1 } #5
25743 }

```

(End definition for `_fp_sqrt_auxx_o:nnnnnnnw` and `_fp_sqrt_auxxi_o:wnnnN`.)

`_fp_sqrt_auxxii_o:nnnnnnnw`
`_fp_sqrt_auxxiii_o:w`

The difference $0 \leq a + 10^{-16} - m^2 \leq 10^{-16} + (\sqrt{a} - m)(\sqrt{a} + m) \leq 2 \cdot 10^{-16}$ was just computed: its first 8 digits vanish, as do the next four, #1, and most of the following four, #2. The guess m is an overestimate if $a + 10^{-16} - m^2 < 10^{-16}$, that is, #1#2 vanishes. Otherwise it is an underestimate, unless $a + 10^{-16} - m^2 = 10^{-16}$ exactly. For an underestimate, call the `auxxiv` function with argument 9998. For an exact result call it with 9999, and for an overestimate call it with 10000.

```

25744 \cs_new:Npn \_fp_sqrt_auxxii_o:nnnnnnnw 0; #1#2#3#4#5#6#7#8 #9;
25745 {
25746   \if_int_compare:w #1#2 > \c_zero_int
25747     \if_int_compare:w #1#2 = \c_one_int
25748       \if_int_compare:w #3#4 = \c_zero_int
25749         \if_int_compare:w #5#6 = \c_zero_int
25750           \if_int_compare:w #7#8 = \c_zero_int
25751             \_fp_sqrt_auxxiii_o:w
25752           \fi:
25753         \fi:
25754       \fi:
25755     \fi:
25756     \exp_after:wN \_fp_sqrt_auxxiv_o:wnnnnnnnN
25757     \int_value:w 9998
25758   \else:
25759     \exp_after:wN \_fp_sqrt_auxxiv_o:wnnnnnnnN
25760     \int_value:w 10000
25761   \fi:
25762 ;
25763 }
25764 \cs_new:Npn \_fp_sqrt_auxxiii_o:w \fi: \fi: \fi: \fi: #1 \fi: ;
25765 {
25766   \fi: \fi: \fi: \fi: \fi:
25767   \_fp_sqrt_auxxiv_o:wnnnnnnnN 9999 ;
25768 }

```

(End definition for `_fp_sqrt_auxxii_o:nnnnnnnw` and `_fp_sqrt_auxxiii_o:w`.)

`_fp_sqrt_auxxiv_o:wnnnnnnnN`

This receives 9998, 9999 or 10000 as #1 when m is an underestimate, exact, or an overestimate, respectively. Then comes m as five blocks of 4 digits, but where the last block #6 may be 0, 5000, or 10000. In the latter case, we need to add a carry, unless m is an overestimate (#1 is then 10000). Then comes a as three arguments. Rounding is done by `_fp_round:NNN`, whose first argument is the final sign 0 (square roots are positive). We fake its second argument. It should be the last digit kept, but this is only used when ties are “rounded to even”, and only when the result is exactly half-way between two representable numbers rational square roots of numbers with 16 significant digits have: this situation never arises for the square root, as any exact square root of a 16 digit number has at most 8 significant digits. Finally, the last argument is the next digit, possibly shifted by 1 when there are further nonzero digits. This is achieved by `_fp_round_digit:Nw`, which receives (after removal of the 10000’s digit) one of 0000, 0001, 4999, 5000, 5001, or 9999, which it converts to 0, 1, 4, 5, 6, and 9, respectively.

```

25769 \cs_new:Npn \_fp_sqrt_auxxiv_o:wnnnnnnnN #1; #2#3#4#5#6 #7#8#9
25770 {
25771   \exp_after:wN \_fp_basics_pack_high:NNNNNw
25772   \int_value:w \_fp_int_eval:w 1 0000 0000 + #2#3

```

```

25773      \exp_after:wN \__fp_basics_pack_low:NNNNNw
25774      \int_value:w \__fp_int_eval:w 1 0000 0000
25775      + #4#5
25776      \if_int_compare:w #6 > #1 \exp_stop_f: + 1 \fi:
25777      + \exp_after:wN \__fp_round:NNN
25778      \exp_after:wN 0
25779      \exp_after:wN 0
25780      \int_value:w
25781      \exp_after:wN \use_i:nn
25782      \exp_after:wN \__fp_round_digit:Nw
25783      \int_value:w \__fp_int_eval:w #6 + 19999 - #1 ;
25784      \exp_after:wN ;
25785      }

```

(End definition for __fp_sqrt_auxxiv_o:wnnnnnnN.)

72.5 About the sign and exponent

__fp_logb_o:w The exponent of a normal number is its *exponent* minus one.
__fp_logb_aux_o:w

```

25786 \cs_new:Npn \__fp_logb_o:w ? \s__fp \__fp_chk:w #1#2; @
25787 {
25788   \if_case:w #1 \exp_stop_f:
25789     \__fp_case_use:nw
25790     { \__fp_division_by_zero_o:Nnw \c_minus_inf_fp { logb } }
25791   \or:   \exp_after:wN \__fp_logb_aux_o:w
25792   \or:   \__fp_case_return_o:Nw \c_inf_fp
25793   \else: \__fp_case_return_same_o:w
25794   \fi:
25795   \s__fp \__fp_chk:w #1 #2;
25796 }
25797 \cs_new:Npn \__fp_logb_aux_o:w \s__fp \__fp_chk:w #1 #2 #3 #4 ;
25798 {
25799   \exp_after:wN \__fp_parse:n \exp_after:wN
25800   { \int_value:w \int_eval:w #3 - 1 \exp_after:wN }
25801 }

```

(End definition for __fp_logb_o:w and __fp_logb_aux_o:w.)

__fp_sign_o:w Find the sign of the floating point: nan, +0, -0, +1 or -1.
__fp_sign_aux_o:w

```

25802 \cs_new:Npn \__fp_sign_o:w ? \s__fp \__fp_chk:w #1#2; @
25803 {
25804   \if_case:w #1 \exp_stop_f:
25805     \__fp_case_return_same_o:w
25806   \or:   \exp_after:wN \__fp_sign_aux_o:w
25807   \or:   \exp_after:wN \__fp_sign_aux_o:w
25808   \else: \__fp_case_return_same_o:w
25809   \fi:
25810   \s__fp \__fp_chk:w #1 #2;
25811 }
25812 \cs_new:Npn \__fp_sign_aux_o:w \s__fp \__fp_chk:w #1 #2 #3 ;
25813 { \exp_after:wN \__fp_set_sign_o:w \exp_after:wN #2 \c_one_fp @ }

```

(End definition for __fp_sign_o:w and __fp_sign_aux_o:w.)

`_fp_set_sign_o:w` This function is used for the unary minus and for `abs`. It leaves the sign of `nan` invariant, turns negative numbers (sign 2) to positive numbers (sign 0) and positive numbers (sign 0) to positive or negative numbers depending on #1. It also expands after itself in the input stream, just like `_fp+_o:ww`.

```

25814 \cs_new:Npn \_fp_set_sign_o:w #1 \s_fp \_fp_chk:w #2#3#4; @
25815 {
25816   \exp_after:wN \_fp_exp_after_o:w
25817   \exp_after:wN \s_fp
25818   \exp_after:wN \_fp_chk:w
25819   \exp_after:wN #2
25820   \int_value:w
25821   \if_case:w #3 \exp_stop_f: #1 \or: 1 \or: 0 \fi: \exp_stop_f:
25822   #4;
25823 }

```

(End definition for `_fp_set_sign_o:w`.)

72.6 Operations on tuples

`_fp_tuple_set_sign_o:w` Two cases: `abs(<tuple>)` for which #1 is 0 (invalid for tuples) and `-<tuple>` for which #1 is 2. In that case, map over all items in the tuple an auxiliary that dispatches to the type-appropriate sign-flipping function.

```

\fp_tuple_set_sign_aux_o:Nnw
\_fp_tuple_set_sign_aux_o:w
25824 \cs_new:Npn \_fp_tuple_set_sign_o:w #1
25825 {
25826   \if_meaning:w 2 #1
25827     \exp_after:wN \_fp_tuple_set_sign_aux_o:Nnw
25828     \fi:
25829     \_fp_invalid_operation_o:nw { abs }
25830   }
25831 \cs_new:Npn \_fp_tuple_set_sign_aux_o:Nnw #1#2#3 @
25832 { \_fp_tuple_map_o:nw \_fp_tuple_set_sign_aux_o:w #3 }
25833 \cs_new:Npn \_fp_tuple_set_sign_aux_o:w #1#2 ;
25834 {
25835   \_fp_change_func_type:NNN #1 \_fp_set_sign_o:w
25836   \_fp_parse_apply_unary_error:NNw
25837   2 #1 #2 ; @
25838 }

```

(End definition for `_fp_tuple_set_sign_o:w`, `_fp_tuple_set_sign_aux_o:Nnw`, and `_fp_tuple_set_sign_aux_o:w`.)

`_fp*_tuple_o:ww` For `<number>*<tuple>` and `<tuple>*<number>` and `<tuple>/<number>`, loop through the `<tuple>` some code that multiplies or divides by the appropriate `<number>`. Importantly we need to dispatch according to the type, and we make sure to apply the operator in the correct order.

```

\_fp_tuple*_o:ww
\_fp_tuple/_o:ww
25839 \cs_new:cpn { \_fp*_tuple_o:ww } #1 ;
25840 { \_fp_tuple_map_o:nw { \_fp_binary_type_o:Nnw * #1 ; } }
25841 \cs_new:cpn { \_fp_tuple*_o:ww } #1 ; #2 ;
25842 { \_fp_tuple_map_o:nw { \_fp_binary_rev_type_o:Nnw * #2 ; } #1 ; }
25843 \cs_new:cpn { \_fp_tuple/_o:ww } #1 ; #2 ;
25844 { \_fp_tuple_map_o:nw { \_fp_binary_rev_type_o:Nnw / #2 ; } #1 ; }

```

(End definition for `_fp*_tuple_o:ww`, `_fp_tuple*_o:ww`, and `_fp_tuple/_o:ww`.)

`__fp_tuple+_tuple_o:ww` Check the two tuples have the same number of items and map through these a helper
`__fp_tuple-_tuple_o:ww` that dispatches appropriately depending on the types. This means $(1,2)+((1,1),2)$
gives $(\text{nan},4)$.

```

25845 \cs_set_protected:Npn \__fp_tmp:w #1
25846 {
25847   \cs_new:cpn { __fp_tuple_#1_tuple_o:ww }
25848     \s__fp_tuple \__fp_tuple_chk:w ##1 ;
25849     \s__fp_tuple \__fp_tuple_chk:w ##2 ;
25850   {
25851     \int_compare:nNnTF
25852       { \__fp_array_count:n {##1} } = { \__fp_array_count:n {##2} }
25853       { \__fp_tuple_mapthread_o:nww { \__fp_binary_type_o:Nww #1 } }
25854       { \__fp_invalid_operation_o:nww #1 }
25855     \s__fp_tuple \__fp_tuple_chk:w {##1} ;
25856     \s__fp_tuple \__fp_tuple_chk:w {##2} ;
25857   }
25858 }
25859 \__fp_tmp:w +
25860 \__fp_tmp:w -

(End definition for \__fp_tuple+_tuple_o:ww and \__fp_tuple-_tuple_o:ww.)

25861 </package>

```

Chapter 73

l3fp-extended implementation

```
25862 <*package>
25863 <@@=fp>
```

73.1 Description of fixed point numbers

This module provides a few functions to manipulate positive floating point numbers with extended precision (24 digits), but mostly provides functions for fixed-point numbers with this precision (24 digits). Those are used in the computation of Taylor series for the logarithm, exponential, and trigonometric functions. Since we eventually only care about the 16 first digits of the final result, some of the calculations are not performed with the full 24-digit precision. In other words, the last two blocks of each fixed point number may be wrong as long as the error is small enough to be rounded away when converting back to a floating point number. The fixed point numbers are expressed as

$$\{\langle a_1 \rangle\} \{\langle a_2 \rangle\} \{\langle a_3 \rangle\} \{\langle a_4 \rangle\} \{\langle a_5 \rangle\} \{\langle a_6 \rangle\} ;$$

where each $\langle a_i \rangle$ is exactly 4 digits (ranging from 0000 to 9999), except $\langle a_1 \rangle$, which may be any “not-too-large” non-negative integer, with or without leading zeros. Here, “not-too-large” depends on the specific function (see the corresponding comments for details). Checking for overflow is the responsibility of the code calling those functions. The fixed point number a corresponding to the representation above is $a = \sum_{i=1}^6 \langle a_i \rangle \cdot 10^{-4i}$.

Most functions we define here have the form

```
\__fp_fixed_<calculation>:wnn <operand1> ; <operand2> ; {\<continuation>}
```

They perform the $\langle calculation \rangle$ on the two $\langle operands \rangle$, then feed the result (6 brace groups followed by a semicolon) to the $\langle continuation \rangle$, responsible for the next step of the calculation. Some functions only accept an N-type $\langle continuation \rangle$. This allows constructions such as

```
\__fp_fixed_add:wnn <X1> ; <X2> ;
\__fp_fixed_mul:wnn <X3> ;
\__fp_fixed_add:wnn <X4> ;
```


to compute $(X_1 + X_2) \cdot X_3 + X_4$. This turns out to be very appropriate for computing continued fractions and Taylor series.

At the end of the calculation, the result is turned back to a floating point number using `__fp_fixed_to_float_o:wN`. This function has to change the exponent of the floating point number: it must be used after starting an integer expression for the overall exponent of the result.

73.2 Helpers for numbers with extended precision

`\c__fp_one_fixed_tl` The fixed-point number 1, used in `l3fp-expo`.

```
25864 \tl_const:Nn \c__fp_one_fixed_tl
25865 { {10000} {0000} {0000} {0000} {0000} {0000} ; }
```

(End definition for `\c__fp_one_fixed_tl`.)

`__fp_fixed_continue:wn` This function simply calls the next function.

```
25866 \cs_new:Npn \__fp_fixed_continue:wn #1; #2 { #2 #1; }
```

(End definition for `__fp_fixed_continue:wn`.)

`__fp_fixed_add_one:wN` `__fp_fixed_add_one:wN <a> ; <continuation>`

This function adds 1 to the fixed point $\langle a \rangle$, by changing a_1 to $10000 + a_1$, then calls the $\langle continuation \rangle$. This requires $a_1 + 10000 < 2^{31}$.

```
25867 \cs_new:Npn \__fp_fixed_add_one:wN #1#2; #3
25868 {
25869   \exp_after:wN #3 \exp_after:wN
25870   { \int_value:w \__fp_int_eval:w \c__fp_myriad_int + #1 } #2 ;
25871 }
```

(End definition for `__fp_fixed_add_one:wN`.)

`__fp_fixed_div_myriad:wn` Divide a fixed point number by 10000. This is a little bit more subtle than just removing the last group and adding a leading group of zeros: the first group `#1` may have any number of digits, and we must split `#1` into the new first group and a second group of exactly 4 digits. The choice of shifts allows `#1` to be in the range $[0, 5 \cdot 10^8 - 1]$.

```
25872 \cs_new:Npn \__fp_fixed_div_myriad:wn #1#2#3#4#5#6;
25873 {
25874   \exp_after:wN \__fp_fixed_mul_after:wnn
25875   \int_value:w \__fp_int_eval:w \c__fp_leading_shift_int
25876   \exp_after:wN \__fp_pack:NNNNNw
25877   \int_value:w \__fp_int_eval:w \c__fp_trailing_shift_int
25878   + #1 ; {#2}{#3}{#4}{#5};
25879 }
```

(End definition for `__fp_fixed_div_myriad:wn`.)

`__fp_fixed_mul_after:wnn` The fixed point operations which involve multiplication end by calling this auxiliary. It braces the last block of digits, and places the $\langle continuation \rangle$ `#3` in front.

```
25880 \cs_new:Npn \__fp_fixed_mul_after:wnn #1; #2; #3 { #3 {#1} #2; }
```

(End definition for `__fp_fixed_mul_after:wnn`.)

73.3 Multiplying a fixed point number by a short one

`_fp_fixed_mul_short:wnn`

```
\_fp_fixed_mul_short:wnn
  {⟨a1⟩} {⟨a2⟩} {⟨a3⟩} {⟨a4⟩} {⟨a5⟩} {⟨a6⟩} ;
  {⟨b0⟩} {⟨b1⟩} {⟨b2⟩} ; {⟨continuation⟩}
```

Computes the product $c = ab$ of $a = \sum_i \langle a_i \rangle 10^{-4i}$ and $b = \sum_i \langle b_i \rangle 10^{-4i}$, rounds it to the closest multiple of 10^{-24} , and leaves $\langle continuation \rangle$ $\{ \langle c_1 \rangle \} \dots \{ \langle c_6 \rangle \}$; in the input stream, where each of the $\langle c_i \rangle$ are blocks of 4 digits, except $\langle c_1 \rangle$, which is any TeX integer. Note that indices for $\langle b \rangle$ start at 0: for instance a second operand of $\{0001\}\{0000\}$ leaves the first operand unchanged (rather than dividing it by 10^4 , as `_fp_fixed_mul:wnn` would).

```
25881 \cs_new:Npn \_fp_fixed_mul_short:wnn #1#2#3#4#5#6; #7#8#9;
25882 {
25883   \exp_after:wN \_fp_fixed_mul_after:wnn
25884   \int_value:w \_fp_int_eval:w \c\_fp_leading_shift_int
25885     + #1*#7
25886   \exp_after:wN \_fp_pack:NNNNNw
25887   \int_value:w \_fp_int_eval:w \c\_fp_middle_shift_int
25888     + #1*#8 + #2*#7
25889   \exp_after:wN \_fp_pack:NNNNNw
25890   \int_value:w \_fp_int_eval:w \c\_fp_middle_shift_int
25891     + #1*#9 + #2*#8 + #3*#7
25892   \exp_after:wN \_fp_pack:NNNNNw
25893   \int_value:w \_fp_int_eval:w \c\_fp_middle_shift_int
25894     + #2*#9 + #3*#8 + #4*#7
25895   \exp_after:wN \_fp_pack:NNNNNw
25896   \int_value:w \_fp_int_eval:w \c\_fp_middle_shift_int
25897     + #3*#9 + #4*#8 + #5*#7
25898   \exp_after:wN \_fp_pack:NNNNNw
25899   \int_value:w \_fp_int_eval:w \c\_fp_trailing_shift_int
25900     + #4*#9 + #5*#8 + #6*#7
25901     + ( #5*#9 + #6*#8 + #6*#9 / \c\_fp_myriad_int )
25902     / \c\_fp_myriad_int ; ;
25903 }
```

(End definition for `_fp_fixed_mul_short:wnn`.)

73.4 Dividing a fixed point number by a small integer

`_fp_fixed_div_int:wnN`

`_fp_fixed_div_int:wnN`

`_fp_fixed_div_int_auxi:wnn`

`_fp_fixed_div_int_auxii:wnn`

`_fp_fixed_div_int_pack:Nw`

`_fp_fixed_div_int_after:Nw`

```
\_fp_fixed_div_int:wnN ⟨a⟩ ; ⟨n⟩ ; ⟨continuation⟩
```

Divides the fixed point number $\langle a \rangle$ by the (small) integer $0 < \langle n \rangle < 10^4$ and feeds the result to the $\langle continuation \rangle$. There is no bound on a_1 .

The arguments of the `i` auxiliary are 1: one of the a_i , 2: n , 3: the `ii` or the `iii` auxiliary. It computes a (somewhat tight) lower bound Q_i for the ratio a_i/n .

The `ii` auxiliary receives Q_i , n , and a_i as arguments. It adds Q_i to a surrounding integer expression, and starts a new one with the initial value 9999, which ensures that the result of this expression has 5 digits. The auxiliary also computes $a_i - n \cdot Q_i$, placing the result in front of the 4 digits of a_{i+1} . The resulting $a'_{i+1} = 10^4(a_i - n \cdot Q_i) + a_{i+1}$ serves as the first argument for a new call to the `i` auxiliary.

When the `iii` auxiliary is called, the situation looks like this:

```

\__fp_fixed_div_int_after:Nw <continuation>
-1 + Q1
\__fp_fixed_div_int_pack:Nw 9999 + Q2
\__fp_fixed_div_int_pack:Nw 9999 + Q3
\__fp_fixed_div_int_pack:Nw 9999 + Q4
\__fp_fixed_div_int_pack:Nw 9999 + Q5
\__fp_fixed_div_int_pack:Nw 9999
\__fp_fixed_div_int_auxii:wnn Q6 ; {<n>} {<a6

```

where expansion is happening from the last line up. The iii auxiliary adds $Q_6 + 2 \simeq a_6/n + 1$ to the last 9999, giving the integer closest to $10000 + a_6/n$.

Each **pack** auxiliary receives 5 digits followed by a semicolon. The first digit is added as a carry to the integer expression above, and the 4 other digits are braced. Each call to the **pack** auxiliary thus produces one brace group. The last brace group is produced by the **after** auxiliary, which places the *<continuation>* as appropriate.

```

25904 \cs_new:Npn \__fp_fixed_div_int:wwN #1#2#3#4#5#6 ; #7 ; #8
25905 {
25906   \exp_after:wN \__fp_fixed_div_int_after:Nw
25907   \exp_after:wN #8
25908   \int_value:w \__fp_int_eval:w - 1
25909   \__fp_fixed_div_int:wnN
25910   #1; {#7} \__fp_fixed_div_int_auxi:wnn
25911   #2; {#7} \__fp_fixed_div_int_auxi:wnn
25912   #3; {#7} \__fp_fixed_div_int_auxi:wnn
25913   #4; {#7} \__fp_fixed_div_int_auxi:wnn
25914   #5; {#7} \__fp_fixed_div_int_auxi:wnn
25915   #6; {#7} \__fp_fixed_div_int_auxii:wnn ;
25916 }
25917 \cs_new:Npn \__fp_fixed_div_int:wnN #1; #2 #3
25918 {
25919   \exp_after:wN #3
25920   \int_value:w \__fp_int_eval:w #1 / #2 - 1 ;
25921   {#2}
25922   {#1}
25923 }
25924 \cs_new:Npn \__fp_fixed_div_int_auxi:wnn #1; #2 #3
25925 {
25926   + #1
25927   \exp_after:wN \__fp_fixed_div_int_pack:Nw
25928   \int_value:w \__fp_int_eval:w 9999
25929   \exp_after:wN \__fp_fixed_div_int:wnN
25930   \int_value:w \__fp_int_eval:w #3 - #1*#2 \__fp_int_eval_end:
25931 }
25932 \cs_new:Npn \__fp_fixed_div_int_auxii:wnn #1; #2 #3 { + #1 + 2 ; }
25933 \cs_new:Npn \__fp_fixed_div_int_pack:Nw #1 #2; { + #1; {#2} }
25934 \cs_new:Npn \__fp_fixed_div_int_after:Nw #1 #2; { #1 {#2} }

```

(End definition for `__fp_fixed_div_int:wwN` and others.)

73.5 Adding and subtracting fixed points

```

\__fp_fixed_add:wnn          \__fp_fixed_add:wnn <a> ; <b> ; {<continuation>}
\__fp_fixed_sub:wnn
\__fp_fixed_add:Nnnnnwnn
\__fp_fixed_add:nnNnnnwn
\__fp_fixed_add_pack:NNNNNwn
\__fp_fixed_add_after:NNNNNwn

```

Computes $a+b$ (resp. $a-b$) and feeds the result to the $\langle continuation \rangle$. This function requires $0 \leq a_1, b_1 \leq 114748$, its result must be positive (this happens automatically for addition) and its first group must have at most 5 digits: $(a \pm b)_1 < 100000$. The two functions only differ by a sign, hence use a common auxiliary. It would be nice to grab the 12 brace groups in one go; only 9 parameters are allowed. Start by grabbing the sign, a_1, \dots, a_4 , the rest of a , and b_1 and b_2 . The second auxiliary receives the rest of a , the sign multiplying b , the rest of b , and the $\langle continuation \rangle$ as arguments. After going down through the various level, we go back up, packing digits and bringing the $\langle continuation \rangle$ (#8, then #7) from the end of the argument list to its start.

```

25935 \cs_new:Npn \__fp_fixed_add:wnn { \__fp_fixed_add:Nnnnnwnn + }
25936 \cs_new:Npn \__fp_fixed_sub:wnn { \__fp_fixed_add:Nnnnnwnn - }
25937 \cs_new:Npn \__fp_fixed_add:Nnnnnwnn #1 #2#3#4#5 #6; #7#8
25938 {
25939   \exp_after:wN \__fp_fixed_add_after:NNNNNwn
25940   \int_value:w \__fp_int_eval:w 9 9999 9998 + #2#3 #1 #7#8
25941   \exp_after:wN \__fp_fixed_add_pack:NNNNNwn
25942   \int_value:w \__fp_int_eval:w 1 9999 9998 + #4#5
25943   \__fp_fixed_add:nnNnnwn #6 #1
25944 }
25945 \cs_new:Npn \__fp_fixed_add:nnNnnwn #1#2 #3 #4#5 #6#7 ; #8
25946 {
25947   #3 #4#5
25948   \exp_after:wN \__fp_fixed_add_pack:NNNNNwn
25949   \int_value:w \__fp_int_eval:w 2 0000 0000 #3 #6#7 + #1#2 ; {#8} ;
25950 }
25951 \cs_new:Npn \__fp_fixed_add_pack:NNNNNwn #1 #2#3#4#5 #6; #7
25952 { + #1 ; {#7} {#2#3#4#5} {#6} }
25953 \cs_new:Npn \__fp_fixed_add_after:NNNNNwn 1 #1 #2#3#4#5 #6; #7
25954 { #7 {#1#2#3#4#5} {#6} }

```

(End definition for `__fp_fixed_add:wnn` and others.)

73.6 Multiplying fixed points

```

\__fp_fixed_mul:wnn
\__fp_fixed_mul:nnnnnnnw

```

`__fp_fixed_mul:wnn` $\langle a \rangle$; $\langle b \rangle$; $\{\langle continuation \rangle\}$

Computes $a \times b$ and feeds the result to $\langle continuation \rangle$. This function requires $0 \leq a_1, b_1 < 10000$. Once more, we need to play around the limit of 9 arguments for \TeX macros. Note that we don't need to obtain an exact rounding, contrarily to the $*$ operator, so things could be harder. We wish to perform carries in

$$\begin{aligned}
a \times b = & a_1 \cdot b_1 \cdot 10^{-8} \\
& + (a_1 \cdot b_2 + a_2 \cdot b_1) \cdot 10^{-12} \\
& + (a_1 \cdot b_3 + a_2 \cdot b_2 + a_3 \cdot b_1) \cdot 10^{-16} \\
& + (a_1 \cdot b_4 + a_2 \cdot b_3 + a_3 \cdot b_2 + a_4 \cdot b_1) \cdot 10^{-20} \\
& + \left(a_2 \cdot b_4 + a_3 \cdot b_3 + a_4 \cdot b_2 \right. \\
& \quad \left. + \frac{a_3 \cdot b_4 + a_4 \cdot b_3 + a_1 \cdot b_6 + a_2 \cdot b_5 + a_5 \cdot b_2 + a_6 \cdot b_1}{10^4} \right. \\
& \quad \left. + a_1 \cdot b_5 + a_5 \cdot b_1 \right) \cdot 10^{-24} + O(10^{-24}),
\end{aligned}$$

where the $O(10^{-24})$ stands for terms which are at most $5 \cdot 10^{-24}$; ignoring those leads to an error of at most 5 ulp. Note how the first 15 terms only depend on a_1, \dots, a_4 and b_1, \dots, b_4 , while the last 6 terms only depend on a_1, a_2, a_5, a_6 , and the corresponding parts of b . Hence, the first function grabs a_1, \dots, a_4 , the rest of a , and b_1, \dots, b_4 , and writes the 15 first terms of the expression, including a left parenthesis for the fraction. The `i` auxiliary receives $a_5, a_6, b_1, b_2, a_1, a_2, b_5, b_6$ and finally the $\langle continuation \rangle$ as arguments. It writes the end of the expression, including the right parenthesis and the denominator of the fraction. The $\langle continuation \rangle$ is finally placed in front of the 6 brace groups by `_fp_fixed_mul_after:wwn`.

```

25955 \cs_new:Npn \_fp\_fixed\_mul:wwn #1#2#3#4 #5; #6#7#8#9
25956 {
25957   \exp\_after:wN \_fp\_fixed\_mul\_after:wwn
25958   \int\_value:w \_fp\_int\_eval:w \c\_fp\_leading\_shift\_int
25959   \exp\_after:wN \_fp\_pack:NNNNNw
25960   \int\_value:w \_fp\_int\_eval:w \c\_fp\_middle\_shift\_int
25961   + #1*#6
25962   \exp\_after:wN \_fp\_pack:NNNNNw
25963   \int\_value:w \_fp\_int\_eval:w \c\_fp\_middle\_shift\_int
25964   + #1*#7 + #2*#6
25965   \exp\_after:wN \_fp\_pack:NNNNNw
25966   \int\_value:w \_fp\_int\_eval:w \c\_fp\_middle\_shift\_int
25967   + #1*#8 + #2*#7 + #3*#6
25968   \exp\_after:wN \_fp\_pack:NNNNNw
25969   \int\_value:w \_fp\_int\_eval:w \c\_fp\_middle\_shift\_int
25970   + #1*#9 + #2*#8 + #3*#7 + #4*#6
25971   \exp\_after:wN \_fp\_pack:NNNNNw
25972   \int\_value:w \_fp\_int\_eval:w \c\_fp\_trailing\_shift\_int
25973   + #2*#9 + #3*#8 + #4*#7
25974   + ( #3*#9 + #4*#8
25975     + \_fp\_fixed\_mul:nnnnnnnw #5 {#6}{#7} {#1}{#2}
25976   )
25977 \cs\_new:Npn \_fp\_fixed\_mul:nnnnnnnw #1#2 #3#4 #5#6 #7#8 ;
25978 {
25979   #1*#4 + #2*#3 + #5*#8 + #6*#7 ) / \c\_fp\_myriad\_int
25980   + #1*#3 + #5*#7 ; ;
25981 }

```

(End definition for `_fp_fixed_mul:wwn` and `_fp_fixed_mul:nnnnnnnw`.)

73.7 Combining product and sum of fixed points

```

\_fp\_fixed\_mul\_add:wwwn <a> ; <b> ; <c> ; {<continuation>}
\_fp\_fixed\_mul\_sub\_back:wwwn <a> ; <b> ; <c> ; {<continuation>}
\_fp\_fixed\_one\_minus\_mul:wwn <a> ; <b> ; {<continuation>}
\_fp\_fixed\_mul\_one\_minus\_mul:wwn

```

Sometimes called FMA (fused multiply-add), these functions compute $a \times b + c$, $c - a \times b$, and $1 - a \times b$ and feed the result to the $\langle continuation \rangle$. Those functions require $0 \leq a_1, b_1, c_1 \leq 10000$. Since those functions are at the heart of the computation of Taylor expansions, we over-optimize them a bit, and in particular we do not factor out the common parts of the three functions.

For definiteness, consider the task of computing $a \times b + c$. We perform carries in

$$\begin{aligned}
a \times b + c = & (a_1 \cdot b_1 + c_1 c_2) \cdot 10^{-8} \\
& + (a_1 \cdot b_2 + a_2 \cdot b_1) \cdot 10^{-12} \\
& + (a_1 \cdot b_3 + a_2 \cdot b_2 + a_3 \cdot b_1 + c_3 c_4) \cdot 10^{-16} \\
& + (a_1 \cdot b_4 + a_2 \cdot b_3 + a_3 \cdot b_2 + a_4 \cdot b_1) \cdot 10^{-20} \\
& + \left(a_2 \cdot b_4 + a_3 \cdot b_3 + a_4 \cdot b_2 \right. \\
& \quad \left. + \frac{a_3 \cdot b_4 + a_4 \cdot b_3 + a_1 \cdot b_6 + a_2 \cdot b_5 + a_5 \cdot b_2 + a_6 \cdot b_1}{10^4} \right. \\
& \quad \left. + a_1 \cdot b_5 + a_5 \cdot b_1 + c_5 c_6 \right) \cdot 10^{-24} + O(10^{-24}),
\end{aligned}$$

where $c_1 c_2$, $c_3 c_4$, $c_5 c_6$ denote the 8-digit number obtained by juxtaposing the two blocks of digits of c , and \cdot denotes multiplication. The task is obviously tough because we have 18 brace groups in front of us.

Each of the three function starts the first two levels (the first, corresponding to 10^{-4} , is empty), with $c_1 c_2$ in the first level, calls the `i` auxiliary with arguments described later, and adds a trailing $+ c_5 c_6$; $\{\langle continuation \rangle\}$; . The $+ c_5 c_6$ piece, which is omitted for `_fp_fixed_one_minus_mul:wnn`, is taken in the integer expression for the 10^{-24} level.

```

25982 \cs_new:Npn \_fp\_fixed\_mul\_add:wwn #1; #2; #3#4#5#6#7#8;
25983 {
25984   \exp\_after:wN \_fp\_fixed\_mul\_after:wnn
25985   \int\_value:w \_fp\_int\_eval:w \c\_fp\_big\_leading\_shift\_int
25986   \exp\_after:wN \_fp\_pack\_big:NNNNNNw
25987   \int\_value:w \_fp\_int\_eval:w \c\_fp\_big\_middle\_shift\_int + #3 #4
25988   \_fp\_fixed\_mul\_add:Nwnnnwnnn +
25989   + #5 #6 ; #2 ; #1 ; #2 ; +
25990   + #7 #8 ; ;
25991 }
25992 \cs_new:Npn \_fp\_fixed\_mul\_sub\_back:wwn #1; #2; #3#4#5#6#7#8;
25993 {
25994   \exp\_after:wN \_fp\_fixed\_mul\_after:wnn
25995   \int\_value:w \_fp\_int\_eval:w \c\_fp\_big\_leading\_shift\_int
25996   \exp\_after:wN \_fp\_pack\_big:NNNNNNw
25997   \int\_value:w \_fp\_int\_eval:w \c\_fp\_big\_middle\_shift\_int + #3 #4
25998   \_fp\_fixed\_mul\_add:Nwnnnwnnn -
25999   + #5 #6 ; #2 ; #1 ; #2 ; -
26000   + #7 #8 ; ;
26001 }
26002 \cs_new:Npn \_fp\_fixed\_one\_minus\_mul:wnn #1; #2;
26003 {
26004   \exp\_after:wN \_fp\_fixed\_mul\_after:wnn
26005   \int\_value:w \_fp\_int\_eval:w \c\_fp\_big\_leading\_shift\_int
26006   \exp\_after:wN \_fp\_pack\_big:NNNNNNw
26007   \int\_value:w \_fp\_int\_eval:w \c\_fp\_big\_middle\_shift\_int +
26008   1 0000 0000
26009   \_fp\_fixed\_mul\_add:Nwnnnwnnn -
26010   ; #2 ; #1 ; #2 ; -
26011   ; ;
26012 }

```

(End definition for `_fp_fixed_mul_add:www`, `_fp_fixed_mul_sub_back:www`, and `_fp_fixed_mul_one_minus_mul:wn`.)

`_fp_fixed_mul_add:Nwnnnwnnn` $\langle op \rangle + \langle c_3 \rangle \langle c_4 \rangle ;$
 $\langle b \rangle ; \langle a \rangle ; \langle b \rangle ; \langle op \rangle$
 $+ \langle c_5 \rangle \langle c_6 \rangle ;$

Here, $\langle op \rangle$ is either $+$ or $-$. Arguments #3, #4, #5 are $\langle b_1 \rangle, \langle b_2 \rangle, \langle b_3 \rangle$; arguments #7, #8, #9 are $\langle a_1 \rangle, \langle a_2 \rangle, \langle a_3 \rangle$. We can build three levels: $a_1 \cdot b_1$ for 10^{-8} , $(a_1 \cdot b_2 + a_2 \cdot b_1)$ for 10^{-12} , and $(a_1 \cdot b_3 + a_2 \cdot b_2 + a_3 \cdot b_1 + c_3 c_4)$ for 10^{-16} . The $a-b$ products use the sign #1. Note that #2 is empty for `_fp_fixed_one_minus_mul:wn`. We call the *ii* auxiliary for levels 10^{-20} and 10^{-24} , keeping the pieces of $\langle a \rangle$ we've read, but not $\langle b \rangle$, since there is another copy later in the input stream.

```
26013 \cs_new:Npn \_fp\_fixed\_mul\_add:Nwnnnwnnn #1 #2; #3#4#5#6; #7#8#9
26014 {
26015   #1 #7*#3
26016   \exp_after:wN \_fp\_pack\_big:NNNNNNw
26017   \int_value:w \_fp\_int\_eval:w \c\_fp\_big\_middle\_shift\_int
26018   #1 #7*#4 #1 #8*#3
26019   \exp_after:wN \_fp\_pack\_big:NNNNNNw
26020   \int_value:w \_fp\_int\_eval:w \c\_fp\_big\_middle\_shift\_int
26021   #1 #7*#5 #1 #8*#4 #1 #9*#3 #2
26022   \exp_after:wN \_fp\_pack\_big:NNNNNNw
26023   \int_value:w \_fp\_int\_eval:w \c\_fp\_big\_middle\_shift\_int
26024   #1 \_fp\_fixed\_mul\_add:nnnnwnnn {#7}{#8}{#9}
26025 }
```

(End definition for `_fp_fixed_mul_add:Nwnnnwnnn`.)

`_fp_fixed_mul_add:nnnnwnnn` $\langle a \rangle ; \langle b \rangle ; \langle op \rangle$
 $+ \langle c_5 \rangle \langle c_6 \rangle ;$

Level 10^{-20} is $(a_1 \cdot b_4 + a_2 \cdot b_3 + a_3 \cdot b_2 + a_4 \cdot b_1)$, multiplied by the sign, which was inserted by the *i* auxiliary. Then we prepare level 10^{-24} . We don't have access to all parts of $\langle a \rangle$ and $\langle b \rangle$ needed to make all products. Instead, we prepare the partial expressions

$$b_1 + a_4 \cdot b_2 + a_3 \cdot b_3 + a_2 \cdot b_4 + a_1$$

$$b_2 + a_4 \cdot b_3 + a_3 \cdot b_4 + a_2.$$

Obviously, those expressions make no mathematical sense: we complete them with $a_5 \cdot$ and $\cdot b_5$, and with $a_6 \cdot b_1 + a_5 \cdot$ and $\cdot b_5 + a_1 \cdot b_6$, and of course with the trailing $+ c_5 c_6$. To do all this, we keep a_1, a_5, a_6 , and the corresponding pieces of $\langle b \rangle$.

```
26026 \cs_new:Npn \_fp\_fixed\_mul\_add:nnnnwnnn #1#2#3#4#5; #6#7#8#9
26027 {
26028   ( #1*#9 + #2*#8 + #3*#7 + #4*#6 )
26029   \exp_after:wN \_fp\_pack\_big:NNNNNNw
26030   \int_value:w \_fp\_int\_eval:w \c\_fp\_big\_trailing\_shift\_int
26031   \_fp\_fixed\_mul\_add:nnnnwnnnwN
26032   { #6 + #4*#7 + #3*#8 + #2*#9 + #1 }
26033   { #7 + #4*#8 + #3*#9 + #2 }
26034   {#1} #5;
26035   {#6}
26036 }
```

(End definition for _fp_fixed_mul_add:nnnnwnnnn.)

```

\_fp_fixed_mul_add:nnnnwnnnN
    \_fp_fixed_mul_add:nnnnwnnnN {<partial1>} {<partial2>}
    {<a1>} {<a5>} {<a6>} ; {<b1>} {<b5>} {<b6>} ;
    <op> + <c5> <c6> ;

```

Complete the $\langle partial_1 \rangle$ and $\langle partial_2 \rangle$ expressions as explained for the `ii` auxiliary. The second one is divided by 10000: this is the carry from level 10^{-28} . The trailing $+ c_5 c_6$ is taken into the expression for level 10^{-24} . Note that the total of level 10^{-24} is in the interval $[-5 \cdot 10^8, 6 \cdot 10^8]$ (give or take a couple of 10000), hence adding it to the shift gives a 10-digit number, as expected by the packing auxiliaries. See `l3fp-aux` for the definition of the shifts and packing auxiliaries.

```

26037 \cs_new:Npn \_fp_fixed_mul_add:nnnnwnnnN #1#2 #3#4#5; #6#7#8; #9
26038 {
26039     #9 (#4* #1 *#7)
26040     #9 (#5*#6+#4* #2 *#7+#3*#8) / \c__fp_myriad_int
26041 }

```

(End definition for _fp_fixed_mul_add:nnnnwnnnN.)

73.8 Extended-precision floating point numbers

In this section we manipulate floating point numbers with roughly 24 significant figures (“extended-precision” numbers, in short, “ep”), which take the form of an integer exponent, followed by a comma, then six groups of digits, ending with a semicolon. The first group of digit may be any non-negative integer, while other groups of digits have 4 digits. In other words, an extended-precision number is an exponent ending in a comma, then a fixed point number. The corresponding value is $0.\langle digits \rangle \cdot 10^{\langle exponent \rangle}$. This convention differs from floating points.

```

\_fp_ep_to_fixed:wwn
\_fp_ep_to_fixed_auxi:www
\_fp_ep_to_fixed_auxii:nnnnnnnnwn

```

Converts an extended-precision number with an exponent at most 4 and a first block less than 10^8 to a fixed point number whose first block has 12 digits, hopefully starting with many zeros.

```

26042 \cs_new:Npn \_fp_ep_to_fixed:wwn #1,#2
26043 {
26044     \exp_after:wN \_fp_ep_to_fixed_auxi:www
26045     \int_value:w \_fp_int_eval:w 1 0000 0000 + #2 \exp_after:wN ;
26046     \exp:w \exp_end_continue_f:w
26047     \prg_replicate:nn { 4 - \int_max:nn {#1} { -32 } } { 0 } ;
26048 }
26049 \cs_new:Npn \_fp_ep_to_fixed_auxi:www #1#; #2; #3#4#5#6#7;
26050 {
26051     \_fp_pack_eight:wNNNNNNNN
26052     \_fp_pack_twice_four:wNNNNNNNN
26053     \_fp_pack_twice_four:wNNNNNNNN
26054     \_fp_pack_twice_four:wNNNNNNNN
26055     \_fp_ep_to_fixed_auxii:nnnnnnnnwn ;
26056     #2 #1#3#4#5#6#7 0000 !
26057 }
26058 \cs_new:Npn \_fp_ep_to_fixed_auxii:nnnnnnnnwn #1#2#3#4#5#6#7; #8! #9
26059 { #9 {#1#2}{#3}{#4}{#5}{#6}{#7}; }

```

(End definition for _fp_ep_to_fixed:wwn, _fp_ep_to_fixed_auxi:www, and _fp_ep_to_fixed_auxii:nnnnnnnnwn.)


```

\__fp_ep_to_ep:wwN
\__fp_ep_to_ep_loop:N
\__fp_ep_to_ep_end:www
\__fp_ep_to_ep_zero:ww

```

Normalize an extended-precision number. More precisely, leading zeros are removed from the mantissa of the argument, decreasing its exponent as appropriate. Then the digits are packed into 6 groups of 4 (discarding any remaining digit, not rounding). Finally, the continuation #8 is placed before the resulting exponent-mantissa pair. The input exponent may in fact be given as an integer expression. The `loop` auxiliary grabs a digit: if it is 0, decrement the exponent and continue looping, and otherwise call the `end` auxiliary, which places all digits in the right order (the digit that was not 0, and any remaining digits), followed by some 0, then packs them up neatly in $3 \times 2 = 6$ blocks of four. At the end of the day, remove with `__fp_use_i:ww` any digit that did not make it in the final mantissa (typically only zeros, unless the original first block has more than 4 digits).

```

26060 \cs_new:Npn \__fp_ep_to_ep:wwN #1,#2#3#4#5#6#7; #8
26061 {
26062   \exp_after:wN #8
26063   \int_value:w \__fp_int_eval:w #1 + 4
26064   \exp_after:wN \use_i:nn
26065   \exp_after:wN \__fp_ep_to_ep_loop:N
26066   \int_value:w \__fp_int_eval:w 1 0000 0000 + #2 \__fp_int_eval_end:
26067   #3#4#5#6#7 ; ; !
26068 }
26069 \cs_new:Npn \__fp_ep_to_ep_loop:N #1
26070 {
26071   \if_meaning:w 0 #1
26072   - 1
26073   \else:
26074     \__fp_ep_to_ep_end:www #1
26075   \fi:
26076   \__fp_ep_to_ep_loop:N
26077 }
26078 \cs_new:Npn \__fp_ep_to_ep_end:www
26079 #1 \fi: \__fp_ep_to_ep_loop:N #2; #3!
26080 {
26081   \fi:
26082   \if_meaning:w ; #1
26083   - 2 * \c__fp_max_exponent_int
26084   \__fp_ep_to_ep_zero:ww
26085   \fi:
26086   \__fp_pack_twice_four:wNNNNNNNN
26087   \__fp_pack_twice_four:wNNNNNNNN
26088   \__fp_pack_twice_four:wNNNNNNNN
26089   \__fp_use_i:ww , ;
26090   #1 #2 0000 0000 0000 0000 0000 0000 ;
26091 }
26092 \cs_new:Npn \__fp_ep_to_ep_zero:ww \fi: #1; #2; #3;
26093 { \fi: , {1000}{0000}{0000}{0000}{0000}{0000} ; }

```

(End definition for `__fp_ep_to_ep:wwN` and others.)

```

\__fp_ep_compare:www
\__fp_ep_compare_aux:www

```

In `l3fp-trig` we need to compare two extended-precision numbers. This is based on the same function for positive floating point numbers, with an extra test if comparing only 16 decimals is not enough to distinguish the numbers. Note that this function only works if the numbers are normalized so that their first block is in `[1000,9999]`.

```

26094 \cs_new:Npn \__fp_ep_compare:www #1,#2#3#4#5#6#7;

```

```

26095 { \_fp_ep_compare_aux:www {#1}{#2}{#3}{#4}{#5}; #6#7; }
26096 \cs_new:Npn \_fp_ep_compare_aux:www #1;#2;#3,#4#5#6#7#8#9;
26097 {
26098   \if_case:w
26099     \_fp_compare_npos:nwnw #1; {#3}{#4}{#5}{#6}{#7}; \exp_stop_f:
26100     \if_int_compare:w #2 = #8#9 \exp_stop_f:
26101       0
26102     \else:
26103       \if_int_compare:w #2 < #8#9 - \fi: 1
26104     \fi:
26105   \or: 1
26106   \else: -1
26107   \fi:
26108 }

```

(End definition for _fp_ep_compare:www and _fp_ep_compare_aux:www.)

_fp_ep_mul:wwwN
_fp_ep_mul_raw:wwwN

Multiply two extended-precision numbers: first normalize them to avoid losing too much precision, then multiply the mantissas #2 and #4 as fixed point numbers, and sum the exponents #1 and #3. The result's first block is in [100,9999].

```

26109 \cs_new:Npn \_fp_ep_mul:wwwN #1,#2; #3,#4;
26110 {
26111   \_fp_ep_to_ep:wwN #3,#4;
26112   \_fp_fixed_continue:wn
26113   {
26114     \_fp_ep_to_ep:wwN #1,#2;
26115     \_fp_ep_mul_raw:wwwN
26116   }
26117   \_fp_fixed_continue:wn
26118 }
26119 \cs_new:Npn \_fp_ep_mul_raw:wwwN #1,#2; #3,#4; #5
26120 {
26121   \_fp_fixed_mul:wn #2; #4;
26122   { \exp_after:wN #5 \int_value:w \_fp_int_eval:w #1 + #3 , }
26123 }

```

(End definition for _fp_ep_mul:wwwN and _fp_ep_mul_raw:wwwN.)

73.9 Dividing extended-precision numbers

Divisions of extended-precision numbers are difficult to perform with exact rounding: the technique used in l3fp-basics for 16-digit floating point numbers does not generalize easily to 24-digit numbers. Thankfully, there is no need for exact rounding.

Let us call $\langle n \rangle$ the numerator and $\langle d \rangle$ the denominator. After a simple normalization step, we can assume that $\langle n \rangle \in [0.1, 1)$ and $\langle d \rangle \in [0.1, 1)$, and compute $\langle n \rangle / (10 \langle d \rangle) \in (0.01, 1)$. In terms of the 6 blocks of digits $\langle n_1 \rangle \cdots \langle n_6 \rangle$ and the 6 blocks $\langle d_1 \rangle \cdots \langle d_6 \rangle$, the condition translates to $\langle n_1 \rangle, \langle d_1 \rangle \in [1000, 9999]$.

We first find an integer estimate $a \simeq 10^8/\langle d \rangle$ by computing

$$\begin{aligned}\alpha &= \left\lfloor \frac{10^9}{\langle d_1 \rangle + 1} \right\rfloor \\ \beta &= \left\lfloor \frac{10^9}{\langle d_1 \rangle} \right\rfloor \\ a &= 10^3\alpha + (\beta - \alpha) \cdot \left(10^3 - \left\lfloor \frac{\langle d_2 \rangle}{10} \right\rfloor \right) - 1250,\end{aligned}$$

where $\left\lfloor \frac{\cdot}{\cdot} \right\rfloor$ denotes ε -TEX's rounding division, which rounds ties away from zero. The idea is to interpolate between $10^3\alpha$ and $10^3\beta$ with a parameter $\langle d_2 \rangle/10^4$, so that when $\langle d_2 \rangle = 0$ one gets $a = 10^3\beta - 1250 \simeq 10^{12}/\langle d_1 \rangle \simeq 10^8/\langle d \rangle$, while when $\langle d_2 \rangle = 9999$ one gets $a = 10^3\alpha - 1250 \simeq 10^{12}/(\langle d_1 \rangle + 1) \simeq 10^8/\langle d \rangle$. The shift by 1250 helps to ensure that a is an underestimate of the correct value. We shall prove that

$$1 - 1.755 \cdot 10^{-5} < \frac{\langle d \rangle a}{10^8} < 1.$$

We can then compute the inverse of $\langle d \rangle a / 10^8 = 1 - \epsilon$ using the relation $1/(1 - \epsilon) \simeq (1 + \epsilon)(1 + \epsilon^2) + \epsilon^4$, which is correct up to a relative error of $\epsilon^5 < 1.6 \cdot 10^{-24}$. This allows us to find the desired ratio as

$$\frac{\langle n \rangle}{\langle d \rangle} = \frac{\langle n \rangle a}{10^8} ((1 + \epsilon)(1 + \epsilon^2) + \epsilon^4).$$

Let us prove the upper bound first (multiplied by 10^{15}). Note that $10^7\langle d \rangle < 10^3\langle d_1 \rangle + 10^{-1}(\langle d_2 \rangle + 1)$, and that ε -TEX's division $\left\lfloor \frac{\langle d_2 \rangle}{10} \right\rfloor$ underestimates $10^{-1}(\langle d_2 \rangle + 1)$ by 0.5 at most, as can be checked for each possible last digit of $\langle d_2 \rangle$. Then,

$$10^7\langle d \rangle a < \left(10^3\langle d_1 \rangle + \left\lfloor \frac{\langle d_2 \rangle}{10} \right\rfloor + \frac{1}{2} \right) \left(\left(10^3 - \left\lfloor \frac{\langle d_2 \rangle}{10} \right\rfloor \right) \beta + \left\lfloor \frac{\langle d_2 \rangle}{10} \right\rfloor \alpha - 1250 \right) \quad (1)$$

$$< \left(10^3\langle d_1 \rangle + \left\lfloor \frac{\langle d_2 \rangle}{10} \right\rfloor + \frac{1}{2} \right) \quad (2)$$

$$\left(\left(10^3 - \left\lfloor \frac{\langle d_2 \rangle}{10} \right\rfloor \right) \left(\frac{10^9}{\langle d_1 \rangle} + \frac{1}{2} \right) + \left\lfloor \frac{\langle d_2 \rangle}{10} \right\rfloor \left(\frac{10^9}{\langle d_1 \rangle + 1} + \frac{1}{2} \right) - 1250 \right) \quad (3)$$

$$< \left(10^3\langle d_1 \rangle + \left\lfloor \frac{\langle d_2 \rangle}{10} \right\rfloor + \frac{1}{2} \right) \left(\frac{10^{12}}{\langle d_1 \rangle} - \left\lfloor \frac{\langle d_2 \rangle}{10} \right\rfloor \frac{10^9}{\langle d_1 \rangle(\langle d_1 \rangle + 1)} - 750 \right) \quad (4)$$

We recognize a quadratic polynomial in $[\langle d_2 \rangle/10]$ with a negative leading coefficient: this polynomial is bounded above, according to $([\langle d_2 \rangle/10] + a)(b - c[\langle d_2 \rangle/10]) \leq (b + ca)^2/(4c)$. Hence,

$$10^7\langle d \rangle a < \frac{10^{15}}{\langle d_1 \rangle(\langle d_1 \rangle + 1)} \left(\langle d_1 \rangle + \frac{1}{2} + \frac{1}{4}10^{-3} - \frac{3}{8} \cdot 10^{-9}\langle d_1 \rangle(\langle d_1 \rangle + 1) \right)^2$$

Since $\langle d_1 \rangle$ takes integer values within $[1000, 9999]$, it is a simple programming exercise to check that the squared expression is always less than $\langle d_1 \rangle(\langle d_1 \rangle + 1)$, hence $10^7\langle d \rangle a < 10^{15}$. The upper bound is proven. We also find that $\frac{3}{8}$ can be replaced by slightly smaller numbers, but nothing less than $0.374563\dots$, and going back through the derivation of

the upper bound, we find that 1250 is as small a shift as we can obtain without breaking the bound.

Now, the lower bound. The same computation as for the upper bound implies

$$10^7 \langle d \rangle a > \left(10^3 \langle d_1 \rangle + \left\lceil \frac{\langle d_2 \rangle}{10} \right\rceil - \frac{1}{2} \right) \left(\frac{10^{12}}{\langle d_1 \rangle} - \left\lceil \frac{\langle d_2 \rangle}{10} \right\rceil \frac{10^9}{\langle d_1 \rangle (\langle d_1 \rangle + 1)} - 1750 \right)$$

This time, we want to find the minimum of this quadratic polynomial. Since the leading coefficient is still negative, the minimum is reached for one of the extreme values $[y/10] = 0$ or $[y/10] = 100$, and we easily check the bound for those values.

We have proven that the algorithm gives us a precise enough answer. Incidentally, the upper bound that we derived tells us that $a < 10^8 / \langle d \rangle \leq 10^9$, hence we can compute a safely as a $\text{T}_{\text{E}}\text{X}$ integer, and even add 10^9 to it to ease grabbing of all the digits. The lower bound implies $10^8 - 1755 < a$, which we do not care about.

`__fp_ep_div:wwwn`

Compute the ratio of two extended-precision numbers. The result is an extended-precision number whose first block lies in the range $[100, 9999]$, and is placed after the $\langle continuation \rangle$ once we are done. First normalize the inputs so that both first block lie in $[1000, 9999]$, then call `__fp_ep_div_esti:wwwn` $\langle denominator \rangle$ $\langle numerator \rangle$, responsible for estimating the inverse of the denominator.

```
26124 \cs_new:Npn \__fp_ep_div:wwwn #1,#2; #3,#4;
26125 {
26126   \__fp_ep_to_ep:wwN #1,#2;
26127   \__fp_fixed_continue:wn
26128   {
26129     \__fp_ep_to_ep:wwN #3,#4;
26130     \__fp_ep_div_esti:wwwn
26131   }
26132 }
```

(End definition for `__fp_ep_div:wwwn`.)

`__fp_ep_div_esti:wwwn`

`__fp_ep_div_estii:wwnnwwn`

`__fp_ep_div_estiii:NNNNNwwn`

The `esti` function evaluates $\alpha = 10^9 / (\langle d_1 \rangle + 1)$, which is used twice in the expression for a , and combines the exponents `#1` and `#4` (with a shift by 1 because we later compute $\langle n \rangle / (10 \langle d \rangle)$). Then the `estii` function evaluates $10^9 + a$, and puts the exponent `#2` after the continuation `#7`: from there on we can forget exponents and focus on the mantissa. The `estiii` function multiplies the denominator `#7` by $10^{-8}a$ (obtained as a split into the single digit `#1` and two blocks of 4 digits, `#2#3#4#5` and `#6`). The result $10^{-8}a \langle d \rangle = (1 - \epsilon)$, and a partially packed $10^{-9}a$ (as a block of four digits, and five individual digits, not packed by lack of available macro parameters here) are passed to `__fp_ep_div_epsilon:wnNNNNn`, which computes $10^{-9}a / (1 - \epsilon)$, that is, $1 / (10 \langle d \rangle)$ and we finally multiply this by the numerator `#8`.

```
26133 \cs_new:Npn \__fp_ep_div_esti:wwwn #1,#2#3; #4,
26134 {
26135   \exp_after:wN \__fp_ep_div_estii:wwnnwwn
26136   \int_value:w \__fp_int_eval:w 10 0000 0000 / ( #2 + 1 )
26137   \exp_after:wN ;
26138   \int_value:w \__fp_int_eval:w #4 - #1 + 1 ,
26139   {#2} #3;
26140 }
26141 \cs_new:Npn \__fp_ep_div_estii:wwnnwwn #1; #2,#3#4#5; #6; #7
26142 {
```

```

26143 \exp_after:wN \_fp_ep_div_estiii:NNNNNwwwn
26144 \int_value:w \_fp_int_eval:w 10 0000 0000 - 1750
26145 + #1 000 + (10 0000 0000 / #3 - #1) * (1000 - #4 / 10) ;
26146 {#3}{#4}#5; #6; { #7 #2, }
26147 }
26148 \cs_new:Npn \_fp_ep_div_estiii:NNNNNwwwn 1#1#2#3#4#5#6; #7;
26149 {
26150 \_fp_fixed_mul_short:wwn #7; {#1}{#2#3#4#5}{#6};
26151 \_fp_ep_div_epsilon:wnNNNNNn {#1#2#3#4}#5#6
26152 \_fp_fixed_mul:wwn
26153 }

```

(End definition for `_fp_ep_div_esti:wwwwn`, `_fp_ep_div_estii:wwnnwwn`, and `_fp_ep_div_estiii:NNNNNwwwn`.)

`_fp_ep_div_epsilon:wnNNNNNn`
`_fp_ep_div_epsilon_pack:NNNNNw`
`_fp_ep_div_epsilonii:wwnnNNNNNn`

The bounds shown above imply that the `epsi` function's first operand is $(1 - \epsilon)$ with $\epsilon \in [0, 1.755 \cdot 10^{-5}]$. The `epsi` function computes ϵ as $1 - (1 - \epsilon)$. Since $\epsilon < 10^{-4}$, its first block vanishes and there is no need to explicitly use `#1` (which is 9999). Then `epsiii` evaluates $10^{-9}a/(1 - \epsilon)$ as $(1 + \epsilon^2)(1 + \epsilon)(10^{-9}a\epsilon) + 10^{-9}a$. Importantly, we compute $10^{-9}a\epsilon$ before multiplying it with the rest, rather than multiplying by ϵ and then $10^{-9}a$, as this second option loses more precision. Also, the combination of `short_mul` and `div_myriad` is both faster and more precise than a simple `mul`.

```

26154 \cs_new:Npn \_fp_ep_div_epsilon:wnNNNNNn #1#2#3#4#5#6;
26155 {
26156 \exp_after:wN \_fp_ep_div_epsilonii:wwnnNNNNNn
26157 \int_value:w \_fp_int_eval:w 1 9998 - #2
26158 \exp_after:wN \_fp_ep_div_epsilon_pack:NNNNNw
26159 \int_value:w \_fp_int_eval:w 1 9999 9998 - #3#4
26160 \exp_after:wN \_fp_ep_div_epsilon_pack:NNNNNw
26161 \int_value:w \_fp_int_eval:w 2 0000 0000 - #5#6 ; ;
26162 }
26163 \cs_new:Npn \_fp_ep_div_epsilon_pack:NNNNNw #1#2#3#4#5#6;
26164 { + #1 ; {#2#3#4#5} {#6} }
26165 \cs_new:Npn \_fp_ep_div_epsilonii:wwnnNNNNNn 1#1; #2; #3#4#5#6#7#8
26166 {
26167 \_fp_fixed_mul:wwn {0000}{#1}#2; {0000}{#1}#2;
26168 \_fp_fixed_add_one:wN
26169 \_fp_fixed_mul:wwn {10000} {#1} #2 ;
26170 {
26171 \_fp_fixed_mul_short:wwn {0000}{#1}#2; {#3}{#4#5#6#7}{#8000};
26172 \_fp_fixed_div_myriad:wn
26173 \_fp_fixed_mul:wwn
26174 }
26175 \_fp_fixed_add:wwn {#3}{#4#5#6#7}{#8000}{0000}{0000}{0000};
26176 }

```

(End definition for `_fp_ep_div_epsilon:wnNNNNNn`, `_fp_ep_div_epsilon_pack:NNNNNw`, and `_fp_ep_div_epsilonii:wwnnNNNNNn`.)

73.10 Inverse square root of extended precision numbers

The idea here is similar to division. Normalize the input, multiplying by powers of 100 until we have $x \in [0.01, 1)$. Then find an integer approximation $r \in [101, 1003]$ of $10^2/\sqrt{x}$, as the fixed point of iterations of the Newton method: essentially $r \mapsto (r + 10^8/(x_1 r))/2$, starting from a guess that optimizes the number of steps before convergence. In fact, just as there is a slight shift when computing divisions to ensure that some inequalities hold, we replace 10^8 by a slightly larger number which ensures that $r^2 x \geq 10^4$. This also causes $r \in [101, 1003]$. Another correction to the above is that the input is actually normalized to $[0.1, 1)$, and we use either 10^8 or 10^9 in the Newton method, depending on the parity of the exponent. Skipping those technical hurdles, once we have the approximation r , we set $y = 10^{-4} r^2 x$ (or rather, the correct power of 10 to get $y \simeq 1$) and compute $y^{-1/2}$ through another application of Newton's method. This time, the starting value is $z = 1$, each step maps $z \mapsto z(1.5 - 0.5yz^2)$, and we perform a fixed number of steps. Our final result combines r with $y^{-1/2}$ as $x^{-1/2} = 10^{-2} r y^{-1/2}$.

```

__fp_ep_isqrt:wwn
__fp_ep_isqrt_aux:wwn
__fp_ep_isqrt_auxii:wwnnwn

```

First normalize the input, then check the parity of the exponent #1. If it is even, the result's exponent will be $-\#1/2$, otherwise it will be $(\#1 - 1)/2$ (except in the case where the input was an exact power of 100). The `auxii` function receives as #1 the result's exponent just computed, as #2 the starting value for the iteration giving r (the values 168 and 535 lead to the least number of iterations before convergence, on average), as #3 and #4 one empty argument and one 0, depending on the parity of the original exponent, as #5 and #6 the normalized mantissa ($\#5 \in [1000, 9999]$), and as #7 the continuation. It sets up the iteration giving r : the `esti` function thus receives the initial two guesses #2 and 0, an approximation #5 of $10^4 x$ (its first block of digits), and the empty/zero arguments #3 and #4, followed by the mantissa and an altered continuation where we have stored the result's exponent.

```

26177 \cs_new:Npn \__fp_ep_isqrt:wwn #1,#2;
26178 {
26179   \__fp_ep_to_ep:wwN #1,#2;
26180   \__fp_ep_isqrt_auxi:wwn
26181 }
26182 \cs_new:Npn \__fp_ep_isqrt_auxi:wwn #1,
26183 {
26184   \exp_after:wN \__fp_ep_isqrt_auxii:wwnnwn
26185   \int_value:w \__fp_int_eval:w
26186   \int_if_odd:nTF {#1}
26187     { (1 - #1) / 2 , 535 , { 0 } { } }
26188     { 1 - #1 / 2 , 168 , { } { 0 } }
26189 }
26190 \cs_new:Npn \__fp_ep_isqrt_auxii:wwnnwn #1, #2, #3#4 #5#6; #7
26191 {
26192   \__fp_ep_isqrt_esti:wwnnwn #2, 0, #5, {#3} {#4}
26193   {#5} #6 ; { #7 #1 , }
26194 }

```

(End definition for `__fp_ep_isqrt:wwn`, `__fp_ep_isqrt_aux:wwn`, and `__fp_ep_isqrt_auxii:wwnnwn`.)

```

__fp_ep_isqrt_esti:wwnnwn
__fp_ep_isqrt_estii:wwnnwn
__fp_ep_isqrt_estiii:NNNNNwwn

```

If the last two approximations gave the same result, we are done: call the `estii` function to clean up. Otherwise, evaluate $(\langle prev \rangle + 1.005 \cdot 10^8 \text{ or } 9 / (\langle prev \rangle \cdot x)) / 2$, as the next approximation: omitting the 1.005 factor, this would be Newton's method. We can

check by brute force that if #4 is empty (the original exponent was even), the process computes an integer slightly larger than $100/\sqrt{x}$, while if #4 is 0 (the original exponent was odd), the result is an integer slightly larger than $100/\sqrt{x/10}$. Once we are done, we evaluate $100r^2/2$ or $10r^2/2$ (when the exponent is even or odd, respectively) and feed that to `esti`. This third auxiliary finds $y_{\text{even}}/2 = 10^{-4}r^2x/2$ or $y_{\text{odd}}/2 = 10^{-5}r^2x/2$ (again, depending on earlier parity). A simple program shows that $y \in [1, 1.0201]$. The number $y/2$ is fed to `_fp_ep_isqrt_epsi:wN`, which computes $1/\sqrt{y}$, and we finally multiply the result by r .

```

26195 \cs_new:Npn \_fp\_ep\_isqrt\_esti:wwnnwn #1, #2, #3, #4
26196 {
26197   \if_int_compare:w #1 = #2 \exp_stop_f:
26198   \exp_after:wN \_fp\_ep\_isqrt\_estii:wwnnwn
26199   \fi:
26200   \exp_after:wN \_fp\_ep\_isqrt\_esti:wwnnwn
26201   \int_value:w \_fp\_int\_eval:w
26202   (#1 + 1 0050 0000 #4 / (#1 * #3)) / 2 ,
26203   #1, #3, {#4}
26204 }
26205 \cs_new:Npn \_fp\_ep\_isqrt\_estii:wwnnwn #1, #2, #3, #4#5
26206 {
26207   \exp_after:wN \_fp\_ep\_isqrt\_estiii:NNNNNwwwn
26208   \int_value:w \_fp\_int\_eval:w 1000 0000 + #2 * #2 #5 * 5
26209   \exp_after:wN , \int_value:w \_fp\_int\_eval:w 10000 + #2 ;
26210 }
26211 \cs_new:Npn \_fp\_ep\_isqrt\_estiii:NNNNNwwwn 1#1#2#3#4#5#6, 1#7#8; #9;
26212 {
26213   \_fp\_fixed\_mul\_short:wwn #9; {#1} {#2#3#4#5} {#600} ;
26214   \_fp\_ep\_isqrt\_epsi:wN
26215   \_fp\_fixed\_mul\_short:wwn {#7} {#80} {0000} ;
26216 }

```

(End definition for `_fp_ep_isqrt_esti:wwnnwn`, `_fp_ep_isqrt_estii:wwnnwn`, and `_fp_ep_isqrt_estiii:NNNNNwwwn`.)

`_fp_ep_isqrt_epsi:wN`
`_fp_ep_isqrt_epsii:wwN`

Here, we receive a fixed point number $y/2$ with $y \in [1, 1.0201]$. Starting from $z = 1$ we iterate $z \mapsto z(3/2 - z^2y/2)$. In fact, we start from the first iteration $z = 3/2 - y/2$ to avoid useless multiplications. The `epsii` auxiliary receives z as #1 and y as #2.

```

26217 \cs_new:Npn \_fp\_ep\_isqrt\_epsi:wN #1;
26218 {
26219   \_fp\_fixed\_sub:wwn {15000}{0000}{0000}{0000}{0000}{0000}; #1;
26220   \_fp\_ep\_isqrt\_epsii:wwN #1;
26221   \_fp\_ep\_isqrt\_epsii:wwN #1;
26222   \_fp\_ep\_isqrt\_epsii:wwN #1;
26223 }
26224 \cs_new:Npn \_fp\_ep\_isqrt\_epsii:wwN #1; #2;
26225 {
26226   \_fp\_fixed\_mul:wwn #1; #1;
26227   \_fp\_fixed\_mul\_sub\_back:wwwn #2;
26228   {15000}{0000}{0000}{0000}{0000}{0000};
26229   \_fp\_fixed\_mul:wwn #1;
26230 }

```

(End definition for `_fp_ep_isqrt_epsi:wN` and `_fp_ep_isqrt_epsii:wwN`.)

73.11 Converting from fixed point to floating point

After computing Taylor series, we wish to convert the result from extended precision (with or without an exponent) to the public floating point format. The functions here should be called within an integer expression for the overall exponent of the floating point.

`__fp_ep_to_float_o:wwN`
`__fp_ep_inv_to_float_o:wwN` An extended-precision number is simply a comma-delimited exponent followed by a fixed point number. Leave the exponent in the current integer expression then convert the fixed point number.

```
26231 \cs_new:Npn \__fp_ep_to_float_o:wwN #1,
26232 { + \__fp_int_eval:w #1 \__fp_fixed_to_float_o:wN }
26233 \cs_new:Npn \__fp_ep_inv_to_float_o:wwN #1,#2;
26234 {
26235   \__fp_ep_div:wwwwN 1,{1000}{0000}{0000}{0000}{0000}{0000}; #1,#2;
26236   \__fp_ep_to_float_o:wwN
26237 }
```

(End definition for `__fp_ep_to_float_o:wwN` and `__fp_ep_inv_to_float_o:wwN`.)

`__fp_fixed_inv_to_float_o:wN` Another function which reduces to converting an extended precision number to a float.

```
26238 \cs_new:Npn \__fp_fixed_inv_to_float_o:wN
26239 { \__fp_ep_inv_to_float_o:wwN 0, }
```

(End definition for `__fp_fixed_inv_to_float_o:wN`.)

`__fp_fixed_to_float_rad_o:wN` Converts the fixed point number #1 from degrees to radians then to a floating point number. This could perhaps remain in `l3fp-trig`.

```
26240 \cs_new:Npn \__fp_fixed_to_float_rad_o:wN #1;
26241 {
26242   \__fp_fixed_mul:wwN #1; {5729}{5779}{5130}{8232}{0876}{7981};
26243   { \__fp_ep_to_float_o:wwN 2, }
26244 }
```

(End definition for `__fp_fixed_to_float_rad_o:wN`.)

`__fp_fixed_to_float_o:wN`
`__fp_fixed_to_float_o:Nw` ... `__fp_int_eval:w` $\langle \text{exponent} \rangle$ `__fp_fixed_to_float_o:wN` $\{\langle a_1 \rangle\} \{\langle a_2 \rangle\} \{\langle a_3 \rangle\} \{\langle a_4 \rangle\} \{\langle a_5 \rangle\} \{\langle a_6 \rangle\}$; $\langle \text{sign} \rangle$
yields
 $\langle \text{exponent}' \rangle$; $\{\langle a_1' \rangle\} \{\langle a_2' \rangle\} \{\langle a_3' \rangle\} \{\langle a_4' \rangle\}$;

And the `to_fixed` version gives six brace groups instead of 4, ensuring that $1000 \leq \langle a_1' \rangle \leq 9999$. At this stage, we know that $\langle a_1 \rangle$ is positive (otherwise, it is sign of an error before), and we assume that it is less than 10^8 .¹¹

```
26245 \cs_new:Npn \__fp_fixed_to_float_o:Nw #1#2;
26246 { \__fp_fixed_to_float_o:wN #2; #1 }
26247 \cs_new:Npn \__fp_fixed_to_float_o:wN #1#2#3#4#5#6; #7
26248 { % for the 8-digit-at-the-start thing
26249   + \__fp_int_eval:w \c__fp_block_int
26250   \exp_after:wN \exp_after:wN
26251   \exp_after:wN \__fp_fixed_to_loop:N
```

¹¹Bruno: I must double check this assumption.


```

26252 \exp_after:wN \use_none:n
26253 \int_value:w \__fp_int_eval:w
26254 1 0000 0000 + #1 \exp_after:wN \__fp_use_none_stop_f:n
26255 \int_value:w 1#2 \exp_after:wN \__fp_use_none_stop_f:n
26256 \int_value:w 1#3#4 \exp_after:wN \__fp_use_none_stop_f:n
26257 \int_value:w 1#5#6
26258 \exp_after:wN ;
26259 \exp_after:wN ;
26260 }
26261 \cs_new:Npn \__fp_fixed_to_loop:N #1
26262 {
26263 \if_meaning:w 0 #1
26264 - 1
26265 \exp_after:wN \__fp_fixed_to_loop:N
26266 \else:
26267 \exp_after:wN \__fp_fixed_to_loop_end:w
26268 \exp_after:wN #1
26269 \fi:
26270 }
26271 \cs_new:Npn \__fp_fixed_to_loop_end:w #1 #2 ;
26272 {
26273 \if_meaning:w ; #1
26274 \exp_after:wN \__fp_fixed_to_float_zero:w
26275 \else:
26276 \exp_after:wN \__fp_pack_twice_four:wNNNNNNNN
26277 \exp_after:wN \__fp_pack_twice_four:wNNNNNNNN
26278 \exp_after:wN \__fp_fixed_to_float_pack:ww
26279 \exp_after:wN ;
26280 \fi:
26281 #1 #2 0000 0000 0000 0000 ;
26282 }
26283 \cs_new:Npn \__fp_fixed_to_float_zero:w ; 0000 0000 0000 0000 ;
26284 {
26285 - 2 * \c__fp_max_exponent_int ;
26286 {0000} {0000} {0000} {0000} ;
26287 }
26288 \cs_new:Npn \__fp_fixed_to_float_pack:ww #1 ; #2#3 ; ;
26289 {
26290 \if_int_compare:w #2 > 4 \exp_stop_f:
26291 \exp_after:wN \__fp_fixed_to_float_round_up:wnnnnw
26292 \fi:
26293 ; #1 ;
26294 }
26295 \cs_new:Npn \__fp_fixed_to_float_round_up:wnnnnw ; #1#2#3#4 ;
26296 {
26297 \exp_after:wN \__fp_basics_pack_high:NNNNNw
26298 \int_value:w \__fp_int_eval:w 1 #1#2
26299 \exp_after:wN \__fp_basics_pack_low:NNNNNw
26300 \int_value:w \__fp_int_eval:w 1 #3#4 + 1 ;
26301 }

```

(End definition for __fp_fixed_to_float_o:wN and __fp_fixed_to_float_o:Nw.)

```

26302 \</package>

```

Chapter 74

l3fp-expo implementation

```

26303 <*package>
26304 <@@=fp>

\__fp_parse_word_exp:N Unary functions.
\__fp_parse_word_ln:N 26305 \cs_new:Npn \__fp_parse_word_exp:N
\__fp_parse_word_fact:N 26306 { \__fp_parse_unary_function:NNN \__fp_exp_o:w ? }
26307 \cs_new:Npn \__fp_parse_word_ln:N
26308 { \__fp_parse_unary_function:NNN \__fp_ln_o:w ? }
26309 \cs_new:Npn \__fp_parse_word_fact:N
26310 { \__fp_parse_unary_function:NNN \__fp_fact_o:w ? }

(End definition for \__fp_parse_word_exp:N, \__fp_parse_word_ln:N, and \__fp_parse_word_fact:N.)

```

74.1 Logarithm

74.1.1 Work plan

As for many other functions, we filter out special cases in `__fp_ln_o:w`. Then `__fp_ln_npos_o:w` receives a positive normal number, which we write in the form $a \cdot 10^b$ with $a \in [0.1, 1)$.

The rest of this section is actually not in sync with the code. Or is the code not in sync with the section? In the current code, $c \in [1, 10]$ is such that $0.7 \leq ac < 1.4$.

We are given a positive normal number, of the form $a \cdot 10^b$ with $a \in [0.1, 1)$. To compute its logarithm, we find a small integer $5 \leq c < 50$ such that $0.91 \leq ac/5 < 1.1$, and use the relation

$$\ln(a \cdot 10^b) = b \cdot \ln(10) - \ln(c/5) + \ln(ac/5).$$

The logarithms $\ln(10)$ and $\ln(c/5)$ are looked up in a table. The last term is computed using the following Taylor series of \ln near 1:

$$\ln\left(\frac{ac}{5}\right) = \ln\left(\frac{1+t}{1-t}\right) = 2t \left(1 + t^2 \left(\frac{1}{3} + t^2 \left(\frac{1}{5} + t^2 \left(\frac{1}{7} + t^2 \left(\frac{1}{9} + \dots\right)\right)\right)\right)\right)$$

where $t = 1 - 10/(ac + 5)$. We can now see one reason for the choice of $ac \sim 5$: then $ac + 5 = 10(1 - \epsilon)$ with $-0.05 < \epsilon \leq 0.045$, hence

$$t = \frac{\epsilon}{1 - \epsilon} = \epsilon(1 + \epsilon)(1 + \epsilon^2)(1 + \epsilon^4) \dots,$$

is not too difficult to compute.

74.1.2 Some constants

A few values of the logarithm as extended fixed point numbers. Those are needed in the implementation. It turns out that we don't need the value of $\ln(5)$.

```

\c__fp_ln_i_fixed_t1
\c__fp_ln_ii_fixed_t1
\c__fp_ln_iii_fixed_t1
\c__fp_ln_iv_fixed_t1
\c__fp_ln_vii_fixed_t1
\c__fp_ln_viii_fixed_t1
\c__fp_ln_ix_fixed_t1
\c__fp_ln_x_fixed_t1
26311 \tl_const:Nn \c__fp_ln_i_fixed_t1 { {0000}{0000}{0000}{0000}{0000}{0000};}
26312 \tl_const:Nn \c__fp_ln_ii_fixed_t1 { {6931}{4718}{0559}{9453}{0941}{7232};}
26313 \tl_const:Nn \c__fp_ln_iii_fixed_t1 { {10986}{1228}{8668}{1096}{9139}{5245};}
26314 \tl_const:Nn \c__fp_ln_iv_fixed_t1 { {13862}{9436}{1119}{8906}{1883}{4464};}
26315 \tl_const:Nn \c__fp_ln_vii_fixed_t1 { {17917}{5946}{9228}{0550}{0081}{2477};}
26316 \tl_const:Nn \c__fp_ln_viii_fixed_t1 { {19459}{1014}{9055}{3133}{0510}{5353};}
26317 \tl_const:Nn \c__fp_ln_viii_fixed_t1 { {20794}{4154}{1679}{8359}{2825}{1696};}
26318 \tl_const:Nn \c__fp_ln_ix_fixed_t1 { {21972}{2457}{7336}{2193}{8279}{0490};}
26319 \tl_const:Nn \c__fp_ln_x_fixed_t1 { {23025}{8509}{2994}{0456}{8401}{7991};}

```

(End definition for `\c__fp_ln_i_fixed_t1` and others.)

74.1.3 Sign, exponent, and special numbers

The logarithm of negative numbers (including $-\infty$ and -0) raises the “invalid” exception. The logarithm of $+0$ is $-\infty$, raising a division by zero exception. The logarithm of $+\infty$ or a `nan` is itself. Positive normal numbers call `__fp_ln_npos_o:w`.

```

26320 \cs_new:Npn \__fp_ln_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
26321 {
26322   \if_meaning:w 2 #3
26323     \__fp_case_use:nw { \__fp_invalid_operation_o:nw { ln } }
26324   \fi:
26325   \if_case:w #2 \exp_stop_f:
26326     \__fp_case_use:nw
26327     { \__fp_division_by_zero_o:Nnw \c_minus_inf_fp { ln } }
26328   \or:
26329   \else:
26330     \__fp_case_return_same_o:w
26331   \fi:
26332   \__fp_ln_npos_o:w \s__fp \__fp_chk:w #2#3#4;
26333 }

```

(End definition for `__fp_ln_o:w`.)

74.1.4 Absolute ln

`__fp_ln_npos_o:w` We catch the case of a significand very close to 0.1 or to 1. In all other cases, the final result is at least 10^{-4} , and then an error of $0.5 \cdot 10^{-20}$ is acceptable.

```

26334 \cs_new:Npn \__fp_ln_npos_o:w \s__fp \__fp_chk:w 10#1#2#3;
26335 { %^A todo: ln(1) should be "exact zero", not "underflow"
26336   \exp_after:wN \__fp_sanitize:Nw
26337   \int_value:w % for the overall sign
26338   \if_int_compare:w #1 < \c_one_int
26339     2
26340   \else:
26341     0
26342   \fi:

```

```

26343     \exp_after:wN \exp_stop_f:
26344     \int_value:w \__fp_int_eval:w % for the exponent
26345     \__fp_ln_significand:NNNNnnnN #2#3
26346     \__fp_ln_exponent:wn {#1}
26347 }

```

(End definition for __fp_ln_npos_o:w.)

__fp_ln_significand:NNNNnnnN __fp_ln_significand:NNNNnnnN $\langle X_1 \rangle$ $\{\langle X_2 \rangle\}$ $\{\langle X_3 \rangle\}$ $\{\langle X_4 \rangle\}$ $\langle continuation \rangle$
This function expands to

$\langle continuation \rangle$ $\{\langle Y_1 \rangle\}$ $\{\langle Y_2 \rangle\}$ $\{\langle Y_3 \rangle\}$ $\{\langle Y_4 \rangle\}$ $\{\langle Y_5 \rangle\}$ $\{\langle Y_6 \rangle\}$;

where $Y = -\ln(X)$ as an extended fixed point.

```

26348 \cs_new:Npn \__fp_ln_significand:NNNNnnnN #1#2#3#4
26349 {
26350     \exp_after:wN \__fp_ln_x_ii:wnnnn
26351     \int_value:w
26352     \if_case:w #1 \exp_stop_f:
26353     \or:
26354         \if_int_compare:w #2 < 4 \exp_stop_f:
26355             \__fp_int_eval:w 10 - #2
26356         \else:
26357             6
26358         \fi:
26359     \or: 4
26360     \or: 3
26361     \or: 2
26362     \or: 2
26363     \or: 2
26364     \else: 1
26365     \fi:
26366     ; { #1 #2 #3 #4 }
26367 }

```

(End definition for __fp_ln_significand:NNNNnnnN.)

__fp_ln_x_ii:wnnnn We have thus found $c \in [1, 10]$ such that $0.7 \leq ac < 1.4$ in all cases. Compute $1 + x = 1 + ac \in [1.7, 2.4)$.

```

26368 \cs_new:Npn \__fp_ln_x_ii:wnnnn #1; #2#3#4#5
26369 {
26370     \exp_after:wN \__fp_ln_div_after:Nw
26371     \cs:w c__fp_ln_ \__fp_int_to_roman:w #1 _fixed_tl \exp_after:wN \cs_end:
26372     \int_value:w
26373     \exp_after:wN \__fp_ln_x_iv:wnnnnnnnn
26374     \int_value:w \__fp_int_eval:w
26375     \exp_after:wN \__fp_ln_x_iii_var:NNNNNw
26376     \int_value:w \__fp_int_eval:w 9999 9990 + #1*#2#3 +
26377     \exp_after:wN \__fp_ln_x_iii:NNNNNw
26378     \int_value:w \__fp_int_eval:w 10 0000 0000 + #1*#4#5 ;
26379     {20000} {0000} {0000} {0000}
26380 } %^^A todo: reoptimize (a generalization attempt failed).
26381 \cs_new:Npn \__fp_ln_x_iii:NNNNNw #1#2 #3#4#5#6 #7;
26382 { #1#2; {#3#4#5#6} {#7} }
26383 \cs_new:Npn \__fp_ln_x_iii_var:NNNNNw #1 #2#3#4#5 #6;

```

```

26384 {
26385   #1#2#3#4#5 + 1 ;
26386   {#1#2#3#4#5} {#6}
26387 }

```

The Taylor series to be used is expressed in terms of $t = (x - 1)/(x + 1) = 1 - 2/(x + 1)$. We now compute the quotient with extended precision, reusing some code from `_fp_/_o:ww`. Note that $1 + x$ is known exactly.

To reuse notations from `l3fp-basics`, we want to compute A/Z with $A = 2$ and $Z = x + 1$. In `l3fp-basics`, we considered the case where both A and Z are arbitrary, in the range $[0.1, 1)$, and we had to monitor the growth of the sequence of remainders A, B, C , etc. to ensure that no overflow occurred during the computation of the next quotient. The main source of risk was our choice to define the quotient as roughly $10^9 \cdot A/10^5 \cdot Z$: then A was bound to be below $2.147 \cdots$, and this limit was never far.

In our case, we can simply work with $10^8 \cdot A$ and $10^4 \cdot Z$, because our reason to work with higher powers has gone: we needed the integer $y \simeq 10^5 \cdot Z$ to be at least 10^4 , and now, the definition $y \simeq 10^4 \cdot Z$ suffices.

Let us thus define $y = \lfloor 10^4 \cdot Z \rfloor + 1 \in (1.7 \cdot 10^4, 2.4 \cdot 10^4]$, and

$$Q_1 = \left\lfloor \frac{\lfloor 10^8 \cdot A \rfloor}{y} - \frac{1}{2} \right\rfloor.$$

(The $1/2$ comes from how ε -TeX rounds.) As for division, it is easy to see that $Q_1 \leq 10^4 A/Z$, *i.e.*, Q_1 is an underestimate.

Exactly as we did for division, we set $B = 10^4 A - Q_1 Z$. Then

$$\begin{aligned}
10^4 B &\leq A_1 A_2 \cdot A_3 A_4 - \left(\frac{A_1 A_2}{y} - \frac{3}{2} \right) 10^4 Z \\
&\leq A_1 A_2 \left(1 - \frac{10^4 Z}{y} \right) + 1 + \frac{3}{2} y \\
&\leq 10^8 \frac{A}{y} + 1 + \frac{3}{2} y
\end{aligned}$$

In the same way, and using $1.7 \cdot 10^4 \leq y \leq 2.4 \cdot 10^4$, and convexity, we get

$$\begin{aligned}
10^4 A &= 2 \cdot 10^4 \\
10^4 B &\leq 10^8 \frac{A}{y} + 1.6y \leq 4.7 \cdot 10^4 \\
10^4 C &\leq 10^8 \frac{B}{y} + 1.6y \leq 5.8 \cdot 10^4 \\
10^4 D &\leq 10^8 \frac{C}{y} + 1.6y \leq 6.3 \cdot 10^4 \\
10^4 E &\leq 10^8 \frac{D}{y} + 1.6y \leq 6.5 \cdot 10^4 \\
10^4 F &\leq 10^8 \frac{E}{y} + 1.6y \leq 6.6 \cdot 10^4
\end{aligned}$$

Note that we compute more steps than for division: since t is not the end result, we need to know it with more accuracy (on the other hand, the ending is much simpler, as we don't need an exact rounding for transcendental functions, but just a faithful rounding).

_fp_ln_x_iv:wnnnnnnnn <1 or 2> <8d> ; {<4d>} {<4d>} <fixed-t1>

The number is x . Compute y by adding 1 to the five first digits.

```

26388 \cs_new:Npn \_fp_ln_x_iv:wnnnnnnnn #1; #2#3#4#5 #6#7#8#9
26389 {
26390   \exp_after:wN \_fp_div_significand_pack:NNN
26391   \int_value:w \_fp_int_eval:w
26392   \_fp_ln_div_i:w #1 ;
26393   #6 #7 ; {#8} {#9}
26394   {#2} {#3} {#4} {#5}
26395   { \exp_after:wN \_fp_ln_div_ii:wnn \int_value:w #1 }
26396   { \exp_after:wN \_fp_ln_div_ii:wnn \int_value:w #1 }
26397   { \exp_after:wN \_fp_ln_div_ii:wnn \int_value:w #1 }
26398   { \exp_after:wN \_fp_ln_div_ii:wnn \int_value:w #1 }
26399   { \exp_after:wN \_fp_ln_div_vi:wnn \int_value:w #1 }
26400 }
26401 \cs_new:Npn \_fp_ln_div_i:w #1;
26402 {
26403   \exp_after:wN \_fp_div_significand_calc:wnnnnnnnn
26404   \int_value:w \_fp_int_eval:w 999999 + 2 0000 0000 / #1 ; % Q1
26405 }
26406 \cs_new:Npn \_fp_ln_div_ii:wnn #1; #2;#3 % y; B1;B2 <- for k=1
26407 {
26408   \exp_after:wN \_fp_div_significand_pack:NNN
26409   \int_value:w \_fp_int_eval:w
26410   \exp_after:wN \_fp_div_significand_calc:wnnnnnnnn
26411   \int_value:w \_fp_int_eval:w 999999 + #2 #3 / #1 ; % Q2
26412   #2 #3 ;
26413 }
26414 \cs_new:Npn \_fp_ln_div_vi:wnn #1; #2;#3#4#5 #6#7#8#9 %y;F1;F2F3F4x1x2x3x4
26415 {
26416   \exp_after:wN \_fp_div_significand_pack:NNN
26417   \int_value:w \_fp_int_eval:w 1000000 + #2 #3 / #1 ; % Q6
26418 }

```

We now have essentially

```

\_fp_ln_div_after:Nw <fixed t1>
\_fp_div_significand_pack:NNN 106 + Q1
\_fp_div_significand_pack:NNN 106 + Q2
\_fp_div_significand_pack:NNN 106 + Q3
\_fp_div_significand_pack:NNN 106 + Q4
\_fp_div_significand_pack:NNN 106 + Q5
\_fp_div_significand_pack:NNN 106 + Q6 ;
<exponent> ; <continuation>

```

where $\langle \text{fixed } t1 \rangle$ holds the logarithm of a number in $[1, 10]$, and $\langle \text{exponent} \rangle$ is the exponent. Also, the expansion is done backwards. Then $_fp_div_significand_pack:NNN$ puts things in the correct order to add the Q_i together and put semicolons between each piece. Once those have been expanded, we get

```

\_fp_ln_div_after:Nw <fixed-t1> <1d> ; <4d> ; <4d> ;
<4d> ; <4d> ; <4d> ; <4d> ; <exponent> ;

```

Just as with division, we know that the first two digits are 1 and 0 because of bounds on the final result of the division $2/(x+1)$, which is between roughly 0.8 and 1.2. We then compute $1 - 2/(x+1)$, after testing whether $2/(x+1)$ is greater than or smaller than 1.

```

26419 \cs_new:Npn \__fp_ln_div_after:Nw #1#2;
26420 {
26421   \if_meaning:w 0 #2
26422     \exp_after:wN \__fp_ln_t_small:Nw
26423   \else:
26424     \exp_after:wN \__fp_ln_t_large:NNw
26425     \exp_after:wN -
26426   \fi:
26427   #1
26428 }
26429 \cs_new:Npn \__fp_ln_t_small:Nw #1 #2; #3; #4; #5; #6; #7;
26430 {
26431   \exp_after:wN \__fp_ln_t_large:NNw
26432   \exp_after:wN + % <sign>
26433   \exp_after:wN #1
26434   \int_value:w \__fp_int_eval:w 9999 - #2 \exp_after:wN ;
26435   \int_value:w \__fp_int_eval:w 9999 - #3 \exp_after:wN ;
26436   \int_value:w \__fp_int_eval:w 9999 - #4 \exp_after:wN ;
26437   \int_value:w \__fp_int_eval:w 9999 - #5 \exp_after:wN ;
26438   \int_value:w \__fp_int_eval:w 9999 - #6 \exp_after:wN ;
26439   \int_value:w \__fp_int_eval:w 1 0000 - #7 ;
26440 }

\__fp_ln_t_large:NNw <sign> <fixed t1>
<t1>; <t2>; <t3>; <t4>; <t5>; <t6>;
<exponent>; <continuation>

```

Compute the square t^2 , and keep t at the end with its sign. We know that $t < 0.1765$, so every piece has at most 4 digits. However, since we were not careful in `__fp_ln_t_small:w`, they can have less than 4 digits.

```

26441 \cs_new:Npn \__fp_ln_t_large:NNw #1 #2 #3; #4; #5; #6; #7; #8;
26442 {
26443   \exp_after:wN \__fp_ln_square_t_after:w
26444   \int_value:w \__fp_int_eval:w 9999 0000 + #3*#3
26445   \exp_after:wN \__fp_ln_square_t_pack:NNNNNw
26446   \int_value:w \__fp_int_eval:w 9999 0000 + 2*#3*#4
26447   \exp_after:wN \__fp_ln_square_t_pack:NNNNNw
26448   \int_value:w \__fp_int_eval:w 9999 0000 + 2*#3*#5 + #4*#4
26449   \exp_after:wN \__fp_ln_square_t_pack:NNNNNw
26450   \int_value:w \__fp_int_eval:w 9999 0000 + 2*#3*#6 + 2*#4*#5
26451   \exp_after:wN \__fp_ln_square_t_pack:NNNNNw
26452   \int_value:w \__fp_int_eval:w
26453     1 0000 0000 + 2*#3*#7 + 2*#4*#6 + #5*#5
26454     + (2*#3*#8 + 2*#4*#7 + 2*#5*#6) / 1 0000
26455     % ; ; ;
26456   \exp_after:wN \__fp_ln_twice_t_after:w
26457   \int_value:w \__fp_int_eval:w -1 + 2*#3
26458   \exp_after:wN \__fp_ln_twice_t_pack:Nw
26459   \int_value:w \__fp_int_eval:w 9999 + 2*#4
26460   \exp_after:wN \__fp_ln_twice_t_pack:Nw

```

```

26461      \int_value:w \__fp_int_eval:w 9999 + 2*#5
26462      \exp_after:wN \__fp_ln_twice_t_pack:Nw
26463      \int_value:w \__fp_int_eval:w 9999 + 2*#6
26464      \exp_after:wN \__fp_ln_twice_t_pack:Nw
26465      \int_value:w \__fp_int_eval:w 9999 + 2*#7
26466      \exp_after:wN \__fp_ln_twice_t_pack:Nw
26467      \int_value:w \__fp_int_eval:w 10000 + 2*#8 ; ;
26468      { \__fp_ln_c:NwNw #1 }
26469      #2
26470    }
26471    \cs_new:Npn \__fp_ln_twice_t_pack:Nw #1 #2; { + #1 ; {#2} }
26472    \cs_new:Npn \__fp_ln_twice_t_after:w #1; { ;; ; {#1} }
26473    \cs_new:Npn \__fp_ln_square_t_pack:NNNNw #1 #2#3#4#5 #6;
26474      { + #1#2#3#4#5 ; {#6} }
26475    \cs_new:Npn \__fp_ln_square_t_after:w 1 0 #1#2#3 #4;
26476      { \__fp_ln_Taylor:wwNw {0#1#2#3} {#4} }

```

(End definition for __fp_ln_x_ii:wnnnn.)

__fp_ln_Taylor:wwNw Denoting $T = t^2$, we get

```

\__fp_ln_Taylor:wwNw
{ \langle T_1 \rangle \langle T_2 \rangle \langle T_3 \rangle \langle T_4 \rangle \langle T_5 \rangle \langle T_6 \rangle ; ;
  \langle (2t)_1 \rangle \langle (2t)_2 \rangle \langle (2t)_3 \rangle \langle (2t)_4 \rangle \langle (2t)_5 \rangle \langle (2t)_6 \rangle ;
  { \__fp_ln_c:NwNw \langle sign \rangle }
  \langle fixed t1 \rangle \langle exponent \rangle ; \langle continuation \rangle

```

And we want to compute

$$\ln\left(\frac{1+t}{1-t}\right) = 2t \left(1 + T \left(\frac{1}{3} + T \left(\frac{1}{5} + T \left(\frac{1}{7} + T \left(\frac{1}{9} + \cdots\right)\right)\right)\right)\right)$$

The process looks as follows

```

\loop 5; A;
\div_int 5; 1.0; \add A; \mul T; {\loop \eval 5-2;}
\add 0.2; A; \mul T; {\loop \eval 5-2;}
\mul B; T; {\loop 3;}
\loop 3; C;

```

This uses the routine for dividing a number by a small integer ($< 10^4$).

```

26477 \cs_new:Npn \__fp_ln_Taylor:wwNw
26478   { \__fp_ln_Taylor_loop:www 21 ; {0000}{0000}{0000}{0000}{0000}{0000} ; }
26479 \cs_new:Npn \__fp_ln_Taylor_loop:www #1; #2; #3;
26480   {
26481     \if_int_compare:w #1 = \c_one_int
26482       \__fp_ln_Taylor_break:w
26483     \fi:
26484     \exp_after:wN \__fp_fixed_div_int:wwN \c__fp_one_fixed_t1 #1;
26485     \__fp_fixed_add:wwN #2;
26486     \__fp_fixed_mul:wwN #3;
26487     {
26488       \exp_after:wN \__fp_ln_Taylor_loop:www
26489       \int_value:w \__fp_int_eval:w #1 - 2 ;
26490     }

```



```

26491     #3;
26492   }
26493 \cs_new:Npn \__fp_ln_Taylor_break:w \fi: #1 \__fp_fixed_add:wwn #2#3; #4 ;;
26494   {
26495     \fi:
26496     \exp_after:wN \__fp_fixed_mul:wwn
26497     \exp_after:wN { \int_value:w \__fp_int_eval:w 10000 + #2 } #3;
26498   }

```

(End definition for __fp_ln_Taylor:wwNw.)

```

\__fp_ln_c:NwNw
    \__fp_ln_c:NwNw <sign>
    {\langle r_1 \rangle} {\langle r_2 \rangle} {\langle r_3 \rangle} {\langle r_4 \rangle} {\langle r_5 \rangle} {\langle r_6 \rangle} ;
    <fixed t1> <exponent> ; <continuation>

```

We are now reduced to finding $\ln(c)$ and $\langle exponent \rangle \ln(10)$ in a table, and adding it to the mixture. The first step is to get $\ln(c) - \ln(x) = -\ln(a)$, then we get $b \ln(10)$ and add or subtract.

For now, $\ln(x)$ is given as $\cdot 10^0$. Unless both the exponent is 1 and $c = 1$, we shift to working in units of $\cdot 10^4$, since the final result is at least $\ln(10/7) \simeq 0.35$.

```

26499 \cs_new:Npn \__fp_ln_c:NwNw #1 #2; #3
26500   {
26501     \if_meaning:w + #1
26502     \exp_after:wN \exp_after:wN \exp_after:wN \__fp_fixed_sub:wwn
26503     \else:
26504     \exp_after:wN \exp_after:wN \exp_after:wN \__fp_fixed_add:wwn
26505     \fi:
26506     #3 #2 ;
26507   }

```

(End definition for __fp_ln_c:NwNw.)

```

\__fp_ln_exponent:wn
    \__fp_ln_exponent:wn
    {\langle s_1 \rangle} {\langle s_2 \rangle} {\langle s_3 \rangle} {\langle s_4 \rangle} {\langle s_5 \rangle} {\langle s_6 \rangle} ;
    {\langle exponent \rangle}

```

Compute $\langle exponent \rangle$ times $\ln(10)$. Apart from the cases where $\langle exponent \rangle$ is 0 or 1, the result is necessarily at least $\ln(10) \simeq 2.3$ in magnitude. We can thus drop the least significant 4 digits. In the case of a very large (positive or negative) exponent, we can (and we need to) drop 4 additional digits, since the result is of order 10^4 . Naively, one would think that in both cases we can drop 4 more digits than we do, but that would be slightly too tight for rounding to happen correctly. Besides, we already have addition and subtraction for 24 digits fixed point numbers.

```

26508 \cs_new:Npn \__fp_ln_exponent:wn #1; #2
26509   {
26510     \if_case:w #2 \exp_stop_f:
26511     0 \__fp_case_return:nw { \__fp_fixed_to_float_o:Nw 2 }
26512     \or:
26513     \exp_after:wN \__fp_ln_exponent_one:ww \int_value:w
26514     \else:
26515     \if_int_compare:w #2 > \c_zero_int
26516     \exp_after:wN \__fp_ln_exponent_small:NNww
26517     \exp_after:wN 0
26518     \exp_after:wN \__fp_fixed_sub:wwn \int_value:w
26519     \else:

```

```

26520         \exp_after:wN \__fp_ln_exponent_small:NNww
26521         \exp_after:wN 2
26522         \exp_after:wN \__fp_fixed_add:wwn \int_value:w -
26523         \fi:
26524     \fi:
26525     #2; #1;
26526 }

```

Now we painfully write all the cases.¹² No overflow nor underflow can happen, except when computing $\ln(1)$.

```

26527 \cs_new:Npn \__fp_ln_exponent_one:ww #1; #1;
26528 {
26529     0
26530     \exp_after:wN \__fp_fixed_sub:wwn \c__fp_ln_x_fixed_tl #1;
26531     \__fp_fixed_to_float_o:wN 0
26532 }

```

For small exponents, we just drop one block of digits, and set the exponent of the log to 4 (minus any shift coming from leading zeros in the conversion from fixed point to floating point). Note that here the exponent has been made positive.

```

26533 \cs_new:Npn \__fp_ln_exponent_small:NNww #1#2#3; #4#5#6#7#8#9;
26534 {
26535     4
26536     \exp_after:wN \__fp_fixed_mul:wwn
26537         \c__fp_ln_x_fixed_tl
26538         {#3}{0000}{0000}{0000}{0000}{0000} ;
26539     #2
26540     {0000}{#4}{#5}{#6}{#7}{#8};
26541     \__fp_fixed_to_float_o:wN #1
26542 }

```

(End definition for `__fp_ln_exponent:wn`.)

74.2 Exponential

74.2.1 Sign, exponent, and special numbers

`__fp_exp_o:w`

```

26543 \cs_new:Npn \__fp_exp_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
26544 {
26545     \if_case:w #2 \exp_stop_f:
26546         \__fp_case_return_o:Nw \c_one_fp
26547     \or:
26548         \exp_after:wN \__fp_exp_normal_o:w
26549     \or:
26550         \if_meaning:w 0 #3
26551             \exp_after:wN \__fp_case_return_o:Nw
26552             \exp_after:wN \c_inf_fp
26553         \else:
26554             \exp_after:wN \__fp_case_return_o:Nw
26555             \exp_after:wN \c_zero_fp
26556         \fi:

```

¹²Bruno: do rounding.

```

26557     \or:
26558         \__fp_case_return_same_o:w
26559     \fi:
26560     \s__fp \__fp_chk:w #2#3#4;
26561 }

```

(End definition for __fp_exp_o:w.)

```

\__fp_exp_normal_o:w
\__fp_exp_pos_o:Nnwnw
\__fp_exp_overflow:NN

```

```

26562 \cs_new:Npn \__fp_exp_normal_o:w \s__fp \__fp_chk:w 1#1
26563 {
26564     \if_meaning:w 0 #1
26565         \__fp_exp_pos_o:Nnwnw + \__fp_fixed_to_float_o:wN
26566     \else:
26567         \__fp_exp_pos_o:Nnwnw - \__fp_fixed_inv_to_float_o:wN
26568     \fi:
26569 }
26570 \cs_new:Npn \__fp_exp_pos_o:Nnwnw #1#2#3 \fi: #4#5;
26571 {
26572     \fi:
26573     \if_int_compare:w #4 > \c__fp_max_exp_exponent_int
26574         \token_if_eq_charcode:NNTF + #1
26575         { \__fp_exp_overflow:NN \__fp_overflow:w \c_inf_fp }
26576         { \__fp_exp_overflow:NN \__fp_underflow:w \c_zero_fp }
26577     \exp:w
26578     \else:
26579         \exp_after:wN \__fp_sanitize:Nw
26580         \exp_after:wN 0
26581         \int_value:w #1 \__fp_int_eval:w
26582         \if_int_compare:w #4 < \c_zero_int
26583             \exp_after:wN \use_i:nn
26584         \else:
26585             \exp_after:wN \use_ii:nn
26586         \fi:
26587         {
26588             0
26589             \__fp_decimate:nNnnnn { - #4 }
26590             \__fp_exp_Taylor:Nnnwn
26591         }
26592         {
26593             \__fp_decimate:nNnnnn { \c__fp_prec_int - #4 }
26594             \__fp_exp_pos_large:NnnNwn
26595         }
26596         #5
26597         {#4}
26598         #1 #2 0
26599         \exp:w
26600     \fi:
26601     \exp_after:wN \exp_end:
26602 }
26603 \cs_new:Npn \__fp_exp_overflow:NN #1#2
26604 {
26605     \exp_after:wN \exp_after:wN
26606     \exp_after:wN #1

```

```

26607     \exp_after:wN #2
26608 }

```

(End definition for `_fp_exp_normal_o:w`, `_fp_exp_pos_o:Nnnwn`, and `_fp_exp_overflow:NN`.)

```

\_fp_exp_Taylor:Nnnwn
\_fp_exp_Taylor_loop:www
\_fp_exp_Taylor_break:Nww

```

This function is called for numbers in the range $[10^{-9}, 10^{-1}]$. We compute 10 terms of the Taylor series. The first argument is irrelevant (rounding digit used by some other functions). The next three arguments, at least 16 digits, delimited by a semicolon, form a fixed point number, so we pack it in blocks of 4 digits.

```

26609 \cs_new:Npn \_fp_exp_Taylor:Nnnwn #1#2#3 #4; #5 #6
26610 {
26611     #6
26612     \_fp_pack_twice_four:wNNNNNNNN
26613     \_fp_pack_twice_four:wNNNNNNNN
26614     \_fp_pack_twice_four:wNNNNNNNN
26615     \_fp_exp_Taylor_ii:ww
26616     ; #2#3#4 0000 0000 ;
26617 }
26618 \cs_new:Npn \_fp_exp_Taylor_ii:ww #1; #2;
26619 { \_fp_exp_Taylor_loop:www 10 ; #1 ; #1 ; \s_fp_stop }
26620 \cs_new:Npn \_fp_exp_Taylor_loop:www #1; #2; #3;
26621 {
26622     \if_int_compare:w #1 = \c_one_int
26623     \exp_after:wN \_fp_exp_Taylor_break:Nww
26624     \fi:
26625     \_fp_fixed_div_int:wwN #3 ; #1 ;
26626     \_fp_fixed_add_one:wN
26627     \_fp_fixed_mul:wwN #2 ;
26628     {
26629         \exp_after:wN \_fp_exp_Taylor_loop:www
26630         \int_value:w \_fp_int_eval:w #1 - 1 ;
26631         #2 ;
26632     }
26633 }
26634 \cs_new:Npn \_fp_exp_Taylor_break:Nww #1 #2; #3 \s_fp_stop
26635 { \_fp_fixed_add_one:wN #2 ; }

```

(End definition for `_fp_exp_Taylor:Nnnwn`, `_fp_exp_Taylor_loop:www`, and `_fp_exp_Taylor_break:Nww`.)

```
\c__fp_exp_intarray
```

The integer array has $6 \times 9 \times 4 = 216$ items encoding the values of $\exp(j \times 10^i)$ for $j = 1, \dots, 9$ and $i = -1, \dots, 4$. Each value is expressed as $\simeq 10^p \times 0.m_1m_2m_3$ with three 8-digit blocks m_1, m_2, m_3 and an integer exponent p (one more than the scientific exponent), and these are stored in the integer array as four items: $p, 10^8 + m_1, 10^8 + m_2, 10^8 + m_3$. The various exponentials are stored in increasing order of $j \times 10^i$.

Storing this data in an integer array makes it slightly harder to access (slower, too), but uses 16 bytes of memory per exponential stored, while storing as tokens used around 40 tokens; tokens have an especially large footprint in Unicode-aware engines.

```

26636 \intarray_const_from_clist:Nn \c__fp_exp_intarray
26637 {
26638     1 , 1 1105 1709 , 1 1807 5647 , 1 6248 1171 ,
26639     1 , 1 1221 4027 , 1 5816 0169 , 1 8339 2107 ,
26640     1 , 1 1349 8588 , 1 0757 6003 , 1 1039 8374 ,
26641     1 , 1 1491 8246 , 1 9764 1270 , 1 3178 2485 ,

```

```

26642      1 , 1 1648 7212 , 1 7070 0128 , 1 1468 4865 ,
26643      1 , 1 1822 1188 , 1 0039 0508 , 1 9748 7537 ,
26644      1 , 1 2013 7527 , 1 0747 0476 , 1 5216 2455 ,
26645      1 , 1 2225 5409 , 1 2849 2467 , 1 6045 7954 ,
26646      1 , 1 2459 6031 , 1 1115 6949 , 1 6638 0013 ,
26647      1 , 1 2718 2818 , 1 2845 9045 , 1 2353 6029 ,
26648      1 , 1 7389 0560 , 1 9893 0650 , 1 2272 3043 ,
26649      2 , 1 2008 5536 , 1 9231 8766 , 1 7740 9285 ,
26650      2 , 1 5459 8150 , 1 0331 4423 , 1 9078 1103 ,
26651      3 , 1 1484 1315 , 1 9102 5766 , 1 0342 1116 ,
26652      3 , 1 4034 2879 , 1 3492 7351 , 1 2260 8387 ,
26653      4 , 1 1096 6331 , 1 5842 8458 , 1 5992 6372 ,
26654      4 , 1 2980 9579 , 1 8704 1728 , 1 2747 4359 ,
26655      4 , 1 8103 0839 , 1 2757 5384 , 1 0077 1000 ,
26656      5 , 1 2202 6465 , 1 7948 0671 , 1 6516 9579 ,
26657      9 , 1 4851 6519 , 1 5409 7902 , 1 7796 9107 ,
26658     14 , 1 1068 6474 , 1 5815 2446 , 1 2146 9905 ,
26659     18 , 1 2353 8526 , 1 6837 0199 , 1 8540 7900 ,
26660     22 , 1 5184 7055 , 1 2858 7072 , 1 4640 8745 ,
26661     27 , 1 1142 0073 , 1 8981 5684 , 1 2836 6296 ,
26662     31 , 1 2515 4386 , 1 7091 9167 , 1 0062 6578 ,
26663     35 , 1 5540 6223 , 1 8439 3510 , 1 0525 7117 ,
26664     40 , 1 1220 4032 , 1 9431 7840 , 1 8020 0271 ,
26665     44 , 1 2688 1171 , 1 4181 6135 , 1 4484 1263 ,
26666     87 , 1 7225 9737 , 1 6812 5749 , 1 2581 7748 ,
26667    131 , 1 1942 4263 , 1 9524 1255 , 1 9365 8421 ,
26668    174 , 1 5221 4696 , 1 8976 4143 , 1 9505 8876 ,
26669    218 , 1 1403 5922 , 1 1785 2837 , 1 4107 3977 ,
26670    261 , 1 3773 0203 , 1 0092 9939 , 1 8234 0143 ,
26671    305 , 1 1014 2320 , 1 5473 5004 , 1 5094 5533 ,
26672    348 , 1 2726 3745 , 1 7211 2566 , 1 5673 6478 ,
26673    391 , 1 7328 8142 , 1 2230 7421 , 1 7051 8866 ,
26674    435 , 1 1970 0711 , 1 1401 7046 , 1 9938 8888 ,
26675    869 , 1 3881 1801 , 1 9428 4368 , 1 5764 8232 ,
26676   1303 , 1 7646 2009 , 1 8905 4704 , 1 8893 1073 ,
26677   1738 , 1 1506 3559 , 1 7005 0524 , 1 9009 7592 ,
26678   2172 , 1 2967 6283 , 1 8402 3667 , 1 0689 6630 ,
26679   2606 , 1 5846 4389 , 1 5650 2114 , 1 7278 5046 ,
26680   3041 , 1 1151 7900 , 1 5080 6878 , 1 2914 4154 ,
26681   3475 , 1 2269 1083 , 1 0850 6857 , 1 8724 4002 ,
26682   3909 , 1 4470 3047 , 1 3316 5442 , 1 6408 6591 ,
26683   4343 , 1 8806 8182 , 1 2566 2921 , 1 5872 6150 ,
26684   8686 , 1 7756 0047 , 1 2598 6861 , 1 0458 3204 ,
26685  13029 , 1 6830 5723 , 1 7791 4884 , 1 1932 7351 ,
26686  17372 , 1 6015 5609 , 1 3095 3052 , 1 3494 7574 ,
26687  21715 , 1 5297 7951 , 1 6443 0315 , 1 3251 3576 ,
26688  26058 , 1 4665 6719 , 1 0099 3379 , 1 5527 2929 ,
26689  30401 , 1 4108 9724 , 1 3326 3186 , 1 5271 5665 ,
26690  34744 , 1 3618 6973 , 1 3140 0875 , 1 3856 4102 ,
26691  39087 , 1 3186 9209 , 1 6113 3900 , 1 6705 9685 ,
26692      }

```

(End definition for \c__fp_exp_intarray.)

```

\__fp_exp_pos_large:NnnNwn
\__fp_exp_large_after:wnn
  \__fp_exp_large:NwN
    \__fp_exp_intarray:w
      \__fp_exp_intarray_aux:w

```

The first two arguments are irrelevant (a rounding digit, and a brace group with 8 zeros).

The third argument is the integer part of our number, then we have the decimal part delimited by a semicolon, and finally the exponent, in the range [0, 5]. Remove leading zeros from the integer part: putting #4 in there too ensures that an integer part of 0 is also removed. Then read digits one by one, looking up $\exp(\langle digit \rangle \cdot 10^{\langle exponent \rangle})$ in a table, and multiplying that to the current total. The loop is done by `__fp_exp_large:NwN`, whose #1 is the $\langle exponent \rangle$, #2 is the current mantissa, and #3 is the $\langle digit \rangle$. At the end, `__fp_exp_large_after:wnn` moves on to the Taylor series, eventually multiplied with the mantissa that we have just computed.

```

26693 \cs_new:Npn \__fp_exp_pos_large:NnnNwn #1#2#3 #4#5; #6
26694 {
26695     \exp_after:wN \exp_after:wN \exp_after:wN \__fp_exp_large:NwN
26696     \exp_after:wN \exp_after:wN \exp_after:wN #6
26697     \exp_after:wN \c__fp_one_fixed_tl
26698     \int_value:w #3 #4 \exp_stop_f:
26699     #5 00000 ;
26700 }
26701 \cs_new:Npn \__fp_exp_large:NwN #1#2; #3
26702 {
26703     \if_case:w #3 ~
26704         \exp_after:wN \__fp_fixed_continue:wn
26705     \else:
26706         \exp_after:wN \__fp_exp_intarray:w
26707         \int_value:w \__fp_int_eval:w 36 * #1 + 4 * #3 \exp_after:wN ;
26708     \fi:
26709     #2;
26710     {
26711         \if_meaning:w 0 #1
26712             \exp_after:wN \__fp_exp_large_after:wnn
26713         \else:
26714             \exp_after:wN \__fp_exp_large:NwN
26715             \int_value:w \__fp_int_eval:w #1 - 1 \exp_after:wN \scan_stop:
26716         \fi:
26717     }
26718 }
26719 \cs_new:Npn \__fp_exp_intarray:w #1 ;
26720 {
26721     +
26722     \__kernel_intarray_item:Nn \c__fp_exp_intarray
26723     { \__fp_int_eval:w #1 - 3 \scan_stop: }
26724     \exp_after:wN \use_i:nnn
26725     \exp_after:wN \__fp_fixed_mul:wnn
26726     \int_value:w 0
26727     \exp_after:wN \__fp_exp_intarray_aux:w
26728     \int_value:w \__kernel_intarray_item:Nn
26729         \c__fp_exp_intarray { \__fp_int_eval:w #1 - 2 }
26730     \exp_after:wN \__fp_exp_intarray_aux:w
26731     \int_value:w \__kernel_intarray_item:Nn
26732         \c__fp_exp_intarray { \__fp_int_eval:w #1 - 1 }
26733     \exp_after:wN \__fp_exp_intarray_aux:w
26734     \int_value:w \__kernel_intarray_item:Nn \c__fp_exp_intarray {#1} ; ;
26735 }
26736 \cs_new:Npn \__fp_exp_intarray_aux:w 1 #1#2#3#4#5 ; { ; {#1#2#3#4} {#5} }
26737 \cs_new:Npn \__fp_exp_large_after:wnn #1; #2; #3

```

```

26738 {
26739     \__fp_exp_Taylor:Nnnwn ? { } { } 0 #2; {} #3
26740     \__fp_fixed_mul:wn #1;
26741 }

```

(End definition for `__fp_exp_pos_large:NnnNwn` and others.)

74.3 Power

Raising a number a to a power b leads to many distinct situations.

a^b	$-\infty$	$(-\infty, -0)$	$-\text{integer}$	± 0	$+\text{integer}$	$(0, \infty)$	$+\infty$	nan
$+\infty$	$+0$		$+0$	$+1$	$+\infty$		$+\infty$	nan
$(1, \infty)$	$+0$		$+ a ^b$	$+1$	$+ a ^b$		$+\infty$	nan
$+1$	$+1$		$+1$	$+1$	$+1$		$+1$	$+1$
$(0, 1)$	$+\infty$		$+ a ^b$	$+1$	$+ a ^b$		$+0$	nan
$+0$	$+\infty$		$+\infty$	$+1$	$+0$		$+0$	nan
-0	$+\infty$	nan	$(-1)^b \infty$	$+1$	$(-1)^b 0$	$+0$	$+0$	nan
$(-1, 0)$	$+\infty$	nan	$(-1)^b a ^b$	$+1$	$(-1)^b a ^b$	nan	$+0$	nan
-1	$+1$	nan	$(-1)^b$	$+1$	$(-1)^b$	nan	$+1$	nan
$(-\infty, -1)$	$+0$	nan	$(-1)^b a ^b$	$+1$	$(-1)^b a ^b$	nan	$+\infty$	nan
$-\infty$	$+0$	$+0$	$(-1)^b 0$	$+1$	$(-1)^b \infty$	nan	$+\infty$	nan
nan	nan	nan	nan	$+1$	nan	nan	nan	nan

We distinguished in this table the cases of finite (positive or negative) integer exponents, as $(-1)^b$ is defined in that case. One peculiarity of this operation is that $\text{nan}^0 = 1^{\text{nan}} = 1$, because this relation is obeyed for any number, even $\pm\infty$.

`__fp^_o:ww` We cram most of the tests into a single function to save csnames. First treat the case $b = 0$: $a^0 = 1$ for any a , even nan . Then test the sign of a .

- If it is positive, and a is a normal number, call `__fp_pow_normal_o:ww` followed by the two `fp` a and b . For $a = +0$ or $+\text{inf}$, call `__fp_pow_zero_or_inf:ww` instead, to return either $+0$ or $+\infty$ as appropriate.
- If a is a nan , then skip to the next semicolon (which happens to be conveniently the end of b) and return nan .
- Finally, if a is negative, compute a^b (`__fp_pow_normal_o:ww` which ignores the sign of its first operand), and keep an extra copy of a and b (the second brace group, containing $\{ b \}$, is inserted between a and b). Then do some tests to find the final sign of the result if it exists.

```

26742 \cs_new:cpn { __fp_ \iow_char:N \^_o:ww }
26743     \s__fp \__fp_chk:w #1#2#3; \s__fp \__fp_chk:w #4#5#6;
26744 {
26745     \if_meaning:w 0 #4
26746         \__fp_case_return_o:Nw \c_one_fp
26747     \fi:
26748     \if_case:w #2 \exp_stop_f:
26749         \exp_after:wN \use_i:nn
26750     \or:
26751         \__fp_case_return_o:Nw \c_nan_fp

```

```

26752 \else:
26753 \exp_after:wN \__fp_pow_neg:ww
26754 \exp:w \exp_end_continue_f:w \exp_after:wN \use:nn
26755 \fi:
26756 {
26757 \if_meaning:w 1 #1
26758 \exp_after:wN \__fp_pow_normal_o:ww
26759 \else:
26760 \exp_after:wN \__fp_pow_zero_or_inf:ww
26761 \fi:
26762 \s__fp \__fp_chk:w #1#2#3;
26763 }
26764 { \s__fp \__fp_chk:w #4#5#6; \s__fp \__fp_chk:w #1#2#3; }
26765 \s__fp \__fp_chk:w #4#5#6;
26766 }

```

(End definition for $__fp_o:ww$.)

$__fp_pow_zero_or_inf:ww$

Raising -0 or $-\infty$ to `nan` yields `nan`. For other powers, the result is $+0$ if 0 is raised to a positive power or ∞ to a negative power, and $+\infty$ otherwise. Thus, if the type of a and the sign of b coincide, the result is 0 , since those conveniently take the same possible values, 0 and 2 . Otherwise, either $a = \pm\infty$ and $b > 0$ and the result is $+\infty$, or $a = \pm 0$ with $b < 0$ and we have a division by zero unless $b = -\infty$.

```

26767 \cs_new:Npn \__fp_pow_zero_or_inf:ww
26768 \s__fp \__fp_chk:w #1#2; \s__fp \__fp_chk:w #3#4
26769 {
26770 \if_meaning:w 1 #4
26771 \__fp_case_return_same_o:w
26772 \fi:
26773 \if_meaning:w #1 #4
26774 \__fp_case_return_o:Nw \c_zero_fp
26775 \fi:
26776 \if_meaning:w 2 #1
26777 \__fp_case_return_o:Nw \c_inf_fp
26778 \fi:
26779 \if_meaning:w 2 #3
26780 \__fp_case_return_o:Nw \c_inf_fp
26781 \else:
26782 \__fp_case_use:nw
26783 {
26784 \__fp_division_by_zero_o:NNww \c_inf_fp ^
26785 \s__fp \__fp_chk:w #1 #2 ;
26786 }
26787 \fi:
26788 \s__fp \__fp_chk:w #3#4
26789 }

```

(End definition for $__fp_pow_zero_or_inf:ww$.)

$__fp_pow_normal_o:ww$

We have in front of us a , and $b \neq 0$, we know that a is a normal number, and we wish to compute $|a|^b$. If $|a| = 1$, we return 1 , unless $a = -1$ and b is `nan`. Indeed, returning 1 at this point would wrongly raise “invalid” when the sign is considered. If $|a| \neq 1$, test the type of b :

0 Impossible, we already filtered $b = \pm 0$.

- 1 Call `__fp_pow_npos_o:Nww`.
- 2 Return $+\infty$ or $+0$ depending on the sign of b and whether the exponent of a is positive or not.
- 3 Return b .

```

26790 \cs_new:Npn \__fp_pow_normal_o:ww
26791   \s__fp \__fp_chk:w 1 #1#2#3; \s__fp \__fp_chk:w #4#5
26792   {
26793     \if:w 0 \__fp_str_if_eq:nn { #2 #3 } { 1 {1000} {0000} {0000} {0000} }
26794     \if_int_compare:w #4 #1 = 32 \exp_stop_f:
26795       \exp_after:wN \__fp_case_return_ii_o:ww
26796       \fi:
26797       \__fp_case_return_o:Nww \c_one_fp
26798     \fi:
26799     \if_case:w #4 \exp_stop_f:
26800     \or:
26801       \exp_after:wN \__fp_pow_npos_o:Nww
26802       \exp_after:wN #5
26803     \or:
26804       \if_meaning:w 2 #5 \exp_after:wN \reverse_if:N \fi:
26805       \if_int_compare:w #2 > \c_zero_int
26806         \exp_after:wN \__fp_case_return_o:Nww
26807         \exp_after:wN \c_inf_fp
26808       \else:
26809         \exp_after:wN \__fp_case_return_o:Nww
26810         \exp_after:wN \c_zero_fp
26811       \fi:
26812     \or:
26813       \__fp_case_return_ii_o:ww
26814     \fi:
26815     \s__fp \__fp_chk:w 1 #1 {#2} #3 ;
26816     \s__fp \__fp_chk:w #4 #5
26817   }

```

(End definition for `__fp_pow_normal_o:ww`.)

`__fp_pow_npos_o:Nww` We now know that $a \neq \pm 1$ is a normal number, and b is a normal number too. We want to compute $|a|^b = (|x| \cdot 10^n)^y \cdot 10^p = \exp((\ln|x| + n \ln(10)) \cdot y \cdot 10^p) = \exp(z)$. To compute the exponential accurately, we need to know the digits of z up to the 16-th position. Since the exponential of 10^5 is infinite, we only need at most 21 digits, hence the fixed point result of `__fp_ln_o:w` is precise enough for our needs. Start an integer expression for the decimal exponent of $e^{|z|}$. If z is negative, negate that decimal exponent, and prepare to take the inverse when converting from the fixed point to the floating point result.

```

26818 \cs_new:Npn \__fp_pow_npos_o:Nww #1 \s__fp \__fp_chk:w 1#2#3
26819   {
26820     \exp_after:wN \__fp_sanitise:Nw
26821     \exp_after:wN 0
26822     \int_value:w
26823     \if:w #1 \if_int_compare:w #3 > \c_zero_int 0 \else: 2 \fi:
26824     \exp_after:wN \__fp_pow_npos_aux:NNww
26825     \exp_after:wN +
26826     \exp_after:wN \__fp_fixed_to_float_o:wN

```

```

26827     \else:
26828         \exp_after:wN \__fp_pow_npos_aux:NNnw
26829         \exp_after:wN -
26830         \exp_after:wN \__fp_fixed_inv_to_float_o:wN
26831     \fi:
26832     {#3}
26833 }

```

(End definition for __fp_pow_npos_o:Nnw.)

__fp_pow_npos_aux:NNnw The first argument is the conversion function from fixed point to float. Then comes an exponent and the 4 brace groups of x , followed by b . Compute $-\ln(x)$.

```

26834 \cs_new:Npn \__fp_pow_npos_aux:NNnw #1#2#3#4#5; \s__fp \__fp_chk:w 1#6#7#8;
26835 {
26836     #1
26837     \__fp_int_eval:w
26838     \__fp_ln_significand:NNNNnnN #4#5
26839     \__fp_pow_exponent:wnN {#3}
26840     \__fp_fixed_mul:wwN #8 {0000}{0000} ;
26841     \__fp_pow_B:wwN #7;
26842     #1 #2 0 % fixed_to_float_o:wN
26843 }
26844 \cs_new:Npn \__fp_pow_exponent:wnN #1; #2
26845 {
26846     \if_int_compare:w #2 > \c_zero_int
26847         \exp_after:wN \__fp_pow_exponent:Nwnnnnw % n\ln(10) - (-\ln(x))
26848         \exp_after:wN +
26849     \else:
26850         \exp_after:wN \__fp_pow_exponent:Nwnnnnw % -(ln|\ln(10) + (-\ln(x)))
26851         \exp_after:wN -
26852     \fi:
26853     #2; #1;
26854 }
26855 \cs_new:Npn \__fp_pow_exponent:Nwnnnnw #1#2; #3#4#5#6#7#8;
26856 { %^^A todo: use that in ln.
26857     \exp_after:wN \__fp_fixed_mul_after:wwN
26858     \int_value:w \__fp_int_eval:w \c__fp_leading_shift_int
26859     \exp_after:wN \__fp_pack:NNNNw
26860     \int_value:w \__fp_int_eval:w \c__fp_middle_shift_int
26861     #1#2*23025 - #1 #3
26862     \exp_after:wN \__fp_pack:NNNNw
26863     \int_value:w \__fp_int_eval:w \c__fp_middle_shift_int
26864     #1 #2*8509 - #1 #4
26865     \exp_after:wN \__fp_pack:NNNNw
26866     \int_value:w \__fp_int_eval:w \c__fp_middle_shift_int
26867     #1 #2*2994 - #1 #5
26868     \exp_after:wN \__fp_pack:NNNNw
26869     \int_value:w \__fp_int_eval:w \c__fp_middle_shift_int
26870     #1 #2*0456 - #1 #6
26871     \exp_after:wN \__fp_pack:NNNNw
26872     \int_value:w \__fp_int_eval:w \c__fp_trailing_shift_int
26873     #1 #2*8401 - #1 #7
26874     #1 ( #2*7991 - #8 ) / 1 0000 ; ;
26875 }

```

```

26876 \cs_new:Npn \__fp_pow_B:wwN #1#2#3#4#5#6; #7;
26877 {
26878   \if_int_compare:w #7 < \c_zero_int
26879     \exp_after:wN \__fp_pow_C_neg:w \int_value:w -
26880   \else:
26881     \if_int_compare:w #7 < 22 \exp_stop_f:
26882     \exp_after:wN \__fp_pow_C_pos:w \int_value:w
26883   \else:
26884     \exp_after:wN \__fp_pow_C_overflow:w \int_value:w
26885   \fi:
26886   \fi:
26887   #7 \exp_after:wN ;
26888   \int_value:w \__fp_int_eval:w 10 0000 + #1 \__fp_int_eval_end:
26889   #2#3#4#5#6 0000 0000 0000 0000 0000 0000 ; %^A todo: how many 0?
26890 }
26891 \cs_new:Npn \__fp_pow_C_overflow:w #1; #2; #3
26892 {
26893   + 2 * \c__fp_max_exponent_int
26894   \exp_after:wN \__fp_fixed_continue:wn \c__fp_one_fixed_tl
26895 }
26896 \cs_new:Npn \__fp_pow_C_neg:w #1 ; 1
26897 {
26898   \exp_after:wN \exp_after:wN \exp_after:wN \__fp_pow_C_pack:w
26899   \prg_replicate:nn {#1} {0}
26900 }
26901 \cs_new:Npn \__fp_pow_C_pos:w #1; 1
26902 { \__fp_pow_C_pos_loop:wN #1; }
26903 \cs_new:Npn \__fp_pow_C_pos_loop:wN #1; #2
26904 {
26905   \if_meaning:w 0 #1
26906     \exp_after:wN \__fp_pow_C_pack:w
26907     \exp_after:wN #2
26908   \else:
26909     \if_meaning:w 0 #2
26910     \exp_after:wN \__fp_pow_C_pos_loop:wN \int_value:w
26911   \else:
26912     \exp_after:wN \__fp_pow_C_overflow:w \int_value:w
26913   \fi:
26914   \__fp_int_eval:w #1 - 1 \exp_after:wN ;
26915   \fi:
26916 }
26917 \cs_new:Npn \__fp_pow_C_pack:w
26918 {
26919   \exp_after:wN \__fp_exp_large:NwN
26920   \exp_after:wN 5
26921   \c__fp_one_fixed_tl
26922 }

```

(End definition for __fp_pow_npos_aux:NNnw.)

__fp_pow_neg:www
 __fp_pow_neg_aux:wNN

This function is followed by three floating point numbers: a^b , $a \in [-\infty, -0]$, and b . If b is an even integer (case -1), $a^b = a^b$. If b is an odd integer (case 0), $a^b = -a^b$, obtained by a call to __fp_pow_neg_aux:wNN. Otherwise, the sign is undefined. This is invalid, unless a^b turns out to be $+0$ or nan , in which case we return that as a^b . In particular, since the

underflow detection occurs before `__fp_pow_neg:www` is called, $(-0.1)**(12345.67)$ gives +0 rather than complaining that the sign is not defined.

```

26923 \cs_new:Npn \__fp_pow_neg:www \s__fp \__fp_chk:w #1#2; #3; #4;
26924 {
26925   \if_case:w \__fp_pow_neg_case:w #4 ;
26926   \exp_after:wN \__fp_pow_neg_aux:wNN
26927   \or:
26928   \if_int_compare:w \__fp_int_eval:w #1 / 2 = \c_one_int
26929   \__fp_invalid_operation_o:Nww ^ #3; #4;
26930   \exp:w \exp_end_continue_f:w
26931   \exp_after:wN \exp_after:wN
26932   \exp_after:wN \__fp_use_none_until_s:w
26933   \fi:
26934   \fi:
26935   \__fp_exp_after_o:w
26936   \s__fp \__fp_chk:w #1#2;
26937 }
26938 \cs_new:Npn \__fp_pow_neg_aux:wNN #1 \s__fp \__fp_chk:w #2#3
26939 {
26940   \exp_after:wN \__fp_exp_after_o:w
26941   \exp_after:wN \s__fp
26942   \exp_after:wN \__fp_chk:w
26943   \exp_after:wN #2
26944   \int_value:w \__fp_int_eval:w 2 - #3 \__fp_int_eval_end:
26945 }

```

(End definition for `__fp_pow_neg:www` and `__fp_pow_neg_aux:wNN`.)

```

\__fp_pow_neg_case:w
\__fp_pow_neg_case_aux:nnnnn
\__fp_pow_neg_case_aux:Nnnw

```

This function expects a floating point number, and determines its “parity”. It should be used after `\if_case:w` or in an integer expression. It gives -1 if the number is an even integer, 0 if the number is an odd integer, and 1 otherwise. Zeros and $\pm\infty$ are even (because very large finite floating points are even), while `nan` is a non-integer. The sign of normal numbers is irrelevant to parity. After `__fp_decimate:nNnnnn` the argument `#1` of `__fp_pow_neg_case_aux:Nnnw` is a rounding digit, 0 if and only if the number was an integer, and `#3` is the 8 least significant digits of that integer.

```

26946 \cs_new:Npn \__fp_pow_neg_case:w \s__fp \__fp_chk:w #1#2#3;
26947 {
26948   \if_case:w #1 \exp_stop_f:
26949   -1
26950   \or: \__fp_pow_neg_case_aux:nnnnn #3
26951   \or: -1
26952   \else: 1
26953   \fi:
26954   \exp_stop_f:
26955 }
26956 \cs_new:Npn \__fp_pow_neg_case_aux:nnnnn #1#2#3#4#5
26957 {
26958   \if_int_compare:w #1 > \c__fp_prec_int
26959   -1
26960   \else:
26961   \__fp_decimate:nNnnnn { \c__fp_prec_int - #1 }
26962   \__fp_pow_neg_case_aux:Nnnw
26963   {#2} {#3} {#4} {#5}

```

```

26964     \fi:
26965   }
26966 \cs_new:Npn \__fp_pow_neg_case_aux:Nnnw #1#2#3#4 ;
26967 {
26968   \if_meaning:w 0 #1
26969     \if_int_odd:w #3 \exp_stop_f:
26970       0
26971     \else:
26972       -1
26973     \fi:
26974   \else:
26975     1
26976   \fi:
26977 }

```

(End definition for __fp_pow_neg_case:w, __fp_pow_neg_case_aux:nnnnn, and __fp_pow_neg_case_aux:Nnnw.)

74.4 Factorial

\c__fp_fact_max_arg_int The maximum integer whose factorial fits in the exponent range is 3248, as $3249! \sim 10^{10000.8}$

```

26978 \int_const:Nn \c__fp_fact_max_arg_int { 3248 }

```

(End definition for \c__fp_fact_max_arg_int.)

__fp_fact_o:w First detect ± 0 and $+\infty$ and `nan`. Then note that factorial of anything with a negative sign (except -0) is undefined. Then call __fp_small_int:wTF to get an integer as the argument, and start a loop. This is not the most efficient way of computing the factorial, but it works all right. Of course we work with 24 digits instead of 16. It is easy to check that computing factorials with this precision is enough.

```

26979 \cs_new:Npn \__fp_fact_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
26980 {
26981   \if_case:w #2 \exp_stop_f:
26982     \__fp_case_return_o:Nw \c_one_fp
26983   \or:
26984   \or:
26985     \if_meaning:w 0 #3
26986       \exp_after:wN \__fp_case_return_same_o:w
26987     \fi:
26988   \or:
26989     \__fp_case_return_same_o:w
26990   \fi:
26991   \if_meaning:w 2 #3
26992     \__fp_case_use:nw { \__fp_invalid_operation_o:fw { fact } }
26993   \fi:
26994   \__fp_fact_pos_o:w
26995   \s__fp \__fp_chk:w #2 #3 #4 ;
26996 }

```

(End definition for __fp_fact_o:w.)

_fp_fact_pos_o:w Then check the input is an integer, and call _fp_facorial_int_o:n with that int as an argument. If it's too big the factorial overflows. Otherwise call _fp_sanitize:Nw with a positive sign marker 0 and an integer expression that will mop up any exponent in the calculation.

```

26997 \cs_new:Npn \_fp_fact_pos_o:w #1;
26998 {
26999   \_fp_small_int:wTF #1;
27000   { \_fp_fact_int_o:n }
27001   { \_fp_invalid_operation_o:fw { fact } #1; }
27002 }
27003 \cs_new:Npn \_fp_fact_int_o:n #1
27004 {
27005   \if_int_compare:w #1 > \c__fp_fact_max_arg_int
27006   \_fp_case_return:nw
27007   {
27008     \exp_after:wN \exp_after:wN \exp_after:wN \_fp_overflow:w
27009     \exp_after:wN \c_inf_fp
27010   }
27011   \fi:
27012   \exp_after:wN \_fp_sanitize:Nw
27013   \exp_after:wN 0
27014   \int_value:w \_fp_int_eval:w
27015   \_fp_fact_loop_o:w #1 . 4 , { 1 } { } { } { } { } { } ;
27016 }

```

(End definition for _fp_fact_pos_o:w and _fp_fact_int_o:w.)

_fp_fact_loop_o:w The loop receives an integer #1 whose factorial we want to compute, which we progressively decrement, and the result so far as an extended-precision number #2 in the form $\langle \text{exponent} \rangle, \langle \text{mantissa} \rangle$; . The loop goes in steps of two because we compute $\#1 \cdot \#1 - 1$ as an integer expression (it must fit since #1 is at most 3248), then multiply with the result so far. We don't need to fill in most of the mantissa with zeros because _fp_ep_mul:wwwN first normalizes the extended precision number to avoid loss of precision. When reaching a small enough number simply use a table of factorials less than 10^8 . This limit is chosen because the normalization step cannot deal with larger integers.

```

27017 \cs_new:Npn \_fp_fact_loop_o:w #1 . #2 ;
27018 {
27019   \if_int_compare:w #1 < 12 \exp_stop_f:
27020   \_fp_fact_small_o:w #1
27021   \fi:
27022   \exp_after:wN \_fp_ep_mul:wwwN
27023   \exp_after:wN 4 \exp_after:wN ,
27024   \exp_after:wN { \int_value:w \_fp_int_eval:w #1 * (#1 - 1) }
27025   { } { } { } { } { } { } ;
27026   #2 ;
27027   {
27028     \exp_after:wN \_fp_fact_loop_o:w
27029     \int_value:w \_fp_int_eval:w #1 - 2 .
27030   }
27031 }
27032 \cs_new:Npn \_fp_fact_small_o:w #1 \fi: #2 ; #3 ; #4
27033 {
27034   \fi:

```

```

27035 \exp_after:wN \_fp_ep_mul:wwwwn
27036 \exp_after:wN 4 \exp_after:wN ,
27037 \exp_after:wN
27038 {
27039   \int_value:w
27040   \if_case:w #1 \exp_stop_f:
27041   1 \or: 1 \or: 2 \or: 6 \or: 24 \or: 120 \or: 720 \or: 5040
27042   \or: 40320 \or: 362880 \or: 3628800 \or: 39916800
27043   \fi:
27044   } { } { } { } { } { } { } ;
27045   #3 ;
27046   \_fp_ep_to_float_o:wwN 0
27047 }

(End definition for \_fp_fact_loop_o:w.)

27048 </package>

```

Chapter 75

l3fp-trig Implementation

```
27049 ⟨*package⟩
27050 ⟨@@=fp⟩

\__fp_parse_word_acos:N
\__fp_parse_word_acosd:N
\__fp_parse_word_acsc:N
\__fp_parse_word_acscd:N
\__fp_parse_word_asec:N
\__fp_parse_word_asecd:N
\__fp_parse_word_asin:N
\__fp_parse_word_asind:N
\__fp_parse_word_cos:N
\__fp_parse_word_cosd:N
\__fp_parse_word_cot:N
\__fp_parse_word_cotd:N
\__fp_parse_word_csc:N
\__fp_parse_word_cscd:N
\__fp_parse_word_sec:N
\__fp_parse_word_secd:N
\__fp_parse_word_sin:N
\__fp_parse_word_sind:N
\__fp_parse_word_tan:N
\__fp_parse_word_tand:N

Unary functions.
27051 \tl_map_inline:nn
27052 {
27053   {acos} {acsc} {asec} {asin}
27054   {cos} {cot} {csc} {sec} {sin} {tan}
27055 }
27056 {
27057   \cs_new:cpx { __fp_parse_word_#1:N }
27058   {
27059     \exp_not:N \__fp_parse_unary_function:NNN
27060     \exp_not:c { __fp_#1_o:w }
27061     \exp_not:N \use_i:nn
27062   }
27063   \cs_new:cpx { __fp_parse_word_#1d:N }
27064   {
27065     \exp_not:N \__fp_parse_unary_function:NNN
27066     \exp_not:c { __fp_#1_o:w }
27067     \exp_not:N \use_ii:nn
27068   }
27069 }

(End definition for \__fp_parse_word_acos:N and others.)

\__fp_parse_word_acot:N
\__fp_parse_word_acotd:N
\__fp_parse_word_atan:N
\__fp_parse_word_atand:N

Those functions may receive a variable number of arguments.
27070 \cs_new:Npn \__fp_parse_word_acot:N
27071 { \__fp_parse_function:NNN \__fp_acot_o:Nw \use_i:nn }
27072 \cs_new:Npn \__fp_parse_word_acotd:N
27073 { \__fp_parse_function:NNN \__fp_acot_o:Nw \use_ii:nn }
27074 \cs_new:Npn \__fp_parse_word_atan:N
27075 { \__fp_parse_function:NNN \__fp_atan_o:Nw \use_i:nn }
27076 \cs_new:Npn \__fp_parse_word_atand:N
27077 { \__fp_parse_function:NNN \__fp_atan_o:Nw \use_ii:nn }

(End definition for \__fp_parse_word_acot:N and others.)
```


75.1 Direct trigonometric functions

The approach for all trigonometric functions (sine, cosine, tangent, cotangent, cosecant, and secant), with arguments given in radians or in degrees, is the same.

- Filter out special cases (± 0 , $\pm \infty$ and **nan**).
- Keep the sign for later, and work with the absolute value $|x|$ of the argument.
- Small numbers ($|x| < 1$ in radians, $|x| < 10$ in degrees) are converted to fixed point numbers (and to radians if $|x|$ is in degrees).
- For larger numbers, we need argument reduction. Subtract a multiple of $\pi/2$ (in degrees, 90) to bring the number to the range to $[0, \pi/2)$ (in degrees, $[0, 90)$).
- Reduce further to $[0, \pi/4]$ (in degrees, $[0, 45]$) using $\sin x = \cos(\pi/2 - x)$, and when working in degrees, convert to radians.
- Use the appropriate power series depending on the octant $\lfloor \frac{x}{\pi/4} \rfloor \bmod 8$ (in degrees, the same formula with $\pi/4 \rightarrow 45$), the sign, and the function to compute.

75.1.1 Filtering special cases

`__fp_sin_o:w` This function, and its analogs for `cos`, `csc`, `sec`, `tan`, and `cot` instead of `sin`, are followed either by `\use_i:nn` and a float in radians or by `\use_ii:nn` and a float in degrees. The sine of ± 0 or **nan** is the same float. The sine of $\pm \infty$ raises an invalid operation exception with the appropriate function name. Otherwise, call the `trig` function to perform argument reduction and if necessary convert the reduced argument to radians. Then, `__fp_sin_series_o:NNwww` is called to compute the Taylor series: this function receives a sign `#3`, an initial octant of 0, and the function `__fp_ep_to_float_o:wwN` which converts the result of the series to a floating point directly rather than taking its inverse, since $\sin(x) = \#3 \sin|x|$.

```

27078 \cs_new:Npn \__fp_sin_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
27079 {
27080     \if_case:w #2 \exp_stop_f:
27081         \__fp_case_return_same_o:w
27082     \or: \__fp_case_use:nw
27083         {
27084             \__fp_trig:NNNNwn #1 \__fp_sin_series_o:NNwww
27085             \__fp_ep_to_float_o:wwN #3 0
27086         }
27087     \or: \__fp_case_use:nw
27088         { \__fp_invalid_operation_o:fw { #1 { sin } { sind } } }
27089     \else: \__fp_case_return_same_o:w
27090     \fi:
27091     \s__fp \__fp_chk:w #2 #3 #4;
27092 }
```

(End definition for `__fp_sin_o:w`.)

`__fp_cos_o:w` The cosine of ± 0 is 1. The cosine of $\pm \infty$ raises an invalid operation exception. The cosine of **nan** is itself. Otherwise, the `trig` function reduces the argument to at most half a right-angle and converts if necessary to radians. We then call the same series as

for sine, but using a positive sign 0 regardless of the sign of x , and with an initial octant of 2, because $\cos(x) = +\sin(\pi/2 + |x|)$.

```

27093 \cs_new:Npn \__fp_cos_o:w #1 \s__fp \__fp_chk:w #2#3; @
27094 {
27095   \if_case:w #2 \exp_stop_f:
27096     \__fp_case_return_o:Nw \c_one_fp
27097   \or: \__fp_case_use:nw
27098     {
27099       \__fp_trig:NNNNNwn #1 \__fp_sin_series_o:NNwww
27100       \__fp_ep_to_float_o:wwN 0 2
27101     }
27102   \or: \__fp_case_use:nw
27103     { \__fp_invalid_operation_o:fw { #1 { cos } { cosd } } }
27104   \else: \__fp_case_return_same_o:w
27105   \fi:
27106   \s__fp \__fp_chk:w #2 #3;
27107 }

```

(End definition for `__fp_cos_o:w`.)

`__fp_csc_o:w` The cosecant of ± 0 is $\pm\infty$ with the same sign, with a division by zero exception (see `__fp_cot_zero_o:Nfw` defined below), which requires the function name. The cosecant of $\pm\infty$ raises an invalid operation exception. The cosecant of `nan` is itself. Otherwise, the `trig` function performs the argument reduction, and converts if necessary to radians before calling the same series as for sine, using the sign #3, a starting octant of 0, and inverting during the conversion from the fixed point sine to the floating point result, because $\csc(x) = \#3(\sin|x|)^{-1}$.

```

27108 \cs_new:Npn \__fp_csc_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
27109 {
27110   \if_case:w #2 \exp_stop_f:
27111     \__fp_cot_zero_o:Nfw #3 { #1 { csc } { cscd } }
27112   \or: \__fp_case_use:nw
27113     {
27114       \__fp_trig:NNNNNwn #1 \__fp_sin_series_o:NNwww
27115       \__fp_ep_inv_to_float_o:wwN #3 0
27116     }
27117   \or: \__fp_case_use:nw
27118     { \__fp_invalid_operation_o:fw { #1 { csc } { cscd } } }
27119   \else: \__fp_case_return_same_o:w
27120   \fi:
27121   \s__fp \__fp_chk:w #2 #3 #4;
27122 }

```

(End definition for `__fp_csc_o:w`.)

`__fp_sec_o:w` The secant of ± 0 is 1. The secant of $\pm\infty$ raises an invalid operation exception. The secant of `nan` is itself. Otherwise, the `trig` function reduces the argument and turns it to radians before calling the same series as for sine, using a positive sign 0, a starting octant of 2, and inverting upon conversion, because $\sec(x) = +1/\sin(\pi/2 + |x|)$.

```

27123 \cs_new:Npn \__fp_sec_o:w #1 \s__fp \__fp_chk:w #2#3; @
27124 {
27125   \if_case:w #2 \exp_stop_f:
27126     \__fp_case_return_o:Nw \c_one_fp

```

```

27127 \or: \__fp_case_use:nw
27128 {
27129     \__fp_trig:NNNNNwn #1 \__fp_sin_series_o:NNwww
27130     \__fp_ep_inv_to_float_o:wwN 0 2
27131 }
27132 \or: \__fp_case_use:nw
27133 { \__fp_invalid_operation_o:fw { #1 { sec } { secd } } }
27134 \else: \__fp_case_return_same_o:w
27135 \fi:
27136 \s__fp \__fp_chk:w #2 #3;
27137 }

```

(End definition for __fp_sec_o:w.)

__fp_tan_o:w The tangent of ± 0 or `nan` is the same floating point number. The tangent of $\pm\infty$ raises an invalid operation exception. Once more, the `trig` function does the argument reduction step and conversion to radians before calling `__fp_tan_series_o:NNwww`, with a sign `#3` and an initial octant of 1 (this shift is somewhat arbitrary). See `__fp_cot_o:w` for an explanation of the 0 argument.

```

27138 \cs_new:Npn \__fp_tan_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
27139 {
27140     \if_case:w #2 \exp_stop_f:
27141         \__fp_case_return_same_o:w
27142     \or: \__fp_case_use:nw
27143         {
27144             \__fp_trig:NNNNNwn #1
27145             \__fp_tan_series_o:NNwww 0 #3 1
27146         }
27147     \or: \__fp_case_use:nw
27148         { \__fp_invalid_operation_o:fw { #1 { tan } { tand } } }
27149     \else: \__fp_case_return_same_o:w
27150     \fi:
27151     \s__fp \__fp_chk:w #2 #3 #4;
27152 }

```

(End definition for __fp_tan_o:w.)

__fp_cot_o:w The cotangent of ± 0 is $\pm\infty$ with the same sign, with a division by zero exception (see `__fp_cot_zero_o:Nfw`). The cotangent of $\pm\infty$ raises an invalid operation exception. The cotangent of `nan` is itself. We use $\cot x = -\tan(\pi/2 + x)$, and the initial octant for the tangent was chosen to be 1, so the octant here starts at 3. The change in sign is obtained by feeding `__fp_tan_series_o:NNwww` two signs rather than just the sign of the argument: the first of those indicates whether we compute tangent or cotangent. Those signs are eventually combined.

```

27153 \cs_new:Npn \__fp_cot_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
27154 {
27155     \if_case:w #2 \exp_stop_f:
27156         \__fp_cot_zero_o:Nfw #3 { #1 { cot } { cotd } }
27157     \or: \__fp_case_use:nw
27158         {
27159             \__fp_trig:NNNNNwn #1
27160             \__fp_tan_series_o:NNwww 2 #3 3
27161         }
27162     \or: \__fp_case_use:nw

```

```

27163         { \__fp_invalid_operation_o:fw { #1 { cot } { cotd } } }
27164     \else: \__fp_case_return_same_o:w
27165     \fi:
27166     \s__fp \__fp_chk:w #2 #3 #4;
27167 }
27168 \cs_new:Npn \__fp_cot_zero_o:Nfw #1#2#3 \fi:
27169 {
27170     \fi:
27171     \token_if_eq_meaning:NNTF 0 #1
27172     { \exp_args:NNf \__fp_division_by_zero_o:Nnw \c_inf_fp }
27173     { \exp_args:NNf \__fp_division_by_zero_o:Nnw \c_minus_inf_fp }
27174     {#2}
27175 }

```

(End definition for __fp_cot_o:w and __fp_cot_zero_o:Nfw.)

75.1.2 Distinguishing small and large arguments

__fp_trig:NNNNNwn The first argument is \use_i:nn if the operand is in radians and \use_ii:nn if it is in degrees. Arguments #2 to #5 control what trigonometric function we compute, and #6 to #8 are pieces of a normal floating point number. Call the `_series` function #2, with arguments #3, either a conversion function (`__fp_ep_to_float_o:wN` or `__fp_ep_inv_to_float_o:wN`) or a sign 0 or 2 when computing tangent or cotangent; #4, a sign 0 or 2; the octant, computed in an integer expression starting with #5 and stopped by a period; and a fixed point number obtained from the floating point number by argument reduction (if necessary) and conversion to radians (if necessary). Any argument reduction adjusts the octant accordingly by leaving a (positive) shift into its integer expression. Let us explain the integer comparison. Two of the four `\exp_after:wN` are expanded, the expansion hits the test, which is true if the float is at least 1 when working in radians, and at least 10 when working in degrees. Then one of the remaining `\exp_after:wN` hits #1, which picks the `trig` or `trigd` function in whichever branch of the conditional was taken. The final `\exp_after:wN` closes the conditional. At the end of the day, a number is `large` if it is ≥ 1 in radians or ≥ 10 in degrees, and `small` otherwise. All four `trig/trigd` auxiliaries receive the operand as an extended-precision number.

```

27176 \cs_new:Npn \__fp_trig:NNNNNwn #1#2#3#4#5 \s__fp \__fp_chk:w 1#6#7#8;
27177 {
27178     \exp_after:wN #2
27179     \exp_after:wN #3
27180     \exp_after:wN #4
27181     \int_value:w \__fp_int_eval:w #5
27182     \exp_after:wN \exp_after:wN \exp_after:wN \exp_after:wN
27183     \if_int_compare:w #7 > #1 0 1 \exp_stop_f:
27184     #1 \__fp_trig_large:ww \__fp_trigd_large:ww
27185     \else:
27186     #1 \__fp_trig_small:ww \__fp_trigd_small:ww
27187     \fi:
27188     #7,#8{0000}{0000};
27189 }

```

(End definition for __fp_trig:NNNNNwn.)

75.1.3 Small arguments

`__fp_trig_small:ww` This receives a small extended-precision number in radians and converts it to a fixed point number. Some trailing digits may be lost in the conversion, so we keep the original floating point number around: when computing sine or tangent (or their inverses), the last step is to multiply by the floating point number (as an extended-precision number) rather than the fixed point number. The period serves to end the integer expression for the octant.

```
27190 \cs_new:Npn \__fp_trig_small:ww #1,#2;
27191 { \__fp_ep_to_fixed:wwn #1,#2; . #1,#2; }
```

(End definition for `__fp_trig_small:ww`.)

`__fp_trigd_small:ww` Convert the extended-precision number to radians, then call `__fp_trig_small:ww` to massage it in the form appropriate for the `_series` auxiliary.

```
27192 \cs_new:Npn \__fp_trigd_small:ww #1,#2;
27193 {
27194   \__fp_ep_mul_raw:wwwN
27195   -1,{1745}{3292}{5199}{4329}{5769}{2369}; #1,#2;
27196   \__fp_trig_small:ww
27197 }
```

(End definition for `__fp_trigd_small:ww`.)

75.1.4 Argument reduction in degrees

`__fp_trigd_large:ww` Note that $25 \times 360 = 9000$, so $10^{k+1} \equiv 10^k \pmod{360}$ for $k \geq 3$. When the exponent `#1` is very large, we can thus safely replace it by 22 (or even 19). We turn the floating point number into a fixed point number with two blocks of 8 digits followed by five blocks of 4 digits. The original float is $100 \times \langle block_1 \rangle \cdots \langle block_3 \rangle . \langle block_4 \rangle \cdots \langle block_7 \rangle$, or is equal to it modulo 360 if the exponent `#1` is very large. The first auxiliary finds $\langle block_1 \rangle + \langle block_2 \rangle \pmod{9}$, a single digit, and prepends it to the 4 digits of $\langle block_3 \rangle$. It also unpacks $\langle block_4 \rangle$ and grabs the 4 digits of $\langle block_7 \rangle$. The second auxiliary grabs the $\langle block_3 \rangle$ plus any contribution from the first two blocks as `#1`, the first digit of $\langle block_4 \rangle$ (just after the decimal point in hundreds of degrees) as `#2`, and the three other digits as `#3`. It finds the quotient and remainder of `#1#2` modulo 9, adds twice the quotient to the integer expression for the octant, and places the remainder (between 0 and 8) before `#3` to form a new $\langle block_4 \rangle$. The resulting fixed point number is $x \in [0, 0.9]$. If $x \geq 0.45$, we add 1 to the octant and feed $0.9 - x$ with an exponent of 2 (to compensate the fact that we are working in units of hundreds of degrees rather than degrees) to `__fp_trigd_small:ww`. Otherwise, we feed it x with an exponent of 2. The third auxiliary also discards digits which were not packed into the various $\langle blocks \rangle$. Since the original exponent `#1` is at least 2, those are all 0 and no precision is lost (`#6` and `#7` are four 0 each).

```
27198 \cs_new:Npn \__fp_trigd_large:ww #1, #2#3#4#5#6#7;
27199 {
27200   \exp_after:wN \__fp_pack_eight:wNNNNNNNN
27201   \exp_after:wN \__fp_pack_eight:wNNNNNNNN
27202   \exp_after:wN \__fp_pack_twice_four:wNNNNNNNN
27203   \exp_after:wN \__fp_pack_twice_four:wNNNNNNNN
27204   \exp_after:wN \__fp_trigd_large_auxi:nnnnwNNNN
27205   \exp_after:wN ;
27206   \exp:w \exp_end_continue_f:w
```

```

27207 \prg_replicate:nn { \int_max:nn { 22 - #1 } { 0 } } { 0 }
27208 #2#3#4#5#6#7 0000 0000 0000 !
27209 }
27210 \cs_new:Npn \__fp_trigd_large_auxi:nnnnwNNNN #1#2#3#4#5; #6#7#8#9
27211 {
27212 \exp_after:wN \__fp_trigd_large_auxii:wNw
27213 \int_value:w \__fp_int_eval:w #1 + #2
27214 - (#1 + #2 - 4) / 9 * 9 \__fp_int_eval_end:
27215 #3;
27216 #4; #5{#6#7#8#9};
27217 }
27218 \cs_new:Npn \__fp_trigd_large_auxii:wNw #1; #2#3;
27219 {
27220 + (#1#2 - 4) / 9 * 2
27221 \exp_after:wN \__fp_trigd_large_auxiii:www
27222 \int_value:w \__fp_int_eval:w #1#2
27223 - (#1#2 - 4) / 9 * 9 \__fp_int_eval_end: #3 ;
27224 }
27225 \cs_new:Npn \__fp_trigd_large_auxiii:www #1; #2; #3!
27226 {
27227 \if_int_compare:w #1 < 4500 \exp_stop_f:
27228 \exp_after:wN \__fp_use_i_until:s:nw
27229 \exp_after:wN \__fp_fixed_continue:wn
27230 \else:
27231 + 1
27232 \fi:
27233 \__fp_fixed_sub:wnn {9000}{0000}{0000}{0000}{0000}{0000};
27234 {#1}#2{0000}{0000};
27235 { \__fp_trigd_small:ww 2, }
27236 }

```

(End definition for `__fp_trigd_large:ww` and others.)

75.1.5 Argument reduction in radians

Arguments greater or equal to 1 need to be reduced to a range where we only need a few terms of the Taylor series. We reduce to the range $[0, 2\pi]$ by subtracting multiples of 2π , then to the smaller range $[0, \pi/2]$ by subtracting multiples of $\pi/2$ (keeping track of how many times $\pi/2$ is subtracted), then to $[0, \pi/4]$ by mapping $x \rightarrow \pi/2 - x$ if appropriate. When the argument is very large, say, 10^{100} , an equally large multiple of 2π must be subtracted, hence we must work with a very good approximation of 2π in order to get a sensible remainder modulo 2π .

Specifically, we multiply the argument by an approximation of $1/(2\pi)$ with 10048 digits, then discard the integer part of the result, keeping 52 digits of the fractional part. From the fractional part of $x/(2\pi)$ we deduce the octant (quotient of the first three digits by 125). We then multiply by 8 or -8 (the latter when the octant is odd), ignore any integer part (related to the octant), and convert the fractional part to an extended precision number, before multiplying by $\pi/4$ to convert back to a value in radians in $[0, \pi/4]$.

It is possible to prove that given the precision of floating points and their range of exponents, the 52 digits may start at most with 24 zeros. The 5 last digits are affected by carries from computations which are not done, hence we are left with at least $52 - 24 - 5 = 23$ significant digits, enough to round correctly up to $0.6 \cdot \text{ulp}$ in all cases.

`\c__fp_trig_intarray` This integer array stores blocks of 8 decimals of $10^{-16}/(2\pi)$. Each entry is 10^8 plus an 8 digit number storing 8 decimals. In total we store 10112 decimals of $10^{-16}/(2\pi)$. The number of decimals we really need is the maximum exponent plus the number of digits we later need, 52, plus 12 (4 – 1 groups of 4 digits). The memory footprint (1/2 byte per digit) is the same as an earlier method of storing the data as a control sequence name, but the major advantage is that we can unpack specific subsets of the digits without unpacking the 10112 decimals.

```

27237 \intarray_const_from_clist:Nn \c__fp_trig_intarray
27238 {
27239     100000000, 100000000, 115915494, 130918953, 135768883, 176337251,
27240     143620344, 159645740, 145644874, 176673440, 158896797, 163422653,
27241     150901138, 102766253, 108595607, 128427267, 157958036, 189291184,
27242     161145786, 152877967, 141073169, 198392292, 139966937, 140907757,
27243     130777463, 196925307, 168871739, 128962173, 197661693, 136239024,
27244     117236290, 111832380, 111422269, 197557159, 140461890, 108690267,
27245     139561204, 189410936, 193784408, 155287230, 199946443, 140024867,
27246     123477394, 159610898, 132309678, 130749061, 166986462, 180469944,
27247     186521878, 181574786, 156696424, 110389958, 174139348, 160998386,
27248     180991999, 162442875, 158517117, 188584311, 117518767, 116054654,
27249     175369880, 109739460, 136475933, 137680593, 102494496, 163530532,
27250     171567755, 103220324, 177781639, 171660229, 146748119, 159816584,
27251     106060168, 103035998, 113391198, 174988327, 186654435, 127975507,
27252     100162406, 177564388, 184957131, 108801221, 199376147, 168137776,
27253     147378906, 133068046, 145797848, 117613124, 127314069, 196077502,
27254     145002977, 159857089, 105690279, 167851315, 125210016, 131774602,
27255     109248116, 106240561, 145620314, 164840892, 148459191, 143521157,
27256     154075562, 100871526, 160680221, 171591407, 157474582, 172259774,
27257     162853998, 175155329, 139081398, 117724093, 158254797, 107332871,
27258     190406999, 175907657, 170784934, 170393589, 182808717, 134256403,
27259     166895116, 162545705, 194332763, 112686500, 126122717, 197115321,
27260     112599504, 138667945, 103762556, 108363171, 116952597, 158128224,
27261     194162333, 143145106, 112353687, 185631136, 136692167, 114206974,
27262     169601292, 150578336, 105311960, 185945098, 139556718, 170995474,
27263     165104316, 123815517, 158083944, 129799709, 199505254, 138756612,
27264     194458833, 106846050, 178529151, 151410404, 189298850, 163881607,
27265     176196993, 107341038, 199957869, 118905980, 193737772, 106187543,
27266     122271893, 101366255, 126123878, 103875388, 181106814, 106765434,
27267     108282785, 126933426, 179955607, 107903860, 160352738, 199624512,
27268     159957492, 176297023, 159409558, 143011648, 129641185, 157771240,
27269     157544494, 157021789, 176979240, 194903272, 194770216, 164960356,
27270     153181535, 144003840, 168987471, 176915887, 163190966, 150696440,
27271     147769706, 187683656, 177810477, 197954503, 153395758, 130188183,
27272     186879377, 166124814, 195305996, 155802190, 183598751, 103512712,
27273     190432315, 180498719, 168687775, 194656634, 162210342, 104440855,
27274     149785037, 192738694, 129353661, 193778292, 187359378, 143470323,
27275     102371458, 137923557, 111863634, 119294601, 183182291, 196416500,
27276     187830793, 131353497, 179099745, 186492902, 167450609, 189368909,
27277     145883050, 133703053, 180547312, 132158094, 131976760, 132283131,
27278     141898097, 149822438, 133517435, 169898475, 101039500, 168388003,
27279     197867235, 199608024, 100273901, 108749548, 154787923, 156826113,
27280     199489032, 168997427, 108349611, 149208289, 103776784, 174303550,
27281     145684560, 183671479, 130845672, 133270354, 185392556, 120208683,
27282     193240995, 162211753, 131839402, 109707935, 170774965, 149880868,

```

27283 160663609, 168661967, 103747454, 121028312, 119251846, 122483499,
27284 111611495, 166556037, 196967613, 199312829, 196077608, 127799010,
27285 107830360, 102338272, 198790854, 102387615, 157445430, 192601191,
27286 100543379, 198389046, 154921248, 129516070, 172853005, 122721023,
27287 160175233, 113173179, 175931105, 103281551, 109373913, 163964530,
27288 157926071, 180083617, 195487672, 146459804, 173977292, 144810920,
27289 109371257, 186918332, 189588628, 139904358, 168666639, 175673445,
27290 114095036, 137327191, 174311388, 106638307, 125923027, 159734506,
27291 105482127, 178037065, 133778303, 121709877, 134966568, 149080032,
27292 169885067, 141791464, 168350828, 116168533, 114336160, 173099514,
27293 198531198, 119733758, 144420984, 116559541, 152250643, 139431286,
27294 144403838, 183561508, 179771645, 101706470, 167518774, 156059160,
27295 187168578, 157939226, 123475633, 117111329, 198655941, 159689071,
27296 198506887, 144230057, 151919770, 156900382, 118392562, 120338742,
27297 135362568, 108354156, 151729710, 188117217, 195936832, 156488518,
27298 174997487, 108553116, 159830610, 113921445, 144601614, 188452770,
27299 125114110, 170248521, 173974510, 138667364, 103872860, 109967489,
27300 131735618, 112071174, 104788993, 168886556, 192307848, 150230570,
27301 157144063, 163863202, 136852010, 174100574, 185922811, 115721968,
27302 100397824, 175953001, 166958522, 112303464, 118773650, 143546764,
27303 164565659, 171901123, 108476709, 193097085, 191283646, 166919177,
27304 169387914, 133315566, 150669813, 121641521, 100895711, 172862384,
27305 126070678, 145176011, 113450800, 169947684, 122356989, 162488051,
27306 157759809, 153397080, 185475059, 175362656, 149034394, 145420581,
27307 178864356, 183042000, 131509559, 147434392, 152544850, 167491429,
27308 108647514, 142303321, 133245695, 111634945, 167753939, 142403609,
27309 105438335, 152829243, 142203494, 184366151, 146632286, 102477666,
27310 166049531, 140657343, 157553014, 109082798, 180914786, 169343492,
27311 127376026, 134997829, 195701816, 119643212, 133140475, 176289748,
27312 140828911, 174097478, 126378991, 181699939, 148749771, 151989818,
27313 172666294, 160183053, 195832752, 109236350, 168538892, 128468247,
27314 125997252, 183007668, 156937583, 165972291, 198244297, 147406163,
27315 181831139, 158306744, 134851692, 185973832, 137392662, 140243450,
27316 119978099, 140402189, 161348342, 173613676, 144991382, 171541660,
27317 163424829, 136374185, 106122610, 186132119, 198633462, 184709941,
27318 183994274, 129559156, 128333990, 148038211, 175011612, 111667205,
27319 119125793, 103552929, 124113440, 131161341, 112495318, 138592695,
27320 184904438, 146807849, 109739828, 108855297, 104515305, 139914009,
27321 188698840, 188365483, 166522246, 168624087, 125401404, 100911787,
27322 142122045, 123075334, 173972538, 114940388, 141905868, 142311594,
27323 163227443, 139066125, 116239310, 162831953, 123883392, 113153455,
27324 163815117, 152035108, 174595582, 101123754, 135976815, 153401874,
27325 107394340, 136339780, 138817210, 104531691, 182951948, 179591767,
27326 139541778, 179243527, 161740724, 160593916, 102732282, 187946819,
27327 136491289, 149714953, 143255272, 135916592, 198072479, 198580612,
27328 169007332, 118844526, 179433504, 155801952, 149256630, 162048766,
27329 116134365, 133992028, 175452085, 155344144, 109905129, 182727454,
27330 165911813, 122232840, 151166615, 165070983, 175574337, 129548631,
27331 120411217, 116380915, 160616116, 157320000, 183306114, 160618128,
27332 103262586, 195951602, 146321661, 138576614, 180471993, 127077713,
27333 116441201, 159496011, 106328305, 120759583, 148503050, 179095584,
27334 198298218, 167402898, 138551383, 123957020, 180763975, 150429225,
27335 198476470, 171016426, 197438450, 143091658, 164528360, 132493360,
27336 143546572, 137557916, 113663241, 120457809, 196971566, 134022158,

27337 180545794, 131328278, 100552461, 132088901, 187421210, 192448910,
27338 141005215, 149680971, 113720754, 100571096, 134066431, 135745439,
27339 191597694, 135788920, 179342561, 177830222, 137011486, 142492523,
27340 192487287, 113132021, 176673607, 156645598, 127260957, 141566023,
27341 143787436, 129132109, 174858971, 150713073, 191040726, 143541417,
27342 197057222, 165479803, 181512759, 157912400, 125344680, 148220261,
27343 173422990, 101020483, 106246303, 137964746, 178190501, 181183037,
27344 151538028, 179523433, 141955021, 135689770, 191290561, 143178787,
27345 192086205, 174499925, 178975690, 118492103, 124206471, 138519113,
27346 188147564, 102097605, 154895793, 178514140, 141453051, 151583964,
27347 128232654, 106020603, 131189158, 165702720, 186250269, 191639375,
27348 115278873, 160608114, 155694842, 110322407, 177272742, 116513642,
27349 134366992, 171634030, 194053074, 180652685, 109301658, 192136921,
27350 141431293, 171341061, 157153714, 106203978, 147618426, 150297807,
27351 186062669, 169960809, 118422347, 163350477, 146719017, 145045144,
27352 161663828, 146208240, 186735951, 102371302, 190444377, 194085350,
27353 134454426, 133413062, 163074595, 113830310, 122931469, 134466832,
27354 185176632, 182415152, 110179422, 164439571, 181217170, 121756492,
27355 119644493, 196532222, 118765848, 182445119, 109401340, 150443213,
27356 198586286, 121083179, 139396084, 143898019, 114787389, 177233102,
27357 186310131, 148695521, 126205182, 178063494, 157118662, 177825659,
27358 188310053, 151552316, 165984394, 109022180, 163144545, 121212978,
27359 197344714, 188741258, 126822386, 102360271, 109981191, 152056882,
27360 134723983, 158013366, 106837863, 128867928, 161973236, 172536066,
27361 185216856, 132011948, 197807339, 158419190, 166595838, 167852941,
27362 124187182, 117279875, 106103946, 106481958, 157456200, 160892122,
27363 184163943, 173846549, 158993202, 184812364, 133466119, 170732430,
27364 195458590, 173361878, 162906318, 150165106, 126757685, 112163575,
27365 188696307, 145199922, 100107766, 176830946, 198149756, 122682434,
27366 179367131, 108412102, 119520899, 148191244, 140487511, 171059184,
27367 141399078, 189455775, 118462161, 190415309, 134543802, 180893862,
27368 180732375, 178615267, 179711433, 123241969, 185780563, 176301808,
27369 184386640, 160717536, 183213626, 129671224, 126094285, 140110963,
27370 121826276, 151201170, 122552929, 128965559, 146082049, 138409069,
27371 107606920, 103954646, 119164002, 115673360, 117909631, 187289199,
27372 186343410, 186903200, 157966371, 103128612, 135698881, 176403642,
27373 152540837, 109810814, 183519031, 121318624, 172281810, 150845123,
27374 169019064, 166322359, 138872454, 163073727, 128087898, 130041018,
27375 194859136, 173742589, 141812405, 167291912, 138003306, 134499821,
27376 196315803, 186381054, 124578934, 150084553, 128031351, 118843410,
27377 107373060, 159565443, 173624887, 171292628, 198074235, 139074061,
27378 178690578, 144431052, 174262641, 176783005, 182214864, 162289361,
27379 192966929, 192033046, 169332843, 181580535, 164864073, 118444059,
27380 195496893, 153773183, 167266131, 130108623, 158802128, 180432893,
27381 144562140, 147978945, 142337360, 158506327, 104399819, 132635916,
27382 168734194, 136567839, 101281912, 120281622, 195003330, 112236091,
27383 185875592, 101959081, 122415367, 194990954, 148881099, 175891989,
27384 108115811, 163538891, 163394029, 123722049, 184837522, 142362091,
27385 100834097, 156679171, 100841679, 157022331, 178971071, 102928884,
27386 189701309, 195339954, 124415335, 106062584, 139214524, 133864640,
27387 134324406, 157317477, 155340540, 144810061, 177612569, 108474646,
27388 114329765, 143900008, 138265211, 145210162, 136643111, 197987319,
27389 102751191, 144121361, 169620456, 193602633, 161023559, 162140467,
27390 102901215, 167964187, 135746835, 187317233, 110047459, 163339773,

27391	124770449,	118885134,	141536376,	100915375,	164267438,	145016622,
27392	113937193,	106748706,	128815954,	164819775,	119220771,	102367432,
27393	189062690,	170911791,	194127762,	112245117,	123546771,	115640433,
27394	135772061,	166615646,	174474627,	130562291,	133320309,	153340551,
27395	138417181,	194605321,	150142632,	180008795,	151813296,	175497284,
27396	167018836,	157425342,	150169942,	131069156,	134310662,	160434122,
27397	105213831,	158797111,	150754540,	163290657,	102484886,	148697402,
27398	187203725,	198692811,	149360627,	140384233,	128749423,	132178578,
27399	177507355,	171857043,	178737969,	134023369,	102911446,	196144864,
27400	197697194,	134527467,	144296030,	189437192,	154052665,	188907106,
27401	162062575,	150993037,	199766583,	167936112,	181374511,	104971506,
27402	115378374,	135795558,	167972129,	135876446,	130937572,	103221320,
27403	124605656,	161129971,	131027586,	191128460,	143251843,	143269155,
27404	129284585,	173495971,	150425653,	199302112,	118494723,	121323805,
27405	116549802,	190991967,	168151180,	122483192,	151273721,	199792134,
27406	133106764,	121874844,	126215985,	112167639,	167793529,	182985195,
27407	185453921,	106957880,	158685312,	132775454,	133229161,	198905318,
27408	190537253,	191582222,	192325972,	178133427,	181825606,	148823337,
27409	160719681,	101448145,	131983362,	137910767,	112550175,	128826351,
27410	183649210,	135725874,	110356573,	189469487,	154446940,	118175923,
27411	106093708,	128146501,	185742532,	149692127,	164624247,	183221076,
27412	154737505,	168198834,	156410354,	158027261,	125228550,	131543250,
27413	139591848,	191898263,	104987591,	115406321,	103542638,	190012837,
27414	142615518,	178773183,	175862355,	117537850,	169565995,	170028011,
27415	158412588,	170150030,	117025916,	174630208,	142412449,	112839238,
27416	105257725,	114737141,	123102301,	172563968,	130555358,	132628403,
27417	183638157,	168682846,	143304568,	105994018,	170010719,	152092970,
27418	117799058,	132164175,	179868116,	158654714,	177489647,	116547948,
27419	183121404,	131836079,	184431405,	157311793,	149677763,	173989893,
27420	102277656,	107058530,	140837477,	152640947,	143507039,	152145247,
27421	101683884,	107090870,	161471944,	137225650,	128231458,	172995869,
27422	173831689,	171268519,	139042297,	111072135,	107569780,	137262545,
27423	181410950,	138270388,	198736451,	162848201,	180468288,	120582913,
27424	153390138,	135649144,	130040157,	106509887,	192671541,	174507066,
27425	186888783,	143805558,	135011967,	145862340,	180595327,	124727843,
27426	182925939,	157715840,	136885940,	198993925,	152416883,	178793572,
27427	179679516,	154076673,	192703125,	164187609,	162190243,	104699348,
27428	159891990,	160012977,	174692145,	132970421,	167781726,	115178506,
27429	153008552,	155999794,	102099694,	155431545,	127458567,	104403686,
27430	168042864,	184045128,	181182309,	179349696,	127218364,	192935516,
27431	120298724,	169583299,	148193297,	183358034,	159023227,	105261254,
27432	121144370,	184359584,	194433836,	138388317,	175184116,	108817112,
27433	151279233,	137457721,	193398208,	119005406,	132929377,	175306906,
27434	160741530,	149976826,	147124407,	176881724,	186734216,	185881509,
27435	191334220,	175930947,	117385515,	193408089,	157124410,	163472089,
27436	131949128,	180783576,	131158294,	100549708,	191802336,	165960770,
27437	170927599,	101052702,	181508688,	197828549,	143403726,	142729262,
27438	110348701,	139928688,	153550062,	106151434,	130786653,	196085995,
27439	100587149,	139141652,	106530207,	100852656,	124074703,	166073660,
27440	153338052,	163766757,	120188394,	197277047,	122215363,	138511354,
27441	183463624,	161985542,	159938719,	133367482,	104220974,	149956672,
27442	170250544,	164232439,	157506869,	159133019,	137469191,	142980999,
27443	134242305,	150172665,	121209241,	145596259,	160554427,	159095199,
27444	168243130,	184279693,	171132070,	121049823,	123819574,	171759855,

```

27445      119501864, 163094029, 175943631, 194450091, 191506160, 149228764,
27446      132319212, 197034460, 193584259, 126727638, 168143633, 109856853,
27447      127860243, 132141052, 133076065, 188414958, 158718197, 107124299,
27448      159592267, 181172796, 144388537, 196763139, 127431422, 179531145,
27449      100064922, 112650013, 132686230, 121550837,
27450  }

```

(End definition for \c__fp_trig_intarray.)

__fp_trig_large:ww The exponent #1 is between 1 and 10000. We wish to look up decimals $10^{\#1-16}/(2\pi)$ starting from the digit #1 + 1. Since they are stored in batches of 8, compute $\lceil \#1/8 \rceil$ and fetch blocks of 8 digits starting there. The numbering of items in \c__fp_trig_intarray starts at 1, so the block $\lceil \#1/8 \rceil + 1$ contains the digit we want, at one of the eight positions. Each call to \int_value:w __kernel_intarray_item:Nn expands the next, until being stopped by __fp_trig_large_auxiii:w using \exp_stop_f:. Once all these blocks are unpacked, the \exp_stop_f: and 0 to 7 digits are removed by \use_none:n...n. Finally, __fp_trig_large_auxii:w packs 64 digits (there are between 65 and 72 at this point) into groups of 4 and the auxv auxiliary is called.

```

27451 \cs_new:Npn \__fp_trig_large:ww #1, #2#3#4#5#6;
27452 {
27453   \exp_after:wN \__fp_trig_large_auxi:w
27454   \int_value:w \__fp_int_eval:w (#1 - 4) / 8 \exp_after:wN ,
27455   \int_value:w #1 , ;
27456   {#2}{#3}{#4}{#5} ;
27457 }
27458 \cs_new:Npn \__fp_trig_large_auxi:w #1, #2,
27459 {
27460   \exp_after:wN \exp_after:wN
27461   \exp_after:wN \__fp_trig_large_auxii:w
27462   \cs:w
27463     use_none:n \prg_replicate:nn { #2 - #1 * 8 } { n }
27464   \exp_after:wN
27465   \cs_end:
27466   \int_value:w
27467   \__kernel_intarray_item:Nn \c__fp_trig_intarray
27468     { \__fp_int_eval:w #1 + 1 \scan_stop: }
27469   \exp_after:wN \__fp_trig_large_auxiii:w \int_value:w
27470   \__kernel_intarray_item:Nn \c__fp_trig_intarray
27471     { \__fp_int_eval:w #1 + 2 \scan_stop: }
27472   \exp_after:wN \__fp_trig_large_auxiii:w \int_value:w
27473   \__kernel_intarray_item:Nn \c__fp_trig_intarray
27474     { \__fp_int_eval:w #1 + 3 \scan_stop: }
27475   \exp_after:wN \__fp_trig_large_auxiii:w \int_value:w
27476   \__kernel_intarray_item:Nn \c__fp_trig_intarray
27477     { \__fp_int_eval:w #1 + 4 \scan_stop: }
27478   \exp_after:wN \__fp_trig_large_auxiii:w \int_value:w
27479   \__kernel_intarray_item:Nn \c__fp_trig_intarray
27480     { \__fp_int_eval:w #1 + 5 \scan_stop: }
27481   \exp_after:wN \__fp_trig_large_auxiii:w \int_value:w
27482   \__kernel_intarray_item:Nn \c__fp_trig_intarray
27483     { \__fp_int_eval:w #1 + 6 \scan_stop: }
27484   \exp_after:wN \__fp_trig_large_auxiii:w \int_value:w
27485   \__kernel_intarray_item:Nn \c__fp_trig_intarray
27486     { \__fp_int_eval:w #1 + 7 \scan_stop: }

```

```

27487 \exp_after:wN \_fp_trig_large_auxiii:w \int_value:w
27488 \_kernel_intarray_item:Nn \c\_fp_trig_intarray
27489 { \_fp_int_eval:w #1 + 8 \scan_stop: }
27490 \exp_after:wN \_fp_trig_large_auxiii:w \int_value:w
27491 \_kernel_intarray_item:Nn \c\_fp_trig_intarray
27492 { \_fp_int_eval:w #1 + 9 \scan_stop: }
27493 \exp_stop_f:
27494 }
27495 \cs_new:Npn \_fp_trig_large_auxii:w
27496 {
27497 \_fp_pack_twice_four:wNNNNNNNN \_fp_pack_twice_four:wNNNNNNNN
27498 \_fp_pack_twice_four:wNNNNNNNN \_fp_pack_twice_four:wNNNNNNNN
27499 \_fp_pack_twice_four:wNNNNNNNN \_fp_pack_twice_four:wNNNNNNNN
27500 \_fp_pack_twice_four:wNNNNNNNN \_fp_pack_twice_four:wNNNNNNNN
27501 \_fp_trig_large_auxv:www ;
27502 }
27503 \cs_new:Npn \_fp_trig_large_auxiii:w 1 { \exp_stop_f: }

```

(End definition for _fp_trig_large:ww and others.)

```

\_fp_trig_large_auxv:www
\_fp_trig_large_auxvi:wNNNNNNNN
\_fp_trig_large_pack:NNNNw

```

First come the first 64 digits of the fractional part of $10^{1-16}/(2\pi)$, arranged in 16 blocks of 4, and ending with a semicolon. Then a few more digits of the same fractional part, ending with a semicolon, then 4 blocks of 4 digits holding the significand of the original argument. Multiply the 16-digit significand with the 64-digit fractional part: the `auxvi` auxiliary receives the significand as `#2#3#4#5` and 16 digits of the fractional part as `#6#7#8#9`, and computes one step of the usual ladder of `pack` functions we use for multiplication (see *e.g.*, `_fp_fixed_mul:wN`), then discards one block of the fractional part to set things up for the next step of the ladder. We perform 13 such steps, replacing the last `middle` shift by the appropriate `trailing` shift, then discard the significand and remaining 3 blocks from the fractional part, as there are not enough digits to compute any more step in the ladder. The last semicolon closes the ladder, and we return control to the `auxvii` auxiliary.

```

27504 \cs_new:Npn \_fp_trig_large_auxv:www #1; #2; #3;
27505 {
27506 \exp_after:wN \_fp_use_i_until_s:nw
27507 \exp_after:wN \_fp_trig_large_auxvii:w
27508 \int_value:w \_fp_int_eval:w \c\_fp_leading_shift_int
27509 \prg_replicate:nn { 13 }
27510 { \_fp_trig_large_auxvi:wNNNNNNNN }
27511 + \c\_fp_trailing_shift_int - \c\_fp_middle_shift_int
27512 \_fp_use_i_until_s:nw
27513 ; #3 #1 ; ;
27514 }
27515 \cs_new:Npn \_fp_trig_large_auxvi:wNNNNNNNN #1; #2#3#4#5#6#7#8#9
27516 {
27517 \exp_after:wN \_fp_trig_large_pack:NNNNw
27518 \int_value:w \_fp_int_eval:w \c\_fp_middle_shift_int
27519 + #2*#9 + #3*#8 + #4*#7 + #5*#6
27520 #1; {#2}{#3}{#4}{#5} {#7}{#8}{#9}
27521 }
27522 \cs_new:Npn \_fp_trig_large_pack:NNNNw #1#2#3#4#5#6;
27523 { + #1#2#3#4#5 ; #6 }

```

(End definition for `_fp_trig_large_auxv:www`, `_fp_trig_large_auxvi:wnnnnnnnn`, and `_fp_trig_large_pack:NNNNnw`.)

`_fp_trig_large_auxvii:w` The `auxvii` auxiliary is followed by 52 digits and a semicolon. We find the octant as the integer part of 8 times what follows, or equivalently as the integer part of `#1#2#3/125`, and add it to the surrounding integer expression for the octant. We then compute 8 times the 52-digit number, with a minus sign if the octant is odd. Again, the last `middle` shift is converted to a `trailing` shift. Any integer part (including negative values which come up when the octant is odd) is discarded by `_fp_use_i_until_s:nw`. The resulting fractional part should then be converted to radians by multiplying by $2\pi/8$, but first, build an extended precision number by abusing `_fp_ep_to_ep_loop:N` with the appropriate trailing markers. Finally, `_fp_trig_small:ww` sets up the argument for the functions which compute the Taylor series.

```

27524 \cs_new:Npn \_fp_trig_large_auxvii:w #1#2#3
27525 {
27526   \exp_after:wN \_fp_trig_large_auxviii:ww
27527   \int_value:w \_fp_int_eval:w (#1#2#3 - 62) / 125 ;
27528   #1#2#3
27529 }
27530 \cs_new:Npn \_fp_trig_large_auxviii:ww #1;
27531 {
27532   + #1
27533   \if_int_odd:w #1 \exp_stop_f:
27534     \exp_after:wN \_fp_trig_large_auxix:Nw
27535     \exp_after:wN -
27536   \else:
27537     \exp_after:wN \_fp_trig_large_auxix:Nw
27538     \exp_after:wN +
27539   \fi:
27540 }
27541 \cs_new:Npn \_fp_trig_large_auxix:Nw
27542 {
27543   \exp_after:wN \_fp_use_i_until_s:nw
27544   \exp_after:wN \_fp_trig_large_auxxi:w
27545   \int_value:w \_fp_int_eval:w \c__fp_leading_shift_int
27546   \prg_replicate:nn { 13 }
27547     { \_fp_trig_large_auxx:wnnnnn }
27548   + \c__fp_trailing_shift_int - \c__fp_middle_shift_int
27549   ;
27550 }
27551 \cs_new:Npn \_fp_trig_large_auxx:wnnnnn #1; #2 #3#4#5#6
27552 {
27553   \exp_after:wN \_fp_trig_large_pack:NNNNnw
27554   \int_value:w \_fp_int_eval:w \c__fp_middle_shift_int
27555     #2 8 * #3#4#5#6
27556     #1; #2
27557 }
27558 \cs_new:Npn \_fp_trig_large_auxxi:w #1;
27559 {
27560   \exp_after:wN \_fp_ep_mul_raw:wwwN
27561   \int_value:w \_fp_int_eval:w 0 \_fp_ep_to_ep_loop:N #1 ; ; !
27562   0,{7853}{9816}{3397}{4483}{0961}{5661};
27563   \_fp_trig_small:ww

```

27564 }

(End definition for `_fp_trig_large_auxvii:w` and others.)

75.1.6 Computing the power series

`_fp_sin_series_o:NNwww` Here we receive a conversion function `_fp_ep_to_float_o:wwN` or `_fp_ep_inv_to_float_o:wwN`, a *sign* (0 or 2), a (non-negative) *octant* delimited by a dot, a *fixed point* number delimited by a semicolon, and an extended-precision number. The auxiliary receives:

- the conversion function #1;
- the final sign, which depends on the octant #3 and the sign #2;
- the octant #3, which controls the series we use;
- the square #4 * #4 of the argument as a fixed point number, computed with `_fp_fixed_mul:wwn`;
- the number itself as an extended-precision number.

If the octant is in $\{1, 2, 5, 6, \dots\}$, we are near an extremum of the function and we use the series

$$\cos(x) = 1 - x^2 \left(\frac{1}{2!} - x^2 \left(\frac{1}{4!} - x^2 \left(\dots \right) \right) \right).$$

Otherwise, the series

$$\sin(x) = x \left(1 - x^2 \left(\frac{1}{3!} - x^2 \left(\frac{1}{5!} - x^2 \left(\dots \right) \right) \right) \right)$$

is used. Finally, the extended-precision number is converted to a floating point number with the given sign, and `_fp_sanitize:Nw` checks for overflow and underflow.

```

27565 \cs_new:Npn \_fp_sin_series_o:NNwww #1#2#3. #4;
27566 {
27567   \_fp_fixed_mul:wwn #4; #4;
27568   {
27569     \exp_after:wN \_fp_sin_series_aux_o:NNwww
27570     \exp_after:wN #1
27571     \int_value:w
27572     \if_int_odd:w \_fp_int_eval:w (#3 + 2) / 4 \_fp_int_eval_end:
27573       #2
27574     \else:
27575       \if_meaning:w #2 0 2 \else: 0 \fi:
27576     \fi:
27577     {#3}
27578   }
27579 }
27580 \cs_new:Npn \_fp_sin_series_aux_o:NNwww #1#2#3 #4; #5,#6;
27581 {
27582   \if_int_odd:w \_fp_int_eval:w #3 / 2 \_fp_int_eval_end:
27583     \exp_after:wN \use_i:nn
27584   \else:
27585     \exp_after:wN \use_ii:nn

```

```

27586 \fi:
27587 { % 1/18!
27588   \__fp_fixed_mul_sub_back:wwwn    {0000}{0000}{0000}{0001}{5619}{2070};
27589                                     #4;{0000}{0000}{0000}{0477}{9477}{3324};
27590   \__fp_fixed_mul_sub_back:wwwn #4;{0000}{0000}{0011}{4707}{4559}{7730};
27591   \__fp_fixed_mul_sub_back:wwwn #4;{0000}{0000}{2087}{6756}{9878}{6810};
27592   \__fp_fixed_mul_sub_back:wwwn #4;{0000}{0027}{5573}{1922}{3985}{8907};
27593   \__fp_fixed_mul_sub_back:wwwn #4;{0000}{2480}{1587}{3015}{8730}{1587};
27594   \__fp_fixed_mul_sub_back:wwwn #4;{0013}{8888}{8888}{8888}{8888}{8889};
27595   \__fp_fixed_mul_sub_back:wwwn #4;{0416}{6666}{6666}{6666}{6666}{6667};
27596   \__fp_fixed_mul_sub_back:wwwn #4;{5000}{0000}{0000}{0000}{0000}{0000};
27597   \__fp_fixed_mul_sub_back:wwwn#4;{10000}{0000}{0000}{0000}{0000}{0000};
27598   { \__fp_fixed_continue:wn 0, }
27599 }
27600 { % 1/17!
27601   \__fp_fixed_mul_sub_back:wwwn    {0000}{0000}{0000}{0028}{1145}{7254};
27602                                     #4;{0000}{0000}{0000}{7647}{1637}{3182};
27603   \__fp_fixed_mul_sub_back:wwwn #4;{0000}{0000}{0160}{5904}{3836}{8216};
27604   \__fp_fixed_mul_sub_back:wwwn #4;{0000}{0002}{5052}{1083}{8544}{1719};
27605   \__fp_fixed_mul_sub_back:wwwn #4;{0000}{0275}{5731}{9223}{9858}{9065};
27606   \__fp_fixed_mul_sub_back:wwwn #4;{0001}{9841}{2698}{4126}{9841}{2698};
27607   \__fp_fixed_mul_sub_back:wwwn #4;{0083}{3333}{3333}{3333}{3333}{3333};
27608   \__fp_fixed_mul_sub_back:wwwn #4;{1666}{6666}{6666}{6666}{6666}{6667};
27609   \__fp_fixed_mul_sub_back:wwwn#4;{10000}{0000}{0000}{0000}{0000}{0000};
27610   { \__fp_ep_mul:wwwn 0, } #5,#6;
27611 }
27612 {
27613   \exp_after:wN \__fp_sanitizewN
27614   \exp_after:wN #2
27615   \int_value:w \__fp_int_eval:w #1
27616 }
27617 #2
27618 }

```

(End definition for __fp_sin_series_o:NNwww and __fp_sin_series_aux_o:NNwww.)

__fp_tan_series_o:NNwww Contrarily to __fp_sin_series_o:NNwww which received a conversion auxiliary as #1, here, #1 is 0 for tangent and 2 for cotangent. Consider first the case of the tangent. The octant #3 starts at 1, which means that it is 1 or 2 for $|x| \in [0, \pi/2]$, it is 3 or 4 for $|x| \in [\pi/2, \pi]$, and so on: the intervals on which $\tan|x| \geq 0$ coincide with those for which $\lfloor (\#3 + 1)/2 \rfloor$ is odd. We also have to take into account the original sign of x to get the sign of the final result; it is straightforward to check that the first \int_value:w expansion produces 0 for a positive final result, and 2 otherwise. A similar story holds for $\cot(x)$.

The auxiliary receives the sign, the octant, the square of the (reduced) input, and the (reduced) input (an extended-precision number) as arguments. It then computes the numerator and denominator of

$$\tan(x) \simeq \frac{x(1 - x^2(a_1 - x^2(a_2 - x^2(a_3 - x^2(a_4 - x^2 a_5))))))}{1 - x^2(b_1 - x^2(b_2 - x^2(b_3 - x^2(b_4 - x^2 b_5)))}.$$

The ratio is computed by __fp_ep_div:wwwn, then converted to a floating point number. For octants #3 (really, quadrants) next to a pole of the functions, the fixed point

numerator and denominator are exchanged before computing the ratio. Note that this `\if_int_odd:w` test relies on the fact that the octant is at least 1.

```

27619 \cs_new:Npn \__fp_tan_series_o:NNwww #1#2#3. #4;
27620 {
27621   \__fp_fixed_mul:wwn #4; #4;
27622   {
27623     \exp_after:wN \__fp_tan_series_aux_o:Nnwww
27624     \int_value:w
27625     \if_int_odd:w \__fp_int_eval:w #3 / 2 \__fp_int_eval_end:
27626     \exp_after:wN \reverse_if:N
27627     \fi:
27628     \if_meaning:w #1#2 2 \else: 0 \fi:
27629     {#3}
27630   }
27631 }
27632 \cs_new:Npn \__fp_tan_series_aux_o:Nnwww #1 #2 #3; #4,#5;
27633 {
27634   \__fp_fixed_mul_sub_back:wwwn {0000}{0000}{1527}{3493}{0856}{7059};
27635   #3; {0000}{0159}{6080}{0274}{5257}{6472};
27636   \__fp_fixed_mul_sub_back:wwwn #3; {0002}{4571}{2320}{0157}{2558}{8481};
27637   \__fp_fixed_mul_sub_back:wwwn #3; {0115}{5830}{7533}{5397}{3168}{2147};
27638   \__fp_fixed_mul_sub_back:wwwn #3; {1929}{8245}{6140}{3508}{7719}{2982};
27639   \__fp_fixed_mul_sub_back:wwwn #3; {10000}{0000}{0000}{0000}{0000}{0000};
27640   { \__fp_ep_mul:wwwn 0, } #4,#5;
27641   {
27642     \__fp_fixed_mul_sub_back:wwwn {0000}{0007}{0258}{0681}{9408}{4706};
27643     #3; {0000}{2343}{7175}{1399}{6151}{7670};
27644     \__fp_fixed_mul_sub_back:wwwn #3; {0019}{2638}{4588}{9232}{8861}{3691};
27645     \__fp_fixed_mul_sub_back:wwwn #3; {0536}{6357}{0691}{4344}{6852}{4252};
27646     \__fp_fixed_mul_sub_back:wwwn #3; {5263}{1578}{9473}{6842}{1052}{6315};
27647     \__fp_fixed_mul_sub_back:wwwn #3; {10000}{0000}{0000}{0000}{0000}{0000};
27648     {
27649       \reverse_if:N \if_int_odd:w
27650       \__fp_int_eval:w (#2 - 1) / 2 \__fp_int_eval_end:
27651       \exp_after:wN \__fp_reverse_args:Nww
27652       \fi:
27653       \__fp_ep_div:wwwn 0,
27654     }
27655   }
27656   {
27657     \exp_after:wN \__fp_sanitize:Nw
27658     \exp_after:wN #1
27659     \int_value:w \__fp_int_eval:w \__fp_ep_to_float_o:wwN
27660   }
27661   #1
27662 }

```

(End definition for `__fp_tan_series_o:NNwww` and `__fp_tan_series_aux_o:Nnwww`.)

75.2 Inverse trigonometric functions

All inverse trigonometric functions (arcsine, arccosine, arctangent, arccotangent, arc-cosecant, and arcsecant) are based on a function often denoted `atan2`. This func-

tion is accessed directly by feeding two arguments to arctangent, and is defined by $\text{atan}(y, x) = \text{atan}(y/x)$ for generic y and x . Its advantages over the conventional arctangent is that it takes values in $[-\pi, \pi]$ rather than $[-\pi/2, \pi/2]$, and that it is better behaved in boundary cases. Other inverse trigonometric functions are expressed in terms of atan as

$$\text{acos } x = \text{atan}(\sqrt{1 - x^2}, x) \quad (5)$$

$$\text{asin } x = \text{atan}(x, \sqrt{1 - x^2}) \quad (6)$$

$$\text{asec } x = \text{atan}(\sqrt{x^2 - 1}, 1) \quad (7)$$

$$\text{acsc } x = \text{atan}(1, \sqrt{x^2 - 1}) \quad (8)$$

$$\text{atan } x = \text{atan}(x, 1) \quad (9)$$

$$\text{acot } x = \text{atan}(1, x). \quad (10)$$

Rather than introducing a new function, **atan2**, the arctangent function **atan** is overloaded: it can take one or two arguments. In the comments below, following many texts, we call the first argument y and the second x , because $\text{atan}(y, x) = \text{atan}(y/x)$ is the angular coordinate of the point (x, y) .

As for direct trigonometric functions, the first step in computing $\text{atan}(y, x)$ is argument reduction. The sign of y gives that of the result. We distinguish eight regions where the point $(x, |y|)$ can lie, of angular size roughly $\pi/8$, characterized by their “octant”, between 0 and 7 included. In each region, we compute an arctangent as a Taylor series, then shift this arctangent by the appropriate multiple of $\pi/4$ and sign to get the result. Here is a list of octants, and how we compute the arctangent (we assume $y > 0$: otherwise replace y by $-y$ below):

0 $0 < |y| < 0.41421x$, then $\text{atan } \frac{|y|}{x}$ is given by a nicely convergent Taylor series;

1 $0 < 0.41421x < |y| < x$, then $\text{atan } \frac{|y|}{x} = \frac{\pi}{4} - \text{atan } \frac{x-|y|}{x+|y|}$;

2 $0 < 0.41421|y| < x < |y|$, then $\text{atan } \frac{|y|}{x} = \frac{\pi}{4} + \text{atan } \frac{-x+|y|}{x+|y|}$;

3 $0 < x < 0.41421|y|$, then $\text{atan } \frac{|y|}{x} = \frac{\pi}{2} - \text{atan } \frac{x}{|y|}$;

4 $0 < -x < 0.41421|y|$, then $\text{atan } \frac{|y|}{x} = \frac{\pi}{2} + \text{atan } \frac{-x}{|y|}$;

5 $0 < 0.41421|y| < -x < |y|$, then $\text{atan } \frac{|y|}{x} = \frac{3\pi}{4} - \text{atan } \frac{x+|y|}{-x+|y|}$;

6 $0 < -0.41421x < |y| < -x$, then $\text{atan } \frac{|y|}{x} = \frac{3\pi}{4} + \text{atan } \frac{-x-|y|}{-x+|y|}$;

7 $0 < |y| < -0.41421x$, then $\text{atan } \frac{|y|}{x} = \pi - \text{atan } \frac{|y|}{-x}$.

In the following, we denote by z the ratio among $|\frac{y}{x}|$, $|\frac{x}{y}|$, $|\frac{x+y}{x-y}|$, $|\frac{x-y}{x+y}|$ which appears in the right-hand side above.

75.2.1 Arctangent and arccotangent

```

__fp_atan_o:Nw
__fp_acot_o:Nw
__fp_atan_default:w

```

The parsing step manipulates **atan** and **acot** like **min** and **max**, reading in an array of operands, but also leaves **\use_i:nn** or **\use_ii:nn** depending on whether the result

should be given in radians or in degrees. The helper `__fp_parse_function_one_two:nnw` checks that the operand is one or two floating point numbers (not tuples) and leaves its second argument or its tail accordingly (its first argument is used for error messages). More precisely if we are given a single floating point number `__fp_atan_default:w` places `\c_one_fp` (expanded) after it; otherwise `__fp_atan_default:w` is omitted by `__fp_parse_function_one_two:nnw`.

```

27663 \cs_new:Npn \__fp_atan_o:Nw #1
27664 {
27665   \__fp_parse_function_one_two:nnw
27666   { #1 { atan } { atand } }
27667   { \__fp_atan_default:w \__fp_atanii_o:Nww #1 }
27668 }
27669 \cs_new:Npn \__fp_acot_o:Nw #1
27670 {
27671   \__fp_parse_function_one_two:nnw
27672   { #1 { acot } { acotd } }
27673   { \__fp_atan_default:w \__fp_acotii_o:Nww #1 }
27674 }
27675 \cs_new:Npx \__fp_atan_default:w #1#2#3 @ { #1 #2 #3 \c_one_fp @ }

```

(End definition for `__fp_atan_o:Nw`, `__fp_acot_o:Nw`, and `__fp_atan_default:w`.)

`__fp_atanii_o:Nww`
`__fp_acotii_o:Nww`

If either operand is `nan`, we return it. If both are normal, we call `__fp_atan_normal_o:NNnwNnw`. If both are zero or both infinity, we call `__fp_atan_inf_o:NNNw` with argument 2, leading to a result among $\{\pm\pi/4, \pm3\pi/4\}$ (in degrees, $\{\pm45, \pm135\}$). Otherwise, one is much bigger than the other, and we call `__fp_atan_inf_o:NNNw` with either an argument of 4, leading to the values $\pm\pi/2$ (in degrees, ±90), or 0, leading to $\{\pm0, \pm\pi\}$ (in degrees, $\{\pm0, \pm180\}$). Since $\text{acot}(x, y) = \text{atan}(y, x)$, `__fp_acotii_o:ww` simply reverses its two arguments.

```

27676 \cs_new:Npn \__fp_atanii_o:Nww
27677 #1 \s__fp \__fp_chk:w #2#3#4; \s__fp \__fp_chk:w #5 #6 @
27678 {
27679   \if_meaning:w 3 #2 \__fp_case_return_i_o:ww \fi:
27680   \if_meaning:w 3 #5 \__fp_case_return_ii_o:ww \fi:
27681   \if_case:w
27682     \if_meaning:w #2 #5
27683       \if_meaning:w 1 #2 10 \else: 0 \fi:
27684     \else:
27685       \if_int_compare:w #2 > #5 \exp_stop_f: 1 \else: 2 \fi:
27686     \fi:
27687     \exp_stop_f:
27688     \__fp_case_return:nw { \__fp_atan_inf_o:NNNw #1 #3 2 }
27689   \or: \__fp_case_return:nw { \__fp_atan_inf_o:NNNw #1 #3 4 }
27690   \or: \__fp_case_return:nw { \__fp_atan_inf_o:NNNw #1 #3 0 }
27691   \fi:
27692   \__fp_atan_normal_o:NNnwNnw #1
27693   \s__fp \__fp_chk:w #2#3#4;
27694   \s__fp \__fp_chk:w #5 #6
27695 }
27696 \cs_new:Npn \__fp_acotii_o:Nww #1#2; #3;
27697 { \__fp_atanii_o:Nww #1#3; #2; }

```

(End definition for `__fp_atanii_o:Nww` and `__fp_acotii_o:Nww`.)

`_fp_atan_inf_o:NNNw` This auxiliary is called whenever one number is ± 0 or $\pm \infty$ (and neither is `nan`). Then the result only depends on the signs, and its value is a multiple of $\pi/4$. We use the same auxiliary as for normal numbers, `_fp_atan_combine_o:NwwwwN`, with arguments the final sign `#2`; the octant `#3`; $\text{atan } z/z = 1$ as a fixed point number; $z = 0$ as a fixed point number; and $z = 0$ as an extended-precision number. Given the values we provide, $\text{atan } z$ is computed to be 0, and the result is $[\#3/2] \cdot \pi/4$ if the sign `#5` of x is positive, and $[(7 - \#3)/2] \cdot \pi/4$ for negative x , where the divisions are rounded up.

```

27698 \cs_new:Npn \_fp_atan_inf_o:NNNw #1#2#3 \s_fp \_fp_chk:w #4#5#6;
27699 {
27700   \exp_after:wN \_fp_atan_combine_o:NwwwwN
27701   \exp_after:wN #2
27702   \int_value:w \_fp_int_eval:w
27703   \if_meaning:w 2 #5 7 - \fi: #3 \exp_after:wN ;
27704   \c_fp_one_fixed_t1
27705   {0000}{0000}{0000}{0000}{0000}{0000};
27706   0,{0000}{0000}{0000}{0000}{0000}{0000}; #1
27707 }

```

(End definition for `_fp_atan_inf_o:NNNw`.)

`_fp_atan_normal_o:NNwNnw` Here we simply reorder the floating point data into a pair of signed extended-precision numbers, that is, a sign, an exponent ending with a comma, and a six-block mantissa ending with a semi-colon. This extended precision is required by other inverse trigonometric functions, to compute things like $\text{atan}(x, \sqrt{1 - x^2})$ without intermediate rounding errors.

```

27708 \cs_new_protected:Npn \_fp_atan_normal_o:NNwNnw
27709   #1 \s_fp \_fp_chk:w 1#2#3#4; \s_fp \_fp_chk:w 1#5#6#7;
27710 {
27711   \_fp_atan_test_o:NwNwNwN
27712   #2 #3, #4{0000}{0000};
27713   #5 #6, #7{0000}{0000}; #1
27714 }

```

(End definition for `_fp_atan_normal_o:NNwNnw`.)

`_fp_atan_test_o:NwNwNwN` This receives: the sign `#1` of y , its exponent `#2`, its 24 digits `#3` in groups of 4, and similarly for x . We prepare to call `_fp_atan_combine_o:NwwwwN` which expects the sign `#1`, the octant, the ratio $(\text{atan } z)/z = 1 - \dots$, and the value of z , both as a fixed point number and as an extended-precision floating point number with a mantissa in $[0.01, 1)$. For now, we place `#1` as a first argument, and start an integer expression for the octant. The sign of x does not affect z , so we simply leave a contribution to the octant: $\langle \text{octant} \rangle \rightarrow 7 - \langle \text{octant} \rangle$ for negative x . Then we order $|y|$ and $|x|$ in a non-decreasing order: if $|y| > |x|$, insert 3- in the expression for the octant, and swap the two numbers. The finer test with 0.41421 is done by `_fp_atan_div:wNwNwNw` after the operands have been ordered.

```

27715 \cs_new:Npn \_fp_atan_test_o:NwNwNwN #1#2,#3; #4#5,#6;
27716 {
27717   \exp_after:wN \_fp_atan_combine_o:NwwwwN
27718   \exp_after:wN #1
27719   \int_value:w \_fp_int_eval:w
27720   \if_meaning:w 2 #4
27721     7 - \_fp_int_eval:w
27722   \fi:

```

```

27723     \if_int_compare:w
27724         \__fp_ep_compare:www #2,#3; #5,#6; > \c_zero_int
27725         3 -
27726         \exp_after:wN \__fp_reverse_args:Nww
27727     \fi:
27728     \__fp_atan_div:wnwnw #2,#3; #5,#6;
27729 }

```

(End definition for __fp_atan_test_o:NwwNwwN.)

```

\__fp_atan_div:wnwnw
\__fp_atan_near:wwn
\__fp_atan_near_aux:wn

```

This receives two positive numbers a and b (equal to $|x|$ and $|y|$ in some order), each as an exponent and 6 blocks of 4 digits, such that $0 < a < b$. If $0.41421b < a$, the two numbers are “near”, hence the point (y, x) that we started with is closer to the diagonals $\{|y| = |x|\}$ than to the axes $\{xy = 0\}$. In that case, the octant is 1 (possibly combined with the 7- and 3- inserted earlier) and we wish to compute $\operatorname{atan} \frac{b-a}{a+b}$. Otherwise, the octant is 0 (again, combined with earlier terms) and we wish to compute $\operatorname{atan} \frac{a}{b}$. In any case, call __fp_atan_auxi:ww followed by z , as a comma-delimited exponent and a fixed point number.

```

27730 \cs_new:Npn \__fp_atan_div:wnwnw #1,#2#3; #4,#5#6;
27731 {
27732     \if_int_compare:w
27733         \__fp_int_eval:w 41421 * #5 < #2 000
27734         \if_case:w \__fp_int_eval:w #4 - #1 \__fp_int_eval_end:
27735             00 \or: 0 \fi:
27736         \exp_stop_f:
27737         \exp_after:wN \__fp_atan_near:wwn
27738     \fi:
27739     0
27740     \__fp_ep_div:wwwn #1,{#2}#3; #4,{#5}#6;
27741     \__fp_atan_auxi:ww
27742 }
27743 \cs_new:Npn \__fp_atan_near:wwn
27744     0 \__fp_ep_div:wwwn #1,#2; #3,
27745     {
27746         1
27747         \__fp_ep_to_fixed:wn #1 - #3, #2;
27748         \__fp_atan_near_aux:wn
27749     }
27750 \cs_new:Npn \__fp_atan_near_aux:wn #1; #2;
27751 {
27752     \__fp_fixed_add:wn #1; #2;
27753     { \__fp_fixed_sub:wn #2; #1; { \__fp_ep_div:wwwn 0, } 0, }
27754 }

```

(End definition for __fp_atan_div:wnwnw, __fp_atan_near:wwn, and __fp_atan_near_aux:wn.)

```

\__fp_atan_auxi:ww
\__fp_atan_auxii:w

```

Convert z from a representation as an exponent and a fixed point number in $[0.01, 1)$ to a fixed point number only, then set up the call to __fp_atan_Taylor_loop:www, followed by the fixed point representation of z and the old representation.

```

27755 \cs_new:Npn \__fp_atan_auxi:ww #1,#2;
27756 { \__fp_ep_to_fixed:wn #1,#2; \__fp_atan_auxii:w #1,#2; }
27757 \cs_new:Npn \__fp_atan_auxii:w #1;
27758 {
27759     \__fp_fixed_mul:wn #1; #1;

```

```

27760 {
27761   \__fp_atan_Taylor_loop:www 39 ;
27762   {0000}{0000}{0000}{0000}{0000}{0000} ;
27763 }
27764 ! #1;
27765 }

```

(End definition for __fp_atan_auxi:ww and __fp_atan_auxii:w.)

__fp_atan_Taylor_loop:www We compute the series of $(\operatorname{atan} z)/z$. A typical intermediate stage has $\#1 = 2k - 1$,
 __fp_atan_Taylor_break:w $\#2 = \frac{1}{2k+1} - z^2(\frac{1}{2k+3} - z^2(\dots - z^2\frac{1}{39}))$, and $\#3 = z^2$. To go to the next step $k \rightarrow k - 1$,
 we compute $\frac{1}{2k-1}$, then subtract from it z^2 times $\#2$. The loop stops when $k = 0$: then
 $\#2$ is $(\operatorname{atan} z)/z$, and there is a need to clean up all the unnecessary data, end the integer
 expression computing the octant with a semicolon, and leave the result $\#2$ afterwards.

```

27766 \cs_new:Npn \__fp_atan_Taylor_loop:www #1; #2; #3;
27767 {
27768   \if_int_compare:w #1 = - \c_one_int
27769     \__fp_atan_Taylor_break:w
27770   \fi:
27771   \exp_after:wN \__fp_fixed_div_int:wwN \c__fp_one_fixed_tl #1;
27772   \__fp_rrot:www \__fp_fixed_mul_sub_back:wwwn #2; #3;
27773   {
27774     \exp_after:wN \__fp_atan_Taylor_loop:www
27775     \int_value:w \__fp_int_eval:w #1 - 2 ;
27776   }
27777   #3;
27778 }
27779 \cs_new:Npn \__fp_atan_Taylor_break:w
27780   \fi: #1 \__fp_fixed_mul_sub_back:wwwn #2; #3 !
27781   { \fi: ; #2 ; }

```

(End definition for __fp_atan_Taylor_loop:www and __fp_atan_Taylor_break:w.)

__fp_atan_combine_o:NwwwwN This receives a $\langle sign \rangle$, an $\langle octant \rangle$, a fixed point value of $(\operatorname{atan} z)/z$, a fixed point num-
 __fp_atan_combine_aux:ww ber z , and another representation of z , as an $\langle exponent \rangle$ and the fixed point number
 $10^{-\langle exponent \rangle} z$, followed by either `\use_i:nn` (when working in radians) or `\use_ii:nn`
 (when working in degrees). The function computes the floating point result

$$\langle sign \rangle \left(\left\lceil \frac{\langle octant \rangle}{2} \right\rceil \frac{\pi}{4} + (-1)^{\langle octant \rangle} \frac{\operatorname{atan} z}{z} \cdot z \right), \quad (11)$$

multiplied by $180/\pi$ if working in degrees, and using in any case the most appropriate representation of z . The floating point result is passed to `__fp_sanitize:Nw`, which checks for overflow or underflow. If the octant is 0, leave the exponent $\#5$ for `__fp_sanitize:Nw`, and multiply $\#3 = \frac{\operatorname{atan} z}{z}$ with $\#6$, the adjusted z . Otherwise, multiply $\#3 = \frac{\operatorname{atan} z}{z}$ with $\#4 = z$, then compute the appropriate multiple of $\frac{\pi}{4}$ and add or subtract the product $\#3 \cdot \#4$. In both cases, convert to a floating point with `__fp_fixed_to_float_o:wN`.

```

27782 \cs_new:Npn \__fp_atan_combine_o:NwwwwN #1 #2; #3; #4; #5,#6; #7
27783 {
27784   \exp_after:wN \__fp_sanitize:Nw
27785   \exp_after:wN #1
27786   \int_value:w \__fp_int_eval:w

```

```

27787     \if_meaning:w 0 #2
27788     \exp_after:wN \use_i:nn
27789   \else:
27790     \exp_after:wN \use_ii:nn
27791   \fi:
27792   { #5 \__fp_fixed_mul:wnn #3; #6; }
27793   {
27794     \__fp_fixed_mul:wnn #3; #4;
27795     {
27796       \exp_after:wN \__fp_atan_combine_aux:ww
27797       \int_value:w \__fp_int_eval:w #2 / 2 ; #2;
27798     }
27799   }
27800   { #7 \__fp_fixed_to_float_o:wN \__fp_fixed_to_float_rad_o:wN }
27801   #1
27802 }
27803 \cs_new:Npn \__fp_atan_combine_aux:ww #1; #2;
27804 {
27805   \__fp_fixed_mul_short:wnn
27806   {7853}{9816}{3397}{4483}{0961}{5661};
27807   {#1}{0000}{0000};
27808   {
27809     \if_int_odd:w #2 \exp_stop_f:
27810     \exp_after:wN \__fp_fixed_sub:wnn
27811   \else:
27812     \exp_after:wN \__fp_fixed_add:wnn
27813   \fi:
27814   }
27815 }

```

(End definition for `__fp_atan_combine_o:NwwwwwN` and `__fp_atan_combine_aux:ww`.)

75.2.2 Arcsine and arccosine

`__fp_asin_o:w` Again, the first argument provided by `l3fp-parse` is `\use_i:nn` if we are to work in radians and `\use_ii:nn` for degrees. Then comes a floating point number. The arcsine of ± 0 or `nan` is the same floating point number. The arcsine of $\pm\infty$ raises an invalid operation exception. Otherwise, call an auxiliary common with `__fp_acos_o:w`, feeding it information about what function is being performed (for “invalid operation” exceptions).

```

27816 \cs_new:Npn \__fp_asin_o:w #1 \s__fp \__fp_chk:w #2#3; @
27817 {
27818   \if_case:w #2 \exp_stop_f:
27819   \__fp_case_return_same_o:w
27820 \or:
27821   \__fp_case_use:nw
27822   { \__fp_asin_normal_o:NfwNnnnnw #1 { #1 { asin } { asind } } }
27823 \or:
27824   \__fp_case_use:nw
27825   { \__fp_invalid_operation_o:fw { #1 { asin } { asind } } }
27826 \else:
27827   \__fp_case_return_same_o:w
27828 \fi:
27829 \s__fp \__fp_chk:w #2 #3;
27830 }

```

(End definition for `_fp_asin_o:w`.)

`_fp_acos_o:w` The arccosine of ± 0 is $\pi/2$ (in degrees, 90). The arccosine of $\pm\infty$ raises an invalid operation exception. The arccosine of `nan` is itself. Otherwise, call an auxiliary common with `_fp_sin_o:w`, informing it that it was called by `acos` or `acosd`, and preparing to swap some arguments down the line.

```

27831 \cs_new:Npn \_fp_acos_o:w #1 \s_fp \_fp_chk:w #2#3; @
27832 {
27833   \if_case:w #2 \exp_stop_f:
27834     \_fp_case_use:nw { \_fp_atan_inf_o:NNnw #1 0 4 }
27835   \or:
27836     \_fp_case_use:nw
27837     {
27838       \_fp_asin_normal_o:NfwNnnnnw #1 { #1 { acos } { acosd } }
27839       \_fp_reverse_args:Nww
27840     }
27841   \or:
27842     \_fp_case_use:nw
27843     { \_fp_invalid_operation_o:fw { #1 { acos } { acosd } } }
27844   \else:
27845     \_fp_case_return_same_o:w
27846   \fi:
27847   \s_fp \_fp_chk:w #2 #3;
27848 }

```

(End definition for `_fp_acos_o:w`.)

`_fp_asin_normal_o:NfwNnnnnw` If the exponent #5 is at most 0, the operand lies within $(-1, 1)$ and the operation is permitted: call `_fp_asin_auxi_o:NnNww` with the appropriate arguments. If the number is exactly ± 1 (the test works because we know that $\#5 \geq 1$, $\#6\#7 \geq 10000000$, $\#8\#9 \geq 0$, with equality only for ± 1), we also call `_fp_asin_auxi_o:NnNww`. Otherwise, `_fp_use_i:ww` gets rid of the `asin` auxiliary, and raises instead an invalid operation, because the operand is outside the domain of arcsine or arccosine.

```

27849 \cs_new:Npn \_fp_asin_normal_o:NfwNnnnnw
27850   #1#2#3 \s_fp \_fp_chk:w 1#4#5#6#7#8#9;
27851 {
27852   \if_int_compare:w #5 < \c_one_int
27853     \exp_after:wN \_fp_use_none_until_s:w
27854   \fi:
27855   \if_int_compare:w \_fp_int_eval:w #5 + #6#7 + #8#9 = 1000 0001 ~
27856     \exp_after:wN \_fp_use_none_until_s:w
27857   \fi:
27858   \_fp_use_i:ww
27859   \_fp_invalid_operation_o:fw {#2}
27860   \s_fp \_fp_chk:w 1#4#{#5}{#6}{#7}{#8}{#9};
27861   \_fp_asin_auxi_o:NnNww
27862   #1 {#3} #4 #5,{#6}{#7}{#8}{#9}{0000}{0000};
27863 }

```

(End definition for `_fp_asin_normal_o:NfwNnnnnw`.)

`_fp_asin_auxi_o:NnNww` We compute $x/\sqrt{1-x^2}$. This function is used by `asin` and `acos`, but also by `acsc` and `asec` after inverting the operand, thus it must manipulate extended-precision numbers. First evaluate $1-x^2$ as $(1+x)(1-x)$: this behaves better near $x = 1$. We do the

addition/subtraction with fixed point numbers (they are not implemented for extended-precision floats), but go back to extended-precision floats to multiply and compute the inverse square root $1/\sqrt{1-x^2}$. Finally, multiply by the (positive) extended-precision float $|x|$, and feed the (signed) result, and the number +1, as arguments to the arctangent function. When computing the arccosine, the arguments $x/\sqrt{1-x^2}$ and +1 are swapped by #2 (`__fp_reverse_args:Nww` in that case) before `__fp_atan_test_o:NwwNwwN` is evaluated. Note that the arctangent function requires normalized arguments, hence the need for `ep_to_ep` and continue after `ep_mul`.

```

27864 \cs_new:Npn \__fp_asin_auxi_o:NnNww #1#2#3#4,#5;
27865 {
27866   \__fp_ep_to_fixed:wwn #4,#5;
27867   \__fp_asin_isqrt:wn
27868   \__fp_ep_mul:wwwwn #4,#5;
27869   \__fp_ep_to_ep:wwN
27870   \__fp_fixed_continue:wn
27871   { #2 \__fp_atan_test_o:NwwNwwN #3 }
27872   0 1,{1000}{0000}{0000}{0000}{0000}{0000}; #1
27873 }
27874 \cs_new:Npn \__fp_asin_isqrt:wn #1;
27875 {
27876   \exp_after:wN \__fp_fixed_sub:wwn \c__fp_one_fixed_tl #1;
27877   {
27878     \__fp_fixed_add_one:wn #1;
27879     \__fp_fixed_continue:wn { \__fp_ep_mul:wwwwn 0, } 0,
27880   }
27881   \__fp_ep_isqrt:wwn
27882 }

```

(End definition for `__fp_asin_auxi_o:NnNww` and `__fp_asin_isqrt:wn`.)

75.2.3 Arccosecant and arcsecant

`__fp_acsc_o:w` Cases are mostly labelled by #2, except when #2 is 2: then we use #3#2, which is 02 = 2 when the number is $+\infty$ and 22 when the number is $-\infty$. The arccosecant of ± 0 raises an invalid operation exception. The arccosecant of $\pm\infty$ is ± 0 with the same sign. The arcosecant of `nan` is itself. Otherwise, `__fp_acsc_normal_o:NfwNnw` does some more tests, keeping the function name (`acsc` or `acscd`) as an argument for invalid operation exceptions.

```

27883 \cs_new:Npn \__fp_acsc_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
27884 {
27885   \if_case:w \if_meaning:w 2 #2 #3 \fi: #2 \exp_stop_f:
27886     \__fp_case_use:nw
27887     { \__fp_invalid_operation_o:fw { #1 { acsc } { acscd } } }
27888   \or: \__fp_case_use:nw
27889     { \__fp_acsc_normal_o:NfwNnw #1 { #1 { acsc } { acscd } } }
27890   \or: \__fp_case_return_o:Nw \c_zero_fp
27891   \or: \__fp_case_return_same_o:w
27892   \else: \__fp_case_return_o:Nw \c_minus_zero_fp
27893   \fi:
27894   \s__fp \__fp_chk:w #2 #3 #4;
27895 }

```

(End definition for `__fp_acsc_o:w`.)

`__fp_asec_o:w` The arcsecant of ± 0 raises an invalid operation exception. The arcsecant of $\pm\infty$ is $\pi/2$ (in degrees, 90). The arcosecant of `nan` is itself. Otherwise, do some more tests, keeping the function name `asec` (or `asecd`) as an argument for invalid operation exceptions, and a `__fp_reverse_args:Nww` following precisely that appearing in `__fp_acos_o:w`.

```

27896 \cs_new:Npn \__fp_asec_o:w #1 \s__fp \__fp_chk:w #2#3; @
27897 {
27898   \if_case:w #2 \exp_stop_f:
27899     \__fp_case_use:nw
27900     { \__fp_invalid_operation_o:fw { #1 { asec } { asecd } } }
27901   \or:
27902     \__fp_case_use:nw
27903     {
27904       \__fp_acsc_normal_o:NfwNnw #1 { #1 { asec } { asecd } }
27905       \__fp_reverse_args:Nww
27906     }
27907   \or: \__fp_case_use:nw { \__fp_atan_inf_o:NNnw #1 0 4 }
27908   \else: \__fp_case_return_same_o:w
27909   \fi:
27910   \s__fp \__fp_chk:w #2 #3;
27911 }

```

(End definition for `__fp_asec_o:w`.)

`__fp_acsc_normal_o:NfwNnw` If the exponent is non-positive, the operand is less than 1 in absolute value, which is always an invalid operation: complain. Otherwise, compute the inverse of the operand, and feed it to `__fp_asin_auxi_o:NnNww` (with all the appropriate arguments). This computes what we want thanks to $\text{acsc}(x) = \text{asin}(1/x)$ and $\text{asec}(x) = \text{acos}(1/x)$.

```

27912 \cs_new:Npn \__fp_acsc_normal_o:NfwNnw #1#2#3 \s__fp \__fp_chk:w 1#4#5#6;
27913 {
27914   \int_compare:nNnTF {#5} < 1
27915   {
27916     \__fp_invalid_operation_o:fw {#2}
27917     \s__fp \__fp_chk:w 1#4{#5}#6;
27918   }
27919   {
27920     \__fp_ep_div:wwwn
27921     1,{1000}{0000}{0000}{0000}{0000}{0000};
27922     #5,#6{0000}{0000};
27923     { \__fp_asin_auxi_o:NnNww #1 {#3} #4 }
27924   }
27925 }

```

(End definition for `__fp_acsc_normal_o:NfwNnw`.)

```

27926 \</package>

```

Chapter 76

l3fp-convert implementation

```
27927 <*package>
27928 <@@=fp>
```

76.1 Dealing with tuples

The first argument is for instance `__fp_to_tl_dispatch:w`, which converts any floating point object to the appropriate representation. We loop through all items, putting ,~ between all of them and making sure to remove the leading ,~.

```
27929 \cs_new:Npn \__fp_tuple_convert:Nw #1 \s__fp_tuple \__fp_tuple_chk:w #2 ;
27930 {
27931   \int_case:nnF { \__fp_array_count:n {#2} }
27932   {
27933     { 0 } { ( ) }
27934     { 1 } { \__fp_tuple_convert_end:w @ { #1 #2 , } }
27935   }
27936   {
27937     \__fp_tuple_convert_loop:nNw { } #1
27938     #2 { ? \__fp_tuple_convert_end:w } ;
27939     @ { \use_none:nn }
27940   }
27941 }
27942 \cs_new:Npn \__fp_tuple_convert_loop:nNw #1#2#3#4; #5 @ #6
27943 {
27944   \use_none:n #3
27945   \exp_args:Nf \__fp_tuple_convert_loop:nNw { #2 #3#4 ; } #2 #5
27946   @ { #6 , ~ #1 }
27947 }
27948 \cs_new:Npn \__fp_tuple_convert_end:w #1 @ #2
27949 { \exp_after:wN ( \exp:w \exp_end_continue_f:w #2 ) }
```

(End definition for `__fp_tuple_convert:Nw`, `__fp_tuple_convert_loop:nNw`, and `__fp_tuple_convert_end:w`.)

76.2 Trimming trailing zeros

```
\__fp_trim_zeros:w
\__fp_trim_zeros_loop:w
\__fp_trim_zeros_dot:w
\__fp_trim_zeros_end:w
```

If #1 ends with a 0, the loop auxiliary takes that zero as an end-delimiter for its first argument, and the second argument is the same loop auxiliary. Once the last trailing

zero is reached, the second argument is the `dot` auxiliary, which removes a trailing dot if any. We then clean-up with the `end` auxiliary, keeping only the number.

```

27950 \cs_new:Npn \__fp_trim_zeros:w #1 ;
27951 {
27952     \__fp_trim_zeros_loop:w #1
27953     ; \__fp_trim_zeros_loop:w 0; \__fp_trim_zeros_dot:w .; \s__fp_stop
27954 }
27955 \cs_new:Npn \__fp_trim_zeros_loop:w #1 0; #2 { #2 #1 ; #2 }
27956 \cs_new:Npn \__fp_trim_zeros_dot:w #1 .; { \__fp_trim_zeros_end:w #1 ; }
27957 \cs_new:Npn \__fp_trim_zeros_end:w #1 ; #2 \s__fp_stop { #1 }

```

(End definition for `__fp_trim_zeros:w` and others.)

76.3 Scientific notation

`\fp_to_scientific:N` The three public functions evaluate their argument, then pass it to `__fp_to_scientific_dispatch:w`.

```

\fp_to_scientific:c
\fp_to_scientific:n
27958 \cs_new:Npn \fp_to_scientific:N #1
27959 { \exp_after:wN \__fp_to_scientific_dispatch:w #1 }
27960 \cs_generate_variant:Nn \fp_to_scientific:N { c }
27961 \cs_new:Npn \fp_to_scientific:n
27962 {
27963     \exp_after:wN \__fp_to_scientific_dispatch:w
27964     \exp:w \exp_end_continue_f:w \__fp_parse:n
27965 }

```

(End definition for `\fp_to_scientific:N` and `\fp_to_scientific:n`. These functions are documented on page 242.)

`__fp_to_scientific_dispatch:w` We allow tuples.

```

\__fp_to_scientific_recover:w
\__fp_tuple_to_scientific:w
27966 \cs_new:Npn \__fp_to_scientific_dispatch:w #1
27967 {
27968     \__fp_change_func_type:NNN
27969     #1 \__fp_to_scientific:w \__fp_to_scientific_recover:w
27970     #1
27971 }
27972 \cs_new:Npn \__fp_to_scientific_recover:w #1 #2 ;
27973 {
27974     \__fp_error:nffn { unknown-type } { \tl_to_str:n { #2 ; } } { } { }
27975     nan
27976 }
27977 \cs_new:Npn \__fp_tuple_to_scientific:w
27978 { \__fp_tuple_convert:Nw \__fp_to_scientific_dispatch:w }

```

(End definition for `__fp_to_scientific_dispatch:w`, `__fp_to_scientific_recover:w`, and `__fp_tuple_to_scientific:w`.)

`__fp_to_scientific:w` Expressing an internal floating point number in scientific notation is quite easy: no rounding, and the format is very well defined. First cater for the sign: negative numbers (`#2 = 2`) start with `-`; we then only need to care about positive numbers and `nan`. Then filter the special cases: ± 0 are represented as `0`; infinities are converted to a number slightly larger than the largest after an “invalid_operation” exception; `nan` is represented as `0` after an “invalid_operation” exception. In the normal case, decrement the exponent

and unbrace the 4 brace groups, then in a second step grab the first digit (previously hidden in braces) to order the various parts correctly.

```

27979 \cs_new:Npn \__fp_to_scientific:w \s__fp \__fp_chk:w #1#2
27980 {
27981   \if_meaning:w 2 #2 \exp_after:wN - \exp:w \exp_end_continue_f:w \fi:
27982   \if_case:w #1 \exp_stop_f:
27983     \__fp_case_return:nw { 0.000000000000000e0 }
27984   \or: \exp_after:wN \__fp_to_scientific_normal:wnnnnn
27985   \or:
27986     \__fp_case_use:nw
27987     {
27988       \__fp_invalid_operation:nnw
27989       { \fp_to_scientific:N \c__fp_overflowing_fp }
27990       { fp_to_scientific }
27991     }
27992   \or:
27993     \__fp_case_use:nw
27994     {
27995       \__fp_invalid_operation:nnw
27996       { \fp_to_scientific:N \c_zero_fp }
27997       { fp_to_scientific }
27998     }
27999   \fi:
28000   \s__fp \__fp_chk:w #1 #2
28001 }
28002 \cs_new:Npn \__fp_to_scientific_normal:wnnnnn
28003 \s__fp \__fp_chk:w 1 #1 #2 #3#4#5#6 ;
28004 {
28005   \exp_after:wN \__fp_to_scientific_normal:wNw
28006   \exp_after:wN e
28007   \int_value:w \__fp_int_eval:w #2 - 1
28008   ; #3 #4 #5 #6 ;
28009 }
28010 \cs_new:Npn \__fp_to_scientific_normal:wNw #1 ; #2#3;
28011 { #2.#3 #1 }

```

(End definition for `__fp_to_scientific:w`, `__fp_to_scientific_normal:wnnnnn`, and `__fp_to_scientific_normal:wNw`.)

76.4 Decimal representation

`\fp_to_decimal:N` All three public variants are based on the same `__fp_to_decimal_dispatch:w` after evaluating their argument to an internal floating point.

```

\fp_to_decimal:c
\fp_to_decimal:n
28012 \cs_new:Npn \fp_to_decimal:N #1
28013 { \exp_after:wN \__fp_to_decimal_dispatch:w #1 }
28014 \cs_generate_variant:Nn \fp_to_decimal:N { c }
28015 \cs_new:Npn \fp_to_decimal:n
28016 {
28017   \exp_after:wN \__fp_to_decimal_dispatch:w
28018   \exp:w \exp_end_continue_f:w \__fp_parse:n
28019 }

```

(End definition for `\fp_to_decimal:N` and `\fp_to_decimal:n`. These functions are documented on page 241.)

```

\__fp_to_decimal_dispatch:w
\__fp_to_decimal_recover:w
\__fp_tuple_to_decimal:w

```

We allow tuples.

```

28020 \cs_new:Npn \__fp_to_decimal_dispatch:w #1
28021 {
28022   \__fp_change_func_type:NNN
28023   #1 \__fp_to_decimal:w \__fp_to_decimal_recover:w
28024   #1
28025 }
28026 \cs_new:Npn \__fp_to_decimal_recover:w #1 #2 ;
28027 {
28028   \__fp_error:nffn { unknown-type } { \tl_to_str:n { #2 ; } } { } { }
28029   nan
28030 }
28031 \cs_new:Npn \__fp_tuple_to_decimal:w
28032 { \__fp_tuple_convert:Nw \__fp_to_decimal_dispatch:w }

```

(End definition for __fp_to_decimal_dispatch:w, __fp_to_decimal_recover:w, and __fp_tuple_to_decimal:w.)

```

\__fp_to_decimal:w
\__fp_to_decimal_normal:wnnnnn
\__fp_to_decimal_large:Nnnw
\__fp_to_decimal_huge:wnnnn

```

The structure is similar to __fp_to_scientific:w. Insert - for negative numbers. Zero gives 0, $\pm\infty$ and nan yield an “invalid operation” exception; note that $\pm\infty$ produces a very large output, which we don’t expand now since it most likely won’t be needed. Normal numbers with an exponent in the range [1,15] have that number of digits before the decimal separator: “decimate” them, and remove leading zeros with \int_value:w, then trim trailing zeros and dot. Normal numbers with an exponent 16 or larger have no decimal separator, we only need to add trailing zeros. When the exponent is non-positive, the result should be 0.<zeros><digits>, trimmed.

```

28033 \cs_new:Npn \__fp_to_decimal:w \s__fp \__fp_chk:w #1#2
28034 {
28035   \if_meaning:w 2 #2 \exp_after:wN - \exp:w \exp_end_continue_f:w \fi:
28036   \if_case:w #1 \exp_stop_f:
28037     \__fp_case_return:nw { 0 }
28038   \or: \exp_after:wN \__fp_to_decimal_normal:wnnnnn
28039   \or:
28040     \__fp_case_use:nw
28041     {
28042       \__fp_invalid_operation:nnw
28043       { \fp_to_decimal:N \c__fp_overflowing_fp }
28044       { fp_to_decimal }
28045     }
28046   \or:
28047     \__fp_case_use:nw
28048     {
28049       \__fp_invalid_operation:nnw
28050       { 0 }
28051       { fp_to_decimal }
28052     }
28053   \fi:
28054   \s__fp \__fp_chk:w #1 #2
28055 }
28056 \cs_new:Npn \__fp_to_decimal_normal:wnnnnn
28057 \s__fp \__fp_chk:w 1 #1 #2 #3#4#5#6 ;
28058 {
28059   \int_compare:nNnTF {#2} > 0
28060   {

```

```

28061 \int_compare:nNnTF {#2} < \c__fp_prec_int
28062 {
28063     \__fp_decimate:nNnnnn { \c__fp_prec_int - #2 }
28064     \__fp_to_decimal_large:Nnnw
28065 }
28066 {
28067     \exp_after:wN \exp_after:wN
28068     \exp_after:wN \__fp_to_decimal_huge:wnnnn
28069     \prg_replicate:nn { #2 - \c__fp_prec_int } { 0 } ;
28070 }
28071 {#3} {#4} {#5} {#6}
28072 }
28073 {
28074     \exp_after:wN \__fp_trim_zeros:w
28075     \exp_after:wN 0
28076     \exp_after:wN .
28077     \exp:w \exp_end_continue_f:w \prg_replicate:nn { - #2 } { 0 }
28078     #3#4#5#6 ;
28079 }
28080 }
28081 \cs_new:Npn \__fp_to_decimal_large:Nnnw #1#2#3#4;
28082 {
28083     \exp_after:wN \__fp_trim_zeros:w \int_value:w
28084     \if_int_compare:w #2 > \c_zero_int
28085     #2
28086     \fi:
28087     \exp_stop_f:
28088     #3.#4 ;
28089 }
28090 \cs_new:Npn \__fp_to_decimal_huge:wnnnn #1; #2#3#4#5 { #2#3#4#5 #1 }

```

(End definition for __fp_to_decimal:w and others.)

76.5 Token list representation

\fp_to_tl:N These three public functions evaluate their argument, then pass it to __fp_to_tl_dispatch:w.
\fp_to_tl:c
\fp_to_tl:n

```

28091 \cs_new:Npn \fp_to_tl:N #1 { \exp_after:wN \__fp_to_tl_dispatch:w #1 }
28092 \cs_generate_variant:Nn \fp_to_tl:N { c }
28093 \cs_new:Npn \fp_to_tl:n
28094 {
28095     \exp_after:wN \__fp_to_tl_dispatch:w
28096     \exp:w \exp_end_continue_f:w \__fp_parse:n
28097 }

```

(End definition for \fp_to_tl:N and \fp_to_tl:n. These functions are documented on page 242.)

__fp_to_tl_dispatch:w We allow tuples.
__fp_to_tl_recover:w
__fp_tuple_to_tl:w

```

28098 \cs_new:Npn \__fp_to_tl_dispatch:w #1
28099 { \__fp_change_func_type:NNN #1 \__fp_to_tl:w \__fp_to_tl_recover:w #1 }
28100 \cs_new:Npn \__fp_to_tl_recover:w #1 #2 ;
28101 {
28102     \__fp_error:nffn { unknown-type } { \tl_to_str:n { #2 ; } } { } { } { }

```

```

28103     nan
28104 }
28105 \cs_new:Npn \__fp_tuple_to_tl:w
28106 { \__fp_tuple_convert:Nw \__fp_to_tl_dispatch:w }

```

(End definition for __fp_to_tl_dispatch:w, __fp_to_tl_recover:w, and __fp_tuple_to_tl:w.)

```

\__fp_to_tl:w
\__fp_to_tl_normal:nnnnn
\__fp_to_tl_scientific:wnnnnn
\__fp_to_tl_scientific:wNw

```

A structure similar to __fp_to_scientific_dispatch:w and __fp_to_decimal_dispatch:w, but without the “invalid operation” exception. First filter special cases. We express normal numbers in decimal notation if the exponent is in the range $[-2, 16]$, and otherwise use scientific notation.

```

28107 \cs_new:Npn \__fp_to_tl:w \s__fp \__fp_chk:w #1#2
28108 {
28109     \if_meaning:w 2 #2 \exp_after:wN - \exp:w \exp_end_continue_f:w \fi:
28110     \if_case:w #1 \exp_stop_f:
28111         \__fp_case_return:nw { 0 }
28112     \or: \exp_after:wN \__fp_to_tl_normal:nnnnn
28113     \or: \__fp_case_return:nw { inf }
28114     \else: \__fp_case_return:nw { nan }
28115     \fi:
28116 }
28117 \cs_new:Npn \__fp_to_tl_normal:nnnnn #1
28118 {
28119     \int_compare:nTF
28120     { -2 <= #1 <= \c__fp_prec_int }
28121     { \__fp_to_decimal_normal:wnnnnn }
28122     { \__fp_to_tl_scientific:wnnnnn }
28123     \s__fp \__fp_chk:w 1 0 {#1}
28124 }
28125 \cs_new:Npn \__fp_to_tl_scientific:wnnnnn
28126 \s__fp \__fp_chk:w 1 #1 #2 #3#4#5#6 ;
28127 {
28128     \exp_after:wN \__fp_to_tl_scientific:wNw
28129     \exp_after:wN e
28130     \int_value:w \__fp_int_eval:w #2 - 1
28131     ; #3 #4 #5 #6 ;
28132 }
28133 \cs_new:Npn \__fp_to_tl_scientific:wNw #1 ; #2#3;
28134 { \__fp_trim_zeros:w #2.#3 ; #1 }

```

(End definition for __fp_to_tl:w and others.)

76.6 Formatting

This is not implemented yet, as it is not yet clear what a correct interface would be, for this kind of structured conversion from a floating point (or other types of variables) to a string. Ideas welcome.

76.7 Convert to dimension or integer

```

\fp_to_dim:N
\fp_to_dim:c
\fp_to_dim:n
\__fp_to_dim_dispatch:w
\__fp_to_dim_recover:w
\__fp_to_dim:w

```

All three public variants are based on the same __fp_to_dim_dispatch:w after evaluating their argument to an internal floating point. We only allow floating point numbers,

not tuples.

```

28135 \cs_new:Npn \fp_to_dim:N #1
28136 { \exp_after:wN \__fp_to_dim_dispatch:w #1 }
28137 \cs_generate_variant:Nn \fp_to_dim:N { c }
28138 \cs_new:Npn \fp_to_dim:n
28139 {
28140   \exp_after:wN \__fp_to_dim_dispatch:w
28141   \exp:w \exp_end_continue_f:w \__fp_parse:n
28142 }
28143 \cs_new:Npn \__fp_to_dim_dispatch:w #1#2 ;
28144 {
28145   \__fp_change_func_type:NNN #1 \__fp_to_dim:w \__fp_to_dim_recover:w
28146   #1 #2 ;
28147 }
28148 \cs_new:Npn \__fp_to_dim_recover:w #1
28149 { \__fp_invalid_operation:nnw { Opt } { fp_to_dim } }
28150 \cs_new:Npn \__fp_to_dim:w #1 ; { \__fp_to_decimal:w #1 ; pt }

```

(End definition for `\fp_to_dim:N` and others. These functions are documented on page 241.)

```

\fp_to_int:N For the most part identical to \fp_to_dim:N but without pt, and where \__fp_to_int:w
\fp_to_int:c does more work. To convert to an integer, first round to 0 places (to the nearest integer),
\fp_to_int:n then express the result as a decimal number: the definition of \__fp_to_decimal_-
\__fp_to_int_dispatch:w dispatch:w is such that there are no trailing dot nor zero.
\__fp_to_int_recover:w
28151 \cs_new:Npn \fp_to_int:N #1 { \exp_after:wN \__fp_to_int_dispatch:w #1 }
28152 \cs_generate_variant:Nn \fp_to_int:N { c }
28153 \cs_new:Npn \fp_to_int:n
28154 {
28155   \exp_after:wN \__fp_to_int_dispatch:w
28156   \exp:w \exp_end_continue_f:w \__fp_parse:n
28157 }
28158 \cs_new:Npn \__fp_to_int_dispatch:w #1#2 ;
28159 {
28160   \__fp_change_func_type:NNN #1 \__fp_to_int:w \__fp_to_int_recover:w
28161   #1 #2 ;
28162 }
28163 \cs_new:Npn \__fp_to_int_recover:w #1
28164 { \__fp_invalid_operation:nnw { 0 } { fp_to_int } }
28165 \cs_new:Npn \__fp_to_int:w #1;
28166 {
28167   \exp_after:wN \__fp_to_decimal:w \exp:w \exp_end_continue_f:w
28168   \__fp_round:Nwn \__fp_round_to_nearest:NNN #1; { 0 }
28169 }

```

(End definition for `\fp_to_int:N` and others. These functions are documented on page 241.)

76.8 Convert from a dimension

```

\dim_to_fp:n The dimension expression (which can in fact be a glue expression) is evaluated, con-
\__fp_from_dim_test:ww verted to a number (i.e., expressed in scaled points), then multiplied by  $2^{-16} =$ 
\__fp_from_dim:wNw 0.0000152587890625 to give a value expressed in points. The auxiliary \__fp_mul_-
\__fp_from_dim:wNNnnnnnn npos_o:Nww expects the desired final sign and two floating point operands (of the form
\__fp_from_dim:wnnnnwNw

```


`\s__fp ... ;`) as arguments. This set of functions is also used to convert dimension registers to floating points while parsing expressions: in this context there is an additional exponent, which is the first argument of `__fp_from_dim_test:ww`, and is combined with the exponent -4 of 2^{-16} . There is also a need to expand afterwards: this is performed by `__fp_mul_npos_o:Nww`, and cancelled by `\prg_do_nothing`: here.

```

28170 \cs_new:Npn \dim_to_fp:n #1
28171 {
28172   \exp_after:wN \__fp_from_dim_test:ww
28173   \exp_after:wN 0
28174   \exp_after:wN ,
28175   \int_value:w \tex_glueexpr:D #1 ;
28176 }
28177 \cs_new:Npn \__fp_from_dim_test:ww #1, #2
28178 {
28179   \if_meaning:w 0 #2
28180     \__fp_case_return:nw { \exp_after:wN \c_zero_fp }
28181   \else:
28182     \exp_after:wN \__fp_from_dim:wNw
28183     \int_value:w \__fp_int_eval:w #1 - 4
28184     \if_meaning:w - #2
28185       \exp_after:wN , \exp_after:wN 2 \int_value:w
28186     \else:
28187       \exp_after:wN , \exp_after:wN 0 \int_value:w #2
28188     \fi:
28189   \fi:
28190 }
28191 \cs_new:Npn \__fp_from_dim:wNw #1,#2#3;
28192 {
28193   \__fp_pack_twice_four:wNNNNNNNN \__fp_from_dim:wNNnnnnnn ;
28194   #3 000 0000 00 {10}987654321; #2 {#1}
28195 }
28196 \cs_new:Npn \__fp_from_dim:wNNnnnnnn #1; #2#3#4#5#6#7#8#9
28197 { \__fp_from_dim:wnnnnwNn #1 {#2#300} {0000} ; }
28198 \cs_new:Npn \__fp_from_dim:wnnnnwNn #1; #2#3#4#5#6; #7#8
28199 {
28200   \__fp_mul_npos_o:Nww #7
28201   \s__fp \__fp_chk:w 1 #7 {#5} #1 ;
28202   \s__fp \__fp_chk:w 1 0 {#8} {1525} {8789} {0625} {0000} ;
28203   \prg_do_nothing:
28204 }

```

(End definition for `\dim_to_fp:n` and others. This function is documented on page 213.)

76.9 Use and eval

`\fp_use:N` Those public functions are simple copies of the decimal conversions.
`\fp_use:c` 28205 `\cs_new_eq:NN \fp_use:N \fp_to_decimal:N`
`\fp_eval:n` 28206 `\cs_generate_variant:Nn \fp_use:N { c }`
28207 `\cs_new_eq:NN \fp_eval:n \fp_to_decimal:n`

(End definition for `\fp_use:N` and `\fp_eval:n`. These functions are documented on page 242.)

\fp_sign:n Trivial but useful. See the implementation of `\fp_add:Nn` for an explanation of why to use `__fp_parse:n`, namely, for better error reporting.

```
28208 \cs_new:Npn \fp_sign:n #1
28209 { \fp_to_decimal:n { sign \__fp_parse:n {#1} } }
```

(End definition for `\fp_sign:n`. This function is documented on page 241.)

\fp_abs:n Trivial but useful. See the implementation of `\fp_add:Nn` for an explanation of why to use `__fp_parse:n`, namely, for better error reporting.

```
28210 \cs_new:Npn \fp_abs:n #1
28211 { \fp_to_decimal:n { abs \__fp_parse:n {#1} } }
```

(End definition for `\fp_abs:n`. This function is documented on page 257.)

\fp_max:nn Similar to `\fp_abs:n`, for consistency with `\int_max:nn`, etc.

```
\fp_min:nn 28212 \cs_new:Npn \fp_max:nn #1#2
28213 { \fp_to_decimal:n { max ( \__fp_parse:n {#1} , \__fp_parse:n {#2} ) } }
28214 \cs_new:Npn \fp_min:nn #1#2
28215 { \fp_to_decimal:n { min ( \__fp_parse:n {#1} , \__fp_parse:n {#2} ) } }
```

(End definition for `\fp_max:nn` and `\fp_min:nn`. These functions are documented on page 257.)

76.10 Convert an array of floating points to a comma list

`__fp_array_to_clist:n` Converts an array of floating point numbers to a comma-list. If speed here ends up irrelevant, we can simplify the code for the auxiliary to become

```
\cs_new:Npn \__fp_array_to_clist_loop:Nw #1#2;
{
  \use_none:n #1
  { , ~ } \fp_to_tl:n { #1 #2 ; }
  \__fp_array_to_clist_loop:Nw
}
```

The `\use_ii:nn` function is expanded after `__fp_expand:n` is done, and it removes `,~` from the start of the representation.

```
28216 \cs_new:Npn \__fp_array_to_clist:n #1
28217 {
28218   \tl_if_empty:nF {#1}
28219   {
28220     \exp_last_unbraced:Ne \use_ii:nn
28221     {
28222       \__fp_array_to_clist_loop:Nw #1 { ? \prg_break: } ;
28223       \prg_break_point:
28224     }
28225   }
28226 }
28227 \cs_new:Npn \__fp_array_to_clist_loop:Nw #1#2;
28228 {
28229   \use_none:n #1
28230   , ~
```

```

28231 \exp_not:f { \_fp_to_tl_dispatch:w #1 #2 ; }
28232 \_fp_array_to_clist_loop:Nw
28233 }

(End definition for \_fp_array_to_clist:n and \_fp_array_to_clist_loop:Nw.)

28234 \end{package}

```

Chapter 77

l3fp-random Implementation

```
28235 <*package>
28236 <@@=fp>

  \__fp_parse_word_rand:N
  \__fp_parse_word_randint:N
Those functions may receive a variable number of arguments. We won't use the argument ?.

28237 \cs_new:Npn \__fp_parse_word_rand:N
28238   { \__fp_parse_function:NNN \__fp_rand_o:Nw ? }
28239 \cs_new:Npn \__fp_parse_word_randint:N
28240   { \__fp_parse_function:NNN \__fp_randint_o:Nw ? }

(End definition for \__fp_parse_word_rand:N and \__fp_parse_word_randint:N.)
```

77.1 Engine support

Most engines provide random numbers, but not all. We write the test twice simply in order to write the `false` branch first.

```
28241 \sys_if_rand_exist:F
28242 {
28243   \msg_new:nnn { kernel } { fp-no-random }
28244   { Random~numbers~unavailable~for~#1 }
28245   \cs_new:Npn \__fp_rand_o:Nw ? #1 @
28246   {
28247     \msg_expandable_error:nnn { kernel } { fp-no-random }
28248     { fp~rand }
28249     \exp_after:wN \c_nan_fp
28250   }
28251   \cs_new_eq:NN \__fp_randint_o:Nw \__fp_rand_o:Nw
28252   \cs_new:Npn \int_rand:nn #1#2
28253   {
28254     \msg_expandable_error:nnn { kernel } { fp-no-random }
28255     { \int_rand:nn {#1} {#2} }
28256     \int_eval:n {#1}
28257   }
28258   \cs_new:Npn \int_rand:n #1
28259   {
28260     \msg_expandable_error:nnn { kernel } { fp-no-random }
28261     { \int_rand:n {#1} }
```

```

28262         1
28263     }
28264 }
28265 \sys_if_rand_exist:T
28266 {

```

Obviously, every word “random” below means “pseudo-random”, as we have no access to entropy (except a very unreliable source of entropy: the time it takes to run some code).

The primitive random number generator (RNG) is provided as `\tex_uniformdeviate:D`. Under the hood, it maintains an array of 55 28-bit numbers, updated with a linear recursion relation (similar to Fibonacci numbers) modulo 2^{28} . When `\tex_uniformdeviate:D` $\langle integer \rangle$ is called (for brevity denote by N the $\langle integer \rangle$), the next 28-bit number is read from the array, scaled by $N/2^{28}$, and rounded. To prevent 0 and N from appearing half as often as other numbers, they are both mapped to the result 0.

This process means that `\tex_uniformdeviate:D` only gives a uniform distribution from 0 to $N-1$ if N is a divisor of 2^{28} , so we will mostly call the RNG with such power of 2 arguments. If N does not divide 2^{28} , then the relative non-uniformity (difference between probabilities of getting different numbers) is about $N/2^{28}$. This implies that detecting deviation from $1/N$ of the probability of a fixed value X requires about $2^{56}/N$ random trials. But collective patterns can reduce this to about $2^{56}/N^2$. For instance with $N = 3 \times 2^k$, the modulo 3 repartition of such random numbers is biased with a non-uniformity about $2^k/2^{28}$ (which is much worse than the circa $3/2^{28}$ non-uniformity from taking directly $N = 3$). This is detectable after about $2^{56}/2^{2k} = 9 \cdot 2^{56}/N^2$ random numbers. For $k = 15$, $N = 98304$, this means roughly 2^{26} calls to the RNG (experimentally this takes at the very least 16 seconds on a 2 giga-hertz processor). While this bias is not quite problematic, it is uncomfortably close to being so, and it becomes worse as N is increased. In our code, we shall thus combine several results from the RNG.

The RNG has three types of unexpected correlations. First, everything is linear modulo 2^{28} , hence the lowest k bits of the random numbers only depend on the lowest k bits of the seed (and of course the number of times the RNG was called since setting the seed). The recommended way to get a number from 0 to $N-1$ is thus to scale the raw 28-bit integer, as the engine’s RNG does. We will go further and in fact typically we discard some of the lowest bits.

Second, suppose that we call the RNG with the same argument N to get a set of K integers in $[0, N-1]$ (throwing away repeats), and suppose that $N > K^3$ and $K > 55$. The recursion used to construct more 28-bit numbers from previous ones is linear: $x_n = x_{n-55} - x_{n-24}$ or $x_n = x_{n-55} - x_{n-24} + 2^{28}$. After rescaling and rounding we find that the result $N_n \in [0, N-1]$ is among $N_{n-55} - N_{n-24} + \{-1, 0, 1\}$ modulo N (a more detailed analysis shows that 0 appears with frequency close to $3/4$). The resulting set thus has more triplets (a, b, c) than expected obeying $a = b + c$ modulo N . Namely it will have of order $(K-55) \times 3/4$ such triplets, when one would expect $K^3/(6N)$. This starts to be detectable around $N = 2^{18} > 55^3$ (earlier if one keeps track of positions too, but this is more subtle than it looks because the array of 28-bit integers is read backwards by the engine). Hopefully the correlation is subtle enough to not affect realistic documents so we do not specifically mitigate against this. Since we typically use two calls to the RNG per `\int_rand:nn` we would need to investigate linear relations between the x_{2n} on the one hand and between the x_{2n+1} on the other hand. Such relations will have more complicated coefficients than ± 1 , which alleviates the issue.

Third, consider successive batches of 165 calls to the RNG (with argument 2^{28} or with argument 2 for instance), then most batches have more odd than even numbers. Note

that this does not mean that there are more odd than even numbers overall. Similar issues are discussed in Knuth's TAOCP volume 2 near exercise 3.3.2-31. We do not have any mitigation strategy for this.

Ideally, our algorithm should be:

- Uniform. The result should be as uniform as possible assuming that the RNG's underlying 28-bit integers are uniform.
- Uncorrelated. The result should not have detectable correlations between different seeds, similar to the lowest-bit ones mentioned earlier.
- Quick. The algorithm should be fast in \TeX , so no “bit twiddling”, but “digit twiddling” is ok.
- Simple. The behaviour must be documentable precisely.
- Predictable. The number of calls to the RNG should be the same for any `\int_rand:nn`, because then the algorithm can be modified later without changing the result of other uses of the RNG.
- Robust. It should work even for `\int_rand:nn { - \c_max_int } { \c_max_int }` where the range is not representable as an integer. In fact, we also provide later a floating-point `randint` whose range can go all the way up to $2 \times 10^{16} - 1$ possible values.

Some of these requirements conflict. For instance, uniformity cannot be achieved with a fixed number of calls to the RNG.

Denote by `random(N)` one call to `\tex_uniformdeviate:D` with argument N , and by `ediv(p, q)` the ε - \TeX rounding division giving $\lfloor p/q + 1/2 \rfloor$. Denote by $\langle min \rangle$, $\langle max \rangle$ and $R = \langle max \rangle - \langle min \rangle + 1$ the arguments of `\int_min:nn` and the number of possible outcomes. Note that $R \in [1, 2^{32} - 1]$ cannot necessarily be represented as an integer (however, $R - 2^{31}$ can). Our strategy is to get two 28-bit integers X and Y from the RNG, split each into 14-bit integers, as $X = X_1 \times 2^{14} + X_0$ and $Y = Y_1 \times 2^{14} + Y_0$ then return essentially $\langle min \rangle + \lfloor R(X_1 \times 2^{-14} + Y_1 \times 2^{-28} + Y_0 \times 2^{-42} + X_0 \times 2^{-56}) \rfloor$. For small R the X_0 term has a tiny effect so we ignore it and we can compute $R \times Y/2^{28}$ much more directly by `random(R)`.

- If $R \leq 2^{17} - 1$ then return `ediv(R random(2^{14}) + random(R) + 2^{13} , 2^{14}) - 1 + $\langle min \rangle$` . The shifts by 2^{13} and -1 convert ε - \TeX division to truncated division. The bound on R ensures that the number obtained after the shift is less than `\c_max_int`. The non-uniformity is at most of order $2^{17}/2^{42} = 2^{-25}$.
- Split $R = R_2 \times 2^{28} + R_1 \times 2^{14} + R_0$, where $R_2 \in [0, 15]$. Compute $\langle min \rangle + R_2 X_1 2^{14} + (R_2 Y_1 + R_1 X_1) + \text{ediv}(R_2 Y_0 + R_1 Y_1 + R_0 X_1 + \text{ediv}(R_2 X_0 + R_0 Y_1 + \text{ediv}((2^{14} R_1 + R_0)(2^{14} Y_0 + X_0), 2^{28}), 2^{14}), 2^{14})$ then map a result of $\langle max \rangle + 1$ to $\langle min \rangle$. Writing each ediv in terms of truncated division with a shift, and using $\lfloor (p + \lfloor r/s \rfloor)/q \rfloor = \lfloor (ps + r)/(sq) \rfloor$, what we compute is equal to $\lfloor \langle exact \rangle + 2^{-29} + 2^{-15} + 2^{-1} \rfloor$ with $\langle exact \rangle = \langle min \rangle + R \times 0.X_1 Y_1 Y_0 X_0$. Given we map $\langle max \rangle + 1$ to $\langle min \rangle$, the shift has no effect on uniformity. The non-uniformity is bounded by $R/2^{56} < 2^{-24}$. It may be possible to speed up the code by dropping tiny terms such as $R_0 X_0$, but the analysis of non-uniformity proves too difficult.

To avoid the overflow when the computation yields $\langle max \rangle + 1$ with $\langle max \rangle = 2^{31} - 1$ (note that R is then arbitrary), we compute the result in two pieces. Compute

$\langle first \rangle = \langle min \rangle + R_2 X_1 2^{14}$ if $R_2 < 8$ or $\langle min \rangle + 8X_1 2^{14} + (R_2 - 8)X_1 2^{14}$ if $R_2 \geq 8$, the expressions being chosen to avoid overflow. Compute $\langle second \rangle = R_2 Y_1 + R_1 X_1 + \text{ediv}(\dots)$, at most $R_2 2^{14} + R_1 2^{14} + R_0 \leq 2^{28} + 15 \times 2^{14} - 1$, not at risk of overflowing. We have $\langle first \rangle + \langle second \rangle = \langle max \rangle + 1 = \langle min \rangle + R$ if and only if $\langle second \rangle = R 2^{14} + R_0 + R_2 2^{14}$ and $2^{14} R_2 X_1 = 2^{28} R_2 - 2^{14} R_2$ (namely $R_2 = 0$ or $X_1 = 2^{14} - 1$). In that case, return $\langle min \rangle$, otherwise return $\langle first \rangle + \langle second \rangle$, which is safe because it is at most $\langle max \rangle$. Note that the decision of what to return does not need $\langle first \rangle$ explicitly so we don't actually compute it, just put it in an integer expression in which $\langle second \rangle$ is eventually added (or not).

- To get a floating point number in $[0, 1)$ just call the $R = 10000 \leq 2^{17} - 1$ procedure above to produce four blocks of four digits.
- To get an integer floating point number in a range (whose size can be up to $2 \times 10^{16} - 1$), work with fixed-point numbers: get six times four digits to build a fixed point number, multiply by R and add $\langle min \rangle$. This requires some care because l3fp-extended only supports non-negative numbers.

`\c__kernel_randint_max_int` Constant equal to $2^{17} - 1$, the maximal size of a range that `\int_range:nn` can do with its “simple” algorithm.

```
28267 \int_const:Nn \c__kernel_randint_max_int { 131071 }
```

(End definition for `\c__kernel_randint_max_int`.)

`__kernel_randint:n` Used in an integer expression, `__kernel_randint:n {R}` gives a random number $1 + \lfloor (R \text{random}(2^{14}) + \text{random}(R)) / 2^{14} \rfloor$ that is in $[1, R]$. Previous code was computing $\lfloor p / 2^{14} \rfloor$ as `\ediv(p - 2^{13}, 2^{14})` but that wrongly gives -1 for $p = 0$.

```
28268 \cs_new:Npn \__kernel_randint:n #1
28269 {
28270   (#1 * \tex_uniformdeviate:D 16384
28271   + \tex_uniformdeviate:D #1 + 8192 ) / 16384
28272 }
```

(End definition for `__kernel_randint:n`.)

`__fp_rand_myriads:n` Used as `__fp_rand_myriads:n {XXX}` with one letter X (specifically) per block of four digit we want; it expands to `;` followed by the requested number of brace groups, each containing four (pseudo-random) digits. Digits are produced as a random number in $[10000, 19999]$ for the usual reason of preserving leading zeros.

```
28273 \cs_new:Npn \__fp_rand_myriads:n #1
28274 { \__fp_rand_myriads_loop:w #1 \prg_break: X \prg_break_point: ; }
28275 \cs_new:Npn \__fp_rand_myriads_loop:w #1 X
28276 {
28277   #1
28278   \exp_after:wN \__fp_rand_myriads_get:w
28279   \int_value:w \__fp_int_eval:w 9999 +
28280   \__kernel_randint:n { 10000 }
28281   \__fp_rand_myriads_loop:w
28282 }
28283 \cs_new:Npn \__fp_rand_myriads_get:w 1 #1 ; { ; {#1} }
```

(End definition for `__fp_rand_myriads:n`, `__fp_rand_myriads_loop:w`, and `__fp_rand_myriads_get:w`.)

77.2 Random floating point

`__fp_rand_o:Nw` First we check that `random` was called without argument. Then get four blocks of four digits and convert that fixed point number to a floating point number (this correctly sets the exponent). This has a minor bug: if all of the random numbers are zero then the result is correctly 0 but it raises the underflow flag; it should not do that.

`__fp_rand_o:w`

```

28284 \cs_new:Npn \__fp_rand_o:Nw ? #1 @
28285 {
28286   \tl_if_empty:nTF {#1}
28287   {
28288     \exp_after:wN \__fp_rand_o:w
28289     \exp:w \exp_end_continue_f:w
28290     \__fp_rand_myriads:n { XXXX } { 0000 } { 0000 } ; 0
28291   }
28292   {
28293     \msg_expandable_error:nnnnn
28294     { fp } { num-args } { rand() } { 0 } { 0 }
28295     \exp_after:wN \c_nan_fp
28296   }
28297 }
28298 \cs_new:Npn \__fp_rand_o:w ;
28299 {
28300   \exp_after:wN \__fp_sanitize:Nw
28301   \exp_after:wN 0
28302   \int_value:w \__fp_int_eval:w \c_zero_int
28303   \__fp_fixed_to_float_o:wN
28304 }

```

(End definition for `__fp_rand_o:Nw` and `__fp_rand_o:w`.)

77.3 Random integer

`__fp_randint_o:Nw` Enforce that there is one argument (then add first argument 1) or two arguments. Call `__fp_randint_default:w` `__fp_randint_badarg:w` on each; this function inserts `1 \exp_stop_f:` to end the `\if_case:w` statement if either the argument is not an integer or if its absolute value is $\geq 10^{16}$. Also bail out if `__fp_compare_back:ww` yields 1, meaning that the bounds are not in the right order. Otherwise an auxiliary converts each argument times 10^{-16} (hence the shift in exponent) to a 24-digit fixed point number (see `l3fp-extended`). Then compute the number of choices, $\langle max \rangle + 1 - \langle min \rangle$. Create a random 24-digit fixed-point number with `__fp_rand_myriads:n`, then use a fused multiply-add instruction to multiply the number of choices to that random number and add it to $\langle min \rangle$. Then truncate to 16 digits (namely select the integer part of 10^{16} times the result) before converting back to a floating point number (`__fp_sanitize:Nw` takes care of zero). To avoid issues with negative numbers, add 1 to all fixed point numbers (namely 10^{16} to the integers they represent), except of course when it is time to convert back to a float.

```

28305 \cs_new:Npn \__fp_randint_o:Nw ?
28306 {
28307   \__fp_parse_function_one_two:nnw
28308   { randint }
28309   { \__fp_randint_default:w \__fp_randint_o:w }
28310 }
28311 \cs_new:Npn \__fp_randint_default:w #1 { \exp_after:wN #1 \c_one_fp }

```



```

28312 \cs_new:Npn \__fp_randint_badarg:w \s__fp \__fp_chk:w #1#2#3;
28313 {
28314   \__fp_int:wTF \s__fp \__fp_chk:w #1#2#3;
28315   {
28316     \if_meaning:w 1 #1
28317     \if_int_compare:w
28318       \__fp_use_i_until_s:nw #3 ; > \c__fp_prec_int
28319       \c_one_int
28320       \fi:
28321       \fi:
28322     }
28323     { \c_one_int }
28324   }
28325 \cs_new:Npn \__fp_randint_o:w #1; #2; @
28326 {
28327   \if_case:w
28328     \__fp_randint_badarg:w #1;
28329     \__fp_randint_badarg:w #2;
28330     \if:w 1 \__fp_compare_back:ww #2; #1; \c_one_int \fi:
28331     \c_zero_int
28332     \__fp_randint_auxi_o:ww #1; #2;
28333   \or:
28334     \__fp_invalid_operation_tl_o:ff
28335     { randint } { \__fp_array_to_clist:n { #1; #2; } }
28336   \exp:w
28337   \fi:
28338   \exp_after:wN \exp_end:
28339 }
28340 \cs_new:Npn \__fp_randint_auxi_o:ww #1 ; #2 ; #3 \exp_end:
28341 {
28342   \fi:
28343   \__fp_randint_auxii:wn #2 ;
28344   { \__fp_randint_auxii:wn #1 ; \__fp_randint_auxiii_o:ww }
28345 }
28346 \cs_new:Npn \__fp_randint_auxii:wn \s__fp \__fp_chk:w #1#2#3#4 ;
28347 {
28348   \if_meaning:w 0 #1
28349   \exp_after:wN \use_i:nn
28350   \else:
28351   \exp_after:wN \use_ii:nn
28352   \fi:
28353   { \exp_after:wN \__fp_fixed_continue:wn \c__fp_one_fixed_tl }
28354   {
28355     \exp_after:wN \__fp_ep_to_fixed:wwn
28356     \int_value:w \__fp_int_eval:w
28357     #3 - \c__fp_prec_int , #4 {0000} {0000} ;
28358     {
28359       \if_meaning:w 0 #2
28360       \exp_after:wN \use_i:nnnn
28361       \exp_after:wN \__fp_fixed_add_one:wN
28362       \fi:
28363       \exp_after:wN \__fp_fixed_sub:wwn \c__fp_one_fixed_tl
28364     }
28365     \__fp_fixed_continue:wn

```

```

28366     }
28367   }
28368   \cs_new:Npn \__fp_randint_auxiii_o:ww #1 ; #2 ;
28369   {
28370     \__fp_fixed_add:wwn #2 ;
28371     {0000} {0000} {0000} {0001} {0000} {0000} ;
28372     \__fp_fixed_sub:wwn #1 ;
28373     {
28374       \exp_after:wN \use_i:nn
28375       \exp_after:wN \__fp_fixed_mul_add:wwwn
28376       \exp:w \exp_end_continue_f:w \__fp_rand_myriads:n { XXXXXX } ;
28377     }
28378     #1 ;
28379     \__fp_randint_auxiv_o:ww
28380     #2 ;
28381     \__fp_randint_auxv_o:w #1 ; @
28382   }
28383   \cs_new:Npn \__fp_randint_auxiv_o:ww #1#2#3#4#5 ; #6#7#8#9
28384   {
28385     \if_int_compare:w
28386       \if_int_compare:w #1#2 > #6#7 \exp_stop_f: 1 \else:
28387       \if_int_compare:w #1#2 < #6#7 \exp_stop_f: - \fi: \fi:
28388       #3#4 > #8#9 \exp_stop_f:
28389     \__fp_use_i_until_s:nw
28390     \fi:
28391     \__fp_randint_auxv_o:w {#1}{#2}{#3}{#4}#5
28392   }
28393   \cs_new:Npn \__fp_randint_auxv_o:w #1#2#3#4#5 ; #6 @
28394   {
28395     \exp_after:wN \__fp_sanitize:Nw
28396     \int_value:w
28397     \if_int_compare:w #1 < 10000 \exp_stop_f:
28398     2
28399     \else:
28400     0
28401     \exp_after:wN \exp_after:wN
28402     \exp_after:wN \__fp_reverse_args:Nww
28403     \fi:
28404     \exp_after:wN \__fp_fixed_sub:wwn \c__fp_one_fixed_tl
28405     {#1} {#2} {#3} {#4} {0000} {0000} ;
28406     {
28407       \exp_after:wN \exp_stop_f:
28408       \int_value:w \__fp_int_eval:w \c__fp_prec_int
28409       \__fp_fixed_to_float_o:wN
28410     }
28411     0
28412     \exp:w \exp_after:wN \exp_end:
28413   }

```

(End definition for __fp_randint_o:Nw and others.)

\int_rand:nn Evaluate the argument and filter out the case where the lower bound #1 is more than
__fp_randint:ww the upper bound #2. Then determine whether the range is narrower than **\c__kernel_-randint_max_int**; #2-#1 may overflow for very large positive #2 and negative #1. If the

range is narrow, call `__kernel_randint:n {⟨choices⟩}` where $\langle choices \rangle$ is the number of possible outcomes. If the range is wide, use somewhat slower code.

```

28414 \cs_new:Npn \int_rand:nn #1#2
28415 {
28416   \int_eval:n
28417   {
28418     \exp_after:wN \__fp_randint:ww
28419     \int_value:w \int_eval:n {#1} \exp_after:wN ;
28420     \int_value:w \int_eval:n {#2} ;
28421   }
28422 }
28423 \cs_new:Npn \__fp_randint:ww #1; #2;
28424 {
28425   \if_int_compare:w #1 > #2 \exp_stop_f:
28426   \msg_expandable_error:nnnn
28427   { kernel } { randint-backward-range } {#1} {#2}
28428   \__fp_randint:ww #2; #1;
28429   \else:
28430     \if_int_compare:w \__fp_int_eval:w #2
28431     \if_int_compare:w #1 > \c_zero_int
28432     - #1 < \__fp_int_eval:w
28433     \else:
28434       < \__fp_int_eval:w #1 +
28435       \fi:
28436       \c_kernel_randint_max_int
28437       \__fp_int_eval_end:
28438       \__kernel_randint:n
28439       { \__fp_int_eval:w #2 - #1 + 1 \__fp_int_eval_end: }
28440       - 1 + #1
28441     \else:
28442       \__kernel_randint:nn {#1} {#2}
28443     \fi:
28444   \fi:
28445 }

```

(End definition for `\int_rand:nn` and `__fp_randint:ww`. This function is documented on page 165.)

`__kernel_randint:nn` Any $n \in [-2^{31} + 1, 2^{31} - 1]$ is uniquely written as $2^{14}n_1 + n_2$ with $n_1 \in [-2^{17}, 2^{17} - 1]$ and $n_2 \in [0, 2^{14} - 1]$. Calling `__fp_randint_split_o:Nw n ;` gives n_1 ; n_2 ; and expands the next token once. We do this for two random numbers and apply `__fp_randint_split_o:Nw` twice to fully decompose the range R . One subtlety is that we compute $R - 2^{31} = \langle max \rangle - \langle min \rangle - (2^{31} - 1) \in [-2^{31} + 1, 2^{31} - 1]$ rather than R to avoid overflow.

Then we have `__fp_randint_wide_aux:w ⟨X1⟩; ⟨X0⟩; ⟨Y1⟩; ⟨Y0⟩; ⟨R2⟩; ⟨R1⟩; ⟨R0⟩;` and we apply the algorithm described earlier.

```

28446 \cs_new:Npn \__kernel_randint:nn #1#2
28447 {
28448   #1
28449   \exp_after:wN \__fp_randint_wide_aux:w
28450   \int_value:w
28451   \exp_after:wN \__fp_randint_split_o:Nw
28452   \tex_uniformdeviate:D 268435456 ;
28453   \int_value:w
28454   \exp_after:wN \__fp_randint_split_o:Nw

```

```

28455         \tex_uniformdeviate:D 268435456 ;
28456     \int_value:w
28457         \exp_after:wN \__fp_randint_split_o:Nw
28458         \int_value:w \__fp_int_eval:w 131072 +
28459         \exp_after:wN \__fp_randint_split_o:Nw
28460         \int_value:w
28461         \__kernel_int_add:nnn {#2} { -#1 } { -\c_max_int } ;
28462     .
28463 }
28464 \cs_new:Npn \__fp_randint_split_o:Nw #1#2 ;
28465 {
28466     \if_meaning:w 0 #1
28467     0 \exp_after:wN ; \int_value:w 0
28468     \else:
28469         \exp_after:wN \__fp_randint_split_aux:w
28470         \int_value:w \__fp_int_eval:w (#1#2 - 8192) / 16384 ;
28471         + #1#2
28472     \fi:
28473     \exp_after:wN ;
28474 }
28475 \cs_new:Npn \__fp_randint_split_aux:w #1 ;
28476 {
28477     #1 \exp_after:wN ;
28478     \int_value:w \__fp_int_eval:w - #1 * 16384
28479 }
28480 \cs_new:Npn \__fp_randint_wide_aux:w #1;#2; #3;#4; #5;#6;#7; .
28481 {
28482     \exp_after:wN \__fp_randint_wide_auxii:w
28483     \int_value:w \__fp_int_eval:w #5 * #3 + #6 * #1 +
28484     (#5 * #4 + #6 * #3 + #7 * #1 +
28485     (#5 * #2 + #7 * #3 +
28486     (16384 * #6 + #7) * (16384 * #4 + #2) / 268435456) / 16384
28487     ) / 16384 \exp_after:wN ;
28488     \int_value:w \__fp_int_eval:w (#5 + #6) * 16384 + #7 ;
28489     #1 ; #5 ;
28490 }
28491 \cs_new:Npn \__fp_randint_wide_auxii:w #1; #2; #3; #4;
28492 {
28493     \if_int_odd:w 0
28494         \if_int_compare:w #1 = #2 \else: \exp_stop_f: \fi:
28495         \if_int_compare:w #4 = \c_zero_int 1 \fi:
28496         \if_int_compare:w #3 = 16383 ~ 1 \fi:
28497         \exp_stop_f:
28498         \exp_after:wN \prg_break:
28499     \fi:
28500     \if_int_compare:w #4 < 8 \exp_stop_f:
28501     + #4 * #3 * 16384
28502     \else:
28503     + 8 * #3 * 16384 + (#4 - 8) * #3 * 16384
28504     \fi:
28505     + #1
28506     \prg_break_point:
28507 }

```

(End definition for __kernel_randint:nn and others.)

`\int_rand:n` Similar to `\int_rand:nn`, but needs fewer checks.

```

\__fp_randint:n 28508 \cs_new:Npn \int_rand:n #1
                28509 {
                28510   \int_eval:n
                28511   { \exp_args:Nf \__fp_randint:n { \int_eval:n {#1} } }
                28512 }
                28513 \cs_new:Npn \__fp_randint:n #1
                28514 {
                28515   \if_int_compare:w #1 < \c_one_int
                28516   \msg_expandable_error:nnnn
                28517   { kernel } { randint-backward-range } { 1 } {#1}
                28518   \__fp_randint:ww #1; 1;
                28519   \else:
                28520   \if_int_compare:w #1 > \c__kernel_randint_max_int
                28521   \__kernel_randint:nn { 1 } {#1}
                28522   \else:
                28523   \__kernel_randint:n {#1}
                28524   \fi:
                28525   \fi:
                28526 }

```

(End definition for `\int_rand:n` and `__fp_randint:n`. This function is documented on page 165.)

End the initial conditional that ensures these commands are only defined in engines that support random numbers.

```

28527 }
28528 </package>

```

Chapter 78

l3fparray implementation

```
28529 <*package>
```

```
28530 <@@=fp>
```

In analogy to l3intarray it would make sense to have <@@=fparray>, but we need direct access to `__fp_parse:n` from l3fp-parse, and a few other (less crucial) internals of the l3fp family.

78.1 Allocating arrays

There are somewhat more than $(2^{31} - 1)^2$ floating point numbers so we store each floating point number as three entries in integer arrays. To avoid having to multiply indices by three or to add 1 etc, a floating point array is just a token list consisting of three tokens: integer arrays of the same size.

`\g__fp_array_int` Used to generate unique names for the three integer arrays.

```
28531 \int_new:N \g__fp_array_int
```

(End definition for \g__fp_array_int.)

`\l__fp_array_loop_int` Used to loop in `__fp_array_gzero:N`.

```
28532 \int_new:N \l__fp_array_loop_int
```

(End definition for \l__fp_array_loop_int.)

`\fparray_new:Nn` Build a three-token token list, then define all three tokens to be integer arrays of the same size. No need to initialize the data: the integer arrays start with zeros, and three zeros denote precisely `\c_zero_fp`, as we want.

`\fparray_new:cn`
`__fp_array_new:nNNN`

```
28533 \cs_new_protected:Npn \fparray_new:Nn #1#2
28534 {
28535   \tl_new:N #1
28536   \prg_replicate:nn { 3 }
28537   {
28538     \int_gincr:N \g__fp_array_int
28539     \exp_args:NNc \tl_gput_right:Nn #1
28540     { g__fp_array_ \__fp_int_to_roman:w \g__fp_array_int _intarray }
28541   }
28542   \exp_last_unbraced:Nfo \__fp_array_new:nNNNN
28543   { \int_eval:n {#2} } #1 #1
```

```

28544     }
28545     \cs_generate_variant:Nn \fparray_new:Nn { c }
28546     \cs_new_protected:Npn \__fp_array_new:nNNNN #1#2#3#4#5
28547     {
28548         \int_compare:nNnTF {#1} < 0
28549         {
28550             \msg_error:nnn { kernel } { negative-array-size } {#1}
28551             \cs_undefine:N #1
28552             \int_gsub:Nn \g__fp_array_int { 3 }
28553         }
28554         {
28555             \intarray_new:Nn #2 {#1}
28556             \intarray_new:Nn #3 {#1}
28557             \intarray_new:Nn #4 {#1}
28558         }
28559     }

```

(End definition for `\fparray_new:Nn` and `__fp_array_new:nNNNN`. This function is documented on page 260.)

`\fparray_count:N` Size of any of the intarrays, here we pick the third.

```

\fparray_count:c
28560 \cs_new:Npn \fparray_count:N #1
28561 {
28562     \exp_after:wN \use_i:nnn
28563     \exp_after:wN \intarray_count:N #1
28564 }
28565 \cs_generate_variant:Nn \fparray_count:N { c }

```

(End definition for `\fparray_count:N`. This function is documented on page 260.)

78.2 Array items

`__fp_array_bounds:NNnTF` See the `l3intarray` analogue: only names change. The functions `\fparray_gset:Nnn` and `__fp_array_bounds_error:NNn` `\fparray_item:Nn` share bounds checking. The T branch is used if #3 is within bounds of the array #2.

```

28566 \cs_new:Npn \__fp_array_bounds:NNnTF #1#2#3#4#5
28567 {
28568     \if_int_compare:w 1 > #3 \exp_stop_f:
28569     \__fp_array_bounds_error:NNn #1 #2 {#3}
28570     #5
28571     \else:
28572     \if_int_compare:w #3 > \fparray_count:N #2 \exp_stop_f:
28573     \__fp_array_bounds_error:NNn #1 #2 {#3}
28574     #5
28575     \else:
28576     #4
28577     \fi:
28578     \fi:
28579 }
28580 \cs_new:Npn \__fp_array_bounds_error:NNn #1#2#3
28581 {
28582     #1 { kernel } { out-of-bounds }
28583     { \token_to_str:N #2 } {#3} { \fparray_count:N #2 }
28584 }

```

(End definition for _fp_array_bounds:NNnTF and _fp_array_bounds_error:NNn.)

\fparray_gset:Nnn Evaluate, then store exponent in one intarray, sign and 8 digits of mantissa in the next,
\fparray_gset:cnn and 8 trailing digits in the last.

```

\_fp_array_gset:NNNNww 28585 \cs_new_protected:Npn \fparray_gset:Nnn #1#2#3
\_fp_array_gset:w 28586 {
\_fp_array_gset_recover:Nw 28587 \exp_after:wN \exp_after:wN
\_fp_array_gset_special:nnNNN 28588 \exp_after:wN \_fp_array_gset:NNNNww
\_fp_array_gset_normal:w 28589 \exp_after:wN #1
28590 \exp_after:wN #1
28591 \int_value:w \int_eval:n {#2} \exp_after:wN ;
28592 \exp:w \exp_end_continue_f:w \_fp_parse:n {#3}
28593 }
28594 \cs_generate_variant:Nn \fparray_gset:Nnn { c }
28595 \cs_new_protected:Npn \_fp_array_gset:NNNNww #1#2#3#4#5 ; #6 ;
28596 {
28597 \_fp_array_bounds:NNnTF \msg_error:nnxxx #4 {#5}
28598 {
28599 \exp_after:wN \_fp_change_func_type:NNN
28600 \_fp_use_i_until_s:nw #6 ;
28601 \_fp_array_gset:w
28602 \_fp_array_gset_recover:Nw
28603 #6 ; {#5} #1 #2 #3
28604 }
28605 { }
28606 }
28607 \cs_new_protected:Npn \_fp_array_gset_recover:Nw #1#2 ;
28608 {
28609 \_fp_error:nffn { unknown-type } { \tl_to_str:n { #2 ; } } { } { }
28610 \exp_after:wN #1 \c_nan_fp
28611 }
28612 \cs_new_protected:Npn \_fp_array_gset:w \s__fp \_fp_chk:w #1#2
28613 {
28614 \if_case:w #1 \exp_stop_f:
28615 \_fp_case_return:nw { \_fp_array_gset_special:nnNNN {#2} }
28616 \or: \exp_after:wN \_fp_array_gset_normal:w
28617 \or: \_fp_case_return:nw { \_fp_array_gset_special:nnNNN { #2 3 } }
28618 \or: \_fp_case_return:nw { \_fp_array_gset_special:nnNNN { 1 } }
28619 \fi:
28620 \s__fp \_fp_chk:w #1 #2
28621 }
28622 \cs_new_protected:Npn \_fp_array_gset_normal:w
28623 \s__fp \_fp_chk:w 1 #1 #2 #3#4#5 ; #6#7#8#9
28624 {
28625 \_kernel_intarray_gset:Nnn #7 {#6} {#2}
28626 \_kernel_intarray_gset:Nnn #8 {#6}
28627 { \if_meaning:w 2 #1 3 \else: 1 \fi: #3#4 }
28628 \_kernel_intarray_gset:Nnn #9 {#6} { 1 \use:nn #5 }
28629 }
28630 \cs_new_protected:Npn \_fp_array_gset_special:nnNNN #1#2#3#4#5
28631 {
28632 \_kernel_intarray_gset:Nnn #3 {#2} {#1}
28633 \_kernel_intarray_gset:Nnn #4 {#2} {0}
28634 \_kernel_intarray_gset:Nnn #5 {#2} {0}

```



```
28635 }
```

(End definition for \fpararray_gset:Nnn and others. This function is documented on page 260.)

\fpararray_gzero:N

\fpararray_gzero:c

```
28636 \cs_new_protected:Npn \fpararray_gzero:N #1
28637 {
28638   \int_zero:N \l__fp_array_loop_int
28639   \prg_replicate:nn { \fpararray_count:N #1 }
28640   {
28641     \int_incr:N \l__fp_array_loop_int
28642     \exp_after:wN \__fp_array_gset_special:nnNNN
28643     \exp_after:wN 0
28644     \exp_after:wN \l__fp_array_loop_int
28645     #1
28646   }
28647 }
28648 \cs_generate_variant:Nn \fpararray_gzero:N { c }
```

(End definition for \fpararray_gzero:N. This function is documented on page 260.)

\fpararray_item:Nn

\fpararray_item:cn

\fpararray_item_to_tl:Nn

\fpararray_item_to_tl:cn

__fp_array_item:NwN

__fp_array_item:NNNnN

__fp_array_item:N

__fp_array_item:w

__fp_array_item_special:w

__fp_array_item_normal:w

```
28649 \cs_new:Npn \fpararray_item:Nn #1#2
28650 {
28651   \exp_after:wN \__fp_array_item:NwN
28652   \exp_after:wN #1
28653   \int_value:w \int_eval:n {#2} ;
28654   \__fp_to_decimal:w
28655 }
28656 \cs_generate_variant:Nn \fpararray_item:Nn { c }
28657 \cs_new:Npn \fpararray_item_to_tl:Nn #1#2
28658 {
28659   \exp_after:wN \__fp_array_item:NwN
28660   \exp_after:wN #1
28661   \int_value:w \int_eval:n {#2} ;
28662   \__fp_to_tl:w
28663 }
28664 \cs_generate_variant:Nn \fpararray_item_to_tl:Nn { c }
28665 \cs_new:Npn \__fp_array_item:NwN #1#2 ; #3
28666 {
28667   \__fp_array_bounds:NNnTF \msg_expandable_error:nnfff #1 {#2}
28668   { \exp_after:wN \__fp_array_item:NNNnN #1 {#2} #3 }
28669   { \exp_after:wN #3 \c_nan_fp }
28670 }
28671 \cs_new:Npn \__fp_array_item:NNNnN #1#2#3#4
28672 {
28673   \exp_after:wN \__fp_array_item:N
28674   \int_value:w \__kernel_intarray_item:Nn #2 {#4} \exp_after:wN ;
28675   \int_value:w \__kernel_intarray_item:Nn #3 {#4} \exp_after:wN ;
28676   \int_value:w \__kernel_intarray_item:Nn #1 {#4} ;
28677 }
28678 \cs_new:Npn \__fp_array_item:N #1
28679 {
28680   \if_meaning:w 0 #1 \exp_after:wN \__fp_array_item_special:w \fi:
28681   \__fp_array_item:w #1
```

```

28682 }
28683 \cs_new:Npn \__fp_array_item:w #1 #2#3#4#5 #6 ; 1 #7 ;
28684 {
28685   \exp_after:wN \__fp_array_item_normal:w
28686   \int_value:w \if_meaning:w #1 1 0 \else: 2 \fi: \exp_stop_f:
28687   #7 ; {#2#3#4#5} {#6} ;
28688 }
28689 \cs_new:Npn \__fp_array_item_special:w #1 ; #2 ; #3 ; #4
28690 {
28691   \exp_after:wN #4
28692   \exp:w \exp_end_continue_f:w
28693   \if_case:w #3 \exp_stop_f:
28694     \exp_after:wN \c_zero_fp
28695   \or: \exp_after:wN \c_nan_fp
28696   \or: \exp_after:wN \c_minus_zero_fp
28697   \or: \exp_after:wN \c_inf_fp
28698   \else: \exp_after:wN \c_minus_inf_fp
28699   \fi:
28700 }
28701 \cs_new:Npn \__fp_array_item_normal:w #1 #2#3#4#5 #6 ; #7 ; #8 ; #9
28702 { #9 \s_fp \__fp_chk:w 1 #1 {#8} #7 {#2#3#4#5} {#6} ; }

```

(End definition for `\farray_item:Nn` and others. These functions are documented on page [260](#).)

```

28703 \endpackage

```

Chapter 79

l3cctab implementation

28704 `\package`

28705 `\cctab`

As LuaTeX offers engine support for category code tables, and this is entirely lacking from the other engines, we need two complementary approaches. (Some future XeTeX may add support, at which point the conditionals below would be different.)

79.1 Variables

`\g__cctab_stack_seq` List of catcode tables saved by nested `\cctab_begin:N`, to restore catcodes at the matching `\cctab_end:.` When popped from the `\g__cctab_stack_seq` the table numbers are stored in `\g__cctab_unused_seq` for later reuse.

28706 `\seq_new:N \g__cctab_stack_seq`

28707 `\seq_new:N \g__cctab_unused_seq`

(End definition for \g__cctab_stack_seq and \g__cctab_unused_seq.)

`\g__cctab_group_seq` A stack to store the group level when a catcode table started.

28708 `\seq_new:N \g__cctab_group_seq`

(End definition for \g__cctab_group_seq.)

`\g__cctab_allocate_int` Integer to keep track of what category code table to allocate. In LuaTeX it is only used in format mode to implement `\cctab_new:N`. In other engines it is used to make csnames for dynamic tables.

28709 `\int_new:N \g__cctab_allocate_int`

(End definition for \g__cctab_allocate_int.)

`\l__cctab_internal_a_tl` Scratch space. For instance, when popping `\g__cctab_stack_seq/\g__cctab_unused_seq`, consists of the catcodetable number (integer denotation) in LuaTeX, or of an intarray variable (as a single token) in other engines.

28710 `\tl_new:N \l__cctab_internal_a_tl`

28711 `\tl_new:N \l__cctab_internal_b_tl`

(End definition for \l__cctab_internal_a_tl and \l__cctab_internal_b_tl.)

`\g__cctab_endlinechar_prop` In LuaTeX we store the `\endlinechar` associated to each `\catcodetable` in a property list, unless it is the default value 13.

28712 `\prop_new:N \g__cctab_endlinechar_prop`

(End definition for `\g__cctab_endlinechar_prop`.)

79.2 Allocating category code tables

`\cctab_new:N` The `__cctab_new:N` auxiliary allocates a new catcode table but does not attempt to set its value consistently across engines. It is used both in `\cctab_new:N`, which sets catcodes to `iniTeX` values, and in `\cctab_begin:N/\cctab_end:` for dynamically allocated tables.

`\cctab_new:c`
`__cctab_new:N`
`__cctab_gstore:Nnn`

First, the LuaTeX case. Creating a new category code table is done like other registers. In ConTeXt, `\newcatcodetable` does not include the initialisation, so that is added explicitly.

```
28713 \sys_if_engine luatex:TF
28714 {
28715   \cs_new_protected:Npn \cctab_new:N #1
28716   {
28717     \__kernel_chk_if_free_cs:N #1
28718     \__cctab_new:N #1
28719   }
28720   \cs_new_protected:Npn \__cctab_new:N #1
28721   {
28722     \newcatcodetable #1
28723     \tex_initcatcodetable:D #1
28724   }
28725 }
```

Now the case for other engines. Here, each table is an integer array. Following the LuaTeX pattern, a new table starts with `iniTeX` codes. The index base is out-by-one, so we have an internal function to handle that. The `iniTeX` `\endlinechar` is 13.

```
28726 {
28727   \cs_new_protected:Npn \__cctab_new:N #1
28728   { \intarray_new:Nn #1 { 257 } }
28729   \cs_new_protected:Npn \__cctab_gstore:Nnn #1#2#3
28730   { \intarray_gset:Nnn #1 { \int_eval:n { #2 + 1 } } {#3} }
28731   \cs_new_protected:Npn \cctab_new:N #1
28732   {
28733     \__kernel_chk_if_free_cs:N #1
28734     \__cctab_new:N #1
28735     \int_step_inline:nn { 256 }
28736     { \__kernel_intarray_gset:Nnn #1 {##1} { 12 } }
28737     \__kernel_intarray_gset:Nnn #1 { 257 } { 13 }
28738     \__cctab_gstore:Nnn #1 { 0 } { 9 }
28739     \__cctab_gstore:Nnn #1 { 13 } { 5 }
28740     \__cctab_gstore:Nnn #1 { 32 } { 10 }
28741     \__cctab_gstore:Nnn #1 { 37 } { 14 }
28742     \int_step_inline:nnn { 65 } { 90 }
28743     { \__cctab_gstore:Nnn #1 {##1} { 11 } }
28744     \__cctab_gstore:Nnn #1 { 92 } { 0 }
28745     \int_step_inline:nnn { 97 } { 122 }
28746     { \__cctab_gstore:Nnn #1 {##1} { 11 } }
28747     \__cctab_gstore:Nnn #1 { 127 } { 15 }
```

```

28748     }
28749   }
28750   \cs_generate_variant:Nn \cctab_new:N { c }

```

(End definition for `\cctab_new:N`, `__cctab_new:N`, and `__cctab_gstore:Nnn`. This function is documented on page 261.)

79.3 Saving category code tables

`__cctab_gset:n` In various functions we need to save the current catcodes (globally) in a table. In LuaTeX, saving the catcodes is a primitives, but the `\endlinechar` needs more work: to avoid filling `\g__cctab_endlinechar_prop` with many entries we special-case the default value 13. In other engines we store 256 current catcodes and the `\endlinechar` in an intarray variable.

```

28751 \sys_if_engine luatex:TF
28752 {
28753   \cs_new_protected:Npn \__cctab_gset:n #1
28754     { \exp_args:Nf \__cctab_gset_aux:n { \int_eval:n {#1} } }
28755   \cs_new_protected:Npn \__cctab_gset_aux:n #1
28756     {
28757       \tex_savecatcodetable:D #1 \scan_stop:
28758       \int_compare:nNnTF { \tex_endlinechar:D } = { 13 }
28759         { \prop_gremove:Nn \g__cctab_endlinechar_prop {#1} }
28760         {
28761           \prop_gput:NnV \g__cctab_endlinechar_prop {#1}
28762             \tex_endlinechar:D
28763         }
28764     }
28765 }
28766 {
28767   \cs_new_protected:Npn \__cctab_gset:n #1
28768     {
28769       \int_step_inline:nn { 256 }
28770       {
28771         \__kernel_intarray_gset:Nnn #1 {##1}
28772         { \char_value_catcode:n { ##1 - 1 } }
28773       }
28774       \__kernel_intarray_gset:Nnn #1 { 257 }
28775       { \tex_endlinechar:D }
28776     }
28777 }

```

(End definition for `__cctab_gset:n` and `__cctab_gset_aux:n`.)

`\cctab_gset:Nn` Category code tables are always global, so only one version of assignments is needed.
`\cctab_gset:cn` Simply run the setup in a group and save the result in a category code table #1, provided it is valid. The internal function is defined above depending on the engine.

```

28778 \cs_new_protected:Npn \cctab_gset:Nn #1#2
28779 {
28780   \__cctab_chk_if_valid:NT #1
28781   {
28782     \group_begin:
28783     \cctab_select:N \c_initex_cctab

```

```

28784         #2 \scan_stop:
28785         \__cctab_gset:n {#1}
28786     \group_end:
28787 }
28788 }
28789 \cs_generate_variant:Nn \cctab_gset:Nn { c }

```

(End definition for `\cctab_gset:Nn`. This function is documented on page 261.)

79.4 Using category code tables

`\g__cctab_internal_cctab`
`__cctab_internal_cctab_name:`

In LuaTeX, we must ensure that the saved tables are read-only. This is done by applying the saved table, then switching immediately to a scratch table. Any later catcode assignment will affect that scratch table rather than the saved one. If we simply switched to the saved tables, then `\char_set_catcode_other:N` in the example below would change `\c_document_cctab` and a later use of that table would give the wrong category code to `_`.

```

\use:n
{
  \cctab_begin:N \c_document_cctab
  \char_set_catcode_other:N \_
  \cctab_end:
  \cctab_begin:N \c_document_cctab
  \int_compare:nTF { \char_value_catcode:n { '_' } = 8 }
  { \TRUE } { \ERROR }
  \cctab_end:
}

```

We must also make sure that a scratch table is never reused in a nested group: in the following example, the scratch table used by the first `\cctab_begin:N` would be changed globally by the second one issuing `\savecatcodetable`, and after `\group_end:` the wrong category codes (those of `\c_str_cctab`) would be imposed. Note that the inner `\cctab_end:` restores the correct catcodes only locally, so the problem really comes up because of the different grouping level. The simplest is to use a scratch table labeled by the `\currentgrouplevel`. We initialize one of them as an example.

```

\use:n
{
  \cctab_begin:N \c_document_cctab
  \group_begin:
  \cctab_begin:N \c_str_cctab
  \cctab_end:
  \group_end:
  \cctab_end:
}

28790 \sys_if_engine luatex:T
28791 {
28792   \__cctab_new:N \g__cctab_internal_cctab
28793   \cs_new:Npn \__cctab_internal_cctab_name:
28794   {

```

```

28795     g__cctab_internal
28796     \tex_romannumeral:D \tex_currentgrouplevel:D
28797     _cctab
28798   }
28799 }

```

(End definition for `\g__cctab_internal_cctab` and `__cctab_internal_cctab_name:`)

`\cctab_select:N` The public function simply checks the `\cctab var` exists before using the engine-dependent `__cctab_select:N`. Skipping these checks would result in low-level engine-dependent errors. First, the LuaTeX case. In other engines, selecting a catcode table is a matter of doing 256 catcode assignments and setting the `\endlinechar`.

```

28800 \cs_new_protected:Npn \cctab_select:N #1
28801 { \__cctab_chk_if_valid:NT #1 { \__cctab_select:N #1 } }
28802 \cs_generate_variant:Nn \cctab_select:N { c }
28803 \sys_if_engine luatex:TF
28804 {
28805   \cs_new_protected:Npn \__cctab_select:N #1
28806   {
28807     \tex_catcodetable:D #1
28808     \prop_get:NVNTF \g__cctab_endlinechar_prop #1 \l__cctab_internal_a_tl
28809     { \int_set:Nn \tex_endlinechar:D { \l__cctab_internal_a_tl } }
28810     { \int_set:Nn \tex_endlinechar:D { 13 } }
28811     \cs_if_exist:cF { \__cctab_internal_cctab_name: }
28812     { \exp_args:Nc \__cctab_new:N { \__cctab_internal_cctab_name: } }
28813     \exp_args:Nc \tex_savecatcodetable:D { \__cctab_internal_cctab_name: }
28814     \exp_args:Nc \tex_catcodetable:D { \__cctab_internal_cctab_name: }
28815   }
28816 }
28817 {
28818   \cs_new_protected:Npn \__cctab_select:N #1
28819   {
28820     \int_step_inline:nn { 256 }
28821     {
28822       \char_set_catcode:nn { ##1 - 1 }
28823       { \__kernel_intarray_item:Nn #1 {##1} }
28824     }
28825     \int_set:Nn \tex_endlinechar:D
28826     { \__kernel_intarray_item:Nn #1 { 257 } }
28827   }
28828 }

```

(End definition for `\cctab_select:N` and `__cctab_select:N`. This function is documented on page 262.)

`\g__cctab_next_cctab` For `\cctab_begin:N/\cctab_end:` we will need to allocate dynamic tables. This is done here by `__cctab_begin_aux:`, which puts a table number (in LuaTeX) or name (in other engines) into `\l__cctab_internal_a_tl`. In LuaTeX this simply calls `__cctab_new:N` and uses the resulting catcodetable number; in other engines we need to give a name to the intarray variable and use that. In LuaTeX, to restore catcodes at `\cctab_end:` we cannot just set `\catcodetable` to its value before `\cctab_begin:N`, because that table may have been altered by other code in the mean time. So we must make sure to save the catcodes in a table we control and restore them at `\cctab_end:`.

```

28829 \sys_if_engine luatex:TF

```

```

28830 {
28831   \cs_new_protected:Npn \__cctab_begin_aux:
28832   {
28833     \__cctab_new:N \g__cctab_next_cctab
28834     \tl_set:NV \l__cctab_internal_a_tl \g__cctab_next_cctab
28835     \cs_undefine:N \g__cctab_next_cctab
28836   }
28837 }
28838 {
28839   \cs_new_protected:Npn \__cctab_begin_aux:
28840   {
28841     \int_gincr:N \g__cctab_allocate_int
28842     \exp_args:Nc \__cctab_new:N
28843     { \g__cctab_ \int_use:N \g__cctab_allocate_int _cctab }
28844     \exp_args:NNc \tl_set:Nn \l__cctab_internal_a_tl
28845     { \g__cctab_ \int_use:N \g__cctab_allocate_int _cctab }
28846   }
28847 }

```

(End definition for \g__cctab_next_cctab and __cctab_begin_aux:.)

\cctab_begin:N Check the $\langle cctab\ var \rangle$ exists, to avoid low-level errors. Get in \l__cctab_internal_a_tl the number/name of a dynamic table, either from \g__cctab_unused_seq where we save tables that are not currently in use, or from __cctab_begin_aux: if none are available. Then save the current catcodes into the table (pointed to by) \l__cctab_internal_a_tl and save that table number in a stack before selecting the desired catcodes.

\cctab_begin:c

```

28848 \cs_new_protected:Npn \cctab_begin:N #1
28849 {
28850   \__cctab_chk_if_valid:NT #1
28851   {
28852     \seq_gpop:NnF \g__cctab_unused_seq \l__cctab_internal_a_tl
28853     { \__cctab_begin_aux: }
28854     \exp_args:Nx \__cctab_chk_group_begin:n
28855     { \__cctab_nesting_number:N \l__cctab_internal_a_tl }
28856     \seq_gpush:NV \g__cctab_stack_seq \l__cctab_internal_a_tl
28857     \exp_args:NV \__cctab_gset:n \l__cctab_internal_a_tl
28858     \__cctab_select:N #1
28859   }
28860 }
28861 \cs_generate_variant:Nn \cctab_begin:N { c }

```

(End definition for \cctab_begin:N. This function is documented on page 261.)

\cctab_end: Make sure a \cctab_begin:N was used some time earlier, get in \l__cctab_internal_a_tl the catcode table number/name in which the prevailing catcodes were stored, then restore these catcodes. The dynamic table is now unused hence stored in \g__cctab_unused_seq for recycling by later \cctab_begin:N.

```

28862 \cs_new_protected:Npn \cctab_end:
28863 {
28864   \seq_gpop:NNTF \g__cctab_stack_seq \l__cctab_internal_a_tl
28865   {
28866     \seq_gpush:NV \g__cctab_unused_seq \l__cctab_internal_a_tl
28867     \exp_args:Nx \__cctab_chk_group_end:n
28868     { \__cctab_nesting_number:N \l__cctab_internal_a_tl }

```



```

28869         \_cctab_select:N \l__cctab_internal_a_tl
28870     }
28871     { \msg_error:nn { cctab } { extra-end } }
28872 }

```

(End definition for `\cctab_end:`. This function is documented on page 262.)

```

\_cctab_chk_group_begin:n
\_cctab_chk_group_end:n

```

Catcode tables are not allowed to be intermixed with groups, so here we check that they are properly nested regarding TeX groups. `_cctab_chk_group_begin:n` stores the current group level in a stack, and locally defines a dummy control sequence `_cctab_group_<cctab-level>_chk:`.

`_cctab_chk_group_end:n` pops the stack, and compares the returned value with `\tex_currentgrouplevel:D`. If they differ, `\cctab_end:` is in a different grouping level than the matching `\cctab_begin:N`. If they are the same, both happened at the same level, however a group might have ended and another started between `\cctab_begin:N` and `\cctab_end:`:

```

\group_begin:
  \cctab_begin:N \c_document_cctab
\group_end:
\group_begin:
  \cctab_end:
\group_end:

```

In this case checking `\tex_currentgrouplevel:D` is not enough, so we locally define `_cctab_group_<cctab-level>_chk:`, and then check if it exist in `\cctab_end:`. If it doesn't, we know there was a group end where it shouldn't.

The `<cctab-level>` in the sentinel macro above cannot be replaced by the more convenient `\tex_currentgrouplevel:D` because with the latter we might be tricked. Suppose:

```

\group_begin:
  \cctab_begin:N \c_code_cctab % A
\group_end:
\group_begin:
  \cctab_begin:N \c_code_cctab % B
  \cctab_end: % C
  \cctab_end: % D
\group_end:

```

The line marked with A would start a `cctab` with a sentinel token named `_cctab_group_1_chk:`, which would disappear at the `\group_end:` that follows. But B would create the same sentinel token, since both are at the same group level. Line C would end the `cctab` from line B correctly, but so would line D because line B created the same sentinel token. Using `<cctab-level>` works correctly because it signals that certain `cctab` level was activated somewhere, but if it doesn't exist when the `\cctab_end:` is reached, we had a problem.

Unfortunately these tests only flag the wrong usage at the `\cctab_end:`, which might be far from the `\cctab_begin:N`. However it isn't possible to signal the wrong usage at the `\group_end:` without using `\tex_aftergroup:D`, which is unsafe in certain types of groups.

The three cases checked here just raise an error, and no recovery is attempted: usually interleaving groups and catcode tables will work predictably.

```

28873 \cs_new_protected:Npn \__cctab_chk_group_begin:n #1
28874 {
28875   \seq_gpush:Nx \g__cctab_group_seq
28876   { \int_use:N \tex_currentgrouplevel:D }
28877   \cs_set_eq:cN { __cctab_group_ #1 _chk: } \prg_do_nothing:
28878 }
28879 \cs_new_protected:Npn \__cctab_chk_group_end:n #1
28880 {
28881   \seq_gpop:NN \g__cctab_group_seq \l__cctab_internal_b_tl
28882   \bool_lazy_and:nnF
28883   {
28884     \int_compare_p:nNn
28885     { \tex_currentgrouplevel:D } = { \l__cctab_internal_b_tl }
28886   }
28887   { \cs_if_exist_p:c { __cctab_group_ #1 _chk: } }
28888   {
28889     \msg_error:nnx { cctab } { group-mismatch }
28890     {
28891       \int_sign:n
28892       { \tex_currentgrouplevel:D - \l__cctab_internal_b_tl }
28893     }
28894   }
28895   \cs_undefine:c { __cctab_group_ #1 _chk: }
28896 }

```

(End definition for __cctab_chk_group_begin:n and __cctab_chk_group_end:n.)

__cctab_nesting_number:N This macro returns the numeric index of the current catcode table. In LuaTeX this is just the argument, which is a count reference to a \catcodetable register. In other engines, the number is extracted from the cctab variable.

```

28897 \sys_if_engine luatex:TF
28898 { \cs_new:Npn \__cctab_nesting_number:N #1 {#1} }
28899 {
29000   \cs_new:Npn \__cctab_nesting_number:N #1
29001   {
29002     \exp_after:wN \exp_after:wN \exp_after:wN \__cctab_nesting_number:w
29003     \exp_after:wN \token_to_str:N #1
29004   }
29005   \use:x
29006   {
29007     \cs_new:Npn \exp_not:N \__cctab_nesting_number:w
29008     ##1 \tl_to_str:n { g__cctab_ } ##2 \tl_to_str:n { _cctab } {##2}
29009   }
29010 }

```

(End definition for __cctab_nesting_number:N and __cctab_nesting_number:w.)

Finally, install some code at the end of the TeX run to check that all \cctab_begin:N were ended by some \cctab_end:.

```

28911 \cs_if_exist:NT \hook_gput_code:nnn
28912 {
28913   \hook_gput_code:nnn { enddocument/end } { cctab }
28914   {
28915     \seq_if_empty:NF \g__cctab_stack_seq
28916     { \msg_error:nn { cctab } { missing-end } }

```

```

28917     }
28918 }

```

\cctab_item:Nn Evaluate the integer argument only once. In most engines the `cctab` variable only has 256 entries so we only look up the catcode for these entries, otherwise we use the current catcode. In particular, for out-of-range values we use whatever fall-back `\char_value_catcode:n`. In LuaTeX, we use the `tex.getcatcode` function.

```

28919 \cs_new:Npn \cctab_item:Nn #1#2
28920 { \exp_args:Nf \__cctab_item:nN { \int_eval:n {#2} } #1 }
28921 \sys_if_engine luatex:TF
28922 {
28923   \cs_new:Npn \__cctab_item:nN #1#2
28924     { \lua_now:e { tex.print(-2, tex.getcatcode(\int_use:N #2, #1)) } }
28925 }
28926 {
28927   \cs_new:Npn \__cctab_item:nN #1#2
28928     {
28929       \int_compare:nNnTF {#1} < { 256 }
28930       { \intarray_item:Nn #2 { #1 + 1 } }
28931       { \char_value_catcode:n {#1} }
28932     }
28933 }
28934 \cs_generate_variant:Nn \cctab_item:Nn { c }

```

(End definition for `\cctab_item:Nn`. This function is documented on page 262.)

79.5 Category code table conditionals

\cctab_if_exist_p:N Checks whether a $\langle cctab\ var \rangle$ is defined.

```

\cctab_if_exist_p:c 28935 \prg_new_eq_conditional:NNn \cctab_if_exist:N \cs_if_exist:N
\cctab_if_exist:NTF 28936 { TF , T , F , p }
\cctab_if_exist:cTF 28937 \prg_new_eq_conditional:NNn \cctab_if_exist:c \cs_if_exist:c
28938 { TF , T , F , p }

```

(End definition for `\cctab_if_exist:NTF`. This function is documented on page 262.)

__cctab_chk_if_valid:N Checks whether the argument is defined and whether it is a valid $\langle cctab\ var \rangle$. In LuaTeX the validity of the $\langle cctab\ var \rangle$ is checked by the engine, which complains if the argument is not a `\chardef`'ed constant. In other engines, check if the given command is an intarray variable (the underlying definition is a copy of the `cmr10` font).

```

28939 \prg_new_protected_conditional:Npnn \__cctab_chk_if_valid:N #1
28940 { TF , T , F }
28941 {
28942   \cctab_if_exist:NTF #1
28943   {
28944     \__cctab_chk_if_valid_aux:NTF #1
28945     { \prg_return_true: }
28946     {
28947       \msg_error:nnx { cctab } { invalid-cctab }
28948       { \token_to_str:N #1 }
28949       \prg_return_false:
28950     }
28951   }

```

```

28952     {
28953         \msg_error:nxx { kernel } { command-not-defined }
28954         { \token_to_str:N #1 }
28955         \prg_return_false:
28956     }
28957 }
28958 \sys_if_engine luatex:TF
28959 {
28960     \cs_new_protected:Npn \__cctab_chk_if_valid_aux:NTF #1
28961     {
28962         \int_compare:nNnTF {#1-1} < { \e@alloc@ccodetable@count }
28963     }
28964     \cs_if_exist:NT \c_syst_catcodes_n
28965     {
28966         \cs_gset_protected:Npn \__cctab_chk_if_valid_aux:NTF #1
28967         {
28968             \int_compare:nTF { #1 <= \c_syst_catcodes_n }
28969         }
28970     }
28971 }
28972 {
28973     \cs_new_protected:Npn \__cctab_chk_if_valid_aux:NTF #1
28974     {
28975         \exp_args:Nf \str_if_in:nnTF
28976         { \cs_meaning:N #1 }
28977         { select~font~cmr10~at~ }
28978     }
28979 }

```

(End definition for __cctab_chk_if_valid:NTF and __cctab_chk_if_valid_aux:NTF.)

79.6 Constant category code tables

\cctab_const:Nn Creates a new $\langle cctab\ var \rangle$ then sets it with the current and user-supplied codes.

```

\cctab_const:cn 28980 \cs_new_protected:Npn \cctab_const:Nn #1#2
28981 {
28982     \cctab_new:N #1
28983     \cctab_gset:Nn #1 {#2}
28984 }
28985 \cs_generate_variant:Nn \cctab_const:Nn { c }

```

(End definition for \cctab_const:Nn. This function is documented on page 261.)

\c_initex_cctab Creating category code tables means thinking starting from $\text{ini}\text{T}_{\text{E}}\text{X}$. For all-other and
\c_other_cctab the standard “string” tables that’s easy.

```

\c_str_cctab 28986 \cctab_new:N \c_initex_cctab
28987 \cctab_const:Nn \c_other_cctab
28988 {
28989     \cctab_select:N \c_initex_cctab
28990     \int_set:Nn \tex_endlinechar:D { -1 }
28991     \int_step_inline:nnn { 0 } { 127 }
28992     { \char_set_catcode_other:n {#1} }
28993 }

```

```

28994 \cctab_const:Nn \c_str_cctab
28995 {
28996   \cctab_select:N \c_other_cctab
28997   \char_set_catcode_space:n { 32 }
28998 }

```

(End definition for `\c_initex_cctab`, `\c_other_cctab`, and `\c_str_cctab`. These variables are documented on page 262.)

`\c_code_cctab`
`\c_document_cctab`

To pick up document-level category codes, we need to delay set up to the end of the format, where that's possible. Also, as there are a *lot* of category codes to set, we avoid using the official interface and store the document codes using internal code. Depending on whether we are in the hook or not, the catcodes may be code or document, so we explicitly set up both correctly.

```

28999 \cs_if_exist:NTF \@expl@finalise@setup@@
29000 { \tl_gput_right:Nn \@expl@finalise@setup@@ }
29001 { \use:n }
29002 {
29003   \__cctab_new:N \c_code_cctab
29004   \group_begin:
29005     \int_set:Nn \tex_endlinechar:D { 32 }
29006     \char_set_catcode_invalid:n { 0 }
29007     \bool_lazy_or:nnTF
29008       { \sys_if_engine_xetex_p: } { \sys_if_engine_luatex_p: }
29009       { \int_step_function:nnN { 31 } { 64 } \char_set_catcode_invalid:n }
29010       { \int_step_function:nnN { 31 } { 64 } \char_set_catcode_active:n }
29011     \int_step_function:nnN { 33 } { 64 } \char_set_catcode_other:n
29012     \int_step_function:nnN { 65 } { 90 } \char_set_catcode_letter:n
29013     \int_step_function:nnN { 91 } { 96 } \char_set_catcode_other:n
29014     \int_step_function:nnN { 97 } { 122 } \char_set_catcode_letter:n
29015     \char_set_catcode_ignore:n { 9 } % tab
29016     \char_set_catcode_other:n { 10 } % lf
29017     \char_set_catcode_active:n { 12 } % ff
29018     \char_set_catcode_end_line:n { 13 } % cr
29019     \char_set_catcode_ignore:n { 32 } % space
29020     \char_set_catcode_parameter:n { 35 } % hash
29021     \char_set_catcode_math_toggle:n { 36 } % dollar
29022     \char_set_catcode_comment:n { 37 } % percent
29023     \char_set_catcode_alignment:n { 38 } % ampersand
29024     \char_set_catcode_letter:n { 58 } % colon
29025     \char_set_catcode_escape:n { 92 } % backslash
29026     \char_set_catcode_math_superscript:n { 94 } % circumflex
29027     \char_set_catcode_letter:n { 95 } % underscore
29028     \char_set_catcode_group_begin:n { 123 } % left brace
29029     \char_set_catcode_other:n { 124 } % pipe
29030     \char_set_catcode_group_end:n { 125 } % right brace
29031     \char_set_catcode_space:n { 126 } % tilde
29032     \char_set_catcode_invalid:n { 127 } % ^^?
29033     \bool_lazy_or:nnF
29034       { \sys_if_engine_xetex_p: } { \sys_if_engine_luatex_p: }
29035       { \int_step_function:nnN { 128 } { 255 } \char_set_catcode_active:n }
29036     \__cctab_gset:n { \c_code_cctab }
29037   \group_end:
29038   \cctab_const:Nn \c_document_cctab

```

```

29039     {
29040         \cctab_select:N \c_code_cctab
29041         \int_set:Nn \tex_endlinechar:D { 13 }
29042         \char_set_catcode_space:n { 9 }
29043         \char_set_catcode_space:n { 32 }
29044         \char_set_catcode_other:n { 58 }
29045         \char_set_catcode_math_subscript:n { 95 }
29046         \char_set_catcode_active:n { 126 }
29047     }
29048 }

```

(End definition for `\c_code_cctab` and `\c_document_cctab`. These variables are documented on page 262.)

79.7 Messages

```

29049 \msg_new:nnnn { cctab } { stack-full }
29050 { The-category-code-table-stack-is-exhausted. }
29051 {
29052     LaTeX-has-been-asked-to-switch-to-a-new-category-code-table,~
29053     but-there-is-no-more-space-to-do-this!
29054 }
29055 \msg_new:nnnn { cctab } { extra-end }
29056 { Extra~\iow_char:N\\cctab_end:~ignored~\msg_line_context:. }
29057 {
29058     LaTeX-came-across-a~\iow_char:N\\cctab_end:~without-a-matching~
29059     \iow_char:N\\cctab_begin:N.~This-command-will-be-ignored.
29060 }
29061 \msg_new:nnnn { cctab } { missing-end }
29062 { Missing~\iow_char:N\\cctab_end:~before-end-of-TeX-run. }
29063 {
29064     LaTeX-came-across-more~\iow_char:N\\cctab_begin:N~than~
29065     \iow_char:N\\cctab_end:.
29066 }
29067 \msg_new:nnnn { cctab } { invalid-cctab }
29068 { Invalid~\iow_char:N\\catcode-table. }
29069 {
29070     You-can-only-switch-to-a~\iow_char:N\\catcode-table-that-is~
29071     initialized-using~\iow_char:N\\cctab_new:N~or~
29072     \iow_char:N\\cctab_const:Nn.
29073 }
29074 \msg_new:nnnn { cctab } { group-mismatch }
29075 {
29076     \iow_char:N\\cctab_end:~occurred-in-a~
29077     \int_case:nn {#1}
29078     {
29079         { 0 } { different-group }
29080         { 1 } { higher-group-level }
29081         { -1 } { lower-group-level }
29082     } ~than~
29083     the-matching~\iow_char:N\\cctab_begin:N.
29084 }
29085 {
29086     Catcode-tables~and~groups~must~be~properly~nested,~but~

```

```

29087     you~tried~to~interleave~them.~LaTeX~will~try~to~proceed,~
29088     but~results~may~be~unexpected.
29089   }
29090   \prop_gput:Nnn \g_msg_module_name_prop { cctab } { LaTeX3 }
29091   \prop_gput:Nnn \g_msg_module_type_prop { cctab } { }
29092 \end{package}

```

Chapter 80

Unicode implementation

```
29093 <*package>
```

```
29094 <@@=char>
```

Case changing both for strings and “text” requires data from the Unicode Consortium. Some of this is build in to the format (as `\lccode` and `\uccode` values) but this covers only the simple one-to-one situations and does not fully handle for example case folding.

As only the data needs to remain at the end of this process, everything is set up inside a group. The only thing that is outside is creating a stream: they are global anyway and it is best to force a stream for all engines. For performance reasons, some of the code here is very low-level: the material is read during loading `expl3` in package mode.

```
29095 \ior_new:N \g__char_data_ior
```

```
29096 \bool_lazy_or:nnTF { \sys_if_engine_luatex_p: } { \sys_if_engine_xetex_p: }
```

```
29097 {
```

```
29098   \group_begin:
```

Access the primitive but suppress further expansion: active chars are otherwise an issue.

```
29099   \cs_set:Npn \__char_generate_char:n #1
```

```
29100     { \tex_detokenize:D \tex_expandafter:D { \tex_Uchar:D " #1 } }
```

A fast local implementation for generating characters; the chars may be active, so we prevent further expansion.

```
29101   \cs_set:Npx \__char_generate:n #1
```

```
29102   {
```

```
29103     \exp_not:N \tex_unexpanded:D \exp_not:N \exp_after:wN
```

```
29104     {
```

```
29105       \exp_not:N \tex_Ucharcat:D
```

```
29106       #1 ~
```

```
29107       \tex_catcode:D #1 ~
```

```
29108     }
```

```
29109   }
```

Parse the main Unicode data file for two things. First, we want the titlecase exceptions: the one-to-one lower- and uppercase mappings it contains are all be covered by the `TeX` data. Second, we need normalization data: at present, just the canonical NFD mappings. Those all yield either one or two codepoints, so the split is relatively easy.

```
29110   \ior_open:Nn \g__char_data_ior { UnicodeData.txt }
```

```
29111   \cs_set_protected:Npn \__char_data_auxi:w
```



```

29112     #1 ; #2 ; #3 ; #4 ; #5 ; #6 ; #7 ; #8 ; #9 ;
29113     {
29114         \tl_if_blank:nF {#6}
29115         {
29116             \tl_if_head_eq_charcode:nNF {#6} < % >
29117             { \__char_data_auxii:w #1 ; #6 ~ \q_stop }
29118         }
29119         \__char_data_auxiii:w #1 ;
29120     }
29121 \cs_set_protected:Npn \__char_data_auxii:w #1 ; #2 ~ #3 \q_stop
29122 {
29123     \tl_const:cx
29124     { c__char_nfd_ \__char_generate_char:n {#1} _tl }
29125     {
29126         \__char_generate:n { "#2 }
29127         \tl_if_blank:nF {#3}
29128         { \__char_generate:n { "#3 } }
29129     }
29130 }
29131 \cs_set_protected:Npn \__char_data_auxiii:w
29132 #1 ; #2 ; #3 ; #4 ; #5 ; #6 ; #7 ~ \q_stop
29133 {
29134     \cs_set_nopar:Npn \l__char_tmpa_tl {#7}
29135     \reverse_if:N \if_meaning:w \l__char_tmpa_tl \c_empty_tl
29136     \cs_set_nopar:Npn \l__char_tmpb_tl {#5}
29137     \reverse_if:N \if_meaning:w \l__char_tmpa_tl \l__char_tmpb_tl
29138     \tl_const:cx
29139     { c__char_titlecase_ \__char_generate_char:n {#1} _tl }
29140     { \__char_generate:n { "#7 } }
29141     \fi:
29142     \fi:
29143 }
29144 \group_begin:
29145     \char_set_catcode_space:n { '\ }%
29146     \ior_map_variable:NNn \g__char_data_ior \l__char_tmpa_tl
29147     {%
29148         \if_meaning:w \l__char_tmpa_tl \c_space_tl
29149         \exp_after:wN \ior_map_break:
29150         \fi:
29151         \exp_after:wN \__char_data_auxi:w \l__char_tmpa_tl \q_stop
29152     }%
29153 \group_end:
29154 \ior_close:N \g__char_data_ior

```

The other data files all use C-style comments so we have to worry about # tokens (and reading as strings). The set up for case folding is in two parts. For the basic (core) mappings, folding is the same as lower casing in most positions so only store the differences. For the more complex foldings, always store the result, splitting up the two or three code points in the input as required.

```

29155     \ior_open:Nn \g__char_data_ior { CaseFolding.txt }
29156 \cs_set_protected:Npn \__char_data_auxi:w #1 ; ~ #2 ; ~ #3 ; #4 \q_stop
29157 {
29158     \if:w \tl_head:n { #2 ? } C
29159     \reverse_if:N \if_int_compare:w

```

```

29160         \char_value_lccode:n {"#1} = "#3 ~
29161         \tl_const:cx
29162         { c__char_foldcase_ \__char_generate_char:n {#1} _tl }
29163         { \__char_generate:n { "#3 } }
29164     \fi:
29165 \else:
29166     \if:w \tl_head:n { #2 ? } F
29167     \__char_data_auxii:w #1 ~ #3 ~ \q_stop
29168 \fi:
29169 \fi:
29170 }
29171 \cs_set_protected:Npn \__char_data_auxii:w #1 ~ #2 ~ #3 ~ #4 \q_stop
29172 {
29173     \tl_const:cx { c__char_foldcase_ \__char_generate_char:n {#1} _tl }
29174     {
29175         \__char_generate:n { "#2 }
29176         \__char_generate:n { "#3 }
29177         \tl_if_blank:nF {#4}
29178         { \__char_generate:n { \int_value:w "#4 } }
29179     }
29180 }
29181 \ior_str_map_inline:Nn \g__char_data_ior
29182 {
29183     \reverse_if:N \if:w \c_hash_str \tl_head:w #1 \c_hash_str \q_stop
29184     \__char_data_auxi:w #1 \q_stop
29185 \fi:
29186 }
29187 \ior_close:N \g__char_data_ior

```

For upper- and lowercasing special situations, there is a bit more to do as we also have title casing to consider, plus we need to stop part-way through the file.

```

29188 \ior_open:Nn \g__char_data_ior { SpecialCasing.txt }
29189 \cs_set_protected:Npn \__char_data_auxi:w
29190 #1 ;~ #2 ;~ #3 ;~ #4 ; #5 \q_stop
29191 {
29192     \use:n { \__char_data_auxii:w #1 ~ lower ~ #2 ~ } ~ \q_stop
29193     \use:n { \__char_data_auxii:w #1 ~ upper ~ #4 ~ } ~ \q_stop
29194     \str_if_eq:nnF {#3} {#4}
29195     { \use:n { \__char_data_auxii:w #1 ~ title ~ #3 ~ } ~ \q_stop }
29196 }
29197 \cs_set_protected:Npn \__char_data_auxii:w
29198 #1 ~ #2 ~ #3 ~ #4 ~ #5 \q_stop
29199 {
29200     \tl_if_empty:nF {#4}
29201     {
29202         \tl_const:cx { c__char_ #2 case_ \__char_generate_char:n {#1} _tl }
29203         {
29204             \__char_generate:n { "#3 }
29205             \__char_generate:n { "#4 }
29206             \tl_if_blank:nF {#5}
29207             { \__char_generate:n { "#5 } }
29208         }
29209     }
29210 }
29211 \ior_str_map_inline:Nn \g__char_data_ior

```

```

29212 {
29213   \str_if_eq:eeTF
29214   { \tl_head:w #1 \c_hash_str \q_stop }
29215   { \c_hash_str }
29216   {
29217     \str_if_eq:eeT
29218     {#1}
29219     { \c_hash_str \c_space_tl Conditional-Mappings }
29220     { \ior_map_break: }
29221   }
29222   { \__char_data_auxi:w #1 \q_stop }
29223 }
29224 \ior_close:N \g__char_data_ior
29225 \group_end:
29226 }

```

For the 8-bit engines, the above is skipped but there is still some set up required. As case changing can only be applied to bytes, and they have to be in the ASCII range, we define a series of data stores to represent them, and the data are used such that only these are ever case-changed. We do open and close one file to force allocation of a read: this keeps all engines in line.

```

29227 {
29228   \group_begin:
29229   \cs_set_protected:Npn \__char_tmp:NN #1#2
29230   {
29231     \quark_if_recursion_tail_stop:N #2
29232     \tl_const:cn { c__char_uppercase_ #2 _tl } {#1}
29233     \tl_const:cn { c__char_lowercase_ #1 _tl } {#2}
29234     \tl_const:cn { c__char_foldcase_ #1 _tl } {#2}
29235     \__char_tmp:NN
29236   }
29237   \__char_tmp:NN
29238   AaBbCcDdEeFfGgHhIiJjKkLlMmNnOoPpQqRrSsTtUuVvWwXxYyZz
29239   ? \q_recursion_tail \q_recursion_stop
29240   \ior_open:Nn \g__char_data_ior { UnicodeData.txt }
29241   \ior_close:N \g__char_data_ior
29242   \group_end:
29243 }
29244 \</package>

```

Chapter 81

l3text implementation

29245 `*package`

29246 `\@@=text`

81.1 Internal auxiliaries

`\s__text_stop` Internal scan marks.

29247 `\scan_new:N \s__text_stop`

(End definition for \s__text_stop.)

`\q__text_nil` Internal quarks.

29248 `\quark_new:N \q__text_nil`

(End definition for \q__text_nil.)

`__text_quark_if_nil_p:n` Branching quark conditional.

`__text_quark_if_nil:nTF` 29249 `__kernel_quark_new_conditional:Nn __text_quark_if_nil:n { TF }`

(End definition for __text_quark_if_nil:nTF.)

`\q__text_recursion_tail` Internal recursion quarks.

`\q__text_recursion_stop` 29250 `\quark_new:N \q__text_recursion_tail`

29251 `\quark_new:N \q__text_recursion_stop`

(End definition for \q__text_recursion_tail and \q__text_recursion_stop.)

`__text_use_i_delimit_by_q_recursion_stop:nw` Functions to gobble up to a quark.

29252 `\cs_new:Npn __text_use_i_delimit_by_q_recursion_stop:nw`

29253 `#1 #2 \q__text_recursion_stop {#1}`

(End definition for __text_use_i_delimit_by_q_recursion_stop:nw.)

`__text_if_recursion_tail_stop_do:Nn` Functions to query recursion quarks.

29254 `__kernel_quark_new_test:N __text_if_recursion_tail_stop_do:Nn`

(End definition for __text_if_recursion_tail_stop_do:Nn.)

81.2 Utilities

The idea here is to take a token and ensure that if it's an implicit char, we output the explicit version. Otherwise, the token needs to be unchanged. First, we have to split between control sequences and everything else.

```

\__text_token_to_explicit:N
  \__text_token_to_explicit_char:N
  \__text_token_to_explicit_cs:N
  \__text_token_to_explicit_cs_aux:N
\__text_token_to_explicit:n
  \__text_token_to_explicit_auxi:w
  \__text_token_to_explicit_auxii:w
  \__text_token_to_explicit_auxiii:w
29255 \group_begin:
29256   \char_set_catcode_active:n { 0 }
29257   \cs_new:Npn \__text_token_to_explicit:N #1
29258     {
29259       \if_catcode:w \exp_not:N #1
29260         \if_catcode:w \scan_stop: \exp_not:N #1
29261         \scan_stop:
29262       \else:
29263         \exp_not:N ^^@
29264       \fi:
29265       \exp_after:wN \__text_token_to_explicit_cs:N
29266     \else:
29267       \exp_after:wN \__text_token_to_explicit_char:N
29268     \fi:
29269     #1
29270   }
29271 \group_end:

```

For control sequences, we can check for macros versus other cases using `\if_meaning:w`, then explicitly check for `\chardef` and `\mathchardef`.

```

29272 \cs_new:Npn \__text_token_to_explicit_cs:N #1
29273   {
29274     \exp_after:wN \if_meaning:w \exp_not:N #1 #1
29275     \exp_after:wN \use:nn \exp_after:wN
29276     \__text_token_to_explicit_cs_aux:N
29277   \else:
29278     \exp_after:wN \exp_not:n
29279   \fi:
29280   {#1}
29281 }
29282 \cs_new:Npn \__text_token_to_explicit_cs_aux:N #1
29283   {
29284     \bool_lazy_or:nnTF
29285     { \token_if_chardef_p:N #1 }
29286     { \token_if_mathchardef_p:N #1 }
29287     {
29288       \char_generate:nn {#1}
29289       { \char_value_catcode:n {#1} }
29290     }
29291     {#1}
29292   }

```

For character tokens, we need to filter out the implicit characters from those that are explicit. That's done here, then if necessary we work out the category code and generate the char. To avoid issues with alignment tabs, that one is done by elimination rather than looking up the code explicitly. The trick with finding the charcode is that the TeX messages are either the *<something>* character *<char>* or the *<type>* *<char>*.

```

29293 \cs_new:Npn \__text_token_to_explicit_char:N #1
29294   {

```

```

29295 \if:w
29296 \if_catcode:w ^ \exp_args:No \str_tail:n { \token_to_str:N #1 } ^
29297 \token_to_str:N #1 #1
29298 \else:
29299 AB
29300 \fi:
29301 \exp_after:wN \exp_not:n
29302 \else:
29303 \exp_after:wN \__text_token_to_explicit:n
29304 \fi:
29305 {#1}
29306 }
29307 \cs_new:Npn \__text_token_to_explicit:n #1
29308 {
29309 \exp_after:wN \__text_token_to_explicit_auxi:w
29310 \int_value:w
29311 \if_catcode:w \c_group_begin_token #1 1 \else:
29312 \if_catcode:w \c_group_end_token #1 2 \else:
29313 \if_catcode:w \c_math_toggle_token #1 3 \else:
29314 \if_catcode:w ## #1 6 \else:
29315 \if_catcode:w ^ #1 7 \else:
29316 \if_catcode:w \c_math_subscript_token #1 8 \else:
29317 \if_catcode:w \c_space_token #1 10 \else:
29318 \if_catcode:w A #1 11 \else:
29319 \if_catcode:w + #1 12 \else:
29320 4 \fi: \fi: \fi: \fi: \fi: \fi: \fi:
29321 \exp_after:wN ;
29322 \token_to_meaning:N #1 \s__text_stop
29323 }
29324 \cs_new:Npn \__text_token_to_explicit_auxi:w #1 ; #2 \s__text_stop
29325 {
29326 \char_generate:nn
29327 {
29328 \if_int_compare:w #1 < 9 \exp_stop_f:
29329 \exp_after:wN \__text_token_to_explicit_auxii:w
29330 \else:
29331 \exp_after:wN \__text_token_to_explicit_auxiii:w
29332 \fi:
29333 #2
29334 }
29335 {#1}
29336 }
29337 \exp_last_unbraced:NNNNo \cs_new:Npn \__text_token_to_explicit_auxii:w
29338 #1 { \tl_to_str:n { character ~ } } { ' }
29339 \cs_new:Npn \__text_token_to_explicit_auxiii:w #1 ~ #2 ~ { ' }

```

(End definition for `__text_token_to_explicit:N` and others.)

`__text_char_catcode:N` An idea from `l3char`: we need to get the category code of a specific token, not the general case.

```

29340 \cs_new:Npn \__text_char_catcode:N #1
29341 {
29342 \if_catcode:w \exp_not:N #1 \c_math_toggle_token
29343 3

```

```

29344 \else:
29345 \if_catcode:w \exp_not:N #1 \c_alignment_token
29346 4
29347 \else:
29348 \if_catcode:w \exp_not:N #1 \c_math_superscript_token
29349 7
29350 \else:
29351 \if_catcode:w \exp_not:N #1 \c_math_subscript_token
29352 8
29353 \else:
29354 \if_catcode:w \exp_not:N #1 \c_space_token
29355 10
29356 \else:
29357 \if_catcode:w \exp_not:N #1 \c_catcode_letter_token
29358 11
29359 \else:
29360 \if_catcode:w \exp_not:N #1 \c_catcode_other_token
29361 12
29362 \else:
29363 13
29364 \fi:
29365 \fi:
29366 \fi:
29367 \fi:
29368 \fi:
29369 \fi:
29370 \fi:
29371 }

```

(End definition for `_text_char_catcode:N`.)

`_text_if_expandable:NTF` Test for tokens that make sense to expand here: that is more restrictive than the engine view.

```

29372 \prg_new_conditional:Npnn \_text_if_expandable:N #1 { T , F , TF }
29373 {
29374 \token_if_expandable:NTF #1
29375 {
29376 \bool_lazy_any:nTF
29377 {
29378 { \token_if_protected_macro_p:N #1 }
29379 { \token_if_protected_long_macro_p:N #1 }
29380 { \token_if_eq_meaning_p:NN \q_text_recursion_tail #1 }
29381 }
29382 { \prg_return_false: }
29383 { \prg_return_true: }
29384 }
29385 { \prg_return_false: }
29386 }

```

(End definition for `_text_if_expandable:NTF`.)

81.3 Configuration variables

`\l_text_accents_tl` `\l_text_letterlike_tl` Special cases for accents and letter-like symbols, which in some cases will need to be converted further.

```

29387 \tl_new:N \l_text_accents_tl
29388 \tl_set:Nn \l_text_accents_tl
29389   { \‘ \’ \^ \~ \= \u \. \" \r \H \v \d \c \k \b \t }
29390 \tl_new:N \l_text_letterlike_tl
29391 \tl_set:Nn \l_text_letterlike_tl
29392   {
29393     \AA \aa
29394     \AE \ae
29395     \DH \dh
29396     \DJ \dj
29397     \IJ \ij
29398     \L \l
29399     \NG \ng
29400     \O \o
29401     \OE \oe
29402     \SS \ss
29403     \TH \th
29404   }

```

(End definition for `\l_text_accents_tl` and `\l_text_letterlike_tl`. These variables are documented on page 267.)

`\l_text_case_exclude_arg_tl` Non-text arguments.

```

29405 \tl_new:N \l_text_case_exclude_arg_tl
29406 \tl_set:Nn \l_text_case_exclude_arg_tl { \begin \cite \end \label \ref }

```

(End definition for `\l_text_case_exclude_arg_tl`. This variable is documented on page 268.)

`\l_text_math_arg_tl` Math mode as arguments.

```

29407 \tl_new:N \l_text_math_arg_tl
29408 \tl_set:Nn \l_text_math_arg_tl { \ensuremath }

```

(End definition for `\l_text_math_arg_tl`. This variable is documented on page 268.)

`\l_text_math_delims_tl` Paired math mode delimiters.

```

29409 \tl_new:N \l_text_math_delims_tl
29410 \tl_set:Nn \l_text_math_delims_tl { $ $ \(\ \) }

```

(End definition for `\l_text_math_delims_tl`. This variable is documented on page 268.)

`\l_text_expand_exclude_tl` Commands which need not to expand.

```

29411 \tl_new:N \l_text_expand_exclude_tl
29412 \tl_set:Nn \l_text_expand_exclude_tl
29413   { \begin \cite \end \label \ref }

```

(End definition for `\l_text_expand_exclude_tl`. This variable is documented on page 268.)

`\l__text_math_mode_tl` Used to control math mode output: internal as there is a dedicated setter.

```

29414 \tl_new:N \l__text_math_mode_tl

```

(End definition for `\l__text_math_mode_tl`.)

81.4 Expansion to formatted text

Markers for implicit char handling.

```

29415 \tex_chardef:D \c__text_chardef_space_token = '\ %
29416 \tex_mathchardef:D \c__text_mathchardef_space_token = '\ %
29417 \tex_chardef:D \c__text_chardef_group_begin_token = '\{ % '\}
29418 \tex_mathchardef:D \c__text_mathchardef_group_begin_token = '\{ % '\} '\{
29419 \tex_chardef:D \c__text_chardef_group_end_token = '\} % '\{
29420 \tex_mathchardef:D \c__text_mathchardef_group_end_token = '\} %

```

(End definition for `\c__text_chardef_space_token` and others.)

After precautions against `&` tokens, start a simple loop: that of course means that “text” cannot contain the two recursion quarks. The loop here must be f-type expandable; we have arbitrary user commands which might be protected *and* take arguments, and if the expansion code is used in a typesetting context, that will otherwise explode. (The same issue applies more clearly to case changing: see the example there.)

```

\text_expand:n
  \__text_expand:n
    \__text_expand_result:n
    \__text_expand_store:n
    \__text_expand_store:o
    \__text_expand_store:nw
    \__text_expand_end:w
    \__text_expand_loop:w
    \__text_expand_group:n
    \__text_expand_space:w
    \__text_expand_N_type:N
\__text_expand_N_type_auxi:N
  \__text_expand_N_type_auxii:N
  \__text_expand_N_type_auxiii:N
  \__text_expand_math_search:NNN
\__text_expand_math_loop:Nw
  \__text_expand_math_N_type:NN
\__text_expand_math_group:Nn
\__text_expand_math_space:Nw
  \__text_expand_implicit:N
  \__text_expand_explicit:N
  \__text_expand_exclude:N
  \__text_expand_exclude:nN
  \__text_expand_exclude:NN
  \__text_expand_exclude:Nw
  \__text_expand_exclude:Nnn
  \__text_expand_accent:N
  \__text_expand_accent:NN
  \__text_expand_letterlike:N
  \__text_expand_letterlike:NN
  \__text_expand_cs:N
  \__text_expand_encoding:N
  \__text_expand_encoding_escape:N
  \__text_expand_protect:N
  \__text_expand_protect:nN
  \__text_expand_protect:Nw
  \__text_expand_testopt:N
  \__text_expand_testopt:NNn
  \__text_expand_replace:N
  \__text_expand_replace:n
  \__text_expand_cs_expand:N
  \__text_expand_unexpanded:w
  \__text_expand_unexpanded_test:w
  \__text_expand_unexpanded:N
  \__text_expand_unexpanded:n

```

```

29421 \cs_new:Npn \text_expand:n #1
29422 {
29423   \__kernel_exp_not:w \exp_after:wN
29424   {
29425     \exp:w
29426     \__text_expand:n {#1}
29427   }
29428 }
29429 \cs_new:Npn \__text_expand:n #1
29430 {
29431   \group_align_safe_begin:
29432   \__text_expand_loop:w #1
29433   \q__text_recursion_tail \q__text_recursion_stop
29434   \__text_expand_result:n { }
29435 }

```

```

29436 \cs_new:Npn \__text_expand_store:n #1
29437 { \__text_expand_store:nw {#1} }
29438 \cs_generate_variant:Nn \__text_expand_store:n { o }
29439 \cs_new:Npn \__text_expand_store:nw #1#2 \__text_expand_result:n #3
29440 { #2 \__text_expand_result:n { #3 #1 } }
29441 \cs_new:Npn \__text_expand_end:w #1 \__text_expand_result:n #2
29442 {
29443   \group_align_safe_end:
29444   \exp_end:
29445   #2
29446 }

```

```

29447 \cs_new:Npn \__text_expand_loop:w #1 \q__text_recursion_stop
29448 {
29449   \tl_if_head_is_N_type:nTF {#1}
29450   { \__text_expand_N_type:N }
29451   {

```

The approach to making the code f-type expandable is to use a marker result token and to shuffle the collected tokens

The main loop is a standard “tl action”; groups are handled recursively, while spaces are just passed through. Thus all of the action is in handling N-type tokens.

```

29452         \tl_if_head_is_group:nTF {#1}
29453         { \__text_expand_group:n }
29454         { \__text_expand_space:w }
29455     }
29456     #1 \q__text_recursion_stop
29457 }
29458 \cs_new:Npn \__text_expand_group:n #1
29459 {
29460     \__text_expand_store:o
29461     {
29462         \exp_after:wN
29463         {
29464             \exp:w
29465             \__text_expand:n {#1}
29466         }
29467     }
29468     \__text_expand_loop:w
29469 }
29470 \exp_last_unbraced:NNo \cs_new:Npn \__text_expand_space:w \c_space_tl
29471 {
29472     \__text_expand_store:n { ~ }
29473     \__text_expand_loop:w
29474 }

```

Before we get into the real work, we have to watch out for problematic implicit characters: spaces and grouping tokens. Converting these to explicit characters later would lead to real issues as they are *not* N-type. A space is the easy case, so it's dealt with first: just insert the explicit token and continue the loop.

```

29475 \cs_new:Npx \__text_expand_N_type:N #1
29476 {
29477     \exp_not:N \__text_if_recursion_tail_stop_do:Nn #1
29478     { \exp_not:N \__text_expand_end:w }
29479     \exp_not:N \bool_lazy_any:nTF
29480     {
29481         { \exp_not:N \token_if_eq_meaning_p:NN #1 \c_space_token }
29482         {
29483             \exp_not:N \token_if_eq_meaning_p:NN #1
29484             \c__text_chardef_space_token
29485         }
29486         {
29487             \exp_not:N \token_if_eq_meaning_p:NN #1
29488             \c__text_mathchardef_space_token
29489         }
29490     }
29491     { \exp_not:N \__text_expand_space:w \c_space_tl }
29492     { \exp_not:N \__text_expand_N_type_auxi:N #1 }
29493 }

```

Implicit `{/}` offer two issues. First, the token could be an implicit brace character: we need to avoid turning that into a brace group, so filter out the cases manually. Then we handle the case where an implicit group is present. That is done in an “open-ended” way: there's the possibility the closing token is hidden somewhere.

```

29494 \cs_new:Npn \__text_expand_N_type_auxi:N #1
29495 {

```

```

29496 \bool_lazy_or:nnTF
29497 { \token_if_eq_meaning_p:NN #1 \c__text_chardef_group_begin_token }
29498 { \token_if_eq_meaning_p:NN #1 \c__text_mathchardef_group_begin_token }
29499 {
29500   \__text_expand_store:o \c_left_brace_str
29501   \__text_expand_loop:w
29502 }
29503 {
29504   \bool_lazy_or:nnTF
29505   { \token_if_eq_meaning_p:NN #1 \c__text_chardef_group_end_token }
29506   { \token_if_eq_meaning_p:NN #1 \c__text_mathchardef_group_end_token }
29507   {
29508     \__text_expand_store:o \c_right_brace_str
29509     \__text_expand_loop:w
29510   }
29511   { \__text_expand_N_type_auxii:N #1 }
29512 }
29513 }
29514 \cs_new:Npn \__text_expand_N_type_auxii:N #1
29515 {
29516   \token_if_eq_meaning:NNTF #1 \c_group_begin_token
29517   {
29518     { \if_false: } \fi:
29519     \__text_expand_loop:w
29520   }
29521   {
29522     \token_if_eq_meaning:NNTF #1 \c_group_end_token
29523     {
29524       \if_false: { \fi: }
29525       \__text_expand_loop:w
29526     }
29527     { \__text_expand_N_type_auxiii:N #1 }
29528   }
29529 }

```

The first step in dealing with N-type tokens is to look for math mode material: that needs to be left alone. The starting function has to be split into two as we need `\quark_if_recursion_tail_stop:N` first before we can trigger the search. We then look for matching pairs of delimiters, allowing for the case where math mode starts but does not end. Within math mode, we simply pass all the tokens through unchanged, just checking the N-type ones against the end marker.

```

29530 \cs_new:Npn \__text_expand_N_type_auxiii:N #1
29531 {
29532   \exp_after:wN \__text_expand_math_search:NNN
29533   \exp_after:wN #1 \l_text_math_delims_tl
29534   \q__text_recursion_tail \q__text_recursion_tail
29535   \q__text_recursion_stop
29536 }
29537 \cs_new:Npn \__text_expand_math_search:NNN #1#2#3
29538 {
29539   \__text_if_recursion_tail_stop_do:Nn #2
29540   { \__text_expand_explicit:N #1 }
29541   \token_if_eq_meaning:NNTF #1 #2
29542   {

```

```

29543     \_text_use_i_delimit_by_q_recursion_stop:nw
29544     {
29545         \_text_expand_store:n {#1}
29546         \_text_expand_math_loop:Nw #3
29547     }
29548 }
29549 { \_text_expand_math_search:NNN #1 }
29550 }
29551 \cs_new:Npn \_text_expand_math_loop:Nw #1#2 \q__text_recursion_stop
29552 {
29553     \tl_if_head_is_N_type:nTF {#2}
29554     { \_text_expand_math_N_type:NN }
29555     {
29556         \tl_if_head_is_group:nTF {#2}
29557         { \_text_expand_math_group:Nn }
29558         { \_text_expand_math_space:Nw }
29559     }
29560     #1#2 \q__text_recursion_stop
29561 }
29562 \cs_new:Npn \_text_expand_math_N_type:NN #1#2
29563 {
29564     \_text_if_recursion_tail_stop_do:Nn #2
29565     { \_text_expand_end:w }
29566     \_text_expand_store:n {#2}
29567     \token_if_eq_meaning:NNTF #2 #1
29568     { \_text_expand_loop:w }
29569     { \_text_expand_math_loop:Nw #1 }
29570 }
29571 \cs_new:Npn \_text_expand_math_group:Nn #1#2
29572 {
29573     \_text_expand_store:n { {#2} }
29574     \_text_expand_math_loop:Nw #1
29575 }
29576 \exp_after:wN \cs_new:Npn \exp_after:wN \_text_expand_math_space:Nw
29577 \exp_after:wN # \exp_after:wN 1 \c_space_tl
29578 {
29579     \_text_expand_store:n { ~ }
29580     \_text_expand_math_loop:Nw #1
29581 }

```

At this stage, either we have a control sequence or a simple character: split and handle.

```

29582 \cs_new:Npn \_text_expand_explicit:N #1
29583 {
29584     \token_if_cs:NNTF #1
29585     { \_text_expand_exclude:N #1 }
29586     {
29587         \_text_expand_store:n {#1}
29588         \_text_expand_loop:w
29589     }
29590 }

```

Next we exclude math commands: this is mainly as there *might* be an `\ensuremath`.

```

29591 \cs_new:Npn \_text_expand_exclude:N #1
29592 {
29593     \exp_args:Ne \_text_expand_exclude:nN

```

```

29594     {
29595         \exp_not:V \l_text_math_arg_tl
29596         \exp_not:V \l_text_expand_exclude_tl
29597     }
29598     #1
29599 }
29600 \cs_new:Npn \__text_expand_exclude:nN #1#2
29601 {
29602     \__text_expand_exclude:NN #2 #1
29603     \q__text_recursion_tail \q__text_recursion_stop
29604 }
29605 \cs_new:Npn \__text_expand_exclude:NN #1#2
29606 {
29607     \__text_if_recursion_tail_stop_do:Nn #2
29608     { \__text_expand_accent:N #1 }
29609     \str_if_eq:nnTF {#1} {#2}
29610     {
29611         \__text_use_i_delimit_by_q_recursion_stop:nw
29612         { \__text_expand_exclude:Nw #1 }
29613     }
29614     { \__text_expand_exclude:NN #1 }
29615 }
29616 \cs_new:Npn \__text_expand_exclude:Nw #1#2#
29617 { \__text_expand_exclude:Nnn #1 {#2} }
29618 \cs_new:Npn \__text_expand_exclude:Nnn #1#2#3
29619 {
29620     \__text_expand_store:n { #1#2 {#3} }
29621     \__text_expand_loop:w
29622 }

```

Accents.

```

29623 \cs_new:Npn \__text_expand_accent:N #1
29624 {
29625     \exp_after:wN \__text_expand_accent:NN \exp_after:wN
29626     #1 \l_text_accents_tl
29627     \q__text_recursion_tail \q__text_recursion_stop
29628 }
29629 \cs_new:Npn \__text_expand_accent:NN #1#2
29630 {
29631     \__text_if_recursion_tail_stop_do:Nn #2
29632     { \__text_expand_letterlike:N #1 }
29633     \cs_if_eq:NNTF #2 #1
29634     {
29635         \__text_use_i_delimit_by_q_recursion_stop:nw
29636         {
29637             \__text_expand_store:n {#1}
29638             \__text_expand_loop:w
29639         }
29640     }
29641     { \__text_expand_accent:NN #1 }
29642 }

```

Another list of exceptions: these ones take no arguments so are easier to handle.

```

29643 \cs_new:Npn \__text_expand_letterlike:N #1
29644 {

```

```

29645 \exp_after:wN \_text_expand_letterlike:NN \exp_after:wN
29646 #1 \l_text_letterlike_tl
29647 \q__text_recursion_tail \q__text_recursion_stop
29648 }
29649 \cs_new:Npn \_text_expand_letterlike:NN #1#2
29650 {
29651 \_text_if_recursion_tail_stop_do:Nn #2
29652 { \_text_expand_cs:N #1 }
29653 \cs_if_eq:NNTF #2 #1
29654 {
29655 \_text_use_i_delimit_by_q_recursion_stop:nw
29656 {
29657 \_text_expand_store:n {#1}
29658 \_text_expand_loop:w
29659 }
29660 }
29661 { \_text_expand_letterlike:NN #1 }
29662 }

```

LaTeX_{2 ϵ} 's `\protect` makes life interesting. Where possible, we simply remove it and replace with the “parent” command; of course, the `\protect` might be explicit, in which case we need to leave it alone if it's required. There is also the case of a straight `\@protected@testopt` to cover.

```

29663 \cs_new:Npx \_text_expand_cs:N #1
29664 {
29665 \exp_not:N \str_if_eq:nnTF {#1} { \exp_not:N \protect }
29666 { \exp_not:N \_text_expand_protect:N }
29667 {
29668 \bool_lazy_and:nnTF
29669 { \cs_if_exist_p:N \fmtname }
29670 { \str_if_eq_p:Vn \fmtname { LaTeX2e } }
29671 { \exp_not:N \_text_expand_testopt:N #1 }
29672 { \exp_not:N \_text_expand_replace:N #1 }
29673 }
29674 }
29675 \cs_new:Npn \_text_expand_protect:N #1
29676 {
29677 \exp_args:Ne \_text_expand_protect:nN
29678 { \cs_to_str:N #1 } #1
29679 }
29680 \cs_new:Npn \_text_expand_protect:nN #1#2
29681 { \_text_expand_protect:Nw #2 #1 \q__text_nil #1 ~ \q__text_nil \q__text_nil \s__text_stop
29682 \cs_new:Npn \_text_expand_protect:Nw #1 #2 ~ \q__text_nil #3 \q__text_nil #4 \s__text_stop
29683 {
29684 \_text_quark_if_nil:nTF {#4}
29685 {
29686 \cs_if_exist:cTF {#2}
29687 { \exp_args:Ne \_text_expand_store:n { \exp_not:c {#2} } }
29688 { \_text_expand_store:n { \protect #1 } }
29689 }
29690 { \_text_expand_store:n { \protect #1 } }
29691 \_text_expand_loop:w
29692 }
29693 \cs_new:Npn \_text_expand_testopt:N #1

```

```

29694 {
29695     \token_if_eq_meaning:NNTF #1 \@protected@testopt
29696     { \__text_expand_testopt:NNn }
29697     { \__text_expand_encoding:N #1 }
29698 }
29699 \cs_new:Npn \__text_expand_testopt:NNn #1#2#3
29700 {
29701     \__text_expand_store:n {#1}
29702     \__text_expand_loop:w
29703 }

```

Deal with encoding-specific commands

```

29704 \cs_new:Npn \__text_expand_encoding:N #1
29705 {
29706     \bool_lazy_or:nnTF
29707     { \cs_if_eq_p:NN #1 \@current@cmd }
29708     { \cs_if_eq_p:NN #1 \@changed@cmd }
29709     { \exp_after:wN \__text_expand_loop:w \__text_expand_encoding_escape:NN }
29710     { \__text_expand_replace:N #1 }
29711 }
29712 \cs_new:Npn \__text_expand_encoding_escape:NN #1#2 { \exp_not:n {#1} }

```

See if there is a dedicated replacement, and if there is, insert it.

```

29713 \cs_new:Npn \__text_expand_replace:N #1
29714 {
29715     \bool_lazy_and:nnTF
29716     { \cs_if_exist_p:c { l__text_expand_ \token_to_str:N #1 _tl } }
29717     {
29718         \bool_lazy_or_p:nn
29719         { \token_if_cs_p:N #1 }
29720         { \token_if_active_p:N #1 }
29721     }
29722     {
29723         \exp_args:Nv \__text_expand_replace:n
29724         { l__text_expand_ \token_to_str:N #1 _tl }
29725     }
29726     { \__text_expand_cs_expand:N #1 }
29727 }
29728 \cs_new:Npn \__text_expand_replace:n #1 { \__text_expand_loop:w #1 }

```

Finally, expand any macros which can be: this then loops back around to deal with what they produce. The only issue is if the token is `\exp_not:n`, as that must apply to the following balanced text.

```

29729 \cs_new:Npn \__text_expand_cs_expand:N #1
29730 {
29731     \__text_if_expandable:NNTF #1
29732     {
29733         \token_if_eq_meaning:NNTF #1 \exp_not:n
29734         { \__text_expand_unexpanded:w }
29735         { \exp_after:wN \__text_expand_loop:w #1 }
29736     }
29737     {
29738         \__text_expand_store:n {#1}
29739         \__text_expand_loop:w
29740     }

```

29741 }

Since `\exp_not:n` is actually a primitive, it allows a strange syntax and it particular the primitive expands what follows and discards spaces and `\scan_stop:` until finding a braced argument (the opening brace can be implicit but we will not support this here). Here, we repeatedly `f-expand` after such an `\exp_not:n`, and test what follows. If it is a brace group, then we found the intended argument of `\exp_not:n`. If it is a space, then the next `f-expansion` will eliminate it. If it is an N-type token then `__text_expand_unexpanded:N` leaves the token to be expanded if it is expandable, and otherwise removes it, assuming that it is `\scan_stop:`. This silently hides errors when `\exp_not:n` is incorrectly followed by some non-expandable token other than `\scan_stop:`, but this should be pretty rare, and there is no good error recovery anyways.

```
29742 \cs_new:Npn \__text_expand_unexpanded:w
29743 {
29744   \exp_after:wN \__text_expand_unexpanded_test:w
29745   \exp:w \exp_end_continue_f:w
29746 }
29747 \cs_new:Npn \__text_expand_unexpanded_test:w #1 \q__text_recursion_stop
29748 {
29749   \tl_if_head_is_group:nTF {#1}
29750   { \__text_expand_unexpanded:n }
29751   {
29752     \__text_expand_unexpanded:w
29753     \tl_if_head_is_N_type:nT {#1} { \__text_expand_unexpanded:N }
29754   }
29755   #1 \q__text_recursion_stop
29756 }
29757 \cs_new:Npn \__text_expand_unexpanded:N #1
29758 {
29759   \exp_after:wN \if_meaning:w \exp_not:N #1 #1
29760   \else:
29761     \exp_after:wN #1
29762   \fi:
29763 }
29764 \cs_new:Npn \__text_expand_unexpanded:n #1
29765 {
29766   \__text_expand_store:n {#1}
29767   \__text_expand_loop:w
29768 }
```

(End definition for `\text_expand:n` and others. This function is documented on page 265.)

`\text_declare_expand_equivalent:Nn` Create equivalents to allow replacement.

```
\text_declare_expand_equivalent:cn
29769 \cs_new_protected:Npn \text_declare_expand_equivalent:Nn #1#2
29770 {
29771   \tl_clear_new:c { l__text_expand_ \token_to_str:N #1 _tl }
29772   \tl_set:cn { l__text_expand_ \token_to_str:N #1 _tl } {#2}
29773 }
29774 \cs_generate_variant:Nn \text_declare_expand_equivalent:Nn { c }
```

(End definition for `\text_declare_expand_equivalent:Nn`. This function is documented on page 265.)

29775 `\package`

Chapter 82

text-case implementation

```
29776 <*package>
29777 <@@=text>
```

82.1 Case changing

```
\l_text_titlecase_check_letter_bool
\bool_new:N \l_text_titlecase_check_letter_bool
\bool_set_true:N \l_text_titlecase_check_letter_bool
```

(End definition for `\l_text_titlecase_check_letter_bool`. This variable is documented on page 268.)

```
\text_lowercase:n
\text_uppercase:n
\text_titlecase:n
\text_titlecase_first:n
\text_lowercase:nn
\text_uppercase:nn
\text_titlecase:nn
\text_titlecase_first:nn
\cs_new:Npn \text_lowercase:n #1
{ \__text_change_case:nnn { lower } { } {#1} }
\cs_new:Npn \text_uppercase:n #1
{ \__text_change_case:nnn { upper } { } {#1} }
\cs_new:Npn \text_titlecase:n #1
{ \__text_change_case:nnn { title } { } {#1} }
\cs_new:Npn \text_titlecase_first:n #1
{ \__text_change_case:nnn { titleonly } { } {#1} }
\cs_new:Npn \text_lowercase:nn #1#2
{ \__text_change_case:nnn { lower } {#1} {#2} }
\cs_new:Npn \text_uppercase:nn #1#2
{ \__text_change_case:nnn { upper } {#1} {#2} }
\cs_new:Npn \text_titlecase:nn #1#2
{ \__text_change_case:nnn { title } {#1} {#2} }
\cs_new:Npn \text_titlecase_first:nn #1#2
{ \__text_change_case:nnn { titleonly } {#1} {#2} }
```

(End definition for `\text_lowercase:n` and others. These functions are documented on page 266.)

```
\__text_change_case:nnn
\__text_change_case_aux:nnn
\__text_change_case_store:n
\__text_change_case_store:o
\__text_change_case_store:V
\__text_change_case_store:v
\__text_change_case_store:e
\__text_change_case_store:nw
\__text_change_case_result:n
\__text_change_case_end:w
\__text_change_case_loop:nnw
\__text_change_case_break:w
\__text_change_case_group_lower:nnn
\__text_change_case_group_upper:nnn
\__text_change_case_group_title:nnn
\__text_change_case_group_titleonly:nnn
```

As for the expansion code, the business end of case changing is the handling of N-type tokens. First, we expand the input fully (so the loops here don't need to worry about awkward look-aheads and the like). Then we split into the different paths.

The code here needs to be f-type expandable to deal with the situation where case changing is applied in running text. There, we might have case changing as a document command and the text containing other non-expandable document commands.

```

\cs_set_eq:NN \MakeLowercase \text_lowercase
...
\MakeLowercase{\enquote*{A} text}

```

If we use an e-type expansion and wrap each token in `\exp_not:n`, that would explode: the document command grabs `\exp_not:n` as an argument, and things go badly wrong. So we have to wrap the entire result in exactly one `\exp_not:n`, or rather in the kernel version.

```

29796 \cs_new:Npn \__text_change_case:nnn #1#2#3
29797 {
29798   \__kernel_exp_not:w \exp_after:wN
29799   {
29800     \exp:w
29801     \exp_args:Ne \__text_change_case_aux:nnn
29802     { \text_expand:n {#3} }
29803     {#1} {#2}
29804   }
29805 }
29806 \cs_new:Npn \__text_change_case_aux:nnn #1#2#3
29807 {
29808   \group_align_safe_begin:
29809   \cs_if_exist_use:c { __text_change_case_boundary_ #2 _ #3 :Nnnw }
29810   \__text_change_case_loop:nnw {#2} {#3} #1
29811   \q__text_recursion_tail \q__text_recursion_stop
29812   \__text_change_case_result:n { }
29813 }

```

As for expansion, collect up the tokens for future use.

```

29814 \cs_new:Npn \__text_change_case_store:n #1
29815 { \__text_change_case_store:nw {#1} }
29816 \cs_generate_variant:Nn \__text_change_case_store:n { o , e , V , v }
29817 \cs_new:Npn \__text_change_case_store:nw #1#2 \__text_change_case_result:n #3
29818 { #2 \__text_change_case_result:n { #3 #1 } }
29819 \cs_new:Npn \__text_change_case_end:w #1 \__text_change_case_result:n #2
29820 {
29821   \group_align_safe_end:
29822   \exp_end:
29823   #2
29824 }

```

The main loop is the standard `tl` action type.

```

29825 \cs_new:Npn \__text_change_case_loop:nnw #1#2#3 \q__text_recursion_stop
29826 {
29827   \tl_if_head_is_N_type:nTF {#3}
29828   { \__text_change_case_N_type:nnN }
29829   {
29830     \tl_if_head_is_group:nTF {#3}
29831     { \use:c { __text_change_case_group_ #1 :nnn } }
29832     { \__text_change_case_space:nnw }
29833   }
29834   {#1} {#2} #3 \q__text_recursion_stop
29835 }
29836 \cs_new:Npn \__text_change_case_break:w #1 \q__text_recursion_tail \q__text_recursion_stop
29837 {

```

```

29838     \_text_change_case_store:n {#1}
29839     \_text_change_case_end:w
29840 }

```

For a group, we *could* worry about whether this contains a character or not. However, that would make life very complex for little gain: exactly what a first character is is rather weakly-defined anyway. So if there is a group, we simply assume that a character has been seen, and for title case we switch to the “rest of the tokens” situation. To avoid having too much testing, we use a two-step process here to allow the titlecase functions to be separate.

```

29841 \cs_new:Npn \_text_change_case_group_lower:nnn #1#2#3
29842 {
29843     \_text_change_case_store:o
29844     {
29845         \exp_after:wN
29846         {
29847             \exp:w
29848             \_text_change_case_aux:nnn {#3} {#1} {#2}
29849         }
29850     }
29851     \_text_change_case_loop:nnw {#1} {#2}
29852 }
29853 \cs_new_eq:NN \_text_change_case_group_upper:nnn
29854 \_text_change_case_group_lower:nnn
29855 \cs_new:Npn \_text_change_case_group_title:nnn #1#2#3
29856 {
29857     \_text_change_case_store:o
29858     {
29859         \exp_after:wN
29860         {
29861             \exp:w
29862             \_text_change_case_aux:nnn {#3} {#1} {#2}
29863         }
29864     }
29865     \_text_change_case_loop:nnw { lower } {#2}
29866 }
29867 \cs_new:Npn \_text_change_case_group_titleonly:nnn #1#2#3
29868 {
29869     \_text_change_case_store:o
29870     {
29871         \exp_after:wN
29872         {
29873             \exp:w
29874             \_text_change_case_aux:nnn {#3} {#1} {#2}
29875         }
29876     }
29877     \_text_change_case_break:w
29878 }
29879 \use:x
29880 {
29881     \cs_new:Npn \exp_not:N \_text_change_case_space:nnw ##1##2 \c_space_tl
29882 }
29883 {
29884     \_text_change_case_store:n { ~ }

```

```

29885 \cs_if_exist_use:c { __text_change_case_boundary_ #1 _ #2 :Nnnw }
29886 \__text_change_case_loop:nnw {#1} {#2}
29887 }

```

The first step of handling N-type tokens is to filter out the end-of-loop. That has to be done separately from the first real step as otherwise we pick up the wrong delimiter. The loop here is the same as the `expand` one, just passing the additional data long. If no close-math token is found then the final clean-up is forced (i.e. there is no assumption of “well-behaved” input in terms of math mode).

```

29888 \cs_new:Npn \__text_change_case_N_type:nnN #1#2#3
29889 {
29890   \__text_if_recursion_tail_stop_do:Nn #3
29891   { \__text_change_case_end:w }
29892   \__text_change_case_N_type_aux:nnN {#1} {#2} #3
29893 }
29894 \cs_new:Npn \__text_change_case_N_type_aux:nnN #1#2#3
29895 {
29896   \exp_args:NV \__text_change_case_N_type:nnnN
29897   \l_text_math_delims_tl {#1} {#2} #3
29898 }
29899 \cs_new:Npn \__text_change_case_N_type:nnnN #1#2#3#4
29900 {
29901   \__text_change_case_math_search:nnNNN {#2} {#3} #4 #1
29902   \q__text_recursion_tail \q__text_recursion_tail
29903   \q__text_recursion_stop
29904 }
29905 \cs_new:Npn \__text_change_case_math_search:nnNNN #1#2#3#4#5
29906 {
29907   \__text_if_recursion_tail_stop_do:Nn #4
29908   { \__text_change_case_cs_check:nnN {#1} {#2} #3 }
29909   \token_if_eq_meaning:NNTF #3 #4
29910   {
29911     \__text_use_i_delimit_by_q_recursion_stop:nw
29912     {
29913       \__text_change_case_store:n {#3}
29914       \__text_change_case_math_loop:nnNw {#1} {#2} #5
29915     }
29916   }
29917   { \__text_change_case_math_search:nnNNN {#1} {#2} #3 }
29918 }
29919 \cs_new:Npn \__text_change_case_math_loop:nnNw #1#2#3#4 \q__text_recursion_stop
29920 {
29921   \tl_if_head_is_N_type:nTF {#4}
29922   { \__text_change_case_math_N_type:nnNN }
29923   {
29924     \tl_if_head_is_group:nTF {#4}
29925     { \__text_change_case_math_group:nnNn }
29926     { \__text_change_case_math_space:nnNw }
29927   }
29928   {#1} {#2} #3 #4 \q__text_recursion_stop
29929 }
29930 \cs_new:Npn \__text_change_case_math_N_type:nnNN #1#2#3#4
29931 {
29932   \__text_if_recursion_tail_stop_do:Nn #4

```

```

29933     { \_text_change_case_end:w }
29934 \_text_change_case_store:n {#4}
29935 \token_if_eq_meaning:NNTF #4 #3
29936     { \_text_change_case_loop:nnw {#1} {#2} }
29937     { \_text_change_case_math_loop:nnNw {#1} {#2} #3 }
29938 }
29939 \cs_new:Npn \_text_change_case_math_group:nnNn #1#2#3#4
29940 {
29941     \_text_change_case_store:n { {#4} }
29942     \_text_change_case_math_loop:nnNw {#1} {#2} #3
29943 }
29944 \use:x
29945 {
29946     \cs_new:Npn \exp_not:N \_text_change_case_math_space:nnNw ##1##2##3
29947         \c_space_tl
29948 }
29949 {
29950     \_text_change_case_store:n { ~ }
29951     \_text_change_case_math_loop:nnNw {#1} {#2} #3
29952 }

```

Once potential math-mode cases are filtered out the next stage is to test if the token grabbed is a control sequence: the two routes the code may take are then very different.

```

29953 \cs_new:Npn \_text_change_case_cs_check:nnN #1#2#3
29954 {
29955     \token_if_cs:NNTF #3
29956     { \_text_change_case_exclude:nnN }
29957     { \use:c { \_text_change_case_char_ #1 :nnN } }
29958     {#1} {#2} #3
29959 }

```

To deal with a control sequence there is first a need to test if it is on the list which indicate that case changing should be skipped. That's done using a loop as for the other special cases. If a hit is found then the argument is grabbed and passed through as-is.

```

29960 \cs_new:Npn \_text_change_case_exclude:nnN #1#2#3
29961 {
29962     \exp_args:Ne \_text_change_case_exclude:nnnN
29963     {
29964         \exp_not:V \l_text_math_arg_tl
29965         \exp_not:V \l_text_case_exclude_arg_tl
29966     }
29967     {#1} {#2} #3
29968 }
29969 \cs_new:Npn \_text_change_case_exclude:nnnN #1#2#3#4
29970 {
29971     \_text_change_case_exclude:nnNN {#2} {#3} #4 #1
29972     \q__text_recursion_tail \q__text_recursion_stop
29973 }
29974 \cs_new:Npn \_text_change_case_exclude:nnNN #1#2#3#4
29975 {
29976     \_text_if_recursion_tail_stop_do:Nn #4
29977     { \use:c { \_text_change_case_letterlike_ #1 :nnN } {#1} {#2} #3 }
29978     \str_if_eq:nnTF {#3} {#4}
29979     {
29980         \_text_use_i_delimit_by_q_recursion_stop:nw

```

```

29981         { \__text_change_case_exclude:nnNw {#1} {#2} #3 }
29982     }
29983     { \__text_change_case_exclude:nnNN {#1} {#2} #3 }
29984 }
29985 \cs_new:Npn \__text_change_case_exclude:nnNw #1#2#3#4#
29986 { \__text_change_case_exclude:nnNnn {#1} {#2} {#3} {#4} }
29987 \cs_new:Npn \__text_change_case_exclude:nnNnn #1#2#3#4#5
29988 {
29989     \__text_change_case_store:n { #3#4 {#5} }
29990     \__text_change_case_loop:nnw {#1} {#2}
29991 }

```

Letter-like commands may still be present: they are set up using a simple lookup approach, so can easily be handled with no loop. If there is no hit, we are at the end of the process: we loop around. Letter-like chars are all available only in upper- and lowercase, so titlecasing maps to the uppercase version.

```

29992 \cs_new:Npn \__text_change_case_letterlike_lower:nnN #1#2#3
29993 { \__text_change_case_letterlike:nnnnN {#1} {#1} {#1} {#2} #3 }
29994 \cs_new_eq:NN \__text_change_case_letterlike_upper:nnN
29995 \__text_change_case_letterlike_lower:nnN
29996 \cs_new:Npn \__text_change_case_letterlike_title:nnN #1#2#3
29997 { \__text_change_case_letterlike:nnnnN { upper } { lower } {#1} {#2} #3 }
29998 \cs_new:Npn \__text_change_case_letterlike_titleonly:nnN #1#2#3
29999 { \__text_change_case_letterlike:nnnnN { upper } { end } {#1} {#2} #3 }
30000 \cs_new:Npn \__text_change_case_letterlike:nnnnN #1#2#3#4#5
30001 {
30002     \cs_if_exist:cTF { c__text_ #1 case_ \token_to_str:N #5 _tl }
30003     {
30004         \__text_change_case_store:v
30005         { c__text_ #1 case_ \token_to_str:N #5 _tl }
30006         \use:c { __text_change_case_char_next_ #2 :nn } {#2} {#4}
30007     }
30008     {
30009         \__text_change_case_store:n {#5}
30010         \cs_if_exist:cTF
30011         {
30012             c__text_
30013             \str_if_eq:nnTF {#1} { lower } { upper } { lower }
30014             case_ \token_to_str:N #5 _tl
30015         }
30016         { \use:c { __text_change_case_char_next_ #2 :nn } {#2} {#4} }
30017         { \__text_change_case_loop:nnw {#3} {#4} }
30018     }
30019 }

```

For upper- and lowercase changes, once we get to this stage there are only a couple of questions remaining: is there a language-specific mapping and is there the special case of a terminal sigma. If not, then we pass to a simple character mapping.

```

30020 \cs_new:Npn \__text_change_case_char_lower:nnN #1#2#3
30021 {
30022     \cs_if_exist_use:cF { __text_change_case_lower_ #2 :nnnN }
30023     { \__text_change_case_lower_sigma:nnnN }
30024     {#1} {#1} {#2} #3
30025 }

```

```

30026 \cs_new:Npn \__text_change_case_char_upper:nnN #1#2#3
30027 {
30028   \cs_if_exist_use:cF { __text_change_case_upper_ #2 :nnnN }
30029   { \__text_change_case_char:nnnN }
30030   {#1} {#1} {#2} #3
30031 }

```

If the current character is an uppercase sigma, the a check is made on the next item in the input. If it is N-type and not a control sequence then there is a look-ahead phase: the logic here is simply based on letters. The one exception is Dutch: see below.

```

30032 \bool_lazy_or:nnTF
30033 { \sys_if_engine luatex_p: }
30034 { \sys_if_engine xetex_p: }
30035 {
30036   \cs_new:Npn \__text_change_case_lower_sigma:nnnN #1#2#3#4
30037   {
30038     \int_compare:nNnTF { '#4 } = { "03A3 }
30039     { \__text_change_case_lower_sigma:nnNw {#2} {#3} #4 }
30040     { \__text_change_case_char:nnnN {#1} {#2} {#3} #4 }
30041   }
30042   \cs_new:Npn \__text_change_case_lower_sigma:nnNw #1#2#3#4 \q__text_recursion_stop
30043   {
30044     \tl_if_head_is_N_type:nTF {#4}
30045     { \__text_change_case_lower_sigma:NnnN #3 }
30046     {
30047       \__text_change_case_store:e
30048       { \char_generate:nn { "03C2 } { \__text_char_catcode:N #3 } }
30049       \__text_change_case_loop:nnw
30050     }
30051     {#1} {#2} #4 \q__text_recursion_stop
30052   }
30053   \cs_new:Npn \__text_change_case_lower_sigma:NnnN #1#2#3#4
30054   {
30055     \__text_change_case_store:e
30056     {
30057       \token_if_letter:NnTF #4
30058       { \char_generate:nn { "03C3 } { \__text_char_catcode:N #1 } }
30059       { \char_generate:nn { "03C2 } { \__text_char_catcode:N #1 } }
30060     }
30061     \__text_change_case_loop:nnw {#2} {#3} #4
30062   }
30063 }

```

In the 8-bit engines, we have to look ahead once we find the first byte of the possible hit.

```

30064 {
30065   \cs_new:Npn \__text_change_case_lower_sigma:nnnN #1#2#3#4
30066   {
30067     \int_compare:nNnTF { '#4 } = { "CE }
30068     { \__text_change_case_lower_sigma:nnnNN }
30069     { \__text_change_case_char:nnnN }
30070     {#1} {#2} {#3} #4
30071   }
30072   \cs_new:Npn \__text_change_case_lower_sigma:nnnNN #1#2#3#4#5
30073   {
30074     \int_compare:nNnTF { '#5 } = { "A3 }

```

```

30075         { \__text_change_case_lower_sigma:nnw {#2} {#3} }
30076         { \__text_change_case_char:nnnN {#1} {#2} {#3} #4#5 }
30077     }
30078 \cs_new:Npn \__text_change_case_lower_sigma:nnw #1#2#3 \q__text_recursion_stop
30079 {
30080     \tl_if_head_is_N_type:nTF {#3}
30081     { \__text_change_case_lower_sigma:nnN }
30082     {
30083         \__text_change_case_store:V \c__text_final_sigma_tl
30084         \__text_change_case_loop:nnw
30085     }
30086     {#1} {#2} #3 \q__text_recursion_stop
30087 }
30088 \cs_new:Npn \__text_change_case_lower_sigma:nnN #1#2#3
30089 {
30090     \bool_lazy_or:nnTF
30091     { \token_if_letter_p:N #3 }
30092     {
30093         \bool_lazy_and_p:nn
30094         { \token_if_active_p:N #3 }
30095         { \int_compare_p:nNn { '#3 } > { "80 } }
30096     }
30097     { \__text_change_case_store:V \c__text_sigma_tl }
30098     { \__text_change_case_store:V \c__text_final_sigma_tl }
30099     \__text_change_case_loop:nnw {#1} {#2} #3
30100 }
30101 }

```

For titlecasing, we need to fully expand the new character to see if it is a letter (or active) But that means looking ahead in the 8-bit case, so we have to grab the required tokens up-front. Life is a lot easier for Unicode T_EX's, where we just have one token to worry about. The one wrinkle here is that for look-ahead we'd get into trouble: luckily, only Dutch has that issue.

```

30102 \cs_new:Npx \__text_change_case_char_title:nnN #1#2#3
30103 {
30104     \exp_not:N \bool_if:NTF \l_text_titlecase_check_letter_bool
30105     {
30106         \bool_lazy_or:nnTF
30107         { \sys_if_engine luatex_p: }
30108         { \sys_if_engine xetex_p: }
30109         { \exp_not:N \token_if_letter:NTF #3 }
30110         {
30111             \exp_not:N \bool_lazy_or:nnTF
30112             { \exp_not:N \token_if_letter_p:N #3 }
30113             { \exp_not:N \token_if_active_p:N #3 }
30114         }
30115         { \exp_not:N \use:c { __text_change_case_char_ #1 :nN } }
30116         { \exp_not:N \__text_change_case_char_title:nnnN { title } {#1} }
30117     }
30118     { \exp_not:N \use:c { __text_change_case_char_ #1 :nN } }
30119     {#2} #3
30120 }
30121 \cs_new_eq:NN \__text_change_case_char_titleonly:nnN
30122 \__text_change_case_char_title:nnN

```



```

30123 \cs_new:Npn \__text_change_case_char_title:nN #1#2
30124 { \__text_change_case_char_title:nnnN { title } { lower } {#1} #2 }
30125 \cs_new:Npn \__text_change_case_char_titleonly:nN #1#2
30126 { \__text_change_case_char_title:nnnN { title } { end } {#1} #2 }
30127 \cs_new:Npn \__text_change_case_char_title:nnnN #1#2#3#4
30128 {
30129   \cs_if_exist_use:cF { __text_change_case_title_ #3 :nnnN }
30130   {
30131     \cs_if_exist_use:cF { __text_change_case_upper_ #3 :nnnN }
30132     { \__text_change_case_char:nnnN }
30133   }
30134   {#1} {#2} {#3} #4
30135 }

```

For Unicode engines we can handle all characters directly. However, for the 8-bit engines the aim is to deal with (a subset of) Unicode (UTF-8) input. They deal with that by making the upper half of the range active, so we look for that and if found work out how many UTF-8 octets there are to deal with. Those can then be grabbed to reconstruct the full Unicode character, which is then used in a lookup. (As will become obvious below, there is no intention here of covering all of Unicode.) For (u)p-TeX there are a limited number of tokens we can touch.

```

30136 \cs_new:Npn \__text_change_case_char:nnnN #1#2#3#4
30137 {
30138   \token_if_active:NTF #4
30139   { \__text_change_case_store:n {#4} }
30140   {
30141     \__text_change_case_store:e
30142     { \use:c { char_ #1 case :N } #4 }
30143   }
30144   \use:c { __text_change_case_char_next_ #2 :nn } {#2} {#3}
30145 }
30146 \bool_lazy_or:nnF
30147 { \sys_if_engine luatex_p: }
30148 { \sys_if_engine xetex_p: }
30149 {
30150   \cs_new_eq:NN \__text_change_case_char_auxi:nnnN
30151   \__text_change_case_char:nnnN
30152   \cs_gset:Npn \__text_change_case_char:nnnN #1#2#3#4
30153   {
30154     \int_compare:nNnTF { '#4 } > { "80 }
30155     {
30156       \int_compare:nNnTF { '#4 } < { "EO }
30157       { \__text_change_case_char_UTFviii:nnnNN }
30158       { \__text_change_case_char_auxii:nnnN }
30159     }
30160     { \__text_change_case_char_auxi:nnnN }
30161     {#1} {#2} {#3} #4
30162   }
30163   \sys_if_engine pdftex:TF
30164   {
30165     \cs_new:Npn \__text_change_case_char_auxii:nnnN #1#2#3#4
30166     {
30167       \int_compare:nNnTF { '#4 } < { "FO }
30168       { \__text_change_case_char_UTFviii:nnnNNN }

```

```

30169         { \_text_change_case_char_UTFviii:nnnNNNN }
30170         {#1} {#2} {#3} #4
30171     }
30172 }
30173 {
30174     \cs_new:Npn \_text_change_case_char_auxii:nnnN #1#2#3#4
30175     {
30176         \_text_change_case_store:n {#4}
30177         \use:c { \_text_change_case_char_next_ #2 :nn } {#2} {#3}
30178     }
30179 }
30180 \cs_new:Npn \_text_change_case_char_UTFviii:nnnNN #1#2#3#4#5
30181 { \_text_change_case_char_UTFviii:nnnn {#1} {#2} {#3} {#4#5} }
30182 \cs_new:Npn \_text_change_case_char_UTFviii:nnnNNN #1#2#3#4#5#6
30183 { \_text_change_case_char_UTFviii:nnnn {#1} {#2} {#3} {#4#5#6} }
30184 \cs_new:Npn \_text_change_case_char_UTFviii:nnnNNNN #1#2#3#4#5#6#7
30185 { \_text_change_case_char_UTFviii:nnnn {#1} {#2} {#3} {#4#5#6#7} }
30186 \cs_new:Npn \_text_change_case_char_UTFviii:nnnn #1#2#3#4
30187 {
30188     \cs_if_exist:cTF { c__text_ #1 case_ \tl_to_str:n {#4} _tl }
30189     {
30190         \_text_change_case_store:v
30191         { c__text_ #1 case_ \tl_to_str:n {#4} _tl }
30192     }
30193     { \_text_change_case_store:n {#4} }
30194     \use:c { \_text_change_case_char_next_ #2 :nn } {#2} {#3}
30195 }
30196 }
30197 \cs_new:Npn \_text_change_case_char_next_lower:nn #1#2
30198 { \_text_change_case_loop:nnw {#1} {#2} }
30199 \cs_new_eq:NN \_text_change_case_char_next_upper:nn
30200 \_text_change_case_char_next_lower:nn
30201 \cs_new_eq:NN \_text_change_case_char_next_title:nn
30202 \_text_change_case_char_next_lower:nn
30203 \cs_new_eq:NN \_text_change_case_char_next_titleonly:nn
30204 \_text_change_case_char_next_lower:nn
30205 \cs_new:Npn \_text_change_case_char_next_end:nn #1#2
30206 { \_text_change_case_break:w }

```

(End definition for _text_change_case:nnn and others.)

_text_change_case_upper_de-alt:nnnN
_text_change_case_upper_de-alt:nnnNN

A simple alternative version for German.

```

30207 \bool_lazy_or:nnTF
30208 { \sys_if_engine luatex_p: }
30209 { \sys_if_engine xetex_p: }
30210 {
30211     \cs_new:cpn { \_text_change_case_upper_de-alt:nnnN } #1#2#3#4
30212     {
30213         \int_compare:nNnTF { '#4 } = { "00DF }
30214         {
30215             \_text_change_case_store:e
30216             { \char_generate:nn { "1E9E } { \_text_char_catcode:N #4 } }
30217             \use:c { \_text_change_case_char_next_ #2 :nn }
30218             {#2} {#3}

```

```

30219     }
30220     { \_text_change_case_char:nnnN {#1} {#2} {#3} #4 }
30221   }
30222 }
30223 {
30224   \cs_new:cpx { \_text_change_case_upper_de-alt:nnnN } #1#2#3#4
30225   {
30226     \exp_not:N \int_compare:nNnTF { '#4 } = { "00C3 }
30227     {
30228       \exp_not:c { \_text_change_case_upper_de-alt:nnnNN }
30229       {#1} {#2} {#3} #4
30230     }
30231     { \exp_not:N \_text_change_case_char:nnnN {#1} {#2} {#3} #4 }
30232   }
30233   \cs_new:cpn { \_text_change_case_upper_de-alt:nnnNN } #1#2#3#4#5
30234   {
30235     \int_compare:nNnTF { '#5 } = { "009F }
30236     {
30237       \_text_change_case_store:V \c__text_grosses_Eszett_tl
30238       \use:c { \_text_change_case_char_next_ #2 :nn } {#2} {#3}
30239     }
30240     { \_text_change_case_char:nnnN {#1} {#2} {#3} #4#5 }
30241   }
30242 }

```

(End definition for _text_change_case_upper_de-alt:nnnN and _text_change_case_upper_de-alt:nnnNN.)

```

\_text_change_case_upper_el:nnnN
\_text_change_case_upper_el:nnN
\_text_change_case_upper_el:nnW
\_text_change_case_upper_el:NnnN
\_text_change_case_upper_el:dialytika:nnN
\_text_change_case_upper_el:dialytika:N
\_text_change_case_upper_el:hiatus:nnW
\_text_change_case_upper_el:hiatus:nnN
\_text_change_case_upper_el:gobble:nnw
\_text_change_case_upper_el:gobble:nnN
\_text_change_case_if_greek:nTF
\_text_change_case_if_greek:p:n
\_text_change_case_if_greek:nTF
\_text_change_case_if_greek_accent:p:n
\_text_change_case_if_greek_accent:nTF
\_text_change_case_if_greek_diacritic:p:n
\_text_change_case_if_greek_diacritic:nTF
\_text_change_case_if_takes_dialytika:nTF

```

For Greek uppercasing, we need to know if characters *in the Greek range* have accents. That means doing a NFD conversion first, then starting a search. As described by the Unicode CLDR, Greek accents need to be found *after* any U+0308 (dieresis) and are done in two groups to allow for the canonical ordering. The implementation here follows the data and examples from ICU (<https://sites.google.com/site/icusite/design/case/greek-upper>), although necessarily the implementation is somewhat different.

```

30243 \bool_lazy_or:nnT
30244 { \sys_if_engine luatex_p: }
30245 { \sys_if_engine xetex_p: }
30246 {
30247   \cs_new:Npn \_text_change_case_upper_el:nnnN #1#2#3#4
30248   {
30249     \_text_change_case_if_greek:nTF { '#4 }
30250     {
30251       \exp_args:Ne \_text_change_case_upper_el:nnn
30252       { \char_to_nfd:N #4 } {#2} {#3}
30253     }
30254     { \_text_change_case_char:nnnN {#1} {#2} {#3} #4 }
30255   }
30256   \cs_new:Npn \_text_change_case_upper_el:nnn #1#2#3
30257   { \_text_change_case_upper_el:nnW {#2} {#3} #1 }

```

At this stage we have the first NFD codepoint as #3. What we need to know is whether after that we have another character token, either from the NFD or directly in the input. If not, we store the changed character at this stage.

```

30258   \cs_new:Npn \_text_change_case_upper_el:nnW #1#2#3#4 \q_text_recursion_stop
30259   {

```

```

30260     \tl_if_head_is_N_type:nTF {#4}
30261     { \__text_change_case_upper_el:NnnN #3 }
30262     {
30263         \__text_change_case_store:e { \char_uppercase:N #3 }
30264         \__text_change_case_loop:nnw
30265     }
30266     {#1} {#2} #4 \q__text_recursion_stop
30267 }

```

Now, we check the detail of the next codepoint: again we filter out the not-a-char cases, before checking if it's an dialytika, accent or diacritic. (The latter do not have the same hiatus behavior as accents.)

```

30268 \cs_new:Npn \__text_change_case_upper_el:NnnN #1#2#3#4
30269 {
30270     \token_if_cs:NTF #4
30271     {
30272         \__text_change_case_store:e { \char_uppercase:N #1 }
30273         \__text_change_case_loop:nnw {#2} {#3} #4
30274     }
30275     {
30276         \int_compare:nNnTF { '#4 } = { "0308 }
30277         { \__text_change_case_upper_el_dialytika:nnN {#2} {#3} #1 }
30278         {
30279             \__text_change_case_if_greek_accent:nTF { '#4 }
30280             { \__text_change_case_upper_el_hiatus:nnNw {#2} {#3} #1 }
30281             {
30282                 \__text_change_case_if_greek_diacritic:nTF { '#4 }
30283                 {
30284                     \__text_change_case_store:e { \char_uppercase:N #1 }
30285                     \__text_change_case_loop:nnw {#2} {#3}
30286                 }
30287                 {
30288                     \__text_change_case_store:e { \char_uppercase:N #1 }
30289                     \__text_change_case_loop:nnw {#2} {#3} #4
30290                 }
30291             }
30292         }
30293     }
30294 }

```

We handle *dialytika* in parts as it's also needed for the hiatus. We know only two letters take it, so we can shortcut here on the second part of the tests.

```

30295 \cs_new:Npn \__text_change_case_upper_el_dialytika:nnN #1#2#3
30296 {
30297     \__text_change_case_if_takes_dialytika:nTF { '#3 }
30298     { \__text_change_case_upper_el_dialytika:N #3 }
30299     { \__text_change_case_store:e { \char_uppercase:N #3 } }
30300     \__text_change_case_upper_el_gobble:nnw {#1} {#2}
30301 }
30302 \cs_new:Npn \__text_change_case_upper_el_dialytika:N #1
30303 {
30304     \__text_change_case_store:e
30305     {
30306         \bool_lazy_or:nnTF
30307         { \int_compare_p:nNn { '#1 } = { "0399 } }

```

```

30308         { \int_compare_p:nNn { '#1 } = { "03B9 } }
30309         { \char_generate:nn { "03AA } { \__text_char_catcode:N #1 } }
30310         { \char_generate:nn { "03AB } { \__text_char_catcode:N #1 } }
30311     }
30312 }

```

Adding a hiatus needs some of the same ideas, but if there is not one we skip this code point, hence needing a separate function.

```

30313 \cs_new:Npn \__text_change_case_upper_el_hiatus:nnNw
30314 #1#2#3#4 \q__text_recursion_stop
30315 {
30316     \__text_change_case_store:e { \char_uppercase:N #3 }
30317     \tl_if_head_is_N_type:nTF {#4}
30318     { \__text_change_case_upper_el_hiatus:nnN }
30319     { \__text_change_case_loop:nnw }
30320     {#1} {#2} #4 \q__text_recursion_stop
30321 }
30322 \cs_new:Npn \__text_change_case_upper_el_hiatus:nnN #1#2#3
30323 {
30324     \token_if_cs:NTF #3
30325     { \__text_change_case_loop:nnw {#1} {#2} #3 }
30326     {
30327         \__text_change_case_if_takes_dialytika:nTF { '#3 }
30328         {
30329             \__text_change_case_upper_el_dialytika:N #3
30330             \__text_change_case_upper_el_gobble:nnw {#1} {#2}
30331         }
30332         { \__text_change_case_loop:nnw {#1} {#2} #3 }
30333     }
30334 }

```

For clearing out trailing combining marks after we have dealt with the first one.

```

30335 \cs_new:Npn \__text_change_case_upper_el_gobble:nnw
30336 #1#2#3 \q__text_recursion_stop
30337 {
30338     \tl_if_head_is_N_type:nTF {#3}
30339     { \__text_change_case_upper_el_gobble:nnN }
30340     { \__text_change_case_loop:nnw }
30341     {#1} {#2} #3 \q__text_recursion_stop
30342 }
30343 \cs_new:Npn \__text_change_case_upper_el_gobble:nnN #1#2#3
30344 {
30345     \bool_lazy_or:nnTF
30346     { \token_if_cs_p:N #3 }
30347     {
30348         ! \bool_lazy_or_p:nn
30349         { \__text_change_case_if_greek_accent_p:n { '#3 } }
30350         { \__text_change_case_if_greek_diacritic_p:n { '#3 } }
30351     }
30352     { \__text_change_case_loop:nnw {#1} {#2} #3 }
30353     { \__text_change_case_upper_el_gobble:nnw {#1} {#2} }
30354 }
30355 }

```

Luckily the Greek range is limited and clear.

```

30356 \prg_new_conditional:Npnn \__text_change_case_if_greek:n #1 { TF }
30357 {
30358   \if_int_compare:w #1 < "0370 \exp_stop_f:
30359     \prg_return_false:
30360   \else:
30361     \if_int_compare:w #1 > "03FF \exp_stop_f:
30362     \if_int_compare:w #1 < "1F00 \exp_stop_f:
30363     \prg_return_false:
30364   \else:
30365     \if_int_compare:w #1 > "1FFF \exp_stop_f:
30366     \prg_return_false:
30367   \else:
30368     \prg_return_true:
30369   \fi:
30370   \fi:
30371 \else:
30372   \prg_return_true:
30373 \fi:
30374 \fi:
30375 }

```

We follow ICU in adding a few extras to the accent list here.

```

30376 \prg_new_conditional:Npnn \__text_change_case_if_greek_accent:n #1 { TF , p }
30377 {
30378   \if_int_compare:w #1 = "0300 \exp_stop_f:
30379     \prg_return_true:
30380   \else:
30381     \if_int_compare:w #1 = "0301 \exp_stop_f:
30382     \prg_return_true:
30383   \else:
30384     \if_int_compare:w #1 = "0342 \exp_stop_f:
30385     \prg_return_true:
30386   \else:
30387     \if_int_compare:w #1 = "0302 \exp_stop_f:
30388     \prg_return_true:
30389   \else:
30390     \if_int_compare:w #1 = "0303 \exp_stop_f:
30391     \prg_return_true:
30392   \else:
30393     \if_int_compare:w #1 = "0311 \exp_stop_f:
30394     \prg_return_true:
30395   \else:
30396     \prg_return_false:
30397   \fi:
30398   \fi:
30399   \fi:
30400   \fi:
30401   \fi:
30402   \fi:
30403 }
30404 \prg_new_conditional:Npnn \__text_change_case_if_greek_diacritic:n
30405 #1 { TF , p }
30406 {
30407   \if_int_compare:w #1 = "0304 \exp_stop_f:
30408     \prg_return_true:

```

```

30409 \else:
30410 \if_int_compare:w #1 = "0306 \exp_stop_f:
30411 \prg_return_true:
30412 \else:
30413 \if_int_compare:w #1 = "0313 \exp_stop_f:
30414 \prg_return_true:
30415 \else:
30416 \if_int_compare:w #1 = "0314 \exp_stop_f:
30417 \prg_return_true:
30418 \else:
30419 \if_int_compare:w #1 = "0343 \exp_stop_f:
30420 \prg_return_true:
30421 \else:
30422 \prg_return_false:
30423 \fi:
30424 \fi:
30425 \fi:
30426 \fi:
30427 \fi:
30428 }
30429 \prg_new_conditional:Npnn \_text_change_case_if_takes_dialytika:n #1 { TF }
30430 {
30431 \if_int_compare:w #1 = "0399 \exp_stop_f:
30432 \prg_return_true:
30433 \else:
30434 \if_int_compare:w #1 = "03B9 \exp_stop_f:
30435 \prg_return_true:
30436 \else:
30437 \if_int_compare:w #1 = "03A5 \exp_stop_f:
30438 \prg_return_true:
30439 \else:
30440 \if_int_compare:w #1 = "03C5 \exp_stop_f:
30441 \prg_return_true:
30442 \else:
30443 \prg_return_false:
30444 \fi:
30445 \fi:
30446 \fi:
30447 \fi:
30448 }

```

(End definition for _text_change_case_upper_el:nnnN and others.)

_text_change_case_boundary_upper_el:Nnnw
_text_change_case_boundary_upper_el:nnN
_text_change_case_boundary_upper_el:nnNw
_text_change_case_boundary_upper_el:NnnN

There is one special case in Greek that needs to be picked up based on being an isolated letter. We do that using a test similar to final sigma, but it has to fire off from the space grabber.

```

30449 \bool_lazy_or:nnT
30450 { \sys_if_engine luatex_p: }
30451 { \sys_if_engine xetex_p: }
30452 {
30453 \cs_new:Npn \_text_change_case_boundary_upper_el:Nnnw
30454 #1#2#3#4 \q__text_recursion_stop
30455 {
30456 \tl_if_head_is_N_type:nTF {#4}

```

```

30457         { \_text_change_case_boundary_upper_el:nnN }
30458         { \_text_change_case_loop:nnw }
30459         {#2} {#3} #4 \q__text_recursion_stop
30460     }
30461 \cs_new:Npn \_text_change_case_boundary_upper_el:nnN #1#2#3
30462 {
30463     \bool_lazy_or:nnTF
30464     { \token_if_cs_p:N #3 }
30465     {
30466         ! \bool_lazy_or_p:nn
30467         { \int_compare_p:nNn { '#3 } = { "03AE } }
30468         { \int_compare_p:nNn { '#3 } = { "1F22 } }
30469     }
30470     { \_text_change_case_loop:nnw }
30471     { \_text_change_case_boundary_upper_el:nnNw }
30472     {#1} {#2} #3
30473 }
30474 \cs_new:Npn \_text_change_case_boundary_upper_el:nnNw
30475 #1#2#3#4 \q__text_recursion_stop
30476 {
30477     \tl_if_head_is_N_type:nTF {#4}
30478     { \_text_change_case_boundary_upper_el:NnnN #3 }
30479     {
30480         \_text_change_case_store:e
30481         { \char_generate:nn { "0389 } { \_text_char_catcode:N #3 } }
30482         \_text_change_case_loop:nnw
30483     }
30484     {#1} {#2} #4 \q__text_recursion_stop
30485 }
30486 \cs_new:Npn \_text_change_case_boundary_upper_el:NnnN #1#2#3#4
30487 {
30488     \token_if_letter:NTF #4
30489     { \_text_change_case_loop:nnw {#2} {#3} #1#4 }
30490     {
30491         \_text_change_case_store:e
30492         { \char_generate:nn { "0389 } { \_text_char_catcode:N #1 } }
30493         \_text_change_case_loop:nnw {#2} {#3} #4
30494     }
30495 }
30496 }

```

(End definition for _text_change_case_boundary_upper_el:Nnnw and others.)

_text_change_case_title_el:nnnN Titlecasing retains accents, but to prevent the uppercasing code from kicking in, there has to be an explicit function here.

```

30497 \bool_lazy_or:nnT
30498 { \sys_if_engine luatex_p: }
30499 { \sys_if_engine xetex_p: }
30500 {
30501     \cs_new:Npn \_text_change_case_title_el:nnnN #1#2#3#4
30502     { \_text_change_case_char:nnnN {#1} {#2} {#3} #4 }
30503 }

```

(End definition for _text_change_case_title_el:nnnN.)


```

\__text_change_cases_lower_lt:nnnN
\__text_change_cases_lower_lt_auxi:nnnN
\__text_change_cases_lower_lt_auxii:nnnN
\__text_change_case_lower_lt:nnw
\__text_change_case_lower_lt:nnN

```

For Lithuanian, the issue to be dealt with is dots over lower case letters: these should be present if there is another accent. The first step is a simple match attempt: look for the three uppercase accented letters which should gain a dot-above char in their lowercase form.

```

30504 \bool_lazy_or:nnT
30505 { \sys_if_engine luatex_p: }
30506 { \sys_if_engine xetex_p: }
30507 {
30508   \cs_new:Npn \__text_change_case_lower_lt:nnnN #1#2#3#4
30509   {
30510     \exp_args:Ne \__text_change_case_lower_lt_auxi:nnnN
30511     {
30512       \int_case:nn { '#4 }
30513       {
30514         { "00CC } { "0300 }
30515         { "00CD } { "0301 }
30516         { "0128 } { "0303 }
30517       }
30518     }
30519     {#2} {#3} #4
30520   }

```

If there was a hit, output the result with the dot-above and move on. Otherwise, look for one of the three letters that can take a combining accent: I, J and I-ogonek.

```

30521   \cs_new:Npn \__text_change_case_lower_lt_auxi:nnnN #1#2#3#4
30522   {
30523     \tl_if_blank:nTF {#1}
30524     {
30525       \exp_args:Ne \__text_change_case_lower_lt_auxii:nnnN
30526       {
30527         \int_case:nn { '#4 }
30528         {
30529           { "0049 } { "0069 }
30530           { "004A } { "006A }
30531           { "012E } { "012F }
30532         }
30533       }
30534       {#2} {#3} #4
30535     }
30536     {
30537       \__text_change_case_store:e
30538       {
30539         \char_generate:nn { "0069 } { \__text_char_catcode:N #4 }
30540         \char_generate:nn { "0307 } { \__text_char_catcode:N #4 }
30541         \char_generate:nn {#1} { \__text_char_catcode:N #4 }
30542       }
30543       \__text_change_case_loop:nnw {#2} {#3}
30544     }
30545   }

```

Again, branch depending on a hit. If there is one, we output the character then need to look for a combining accent: as usual, we need to be aware of the loop situation.

```

30546   \cs_new:Npn \__text_change_case_lower_lt_auxii:nnnN #1#2#3#4
30547   {

```

```

30548 \tl_if_blank:nTF {#1}
30549 { \__text_change_case_lower_sigma:nnnN {#2} {#2} {#3} #4 }
30550 {
30551 \__text_change_case_store:e
30552 { \char_generate:nn {#1} { \__text_char_catcode:N #4 } }
30553 \__text_change_case_lower_lt:nnw {#2} {#3}
30554 }
30555 }
30556 \cs_new:Npn \__text_change_case_lower_lt:nnw #1#2#3 \q__text_recursion_stop
30557 {
30558 \tl_if_head_is_N_type:nTF {#3}
30559 { \__text_change_case_lower_lt:nnN }
30560 { \__text_change_case_loop:nnw }
30561 {#1} {#2} #3 \q__text_recursion_stop
30562 }
30563 \cs_new:Npn \__text_change_case_lower_lt:nnN #1#2#3
30564 {
30565 \bool_lazy_and:nnT
30566 { ! \token_if_cs_p:N #3 }
30567 {
30568 \bool_lazy_any_p:n
30569 {
30570 { \int_compare_p:nNn { '#3 } = { "0300 } }
30571 { \int_compare_p:nNn { '#3 } = { "0301 } }
30572 { \int_compare_p:nNn { '#3 } = { "0303 } }
30573 }
30574 }
30575 {
30576 \__text_change_case_store:e
30577 { \char_generate:nn { "0307 } { \__text_char_catcode:N #3 } }
30578 }
30579 \__text_change_case_loop:nnw {#1} {#2} #3
30580 }
30581 }

```

(End definition for __text_change_cases_lower_lt:nnnN and others.)

```

\__text_change_cases_upper_lt:nnnN
\__text_change_cases_upper_lt_aux:nnnN
\__text_change_case_upper_lt:nnw
\__text_change_case_upper_lt:nnN

```

The uppercasing version: first find i/j/i-ogonek, then look for the combining char: drop it if present.

```

30582 \bool_lazy_or:nnT
30583 { \sys_if_engine luatex_p: }
30584 { \sys_if_engine xetex_p: }
30585 {
30586 \cs_new:Npn \__text_change_case_upper_lt:nnnN #1#2#3#4
30587 {
30588 \exp_args:Ne \__text_change_case_upper_lt_aux:nnnN
30589 {
30590 \int_case:nn { '#4 }
30591 {
30592 { "0069 } { "0049 }
30593 { "006A } { "004A }
30594 { "012F } { "012E }
30595 }
30596 }

```

```

30597         {#2} {#3} #4
30598     }
30599 \cs_new:Npn \__text_change_case_upper_lt_aux:nnnN #1#2#3#4
30600 {
30601     \tl_if_blank:nTF {#1}
30602     { \__text_change_case_char:nnnN { upper } {#2} {#3} #4 }
30603     {
30604         \__text_change_case_store:e
30605         { \char_generate:nn {#1} { \__text_char_catcode:N #4 } }
30606         \__text_change_case_upper_lt:nnw {#2} {#3}
30607     }
30608 }
30609 \cs_new:Npn \__text_change_case_upper_lt:nnw #1#2#3 \q__text_recursion_stop
30610 {
30611     \tl_if_head_is_N_type:nTF {#3}
30612     { \__text_change_case_upper_lt:nnN }
30613     { \use:c { __text_change_case_char_next_ #1 :nn } }
30614     {#1} {#2} #3 \q__text_recursion_stop
30615 }
30616 \cs_new:Npn \__text_change_case_upper_lt:nnN #1#2#3
30617 {
30618     \bool_lazy_and:nnTF
30619     { ! \token_if_cs_p:N #3 }
30620     { \int_compare_p:nNn { '#3 } = { "0307 } }
30621     { \use:c { __text_change_case_char_next_ #1 :nn } {#1} {#2} }
30622     { \use:c { __text_change_case_char_next_ #1 :nn } {#1} {#2} #3 }
30623 }
30624 }

```

(End definition for __text_change_cases_upper_lt:nnnN and others.)

```

\__text_change_case_title_nl:nnnN
\__text_change_case_title_nl:nnw
\__text_change_case_title_nl:nnN

```

For Dutch, there is a single look-ahead test for ij when title casing. If the appropriate letters are found, produce IJ and gobble the j/J.

```

30625 \cs_new:Npn \__text_change_case_title_nl:nnnN #1#2#3#4
30626 {
30627     \bool_lazy_or:nnTF
30628     { \int_compare_p:nNn { '#4 } = { "0049 } }
30629     { \int_compare_p:nNn { '#4 } = { "0069 } }
30630     {
30631         \__text_change_case_store:e
30632         { \char_generate:nn { "0049 } { \__text_char_catcode:N #4 } }
30633         \__text_change_case_title_nl:nnw {#2} {#3}
30634     }
30635     { \__text_change_case_char:nnnN {#1} {#2} {#3} #4 }
30636 }
30637 \cs_new:Npn \__text_change_case_title_nl:nnw #1#2#3 \q__text_recursion_stop
30638 {
30639     \tl_if_head_is_N_type:nTF {#3}
30640     { \__text_change_case_title_nl:nnN }
30641     { \use:c { __text_change_case_char_next_ #1 :nn } }
30642     {#1} {#2} #3 \q__text_recursion_stop
30643 }
30644 \cs_new:Npn \__text_change_case_title_nl:nnN #1#2#3
30645 {

```

```

30646 \bool_lazy_and:nnTF
30647 { ! \token_if_cs_p:N #3 }
30648 {
30649   \bool_lazy_or_p:nn
30650   { \int_compare_p:nNn { '#3 } = { "004A } }
30651   { \int_compare_p:nNn { '#3 } = { "006A } }
30652 }
30653 {
30654   \__text_change_case_store:e
30655   { \char_generate:nn { "004A } { \__text_char_catcode:N #3 } }
30656   \use:c { __text_change_case_char_next_ #1 :nn } {#1} {#2}
30657 }
30658 { \use:c { __text_change_case_char_next_ #1 :nn } {#1} {#2} #3 }
30659 }

```

(End definition for __text_change_case_title_nl:nnnN, __text_change_case_title_nl:nnw, and __text_change_case_title_nl:nnN.)

```

\__text_change_case_lower_tr:nnnN
\__text_change_case_lower_tr:nnNw
\__text_change_case_lower_tr:NnnN
\__text_change_case_lower_tr:nnnN

```

The Turkic languages need special treatment for dotted-i and dotless-i. The lower casing rule can be expressed in terms of searching first for either a dotless-I or a dotted-I. In the latter case the mapping is easy, but in the former there is a second stage search.

```

30660 \bool_lazy_or:nnTF
30661 { \sys_if_engine_luatex_p: }
30662 { \sys_if_engine_xetex_p: }
30663 {
30664   \cs_new:Npn \__text_change_case_lower_tr:nnnN #1#2#3#4
30665   {
30666     \int_compare:nNnTF { '#4 } = { "0049 }
30667     { \__text_change_case_lower_tr:nnNw {#1} {#3} #4 }
30668     {
30669       \int_compare:nNnTF { '#4 } = { "0130 }
30670       {
30671         \__text_change_case_store:e
30672         { \char_generate:nn { "0069 } { \__text_char_catcode:N #4 } }
30673         \__text_change_case_loop:nnw {#1} {#3}
30674       }
30675       { \__text_change_case_lower_sigma:nnnN {#1} {#2} {#3} #4 }
30676     }
30677   }

```

After a dotless-I there may be a dot-above character. If there is then a dotted-i should be produced, otherwise output a dotless-i. When the combination is found both the dotless-I and the dot-above char have to be removed from the input.

```

30678 \cs_new:Npn \__text_change_case_lower_tr:nnNw #1#2#3#4 \q__text_recursion_stop
30679 {
30680   \tl_if_head_is_N_type:nTF {#4}
30681   { \__text_change_case_lower_tr:NnnN #3 }
30682   {
30683     \__text_change_case_store:e
30684     { \char_generate:nn { "0131 } { \__text_char_catcode:N #3 } }
30685     \__text_change_case_loop:nnw
30686   }
30687   {#1} {#2} #4 \q__text_recursion_stop
30688 }

```

```

30689 \cs_new:Npn \__text_change_case_lower_tr:NnnN #1#2#3#4
30690 {
30691   \bool_lazy_or:nnTF
30692     { \token_if_cs_p:N #4 }
30693     { ! \int_compare_p:nNn { '#4 } = { "0307 } }
30694     {
30695       \__text_change_case_store:e
30696       { \char_generate:nn { "0131 } { \__text_char_catcode:N #1 } }
30697       \__text_change_case_loop:nnw {#2} {#3} #4
30698     }
30699     {
30700       \__text_change_case_store:e
30701       { \char_generate:nn { "0069 } { \__text_char_catcode:N #1 } }
30702       \__text_change_case_loop:nnw {#2} {#3}
30703     }
30704   }
30705 }

```

For 8-bit engines, dot-above is not available so there is a simple test for an upper-case I. Then we can look for the UTF-8 representation of an upper case dotted-I without the combining char. If it's not there, preserve the UTF-8 sequence as-is. With 8bit engines, we cannot completely preserve category codes, so we have to make some assumptions: output a “normal” i for the dotted case. As the original character here is catcode-13, we have to make a choice about handling of i: generate a “normal” one.

```

30706 {
30707   \cs_new:Npn \__text_change_case_lower_tr:nnnN #1#2#3#4
30708   {
30709     \int_compare:nNnTF { '#4 } = { "0049 }
30710     {
30711       \__text_change_case_store:V \c__text_dotless_i_tl
30712       \__text_change_case_loop:nnw {#1} {#3}
30713     }
30714     {
30715       \int_compare:nNnTF { '#4 } = { "00C4 }
30716       { \__text_change_case_lower_tr:nnnN {#1} {#2} {#3} #4 }
30717       { \__text_change_case_char:nnnN {#1} {#2} {#3} #4 }
30718     }
30719   }
30720   \cs_new:Npn \__text_change_case_lower_tr:nnnNN #1#2#3#4#5
30721   {
30722     \int_compare:nNnTF { '#5 } = { "00B0 }
30723     {
30724       \__text_change_case_store:e
30725       {
30726         \char_generate:nn { "0069 }
30727         { \char_value_catcode:n { "0069 } }
30728       }
30729       \__text_change_case_loop:nnw {#1} {#3}
30730     }
30731     { \__text_change_case_char:nnnN {#1} {#2} {#3} #4#5 }
30732   }
30733 }

```

(End definition for __text_change_case_lower_tr:nnnN and others.)

_text_change_case_upper_tr:nnnN

Uppercasing is easier: just one exception with no context.

```
30734 \cs_new:Npx \__text_change_case_upper_tr:nnnN #1#2#3#4
30735 {
30736   \exp_not:N \int_compare:nNnTF { '#4 } = { "0069 }
30737   {
30738     \bool_lazy_or:nnTF
30739     { \sys_if_engine luatex_p: }
30740     { \sys_if_engine xetex_p: }
30741     {
30742       \exp_not:N \__text_change_case_store:e
30743       {
30744         \exp_not:N \char_generate:nn { "0130 }
30745         { \exp_not:N \__text_char_catcode:N #4 }
30746       }
30747     }
30748     {
30749       \exp_not:N \__text_change_case_store:V
30750       \exp_not:N \c__text_dotted_I_tl
30751     }
30752   \exp_not:N \use:c { __text_change_case_char_next_ #2 :nn } {#2} {#3}
30753 }
30754 { \exp_not:N \__text_change_case_char:nnnN {#1} {#2} {#3} #4 }
30755 }
```

(End definition for _text_change_case_upper_tr:nnnN.)

_text_change_case_lower_az:nnnN

Straight copies.

_text_change_case_upper_az:nnnN

```
30756 \cs_new_eq:NN \__text_change_case_lower_az:nnnN
30757   \__text_change_case_lower_tr:nnnN
30758 \cs_new_eq:NN \__text_change_case_upper_az:nnnN
30759   \__text_change_case_upper_tr:nnnN
```

(End definition for _text_change_case_lower_az:nnnN and _text_change_case_upper_az:nnnN.)

82.2 Case changing data for 8-bit engines

\c__text_dotless_i_tl
\c__text_dotted_I_tl
\c__text_i_ogonek_tl
\c__text_I_ogonek_tl
\c__text_final_sigma_tl
\c__text_sigma_tl
\c__text_grosses_Eszett_tl

For cases where there is an 8-bit option in the T1 font set up, a variant is provided in both cases. There are also a few extras for LGR.

```
30760 \group_begin:
30761   \bool_lazy_or:nnF
30762   { \sys_if_engine luatex_p: }
30763   { \sys_if_engine xetex_p: }
30764   {
30765     \cs_set_protected:Npn \__text_tmp:w #1#2
30766     {
30767       \group_begin:
30768       \cs_set_protected:Npn \__text_tmp:w ##1##2##3##4
30769       {
30770         \tl_const:Nx #1
30771         {
30772           \exp_after:wN \exp_after:wN \exp_after:wN
30773           \exp_not:N \char_generate:nn {##1} { 13 }
30774           \exp_after:wN \exp_after:wN \exp_after:wN
```

```

30775         \exp_not:N \char_generate:nn {##2} { 13 }
30776     \tl_if_blank:nF {##3}
30777     {
30778         \exp_after:wN \exp_after:wN \exp_after:wN
30779         \exp_not:N \char_generate:nn {##3} { 13 }
30780     }
30781 }
30782 }
30783 \use:x
30784 { \__text_tmp:w \char_to_utfviii_bytes:n { "#2 } }
30785 \group_end:
30786 }
30787 \__text_tmp:w \c__text_dotless_i_tl      { 0131 }
30788 \__text_tmp:w \c__text_dotted_I_tl      { 0130 }
30789 \__text_tmp:w \c__text_i_ogonek_tl      { 012F }
30790 \__text_tmp:w \c__text_I_ogonek_tl      { 012E }
30791 \__text_tmp:w \c__text_final_sigma_tl   { 03C2 }
30792 \__text_tmp:w \c__text_sigma_tl         { 03C3 }
30793 \__text_tmp:w \c__text_grosses_Eszett_tl { 1E9E }
30794 }
30795 \group_end:

```

(End definition for `\c__text_dotless_i_tl` and others.)

For 8-bit engines we now need to define the case-change data for the multi-octet mappings. This data is here not in the `char` module as the multi-byte nature means they are never N-type. These need a list of what code points are doable in T1 so the list is hard coded (there's no saving in loading the mappings dynamically). All of the straight-forward ones have two octets, so that is taken as read.

```

30796 \group_begin:
30797     \bool_lazy_or:nnF
30798     { \sys_if_engine luatex_p: }
30799     { \sys_if_engine xetex_p: }
30800     {
30801         \cs_set_protected:Npn \__text_loop:nn #1#2
30802         {
30803             \quark_if_recursion_tail_stop:n {#1}
30804             \use:x
30805             {
30806                 \__text_tmp:w
30807                 \char_to_utfviii_bytes:n { "#1 }
30808                 \char_to_utfviii_bytes:n { "#2 }
30809             }
30810             \__text_loop:nn
30811         }
30812         \cs_set_protected:Npn \__text_tmp:nnnn #1#2#3#4#5
30813         {
30814             \tl_const:cx
30815             {
30816                 c__text_ #1 case_
30817                 \char_generate:nn {#2} { 12 }
30818                 \char_generate:nn {#3} { 12 }
30819                 _tl
30820             }
30821         {

```

```

30822         \exp_after:wN \exp_after:wN \exp_after:wN
30823         \exp_not:N \char_generate:nn {#4} { 13 }
30824         \exp_after:wN \exp_after:wN \exp_after:wN
30825         \exp_not:N \char_generate:nn {#5} { 13 }
30826     }
30827 }
30828 \cs_set_protected:Npn \__text_tmp:w #1#2#3#4#5#6#7#8
30829 {
30830     \tl_const:cx
30831     {
30832         c__text_lowercase_
30833         \char_generate:nn {#1} { 12 }
30834         \char_generate:nn {#2} { 12 }
30835         _tl
30836     }
30837     {
30838         \exp_after:wN \exp_after:wN \exp_after:wN
30839         \exp_not:N \char_generate:nn {#5} { 13 }
30840         \exp_after:wN \exp_after:wN \exp_after:wN
30841         \exp_not:N \char_generate:nn {#6} { 13 }
30842     }
30843     \__text_tmp:nnnn { upper } {#5} {#6} {#1} {#2}
30844     \__text_tmp:nnnn { title } {#5} {#6} {#1} {#2}
30845 }
30846 \__text_loop:nn
30847 { 00C0 } { 00E0 }
30848 { 00C1 } { 00E1 }
30849 { 00C2 } { 00E2 }
30850 { 00C3 } { 00E3 }
30851 { 00C4 } { 00E4 }
30852 { 00C5 } { 00E5 }
30853 { 00C6 } { 00E6 }
30854 { 00C7 } { 00E7 }
30855 { 00C8 } { 00E8 }
30856 { 00C9 } { 00E9 }
30857 { 00CA } { 00EA }
30858 { 00CB } { 00EB }
30859 { 00CC } { 00EC }
30860 { 00CD } { 00ED }
30861 { 00CE } { 00EE }
30862 { 00CF } { 00EF }
30863 { 00D0 } { 00F0 }
30864 { 00D1 } { 00F1 }
30865 { 00D2 } { 00F2 }
30866 { 00D3 } { 00F3 }
30867 { 00D4 } { 00F4 }
30868 { 00D5 } { 00F5 }
30869 { 00D6 } { 00F6 }
30870 { 00D8 } { 00F8 }
30871 { 00D9 } { 00F9 }
30872 { 00DA } { 00FA }
30873 { 00DB } { 00FB }
30874 { 00DC } { 00FC }
30875 { 00DD } { 00FD }

```


30876	{ 00DE }	{ 00FE }
30877	{ 0100 }	{ 0101 }
30878	{ 0102 }	{ 0103 }
30879	{ 0104 }	{ 0105 }
30880	{ 0106 }	{ 0107 }
30881	{ 0108 }	{ 0109 }
30882	{ 010A }	{ 010B }
30883	{ 010C }	{ 010D }
30884	{ 010E }	{ 010F }
30885	{ 0110 }	{ 0111 }
30886	{ 0112 }	{ 0113 }
30887	{ 0114 }	{ 0115 }
30888	{ 0116 }	{ 0117 }
30889	{ 0118 }	{ 0119 }
30890	{ 011A }	{ 011B }
30891	{ 011C }	{ 011D }
30892	{ 011E }	{ 011F }
30893	{ 0120 }	{ 0121 }
30894	{ 0122 }	{ 0123 }
30895	{ 0124 }	{ 0125 }
30896	{ 0128 }	{ 0129 }
30897	{ 012A }	{ 012B }
30898	{ 012C }	{ 012D }
30899	{ 012E }	{ 012F }
30900	{ 0132 }	{ 0133 }
30901	{ 0134 }	{ 0135 }
30902	{ 0136 }	{ 0137 }
30903	{ 0139 }	{ 013A }
30904	{ 013B }	{ 013C }
30905	{ 013E }	{ 013F }
30906	{ 0141 }	{ 0142 }
30907	{ 0143 }	{ 0144 }
30908	{ 0145 }	{ 0146 }
30909	{ 0147 }	{ 0148 }
30910	{ 014A }	{ 014B }
30911	{ 014C }	{ 014D }
30912	{ 014E }	{ 014F }
30913	{ 0150 }	{ 0151 }
30914	{ 0152 }	{ 0153 }
30915	{ 0154 }	{ 0155 }
30916	{ 0156 }	{ 0157 }
30917	{ 0158 }	{ 0159 }
30918	{ 015A }	{ 015B }
30919	{ 015C }	{ 015D }
30920	{ 015E }	{ 015F }
30921	{ 0160 }	{ 0161 }
30922	{ 0162 }	{ 0163 }
30923	{ 0164 }	{ 0165 }
30924	{ 0168 }	{ 0169 }
30925	{ 016A }	{ 016B }
30926	{ 016C }	{ 016D }
30927	{ 016E }	{ 016F }
30928	{ 0170 }	{ 0171 }
30929	{ 0172 }	{ 0173 }

30930	{ 0174 }	{ 0175 }
30931	{ 0176 }	{ 0177 }
30932	{ 0178 }	{ 00FF }
30933	{ 0179 }	{ 017A }
30934	{ 017B }	{ 017C }
30935	{ 017D }	{ 017E }
30936	{ 01CD }	{ 01CE }
30937	{ 01CF }	{ 01D0 }
30938	{ 01D1 }	{ 01D2 }
30939	{ 01D3 }	{ 01D4 }
30940	{ 01E2 }	{ 01E3 }
30941	{ 01E6 }	{ 01E7 }
30942	{ 01E8 }	{ 01E9 }
30943	{ 01EA }	{ 01EB }
30944	{ 01F4 }	{ 01F5 }
30945	{ 0218 }	{ 0219 }
30946	{ 021A }	{ 021B }

Add T2 (Cyrillic) as this is doable using a classical \MakeUppercase approach.

30947	{ 0400 }	{ 0450 }
30948	{ 0401 }	{ 0451 }
30949	{ 0402 }	{ 0452 }
30950	{ 0403 }	{ 0453 }
30951	{ 0404 }	{ 0454 }
30952	{ 0405 }	{ 0455 }
30953	{ 0406 }	{ 0456 }
30954	{ 0407 }	{ 0457 }
30955	{ 0408 }	{ 0458 }
30956	{ 0409 }	{ 0459 }
30957	{ 040A }	{ 045A }
30958	{ 040B }	{ 045B }
30959	{ 040C }	{ 045C }
30960	{ 040D }	{ 045D }
30961	{ 040E }	{ 045E }
30962	{ 040F }	{ 045F }
30963	{ 0410 }	{ 0430 }
30964	{ 0411 }	{ 0431 }
30965	{ 0412 }	{ 0432 }
30966	{ 0413 }	{ 0433 }
30967	{ 0414 }	{ 0434 }
30968	{ 0415 }	{ 0435 }
30969	{ 0416 }	{ 0436 }
30970	{ 0417 }	{ 0437 }
30971	{ 0418 }	{ 0438 }
30972	{ 0419 }	{ 0439 }
30973	{ 041A }	{ 043A }
30974	{ 041B }	{ 043B }
30975	{ 041C }	{ 043C }
30976	{ 041D }	{ 043D }
30977	{ 041E }	{ 043E }
30978	{ 041F }	{ 043F }
30979	{ 0420 }	{ 0440 }
30980	{ 0421 }	{ 0441 }
30981	{ 0422 }	{ 0442 }
30982	{ 0423 }	{ 0443 }

30983	{ 0424 }	{ 0444 }
30984	{ 0425 }	{ 0445 }
30985	{ 0426 }	{ 0446 }
30986	{ 0427 }	{ 0447 }
30987	{ 0428 }	{ 0448 }
30988	{ 0429 }	{ 0449 }
30989	{ 042A }	{ 044A }
30990	{ 042B }	{ 044B }
30991	{ 042C }	{ 044C }
30992	{ 042D }	{ 044D }
30993	{ 042E }	{ 044E }
30994	{ 042F }	{ 044F }

Greek support: everything in the two-octet range.

30995	{ 0370 }	{ 0371 }
30996	{ 0372 }	{ 0373 }
30997	{ 0376 }	{ 0377 }
30998	{ 03FD }	{ 037B }
30999	{ 03FE }	{ 037C }
31000	{ 03FF }	{ 037D }
31001	{ 0386 }	{ 03AC }
31002	{ 0388 }	{ 03AD }
31003	{ 0389 }	{ 03AE }
31004	{ 038A }	{ 03AF }
31005	{ 0391 }	{ 03B1 }
31006	{ 0392 }	{ 03B2 }
31007	{ 0393 }	{ 03B3 }
31008	{ 0394 }	{ 03B4 }
31009	{ 0395 }	{ 03B5 }
31010	{ 0396 }	{ 03B6 }
31011	{ 0397 }	{ 03B7 }
31012	{ 0398 }	{ 03B8 }
31013	{ 0399 }	{ 03B9 }
31014	{ 039A }	{ 03BA }
31015	{ 039B }	{ 03BB }
31016	{ 039C }	{ 03BC }
31017	{ 039D }	{ 03BD }
31018	{ 039E }	{ 03BE }
31019	{ 039F }	{ 03BF }
31020	{ 03A0 }	{ 03C0 }
31021	{ 03A1 }	{ 03C1 }
31022	{ 03A3 }	{ 03C3 }
31023	{ 03A4 }	{ 03C4 }
31024	{ 03A5 }	{ 03C5 }
31025	{ 03A6 }	{ 03C6 }
31026	{ 03A7 }	{ 03C7 }
31027	{ 03A8 }	{ 03C8 }
31028	{ 03A9 }	{ 03C9 }
31029	{ 03AA }	{ 03CA }
31030	{ 03AB }	{ 03CB }
31031	{ 038C }	{ 03CC }
31032	{ 038E }	{ 03CD }
31033	{ 038F }	{ 03CE }
31034	{ 03CF }	{ 03D7 }
31035	{ 03D8 }	{ 03D9 }

```

31036      { 03DA } { 03DB }
31037      { 03DC } { 03DD }
31038      { 03DE } { 03DF }
31039      { 03E0 } { 03E1 }
31040      { 03E2 } { 03E3 }
31041      { 03E4 } { 03E5 }
31042      { 03E6 } { 03E7 }
31043      { 03E8 } { 03E9 }
31044      { 03EA } { 03EB }
31045      { 03EC } { 03ED }
31046      { 03EE } { 03EF }
31047      { 03F9 } { 03F2 }
31048      { 037F } { 03F3 }
31049      { 03F7 } { 03F8 }
31050      { 03FA } { 03FB }
31051      \q_recursion_tail ?
31052      \q_recursion_stop

```

Odds and ends for Greek; mainly symbols that are for compatibility, but also things like the terminal sigma. Almost all are uppercase mappings, but there is one that is not!

```

31053      \cs_set_protected:Npn \__text_tmp:w #1#2#3
31054      {
31055          \group_begin:
31056              \cs_set_protected:Npn \__text_tmp:w ##1##2##3##4##5##6##7##8
31057              {
31058                  \tl_const:cx
31059                  {
31060                      c__text_ #3 case_
31061                      \char_generate:nn {##1} { 12 }
31062                      \char_generate:nn {##2} { 12 }
31063                      _tl
31064                  }
31065                  {
31066                      \exp_after:wN \exp_after:wN \exp_after:wN
31067                      \exp_not:N \char_generate:nn {##5} { 13 }
31068                      \exp_after:wN \exp_after:wN \exp_after:wN
31069                      \exp_not:N \char_generate:nn {##6} { 13 }
31070                  }
31071              }
31072          \use:x
31073          {
31074              \__text_tmp:w
31075              \char_to_utfviii_bytes:n { "#1 }
31076              \char_to_utfviii_bytes:n { "#2 }
31077          }
31078          \group_end:
31079      }
31080      \__text_tmp:w { 0345 } { 0399 } { upper }
31081      \__text_tmp:w { 03C2 } { 03A3 } { upper }
31082      \__text_tmp:w { 03D0 } { 0392 } { upper }
31083      \__text_tmp:w { 03D1 } { 0398 } { upper }
31084      \__text_tmp:w { 03D5 } { 03A6 } { upper }
31085      \__text_tmp:w { 03D6 } { 03A0 } { upper }
31086      \__text_tmp:w { 03F0 } { 039A } { upper }

```

```

31087     \__text_tmp:w { 03F1 } { 03A1 } { upper }
31088     \__text_tmp:w { 03F4 } { 03B8 } { lower }
31089     \__text_tmp:w { 03F5 } { 0395 } { upper }

```

Odds and ends that are not simple one-to-one mappings. These are still two-octet code points.

```

31090     \cs_set_protected:Npn \__text_tmp:w #1#2#3
31091     {
31092         \group_begin:
31093         \cs_set_protected:Npn \__text_tmp:w ##1##2##3##4
31094         {
31095             \tl_const:cn
31096             {
31097                 c__text_ #3 case_
31098                 \char_generate:nn {##1} { 12 }
31099                 \char_generate:nn {##2} { 12 }
31100                 _tl
31101             }
31102             {#2}
31103         }
31104         \use:x
31105         { \__text_tmp:w \char_to_utfviii_bytes:n { "#1 } }
31106     \group_end:
31107 }
31108 \__text_tmp:w { 00DF } { SS } { upper }
31109 \__text_tmp:w { 00DF } { Ss } { title }
31110 \__text_tmp:w { 0131 } { I } { upper }

```

Greek support: the three-octet code points.

```

31111     \cs_set_protected:Npn \__text_tmp:nnnnn #1#2#3#4#5#6#7
31112     {
31113         \tl_const:cx
31114         {
31115             c__text_ #1 case_
31116             \char_generate:nn {#2} { 12 }
31117             \char_generate:nn {#3} { 12 }
31118             \char_generate:nn {#4} { 12 }
31119             _tl
31120         }
31121         {
31122             \exp_after:wN \exp_after:wN \exp_after:wN
31123             \exp_not:N \char_generate:nn {#5} { 13 }
31124             \exp_after:wN \exp_after:wN \exp_after:wN
31125             \exp_not:N \char_generate:nn {#6} { 13 }
31126             \exp_after:wN \exp_after:wN \exp_after:wN
31127             \exp_not:N \char_generate:nn {#7} { 13 }
31128         }
31129     }
31130     \cs_set_protected:Npn \__text_tmp:w #1#2#3#4#5#6#7#8
31131     {
31132         \tl_const:cx
31133         {
31134             c__text_lowercase_
31135             \char_generate:nn {#1} { 12 }
31136             \char_generate:nn {#2} { 12 }

```

```

31137         \char_generate:nn {#3} { 12 }
31138         _tl
31139     }
31140     {
31141         \exp_after:wN \exp_after:wN \exp_after:wN
31142         \exp_not:N \char_generate:nn {#5} { 13 }
31143         \exp_after:wN \exp_after:wN \exp_after:wN
31144         \exp_not:N \char_generate:nn {#6} { 13 }
31145         \exp_after:wN \exp_after:wN \exp_after:wN
31146         \exp_not:N \char_generate:nn {#7} { 13 }
31147     }
31148     \__text_tmp:nnnnnn { upper } {#5} {#6} {#7} {#1} {#2} {#3}
31149     \__text_tmp:nnnnnn { title } {#5} {#6} {#7} {#1} {#2} {#3}
31150 }
31151 \__text_loop:nn
31152 { 1F08 } { 1F00 }
31153 { 1F09 } { 1F01 }
31154 { 1F0A } { 1F02 }
31155 { 1F0B } { 1F03 }
31156 { 1F0C } { 1F04 }
31157 { 1F0D } { 1F05 }
31158 { 1F0E } { 1F06 }
31159 { 1F0F } { 1F07 }
31160 { 1F18 } { 1F10 }
31161 { 1F19 } { 1F11 }
31162 { 1F1A } { 1F12 }
31163 { 1F1B } { 1F13 }
31164 { 1F1C } { 1F14 }
31165 { 1F1D } { 1F15 }
31166 { 1F28 } { 1F20 }
31167 { 1F29 } { 1F21 }
31168 { 1F2A } { 1F22 }
31169 { 1F2B } { 1F23 }
31170 { 1F2C } { 1F24 }
31171 { 1F2D } { 1F25 }
31172 { 1F2E } { 1F26 }
31173 { 1F2F } { 1F27 }
31174 { 1F38 } { 1F30 }
31175 { 1F39 } { 1F31 }
31176 { 1F3A } { 1F32 }
31177 { 1F3B } { 1F33 }
31178 { 1F3C } { 1F34 }
31179 { 1F3D } { 1F35 }
31180 { 1F3E } { 1F36 }
31181 { 1F3F } { 1F37 }
31182 { 1F48 } { 1F40 }
31183 { 1F49 } { 1F41 }
31184 { 1F4A } { 1F42 }
31185 { 1F4B } { 1F43 }
31186 { 1F4C } { 1F44 }
31187 { 1F4D } { 1F45 }
31188 { 1F59 } { 1F51 }
31189 { 1F5B } { 1F53 }
31190 { 1F5D } { 1F55 }

```

31191	{ 1F5F }	{ 1F57 }
31192	{ 1F68 }	{ 1F60 }
31193	{ 1F69 }	{ 1F61 }
31194	{ 1F6A }	{ 1F62 }
31195	{ 1F6B }	{ 1F63 }
31196	{ 1F6C }	{ 1F64 }
31197	{ 1F6D }	{ 1F65 }
31198	{ 1F6E }	{ 1F66 }
31199	{ 1F6F }	{ 1F67 }
31200	{ 1FBA }	{ 1F70 }
31201	{ 1FBB }	{ 1F71 }
31202	{ 1FC8 }	{ 1F72 }
31203	{ 1FC9 }	{ 1F73 }
31204	{ 1FCA }	{ 1F74 }
31205	{ 1FCB }	{ 1F75 }
31206	{ 1FDA }	{ 1F76 }
31207	{ 1FDB }	{ 1F77 }
31208	{ 1FF8 }	{ 1F78 }
31209	{ 1FF9 }	{ 1F79 }
31210	{ 1FEA }	{ 1F7A }
31211	{ 1FEB }	{ 1F7B }
31212	{ 1FFA }	{ 1F7C }
31213	{ 1FFB }	{ 1F7D }
31214	{ 1F88 }	{ 1F80 }
31215	{ 1F89 }	{ 1F81 }
31216	{ 1F8A }	{ 1F82 }
31217	{ 1F8B }	{ 1F83 }
31218	{ 1F8C }	{ 1F84 }
31219	{ 1F8D }	{ 1F85 }
31220	{ 1F8E }	{ 1F86 }
31221	{ 1F8F }	{ 1F87 }
31222	{ 1F98 }	{ 1F90 }
31223	{ 1F99 }	{ 1F91 }
31224	{ 1F9A }	{ 1F92 }
31225	{ 1F9B }	{ 1F93 }
31226	{ 1F9C }	{ 1F94 }
31227	{ 1F9D }	{ 1F95 }
31228	{ 1F9E }	{ 1F96 }
31229	{ 1F9F }	{ 1F97 }
31230	{ 1FA8 }	{ 1FA0 }
31231	{ 1FA9 }	{ 1FA1 }
31232	{ 1FAA }	{ 1FA2 }
31233	{ 1FAB }	{ 1FA3 }
31234	{ 1FAC }	{ 1FA4 }
31235	{ 1FAD }	{ 1FA5 }
31236	{ 1FAE }	{ 1FA6 }
31237	{ 1FAF }	{ 1FA7 }
31238	{ 1FB8 }	{ 1FB0 }
31239	{ 1FB9 }	{ 1FB1 }
31240	{ 1FBC }	{ 1FB3 }
31241	{ 1FCC }	{ 1FC3 }
31242	{ 1FD8 }	{ 1FD0 }
31243	{ 1FD9 }	{ 1FD1 }
31244	{ 1FE8 }	{ 1FE0 }

```

31245     { 1FE9 } { 1FE1 }
31246     { 1FEC } { 1FE5 }
31247     { 1FFC } { 1FF3 }
31248     \q_recursion_tail ?
31249     \q_recursion_stop

```

One three-octet special case for Greek: it also moves to two-octets!

```

31250     \cs_set_protected:Npn \__text_tmp:w #1#2#3
31251     {
31252         \group_begin:
31253         \cs_set_protected:Npn \__text_tmp:w ##1##2##3##4##5##6##7##8
31254         {
31255             \tl_const:cx
31256             {
31257                 c__text_ #3 case_
31258                 \char_generate:nn {##1} { 12 }
31259                 \char_generate:nn {##2} { 12 }
31260                 \char_generate:nn {##3} { 12 }
31261                 _tl
31262             }
31263             {
31264                 \exp_after:wN \exp_after:wN \exp_after:wN
31265                 \exp_not:N \char_generate:nn {##5} { 13 }
31266                 \exp_after:wN \exp_after:wN \exp_after:wN
31267                 \exp_not:N \char_generate:nn {##6} { 13 }
31268             }
31269         }
31270         \use:x
31271         {
31272             \__text_tmp:w
31273             \char_to_utfviii_bytes:n { "#1 }
31274             \char_to_utfviii_bytes:n { "#2 }
31275         }
31276         \group_end:
31277     }
31278     \__text_tmp:w { 1FBE } { 0399 } { upper }
31279 }
31280 \group_end:

```

The (fixed) look-up mappings for letter-like control sequences.

```

31281 \group_begin:
31282 \cs_set_protected:Npn \__text_change_case_setup:NN #1#2
31283 {
31284     \quark_if_recursion_tail_stop:N #1
31285     \tl_const:cn { c__text_lowercase_ \token_to_str:N #1 _tl }
31286     { #2 }
31287     \tl_const:cn { c__text_uppercase_ \token_to_str:N #2 _tl }
31288     { #1 }
31289     \__text_change_case_setup:NN
31290 }
31291 \__text_change_case_setup:NN
31292 \AA \aa
31293 \AE \ae
31294 \DH \dh
31295 \DJ \dj

```



```

31296 \IJ \ij
31297 \L \l
31298 \NG \ng
31299 \O \o
31300 \OE \oe
31301 \SS \ss
31302 \TH \th
31303 \q_recursion_tail ?
31304 \q_recursion_stop
31305 \tl_const:cn { c__text_uppercase_ \token_to_str:N \i _tl } { I }
31306 \tl_const:cn { c__text_uppercase_ \token_to_str:N \j _tl } { J }
31307 \group_end:

```

To deal with possible encoding-specific extensions to \@uclclist, we check at the end of the preamble. This will therefore only apply to L^AT_EX 2_ε package mode.

```

31308 \tl_if_exist:NT \@expl@finalise@setup@@
31309 {
31310   \tl_gput_right:Nn \@expl@finalise@setup@@
31311   {
31312     \tl_gput_right:Nn \@kernel@after@begindocument
31313     {
31314       \group_begin:
31315       \cs_set_protected:Npn \__text_change_case_setup:Nn #1#2
31316       {
31317         \quark_if_recursion_tail_stop:N #1
31318         \tl_if_single_token:nT {#2}
31319         {
31320           \cs_if_exist:cF
31321           { c__text_uppercase_ \token_to_str:N #1 _tl }
31322           {
31323             \tl_const:cn
31324             { c__text_uppercase_ \token_to_str:N #1 _tl }
31325             { #2 }
31326           }
31327           \cs_if_exist:cF
31328           { c__text_lowercase_ \token_to_str:N #2 _tl }
31329           {
31330             \tl_const:cn
31331             { c__text_lowercase_ \token_to_str:N #2 _tl }
31332             { #1 }
31333           }
31334         }
31335         \__text_change_case_setup:Nn
31336       }
31337       \exp_after:wN \__text_change_case_setup:Nn \@uclclist
31338       \q_recursion_tail ?
31339       \q_recursion_stop
31340     }
31341   }
31342 }
31343 }
31344 </package>

```

Chapter 83

l3text-purify implementation

```
31345 <*package>
31346 <@@=text>
```

83.1 Purifying text

```
\_text_if_recursion_tail_stop:N
```

Functions to query recursion quarks.

```
31347 \_kernel_quark_new_test:N \_text_if_recursion_tail_stop:N
```

(End definition for _text_if_recursion_tail_stop:N.)

```
\text_purify:n
```

As in the other parts of the module, we start off with a standard “action” loop, with expansion applied up-front.

```
\_text_purify:n
```

```
\_text_purify_store:n
```

```
31348 \cs_new:Npn \text_purify:n #1
```

```
\_text_purify_store:nw
```

```
31349 {
```

```
\_text_purify_end:w
```

```
31350 \_kernel_exp_not:w \exp_after:wN
```

```
\_text_purify_loop:w
```

```
31351 {
```

```
\_text_purify_group:n
```

```
31352 \exp:w
```

```
\_text_purify_space:w
```

```
31353 \exp_args:Ne \_text_purify:n
```

```
\_text_purify_N_type:N
```

```
31354 { \text_expand:n {#1} }
```

```
31355 }
```

```
\_text_purify_N_type_aux:N
```

```
31356 }
```

```
\_text_purify_math_search:NNN
```

```
31357 \cs_new:Npn \_text_purify:n #1
```

```
\_text_purify_math_start:NNw
```

```
31358 {
```

```
\_text_purify_math_store:n
```

```
31359 \group_align_safe_begin:
```

```
\_text_purify_math_store:nw
```

```
31360 \_text_purify_loop:w #1
```

```
\_text_purify_math_end:w
```

```
31361 \q__text_recursion_tail \q__text_recursion_stop
```

```
\_text_purify_math_loop:NNw
```

```
31362 \_text_purify_result:n { }
```

```
\_text_purify_math_N_type:NNN
```

```
31363 }
```

```
\_text_purify_math_group:NNn
```

As for expansion, collect up the tokens for future use.

```
\_text_purify_math_space:NNw
```

```
31364 \cs_new:Npn \_text_purify_store:n #1
```

```
31365 { \_text_purify_store:nw {#1} }
```

```
\_text_purify_math_cmd:N
```

```
31366 \cs_new:Npn \_text_purify_store:nw #1#2 \_text_purify_result:n #3
```

```
\_text_purify_math_cmd:NN
```

```
31367 { #2 \_text_purify_result:n { #3 #1 } }
```

```
\_text_purify_math_cmd:Nn
```

```
31368 \cs_new:Npn \_text_purify_end:w #1 \_text_purify_result:n #2
```

```
\_text_purify_replace:N
```

```
31369 {
```

```
\_text_purify_replace:n
```

```
31370 \group_align_safe_end:
```

```
\_text_purify_expand:N
```

```
31371 \exp_end:
```

```
\_text_purify_protect:N
```

```
\_text_purify_encoding:N
```

```
\_text_purify_encoding_escape:NN
```

```

31372     #2
31373   }

```

The main loop is a standard “tl action”. Unlike the expansion or case changing, here any groups have to be run inline. Most of the business end is as before in the N-type token processing.

```

31374 \cs_new:Npn \__text_purify_loop:w #1 \q__text_recursion_stop
31375   {
31376     \tl_if_head_is_N_type:nTF {#1}
31377       { \__text_purify_N_type:N }
31378       {
31379         \tl_if_head_is_group:nTF {#1}
31380           { \__text_purify_group:n }
31381           { \__text_purify_space:w }
31382       }
31383     #1 \q__text_recursion_stop
31384   }
31385 \cs_new:Npn \__text_purify_group:n #1 { \__text_purify_loop:w #1 }
31386 \exp_last_unbraced:NNo \cs_new:Npn \__text_purify_space:w \c_space_tl
31387   {
31388     \__text_purify_store:n { ~ }
31389     \__text_purify_loop:w
31390   }

```

The first part of handling math mode is exactly the same as in the other functions: look for a start-of-math mode token and if found start a new loop tracking the closing token.

```

31391 \cs_new:Npn \__text_purify_N_type:N #1
31392   {
31393     \__text_if_recursion_tail_stop_do:Nn #1 { \__text_purify_end:w }
31394     \__text_purify_N_type_aux:N #1
31395   }
31396 \cs_new:Npn \__text_purify_N_type_aux:N #1
31397   {
31398     \exp_after:wN \__text_purify_math_search:NNN
31399     \exp_after:wN #1 \l_text_math_delims_tl
31400     \q__text_recursion_tail ?
31401     \q__text_recursion_stop
31402   }
31403 \cs_new:Npn \__text_purify_math_search:NNN #1#2#3
31404   {
31405     \__text_if_recursion_tail_stop_do:Nn #2
31406       { \__text_purify_math_cmd:N #1 }
31407     \token_if_eq_meaning:NNTF #1 #2
31408       {
31409         \__text_use_i_delimit_by_q_recursion_stop:nw
31410         { \__text_purify_math_start:NNw #2 #3 }
31411       }
31412     { \__text_purify_math_search:NNN #1 }
31413   }
31414 \cs_new:Npn \__text_purify_math_start:NNw #1#2#3 \q__text_recursion_stop
31415   {
31416     \__text_purify_math_loop:NNw #1#2#3 \q__text_recursion_stop
31417     \__text_purify_math_result:n { }
31418   }
31419 \cs_new:Npn \__text_purify_math_store:n #1

```

```

31420 { \_text_purify_math_store:nw {#1} }
31421 \cs_new:Npn \_text_purify_math_store:nw #1#2 \_text_purify_math_result:n #3
31422 { #2 \_text_purify_math_result:n { #3 #1 } }
31423 \cs_new:Npn \_text_purify_math_end:w #1 \_text_purify_math_result:n #2
31424 {
31425   \_text_purify_store:n { $ #2 $ }
31426   \_text_purify_loop:w #1
31427 }
31428 \cs_new:Npn \_text_purify_math_stop:Nw #1 \_text_purify_math_result:n #2
31429 {
31430   \_text_purify_store:n {#1#2}
31431   \_text_purify_end:w
31432 }
31433 \cs_new:Npn \_text_purify_math_loop:NNw #1#2#3 \q_text_recursion_stop
31434 {
31435   \tl_if_head_is_N_type:nTF {#3}
31436   { \_text_purify_math_N_type:NNN }
31437   {
31438     \tl_if_head_is_group:nTF {#3}
31439     { \_text_purify_math_group:NNn }
31440     { \_text_purify_math_space:NNw }
31441   }
31442   #1#2#3 \q_text_recursion_stop
31443 }
31444 \cs_new:Npn \_text_purify_math_N_type:NNN #1#2#3
31445 {
31446   \_text_if_recursion_tail_stop_do:Nn #3
31447   { \_text_purify_math_stop:Nw #1 }
31448   \token_if_eq_meaning:NNTF #3 #2
31449   { \_text_purify_math_end:w }
31450   {
31451     \_text_purify_math_store:n {#3}
31452     \_text_purify_math_loop:NNw #1#2
31453   }
31454 }
31455 \cs_new:Npn \_text_purify_math_group:NNn #1#2#3
31456 {
31457   \_text_purify_math_store:n { {#3} }
31458   \_text_purify_math_loop:NNw #1#2
31459 }
31460 \exp_after:wN \cs_new:Npn \exp_after:wN \_text_purify_math_space:NNw
31461 \exp_after:wN # \exp_after:wN 1
31462 \exp_after:wN # \exp_after:wN 2 \c_space_tl
31463 {
31464   \_text_purify_math_store:n { ~ }
31465   \_text_purify_math_loop:NNw #1#2
31466 }

```

Then handle math mode as an argument: same outcomes, different input syntax.

```

31467 \cs_new:Npn \_text_purify_math_cmd:N #1
31468 {
31469   \exp_after:wN \_text_purify_math_cmd:NN \exp_after:wN #1
31470   \l_text_math_arg_tl \q_text_recursion_tail \q_text_recursion_stop
31471 }
31472 \cs_new:Npn \_text_purify_math_cmd:NN #1#2

```

```

31473 {
31474   \__text_if_recursion_tail_stop_do:Nn #2
31475   { \__text_purify_replace:N #1 }
31476   \cs_if_eq:NNTF #2 #1
31477   {
31478     \__text_use_i_delimit_by_q_recursion_stop:nw
31479     { \__text_purify_math_cmd:n }
31480   }
31481   { \__text_purify_math_cmd:NN #1 }
31482 }
31483 \cs_new:Npn \__text_purify_math_cmd:n #1
31484 { \__text_purify_math_end:w \__text_purify_math_result:n {#1} }

```

For N-type tokens, we first look for a string-context replacement before anything else: this can therefore cover anything. Assuming we don't find one, check to see if we can expand control sequences: if not, they have to be dropped. We also allow for L^AT_EX 2_ε \protect: there's an assumption that we don't have \protect { \oops } or similar, but that's also in the expansion code and seems like a reasonable balance.

```

31485 \cs_new:Npn \__text_purify_replace:N #1
31486 {
31487   \bool_lazy_and:nnTF
31488   { \cs_if_exist_p:c { l__text_purify_ \token_to_str:N #1 _tl } }
31489   {
31490     \bool_lazy_or_p:nn
31491     { \token_if_cs_p:N #1 }
31492     { \token_if_active_p:N #1 }
31493   }
31494   {
31495     \exp_args:Nv \__text_purify_replace:n
31496     { l__text_purify_ \token_to_str:N #1 _tl }
31497   }
31498   {
31499     \token_if_cs:NTF #1
31500     { \__text_purify_expand:N #1 }
31501     {
31502       \exp_args:Ne \__text_purify_store:n
31503       { \__text_token_to_explicit:N #1 }
31504       \__text_purify_loop:w
31505     }
31506   }
31507 }
31508 \cs_new:Npn \__text_purify_replace:n #1 { \__text_purify_loop:w #1 }
31509 \cs_new:Npn \__text_purify_expand:N #1
31510 {
31511   \str_if_eq:nnTF {#1} { \protect }
31512   { \__text_purify_protect:N }
31513   { \__text_purify_encoding:N #1 }
31514 }
31515 \cs_new:Npn \__text_purify_protect:N #1
31516 {
31517   \__text_if_recursion_tail_stop_do:Nn #1 { \__text_purify_end:w }
31518   \__text_purify_loop:w
31519 }

```

Handle encoding commands, as detailed for expansion.

```

31520 \cs_new:Npn \__text_purify_encoding:N #1
31521 {
31522   \bool_lazy_or:nnTF
31523     { \cs_if_eq_p:NN #1 \@current@cmd }
31524     { \cs_if_eq_p:NN #1 \@changed@cmd }
31525     { \__text_purify_encoding_escape:NN }
31526     {
31527       \__text_if_expandable:NTF #1
31528         { \exp_after:wN \__text_purify_loop:w #1 }
31529         { \__text_purify_loop:w }
31530     }
31531 }
31532 \cs_new:Npn \__text_purify_encoding_escape:NN #1#2
31533 {
31534   \__text_purify_store:n {#1}
31535   \__text_purify_loop:w
31536 }

```

(End definition for `\text_purify:n` and others. This function is documented on page 267.)

`\text_declare_purify_equivalent:Nn`
`\text_declare_purify_equivalent:Nx`

```

31537 \cs_new_protected:Npn \text_declare_purify_equivalent:Nn #1#2
31538 {
31539   \tl_clear_new:c { l__text_purify_ \token_to_str:N #1 _tl }
31540   \tl_set:cn { l__text_purify_ \token_to_str:N #1 _tl } {#2}
31541 }
31542 \cs_generate_variant:Nn \text_declare_purify_equivalent:Nn { Nx }

```

(End definition for `\text_declare_purify_equivalent:Nn`. This function is documented on page 267.)

Now pre-define a range of standard commands that need dedicated definitions in purified text. First handle font-related stuff: all of this needs to be disabled.

```

31543 \tl_map_inline:nn
31544 {
31545   \fontencoding
31546   \fontfamily
31547   \fontseries
31548   \fontshape
31549 }
31550 { \text_declare_purify_equivalent:Nn #1 { \use_none:n } }
31551 \text_declare_purify_equivalent:Nn \fontsize { \use_none:nn }
31552 \text_declare_purify_equivalent:Nn \selectfont { }
31553 \text_declare_purify_equivalent:Nn \usefont { \use_none:nnnn }
31554 \tl_map_inline:nn
31555 {
31556   \emph
31557   \text
31558   \textnormal
31559   \textrm
31560   \textsf
31561   \texttt
31562   \textbf
31563   \textmd
31564   \textit
31565   \textsl

```

```

31566 \textup
31567 \textsc
31568 \textulc
31569 }
31570 { \text_declare_purify_equivalent:Nn #1 { \use:n } }
31571 \tl_map_inline:nn
31572 {
31573 \normalfont
31574 \rmfamily
31575 \sffamily
31576 \ttfamily
31577 \bfseries
31578 \mdseries
31579 \itshape
31580 \scshape
31581 \slshape
31582 \upshape
31583 \em
31584 \Huge
31585 \LARGE
31586 \Large
31587 \footnotesize
31588 \huge
31589 \large
31590 \normalsize
31591 \scriptsize
31592 \small
31593 \tiny
31594 }
31595 { \text_declare_purify_equivalent:Nn #1 { } }
31596 \exp_args:Nc \text_declare_purify_equivalent:Nn
31597 { @protected@testopt } { \use_none:nnn }

```

Environments have to be handled by pure expansion.

`__text_end_env:n`

```

31598 \text_declare_purify_equivalent:Nn \begin { \use:c }
31599 \text_declare_purify_equivalent:Nn \end { \__text_end_env:n }
31600 \cs_new:Npn \__text_end_env:n #1 { \cs:w end #1 \cs_end: }

```

(End definition for `__text_end_env:n`.)

Some common symbols and similar ideas.

```

31601 \text_declare_purify_equivalent:Nn \ { }
31602 \tl_map_inline:nn
31603 { \{ \} \# \$ \% \_ }
31604 { \text_declare_purify_equivalent:Nx #1 { \cs_to_str:N #1 } }

```

Cross-referencing.

```

31605 \text_declare_purify_equivalent:Nn \label { \use_none:n }

```

Spaces.

```

31606 \group_begin:
31607 \char_set_catcode_active:N \~
31608 \use:n
31609 {

```

```

31610 \group_end:
31611 \text_declare_purify_equivalent:Nx ~ { \c_space_tl }
31612 }
31613 \text_declare_purify_equivalent:Nn \nobreakspace { ~ }
31614 \text_declare_purify_equivalent:Nn \ { ~ }
31615 \text_declare_purify_equivalent:Nn \, { ~ }

```

83.2 Accent and letter-like data for purifying text

In contrast to case changing, both 8-bit and Unicode engines need information for text purification to handle accents and letter-like functions: these all need to be removed. However, the results are of course engine-dependent.

For the letter-like commands, life is relatively easy: they are all simply added as standard exceptions. The only oddity is `\SS`, which gets converted to two letters. (At some stage an alternative version can presumably be added to `babel` or similar.)

```

31616 \bool_lazy_or:nnTF
31617 { \sys_if_engine_luatex_p: }
31618 { \sys_if_engine_xetex_p: }
31619 {
31620   \cs_set_protected:Npn \__text_loop:Nn #1#2
31621   {
31622     \quark_if_recursion_tail_stop:N #1
31623     \text_declare_purify_equivalent:Nx #1
31624     {
31625       \char_generate:nn { "#2 }
31626       { \char_value_catcode:n { "#2 } }
31627     }
31628     \__text_loop:Nn
31629   }
31630 }
31631 {
31632   \cs_set_protected:Npn \__text_loop:Nn #1#2
31633   {
31634     \quark_if_recursion_tail_stop:N #1
31635     \text_declare_purify_equivalent:Nx #1
31636     {
31637       \exp_args:Ne \__text_tmp:n
31638       { \char_to_utfviii_bytes:n { "#2 } }
31639     }
31640     \__text_loop:Nn
31641   }
31642   \cs_set:Npn \__text_tmp:n #1 { \__text_tmp:nnnn #1 }
31643   \cs_set:Npn \__text_tmp:nnnn #1#2#3#4
31644   {
31645     \exp_after:wN \exp_after:wN \exp_after:wN
31646     \exp_not:N \char_generate:nn {#1} { 13 }
31647     \exp_after:wN \exp_after:wN \exp_after:wN
31648     \exp_not:N \char_generate:nn {#2} { 13 }
31649   }
31650 }
31651 \__text_loop:Nn
31652 \AA { 00C5 }

```



```

31653 \AE { 00C6 }
31654 \DH { 00D0 }
31655 \DJ { 0110 }
31656 \IJ { 0132 }
31657 \L { 0141 }
31658 \NG { 014A }
31659 \O { 00D8 }
31660 \OE { 0152 }
31661 \TH { 00DE }
31662 \aa { 00E5 }
31663 \ae { 00E6 }
31664 \dh { 00F0 }
31665 \dj { 0111 }
31666 \i { 0131 }
31667 \j { 0237 }
31668 \ij { 0132 }
31669 \l { 0142 }
31670 \ng { 014B }
31671 \o { 00F8 }
31672 \oe { 0153 }
31673 \ss { 00DF }
31674 \th { 00FE }
31675 \q_recursion_tail ?
31676 \q_recursion_stop
31677 \text_declare_purify_equivalent:Nn \SS { SS }

```

__text_purify_accent:NN Accent LICR handling is a little more complex. Accents may exist as pre-composed codepoints or as independent glyphs. The former are all saved as single token lists, whilst for the latter the combining accent needs to be re-ordered compared to the character it applies to.

```

31678 \cs_new:Npn \__text_purify_accent:NN #1#2
31679 {
31680   \cs_if_exist:cTF
31681   { c__text_purify_ \token_to_str:N #1 _ \token_to_str:N #2 _tl }
31682   {
31683     \exp_not:v
31684     { c__text_purify_ \token_to_str:N #1 _ \token_to_str:N #2 _tl }
31685   }
31686   {
31687     \exp_not:n {#2}
31688     \exp_not:v { c__text_purify_ \token_to_str:N #1 _tl }
31689   }
31690 }
31691 \tl_map_inline:Nn \l_text_accents_tl
31692 { \text_declare_purify_equivalent:Nn #1 { \__text_purify_accent:NN #1 } }

```

First set up the combining accents.

```

31693 \group_begin:
31694 \cs_set_protected:Npn \__text_loop:Nn #1#2
31695 {
31696   \quark_if_recursion_tail_stop:N #1
31697   \tl_const:cx { c__text_purify_ \token_to_str:N #1 _tl }
31698   { \__text_tmp:n {#2} }
31699   \__text_loop:Nn

```

```

31700     }
31701     \bool_lazy_or:nnTF
31702     { \sys_if_engine luatex_p: }
31703     { \sys_if_engine xetex_p: }
31704     {
31705         \cs_set:Npn \__text_tmp:n #1
31706         {
31707             \char_generate:nn { "#1 }
31708             { \char_value_catcode:n { "#1 } }
31709         }
31710     }
31711     {
31712         \cs_set:Npn \__text_tmp:n #1
31713         {
31714             \exp_args:Ne \__text_tmp_aux:n
31715             { \char_to_utfviii_bytes:n { "#1 } }
31716         }
31717         \cs_set:Npn \__text_tmp_aux:n #1 { \__text_tmp:nnnn #1 }
31718         \cs_set:Npn \__text_tmp:nnnn #1#2#3#4
31719         {
31720             \exp_after:wN \exp_after:wN \exp_after:wN
31721             \exp_not:N \char_generate:nn {#1} { 13 }
31722             \exp_after:wN \exp_after:wN \exp_after:wN
31723             \exp_not:N \char_generate:nn {#2} { 13 }
31724         }
31725     }
31726     \__text_loop:Nn
31727     \‘ { 0300 }
31728     \’ { 0301 }
31729     \^ { 0302 }
31730     \~ { 0303 }
31731     \= { 0304 }
31732     \u { 0306 }
31733     \. { 0307 }
31734     \" { 0308 }
31735     \r { 030A }
31736     \H { 030B }
31737     \v { 030C }
31738     \d { 0323 }
31739     \c { 0327 }
31740     \k { 0328 }
31741     \b { 0331 }
31742     \t { 0361 }
31743     \q_recursion_tail { }
31744     \q_recursion_stop

```

Now we handle the pre-composed accents: the list here is taken from `puenc.def`. All of the precomposed cases take a single letter as their second argument. We do not try to cover the case where an accent is added to a “real” dotless-i or -j, or a æ/Æ. Rather, we assume that if the UTF-8 character is used, it will have the real accent character too.

```

31745     \cs_set_protected:Npn \__text_loop:NNn #1#2#3
31746     {
31747         \quark_if_recursion_tail_stop:N #1
31748         \tl_const:cx

```

```

31749         { c__text_purify_ \token_to_str:N #1 _ \token_to_str:N #2 _tl }
31750         { \__text_tmp:n {#3} }
31751     \__text_loop:NNn
31752 }
31753 \__text_loop:NNn
31754 \‘ A { 00C0 }
31755 \’ A { 00C1 }
31756 \^ A { 00C2 }
31757 \~ A { 00C3 }
31758 \" A { 00C4 }
31759 \r A { 00C5 }
31760 \c C { 00C7 }
31761 \‘ E { 00C8 }
31762 \’ E { 00C9 }
31763 \^ E { 00CA }
31764 \" E { 00CB }
31765 \‘ I { 00CC }
31766 \’ I { 00CD }
31767 \^ I { 00CE }
31768 \" I { 00CF }
31769 \~ N { 00D1 }
31770 \‘ O { 00D2 }
31771 \’ O { 00D3 }
31772 \^ O { 00D4 }
31773 \~ O { 00D5 }
31774 \" O { 00D6 }
31775 \‘ U { 00D9 }
31776 \’ U { 00DA }
31777 \^ U { 00DB }
31778 \" U { 00DC }
31779 \’ Y { 00DD }
31780 \‘ a { 00E0 }
31781 \’ a { 00E1 }
31782 \^ a { 00E2 }
31783 \~ a { 00E3 }
31784 \" a { 00E4 }
31785 \r a { 00E5 }
31786 \c c { 00E7 }
31787 \‘ e { 00E8 }
31788 \’ e { 00E9 }
31789 \^ e { 00EA }
31790 \" e { 00EB }
31791 \‘ i { 00EC }
31792 \‘ \i { 00EC }
31793 \’ i { 00ED }
31794 \’ \i { 00ED }
31795 \^ i { 00EE }
31796 \^ \i { 00EE }
31797 \" i { 00EF }
31798 \" \i { 00EF }
31799 \~ n { 00F1 }
31800 \‘ o { 00F2 }
31801 \’ o { 00F3 }
31802 \^ o { 00F4 }

```

31803	\~ o	{ 00F5 }
31804	\" o	{ 00F6 }
31805	\‘ u	{ 00F9 }
31806	\’ u	{ 00FA }
31807	\^ u	{ 00FB }
31808	\" u	{ 00FC }
31809	\’ y	{ 00FD }
31810	\" y	{ 00FF }
31811	\= A	{ 0100 }
31812	\= a	{ 0101 }
31813	\u A	{ 0102 }
31814	\u a	{ 0103 }
31815	\k A	{ 0104 }
31816	\k a	{ 0105 }
31817	\’ C	{ 0106 }
31818	\’ c	{ 0107 }
31819	\^ C	{ 0108 }
31820	\^ c	{ 0109 }
31821	\. C	{ 010A }
31822	\. c	{ 010B }
31823	\v C	{ 010C }
31824	\v c	{ 010D }
31825	\v D	{ 010E }
31826	\v d	{ 010F }
31827	\= E	{ 0112 }
31828	\= e	{ 0113 }
31829	\u E	{ 0114 }
31830	\u e	{ 0115 }
31831	\. E	{ 0116 }
31832	\. e	{ 0117 }
31833	\k E	{ 0118 }
31834	\k e	{ 0119 }
31835	\v E	{ 011A }
31836	\v e	{ 011B }
31837	\^ G	{ 011C }
31838	\^ g	{ 011D }
31839	\u G	{ 011E }
31840	\u g	{ 011F }
31841	\. G	{ 0120 }
31842	\. g	{ 0121 }
31843	\c G	{ 0122 }
31844	\c g	{ 0123 }
31845	\^ H	{ 0124 }
31846	\^ h	{ 0125 }
31847	\~ I	{ 0128 }
31848	\~ i	{ 0129 }
31849	\~ \i	{ 0129 }
31850	\= I	{ 012A }
31851	\= i	{ 012B }
31852	\= \i	{ 012B }
31853	\u I	{ 012C }
31854	\u i	{ 012D }
31855	\u \i	{ 012D }
31856	\k I	{ 012E }

31857	\k i	{ 012F }
31858	\k \i	{ 012F }
31859	\. I	{ 0130 }
31860	\^ J	{ 0134 }
31861	\^ j	{ 0135 }
31862	\^ \j	{ 0135 }
31863	\c K	{ 0136 }
31864	\c k	{ 0137 }
31865	\' L	{ 0139 }
31866	\' l	{ 013A }
31867	\c L	{ 013B }
31868	\c l	{ 013C }
31869	\v L	{ 013D }
31870	\v l	{ 013E }
31871	\. L	{ 013F }
31872	\. l	{ 0140 }
31873	\' N	{ 0143 }
31874	\' n	{ 0144 }
31875	\c N	{ 0145 }
31876	\c n	{ 0146 }
31877	\v N	{ 0147 }
31878	\v n	{ 0148 }
31879	\= O	{ 014C }
31880	\= o	{ 014D }
31881	\u O	{ 014E }
31882	\u o	{ 014F }
31883	\H O	{ 0150 }
31884	\H o	{ 0151 }
31885	\' R	{ 0154 }
31886	\' r	{ 0155 }
31887	\c R	{ 0156 }
31888	\c r	{ 0157 }
31889	\v R	{ 0158 }
31890	\v r	{ 0159 }
31891	\' S	{ 015A }
31892	\' s	{ 015B }
31893	\^ S	{ 015C }
31894	\^ s	{ 015D }
31895	\c S	{ 015E }
31896	\c s	{ 015F }
31897	\v S	{ 0160 }
31898	\v s	{ 0161 }
31899	\c T	{ 0162 }
31900	\c t	{ 0163 }
31901	\v T	{ 0164 }
31902	\v t	{ 0165 }
31903	\~ U	{ 0168 }
31904	\~ u	{ 0169 }
31905	\= U	{ 016A }
31906	\= u	{ 016B }
31907	\u U	{ 016C }
31908	\u u	{ 016D }
31909	\r U	{ 016E }
31910	\r u	{ 016F }

```

31911      \H U   { 0170 }
31912      \H u   { 0171 }
31913      \k U   { 0172 }
31914      \k u   { 0173 }
31915      \^ W   { 0174 }
31916      \^ w   { 0175 }
31917      \^ Y   { 0176 }
31918      \^ y   { 0177 }
31919      \" Y   { 0178 }
31920      \' Z   { 0179 }
31921      \' z   { 017A }
31922      \. Z   { 017B }
31923      \. z   { 017C }
31924      \v Z   { 017D }
31925      \v z   { 017E }
31926      \v A   { 01CD }
31927      \v a   { 01CE }
31928      \v I   { 01CF }
31929      \v \i  { 01D0 }
31930      \v i   { 01D0 }
31931      \v O   { 01D1 }
31932      \v o   { 01D2 }
31933      \v U   { 01D3 }
31934      \v u   { 01D4 }
31935      \v G   { 01E6 }
31936      \v g   { 01E7 }
31937      \v K   { 01E8 }
31938      \v k   { 01E9 }
31939      \k O   { 01EA }
31940      \k o   { 01EB }
31941      \v \j  { 01F0 }
31942      \v j   { 01F0 }
31943      \' G   { 01F4 }
31944      \' g   { 01F5 }
31945      \' N   { 01F8 }
31946      \' n   { 01F9 }
31947      \' \AE { 01FC }
31948      \' \ae { 01FD }
31949      \' \O  { 01FE }
31950      \' \o  { 01FF }
31951      \v H   { 021E }
31952      \v h   { 021F }
31953      \. A   { 0226 }
31954      \. a   { 0227 }
31955      \c E   { 0228 }
31956      \c e   { 0229 }
31957      \. O   { 022E }
31958      \. o   { 022F }
31959      \= Y   { 0232 }
31960      \= y   { 0233 }
31961      \q_recursion_tail ? { }
31962      \q_recursion_stop
31963 \group_end:

```

(End definition for _text_purify_accent:NM.)

31964 \langle /package \rangle

Chapter 84

l3box implementation

```
31965 <*package>
31966 <@@=box>
```

84.1 Support code

`__box_dim_eval:w` Evaluating a dimension expression expandably. The only difference with `\dim_eval:n` is the lack of `\dim_use:N`, to produce an internal dimension rather than expand it into characters.

```
31967 \cs_new_eq:NN \__box_dim_eval:w \tex_dimexpr:D
31968 \cs_new:Npn \__box_dim_eval:n #1
31969 { \__box_dim_eval:w #1 \scan_stop: }
```

(End definition for __box_dim_eval:w and __box_dim_eval:n.)

`__kernel_kern:n` We need kerns in a few places. At present, we don't have a module for this concept, so it goes in at first use: here. The idea is to avoid repeated use of the bare primitive.

```
31970 \cs_new_protected:Npn \__kernel_kern:n #1
31971 { \tex_kern:D \__box_dim_eval:n {#1} }
```

(End definition for __kernel_kern:n.)

84.2 Creating and initialising boxes

The following test files are used for this code: m3box001.lvt.

`\box_new:N` Defining a new `<box>` register: remember that box 255 is not generally available.

```
\box_new:c
31972 \cs_new_protected:Npn \box_new:N #1
31973 {
31974   \__kernel_chk_if_free_cs:N #1
31975   \cs:w newbox \cs_end: #1
31976 }
31977 \cs_generate_variant:Nn \box_new:N { c }
```


Clear a $\langle box \rangle$ register.

```

31978 \cs_new_protected:Npn \box_clear:N #1
\box_clear:N { \box_set_eq:NN #1 \c_empty_box }
31979
\box_clear:c 31980 \cs_new_protected:Npn \box_gclear:N #1
\box_gclear:N { \box_gset_eq:NN #1 \c_empty_box }
31981
\box_gclear:c 31982 \cs_generate_variant:Nn \box_clear:N { c }
31983 \cs_generate_variant:Nn \box_gclear:N { c }

```

Clear or new.

```

31984 \cs_new_protected:Npn \box_clear_new:N #1
\box_clear_new:N { \box_if_exist:NTF #1 { \box_clear:N #1 } { \box_new:N #1 } }
31985
\box_clear_new:c 31986 \cs_new_protected:Npn \box_gclear_new:N #1
\box_gclear_new:N { \box_if_exist:NTF #1 { \box_gclear:N #1 } { \box_new:N #1 } }
31987
\box_gclear_new:c 31988 \cs_generate_variant:Nn \box_clear_new:N { c }
31989 \cs_generate_variant:Nn \box_gclear_new:N { c }

```

Assigning the contents of a box to be another box.

```

31990 \cs_new_protected:Npn \box_set_eq:NN #1#2
\box_set_eq:NN { \tex_setbox:D #1 \tex_copy:D #2 }
31991
\box_set_eq:cN 31992 \cs_new_protected:Npn \box_gset_eq:NN #1#2
\box_set_eq:Nc 31993 { \tex_global:D \tex_setbox:D #1 \tex_copy:D #2 }
\box_set_eq:cc 31994 \cs_generate_variant:Nn \box_set_eq:NN { c , Nc , cc }
\box_gset_eq:NN 31995 \cs_generate_variant:Nn \box_gset_eq:NN { c , Nc , cc }

```

Assigning the contents of a box to be another box, then drops the original box.

```

31996 \cs_new_protected:Npn \box_set_eq_drop:NN #1#2
\box_set_eq_drop:NN { \tex_setbox:D #1 \tex_box:D #2 }
31997
\box_set_eq_drop:cN 31998 \cs_new_protected:Npn \box_gset_eq_drop:NN #1#2
\box_set_eq_drop:Nc 31999 { \tex_global:D \tex_setbox:D #1 \tex_box:D #2 }
\box_set_eq_drop:cc 32000 \cs_generate_variant:Nn \box_set_eq_drop:NN { c , Nc , cc }
\box_gset_eq_drop:NN 32001 \cs_generate_variant:Nn \box_gset_eq_drop:NN { c , Nc , cc }
\box_gset_eq_drop:cN
\box_gset_eq_drop:Nc
\box_gset_eq_drop:cc
\box_if_exist:N 32002 \prg_new_eq_conditional:NNn \box_if_exist:N \cs_if_exist:N
\box_if_exist:N { TF , T , F , p }
32003
\box_if_exist:p:c 32004 \prg_new_eq_conditional:NNn \box_if_exist:c \cs_if_exist:c
\box_if_exist:N\TF 32005 { TF , T , F , p }
\box_if_exist:c\TF

```

Copies of the cs functions defined in l3basics.

```

32002 \prg_new_eq_conditional:NNn \box_if_exist:N \cs_if_exist:N
32003 { TF , T , F , p }
32004 \prg_new_eq_conditional:NNn \box_if_exist:c \cs_if_exist:c
32005 { TF , T , F , p }

```

84.3 Measuring and setting box dimensions

Accessing the height, depth, and width of a $\langle box \rangle$ register.

```

32006 \cs_new_eq:NN \box_ht:N \tex_ht:D
\box_ht:N 32007 \cs_new_eq:NN \box_dp:N \tex_dp:D
\box_ht:c 32008 \cs_new_eq:NN \box_wd:N \tex_wd:D
\box_dp:N 32009 \cs_generate_variant:Nn \box_ht:N { c }
\box_dp:c 32010 \cs_generate_variant:Nn \box_dp:N { c }
\box_wd:N 32011 \cs_generate_variant:Nn \box_wd:N { c }
\box_wd:c

```

The $\backslash\text{box_ht:N}$ and $\backslash\text{box_dp:N}$ primitives do not expand but rather are suitable for use after $\backslash\text{the}$ or inside dimension expressions. Here we obtain the same behaviour by using $\backslash_\text{box_dim_eval:n}$ (basically $\backslash\text{dimexpr}$) rather than $\backslash\text{dim_eval:n}$ (basically $\backslash\text{the dimexpr}$).

```

32012 \cs_new_protected:Npn \box_ht_plus_dp:N #1
32013 { \__box_dim_eval:n { \box_ht:N #1 + \box_dp:N #1 } }
32014 \cs_generate_variant:Nn \box_ht_plus_dp:N { c }

```

Setting the size whilst respecting local scope requires copying; the same issue does not come up when working globally. When debugging, the dimension expression #2 is surrounded by parentheses to catch early termination.

```

\box_set_ht:Nn
\box_set_ht:cn
\box_gset_ht:Nn
\box_gset_ht:cn
\box_set_dp:Nn
\box_set_dp:cn
\box_gset_dp:Nn
\box_gset_dp:cn
\box_set_wd:Nn
\box_set_wd:cn
\box_gset_wd:Nn
\box_gset_wd:cn
32015 \cs_new_protected:Npn \box_set_dp:Nn #1#2
32016 {
32017   \tex_setbox:D #1 = \tex_copy:D #1
32018   \box_dp:N #1 \__box_dim_eval:n {#2}
32019 }
32020 \cs_generate_variant:Nn \box_set_dp:Nn { c }
32021 \cs_new_protected:Npn \box_gset_dp:Nn #1#2
32022 { \box_dp:N #1 \__box_dim_eval:n {#2} }
32023 \cs_generate_variant:Nn \box_gset_dp:Nn { c }
32024 \cs_new_protected:Npn \box_set_ht:Nn #1#2
32025 {
32026   \tex_setbox:D #1 = \tex_copy:D #1
32027   \box_ht:N #1 \__box_dim_eval:n {#2}
32028 }
32029 \cs_generate_variant:Nn \box_set_ht:Nn { c }
32030 \cs_new_protected:Npn \box_gset_ht:Nn #1#2
32031 { \box_ht:N #1 \__box_dim_eval:n {#2} }
32032 \cs_generate_variant:Nn \box_gset_ht:Nn { c }
32033 \cs_new_protected:Npn \box_set_wd:Nn #1#2
32034 {
32035   \tex_setbox:D #1 = \tex_copy:D #1
32036   \box_wd:N #1 \__box_dim_eval:n {#2}
32037 }
32038 \cs_generate_variant:Nn \box_set_wd:Nn { c }
32039 \cs_new_protected:Npn \box_gset_wd:Nn #1#2
32040 { \box_wd:N #1 \__box_dim_eval:n {#2} }
32041 \cs_generate_variant:Nn \box_gset_wd:Nn { c }

```

84.4 Using boxes

Using a $\langle box \rangle$. These are just \TeX primitives with meaningful names.

```

\box_use_drop:N
\box_use_drop:c
\box_use:N
\box_use:c
32042 \cs_new_eq:NN \box_use_drop:N \tex_box:D
32043 \cs_new_eq:NN \box_use:N \tex_copy:D
32044 \cs_generate_variant:Nn \box_use_drop:N { c }
32045 \cs_generate_variant:Nn \box_use:N { c }

```

Move box material in different directions. When debugging, the dimension expression #1 is surrounded by parentheses to catch early termination.

```

\box_move_left:nn
\box_move_right:nn
\box_move_up:nn
\box_move_down:nn
32046 \cs_new_protected:Npn \box_move_left:nn #1#2
32047 { \tex_moveleft:D \__box_dim_eval:n {#1} #2 }
32048 \cs_new_protected:Npn \box_move_right:nn #1#2
32049 { \tex_moveright:D \__box_dim_eval:n {#1} #2 }
32050 \cs_new_protected:Npn \box_move_up:nn #1#2
32051 { \tex_raise:D \__box_dim_eval:n {#1} #2 }
32052 \cs_new_protected:Npn \box_move_down:nn #1#2
32053 { \tex_lower:D \__box_dim_eval:n {#1} #2 }

```

84.5 Box conditionals

The primitives for testing if a $\langle box \rangle$ is empty/void or which type of box it is.

```

\if_hbox:N
\if_vbox:N
\if_box_empty:N

32054 \cs_new_eq:NN \if_hbox:N \tex_ifhbox:D
32055 \cs_new_eq:NN \if_vbox:N \tex_ifvbox:D
32056 \cs_new_eq:NN \if_box_empty:N \tex_ifvoid:D

\prg_new_conditional:Npnn \box_if_horizontal:N #1 { p , T , F , TF }
{ \if_hbox:N #1 \prg_return_true: \else: \prg_return_false: \fi: }
\prg_new_conditional:Npnn \box_if_vertical:N #1 { p , T , F , TF }
{ \if_vbox:N #1 \prg_return_true: \else: \prg_return_false: \fi: }
\prg_generate_conditional_variant:Nnn \box_if_horizontal:N
{ c } { p , T , F , TF }
\prg_generate_conditional_variant:Nnn \box_if_vertical:N
{ c } { p , T , F , TF }

\box_if_horizontal_p:N
\box_if_horizontal_p:c
\box_if_horizontal:N $\underline{TF}$ 
\box_if_horizontal:c $\underline{TF}$ 
\box_if_vertical_p:N
\box_if_vertical_p:c
\box_if_vertical:N $\underline{TF}$ 
\box_if_vertical:c $\underline{TF}$ 

```

Testing if a $\langle box \rangle$ is empty/void.

```

\prg_new_conditional:Npnn \box_if_empty:N #1 { p , T , F , TF }
{ \if_box_empty:N #1 \prg_return_true: \else: \prg_return_false: \fi: }
\prg_generate_conditional_variant:Nnn \box_if_empty:N
{ c } { p , T , F , TF }

\box_if_empty_p:N
\box_if_empty_p:c
\box_if_empty:N $\underline{TF}$ 
\box_if_empty:c $\underline{TF}$ 

```

(End definition for $\backslash box_new:N$ and others. These functions are documented on page 270.)

84.6 The last box inserted

```

\box_set_to_last:N
\box_set_to_last:c
\box_gset_to_last:N
\box_gset_to_last:c

32069 \cs_new_protected:Npn \box_set_to_last:N #1
32070 { \tex_setbox:D #1 \tex_lastbox:D }
32071 \cs_new_protected:Npn \box_gset_to_last:N #1
32072 { \tex_global:D \tex_setbox:D #1 \tex_lastbox:D }
32073 \cs_generate_variant:Nn \box_set_to_last:N { c }
32074 \cs_generate_variant:Nn \box_gset_to_last:N { c }

```

(End definition for $\backslash box_set_to_last:N$ and $\backslash box_gset_to_last:N$. These functions are documented on page 273.)

84.7 Constant boxes

$\backslash c_empty_box$ A box we never use.

```

32075 \box_new:N \c_empty_box

```

(End definition for $\backslash c_empty_box$. This variable is documented on page 273.)

84.8 Scratch boxes

```
\l_tmpa_box Scratch boxes.
\l_tmpb_box 32076 \box_new:N \l_tmpa_box
\g_tmpa_box 32077 \box_new:N \l_tmpb_box
\g_tmpb_box 32078 \box_new:N \g_tmpa_box
32079 \box_new:N \g_tmpb_box
```

(End definition for `\l_tmpa_box` and others. These variables are documented on page 273.)

84.9 Viewing box contents

TeX's `\showbox` is not really that helpful in many cases, and it is also inconsistent with other L^AT_EX3 show functions as it does not actually shows material in the terminal. So we provide a richer set of functionality.

```
\box_show:N Essentially a wrapper around the internal function, but evaluating the breadth and depth
\box_show:c arguments now outside the group.
\box_show:Nnn 32080 \cs_new_protected:Npn \box_show:N #1
\box_show:cnn 32081 { \box_show:Nnn #1 \c_max_int \c_max_int }
32082 \cs_generate_variant:Nn \box_show:N { c }
32083 \cs_new_protected:Npn \box_show:Nnn #1#2#3
32084 { \__box_show:NNff 1 #1 { \int_eval:n {#2} } { \int_eval:n {#3} } }
32085 \cs_generate_variant:Nn \box_show:Nnn { c }
```

(End definition for `\box_show:N` and `\box_show:Nnn`. These functions are documented on page 274.)

```
\box_log:N Getting TeX to write to the log without interruption the run is done by altering the
\box_log:c interaction mode. For that, the  $\epsilon$ -TeX extensions are needed.
\box_log:Nnn 32086 \cs_new_protected:Npn \box_log:N #1
\box_log:cnn 32087 { \box_log:Nnn #1 \c_max_int \c_max_int }
\__box_log:nNnn 32088 \cs_generate_variant:Nn \box_log:N { c }
32089 \cs_new_protected:Npn \box_log:Nnn
32090 { \exp_args:No \__box_log:nNnn { \tex_the:D \tex_interactionmode:D } }
32091 \cs_new_protected:Npn \__box_log:nNnn #1#2#3#4
32092 {
32093 \int_set:Nn \tex_interactionmode:D { 0 }
32094 \__box_show:NNff 0 #2 { \int_eval:n {#3} } { \int_eval:n {#4} }
32095 \int_set:Nn \tex_interactionmode:D {#1}
32096 }
32097 \cs_generate_variant:Nn \box_log:Nnn { c }
```

(End definition for `\box_log:N`, `\box_log:Nnn`, and `__box_log:nNnn`. These functions are documented on page 274.)

```
\__box_show:NNnn The internal auxiliary to actually do the output uses a group to deal with breadth and
\__box_show:NNff depth values. The \use:n here gives better output appearance. Setting \tracingonline
and \errorcontextlines is used to control what appears in the terminal.
32098 \cs_new_protected:Npn \__box_show:NNnn #1#2#3#4
32099 {
32100 \box_if_exist:NTF #2
32101 {
32102 \group_begin:
```

```

32103         \int_set:Nn \tex_showboxbreadth:D {#3}
32104         \int_set:Nn \tex_showboxdepth:D {#4}
32105         \int_set:Nn \tex_tracingonline:D {#1}
32106         \int_set:Nn \tex_errorcontextlines:D { -1 }
32107         \tex_showbox:D \use:n {#2}
32108     \group_end:
32109 }
32110 {
32111     \msg_error:nnx { kernel } { variable-not-defined }
32112     { \token_to_str:N #2 }
32113 }
32114 }
32115 \cs_generate_variant:Nn \__box_show:NNnn { NNff }

```

(End definition for `__box_show:NNnn`.)

84.10 Horizontal mode boxes

`\hbox:n` (The test suite for this command, and others in this file, is `m3box002.lvt`.)

Put a horizontal box directly into the input stream.

```

32116 \cs_new_protected:Npn \hbox:n #1
32117 { \tex_hbox:D \scan_stop: { \color_group_begin: #1 \color_group_end: } }

```

(End definition for `\hbox:n`. This function is documented on page 274.)

```

\hbox_set:Nn
\hbox_set:cn
\hbox_gset:Nn
\hbox_gset:cn
32118 \cs_new_protected:Npn \hbox_set:Nn #1#2
32119 {
32120     \tex_setbox:D #1 \tex_hbox:D
32121     { \color_group_begin: #2 \color_group_end: }
32122 }
32123 \cs_new_protected:Npn \hbox_gset:Nn #1#2
32124 {
32125     \tex_global:D \tex_setbox:D #1 \tex_hbox:D
32126     { \color_group_begin: #2 \color_group_end: }
32127 }
32128 \cs_generate_variant:Nn \hbox_set:Nn { c }
32129 \cs_generate_variant:Nn \hbox_gset:Nn { c }

```

(End definition for `\hbox_set:Nn` and `\hbox_gset:Nn`. These functions are documented on page 274.)

`\hbox_set_to_wd:Nnn` Storing material in a horizontal box with a specified width. Again, put the dimension expression in parentheses when debugging.

```

\hbox_set_to_wd:cn
\hbox_gset_to_wd:Nnn
\hbox_gset_to_wd:cn
32130 \cs_new_protected:Npn \hbox_set_to_wd:Nnn #1#2#3
32131 {
32132     \tex_setbox:D #1 \tex_hbox:D to \__box_dim_eval:n {#2}
32133     { \color_group_begin: #3 \color_group_end: }
32134 }
32135 \cs_new_protected:Npn \hbox_gset_to_wd:Nnn #1#2#3
32136 {
32137     \tex_global:D \tex_setbox:D #1 \tex_hbox:D to \__box_dim_eval:n {#2}
32138     { \color_group_begin: #3 \color_group_end: }
32139 }
32140 \cs_generate_variant:Nn \hbox_set_to_wd:Nnn { c }
32141 \cs_generate_variant:Nn \hbox_gset_to_wd:Nnn { c }

```

(End definition for `\hbox_set_to_wd:Nnn` and `\hbox_gset_to_wd:Nnn`. These functions are documented on page 275.)

`\hbox_set:Nw` Storing material in a horizontal box. This type is useful in environment definitions.

`\hbox_set:cw` 32142 `\cs_new_protected:Npn \hbox_set:Nw #1`

`\hbox_gset:Nw` 32143 `{`

`\hbox_gset:cw` 32144 `\tex_setbox:D #1 \tex_hbox:D`

`\hbox_set_end:` 32145 `\c_group_begin_token`

`\hbox_gset_end:` 32146 `\color_group_begin:`

32147 `}`

32148 `\cs_new_protected:Npn \hbox_gset:Nw #1`

32149 `{`

32150 `\tex_global:D \tex_setbox:D #1 \tex_hbox:D`

32151 `\c_group_begin_token`

32152 `\color_group_begin:`

32153 `}`

32154 `\cs_generate_variant:Nn \hbox_set:Nw { c }`

32155 `\cs_generate_variant:Nn \hbox_gset:Nw { c }`

32156 `\cs_new_protected:Npn \hbox_set_end:`

32157 `{`

32158 `\color_group_end:`

32159 `\c_group_end_token`

32160 `}`

32161 `\cs_new_eq:NN \hbox_gset_end: \hbox_set_end:`

(End definition for `\hbox_set:Nw` and others. These functions are documented on page 275.)

`\hbox_set_to_wd:Nnw` Combining the above ideas.

`\hbox_set_to_wd:cnw` 32162 `\cs_new_protected:Npn \hbox_set_to_wd:Nnw #1#2`

`\hbox_gset_to_wd:Nnw` 32163 `{`

`\hbox_gset_to_wd:cnw` 32164 `\tex_setbox:D #1 \tex_hbox:D to _box_dim_eval:n {#2}`

32165 `\c_group_begin_token`

32166 `\color_group_begin:`

32167 `}`

32168 `\cs_new_protected:Npn \hbox_gset_to_wd:Nnw #1#2`

32169 `{`

32170 `\tex_global:D \tex_setbox:D #1 \tex_hbox:D to _box_dim_eval:n {#2}`

32171 `\c_group_begin_token`

32172 `\color_group_begin:`

32173 `}`

32174 `\cs_generate_variant:Nn \hbox_set_to_wd:Nnw { c }`

32175 `\cs_generate_variant:Nn \hbox_gset_to_wd:Nnw { c }`

(End definition for `\hbox_set_to_wd:Nnw` and `\hbox_gset_to_wd:Nnw`. These functions are documented on page 275.)

`\hbox_to_wd:nn` Put a horizontal box directly into the input stream.

`\hbox_to_zero:nn` 32176 `\cs_new_protected:Npn \hbox_to_wd:nn #1#2`

32177 `{`

32178 `\tex_hbox:D to _box_dim_eval:n {#1}`

32179 `{ \color_group_begin: #2 \color_group_end: }`

32180 `}`

32181 `\cs_new_protected:Npn \hbox_to_zero:nn #1`

32182 `{`

32183 `\tex_hbox:D to \c_zero_dim`

```

32184         { \color_group_begin: #1 \color_group_end: }
32185     }

```

(End definition for `\hbox_to_wd:nn` and `\hbox_to_zero:n`. These functions are documented on page 274.)

`\hbox_overlap_center:n` Put a zero-sized box with the contents pushed against one side (which makes it stick out on the other) directly into the input stream.

```

\hbox_overlap_left:n
\hbox_overlap_right:n
32186 \cs_new_protected:Npn \hbox_overlap_center:n #1
32187     { \hbox_to_zero:n { \tex_hss:D #1 \tex_hss:D } }
32188 \cs_new_protected:Npn \hbox_overlap_left:n #1
32189     { \hbox_to_zero:n { \tex_hss:D #1 } }
32190 \cs_new_protected:Npn \hbox_overlap_right:n #1
32191     { \hbox_to_zero:n { #1 \tex_hss:D } }

```

(End definition for `\hbox_overlap_center:n`, `\hbox_overlap_left:n`, and `\hbox_overlap_right:n`. These functions are documented on page 275.)

```

\hbox_unpack:N      Unpacking a box and if requested also clear it.
\hbox_unpack:c      32192 \cs_new_eq:NN \hbox_unpack:N \tex_unhcopy:D
\hbox_unpack_drop:N 32193 \cs_new_eq:NN \hbox_unpack_drop:N \tex_unhbox:D
\hbox_unpack_drop:c 32194 \cs_generate_variant:Nn \hbox_unpack:N { c }
                    32195 \cs_generate_variant:Nn \hbox_unpack_drop:N { c }

```

(End definition for `\hbox_unpack:N` and `\hbox_unpack_drop:N`. These functions are documented on page 275.)

84.11 Vertical mode boxes

TeX ends these boxes directly with the internal `end_graf` routine. This means that there is no `\par` at the end of vertical boxes unless we insert one. Thus all vertical boxes include a `\par` just before closing the color group.

`\vbox:n` The following test files are used for this code: `m3box003.lvt`.

The following test files are used for this code: `m3box003.lvt`.

```

\vbox_top:n      Put a vertical box directly into the input stream.
32196 \cs_new_protected:Npn \vbox:n #1
32197     { \tex_vbox:D { \color_group_begin: #1 \par \color_group_end: } }
32198 \cs_new_protected:Npn \vbox_top:n #1
32199     { \tex_vtop:D { \color_group_begin: #1 \par \color_group_end: } }

```

(End definition for `\vbox:n` and `\vbox_top:n`. These functions are documented on page 276.)

```

\vbox_to_ht:nn    Put a vertical box directly into the input stream.
\vbox_to_zero:n    32200 \cs_new_protected:Npn \vbox_to_ht:nn #1#2
\vbox_to_ht:nn    32201     {
\vbox_to_zero:n    32202         \tex_vbox:D to \__box_dim_eval:n {#1}
                    32203         { \color_group_begin: #2 \par \color_group_end: }
32204     }
32205 \cs_new_protected:Npn \vbox_to_zero:n #1
32206     {
32207         \tex_vbox:D to \c_zero_dim
32208         { \color_group_begin: #1 \par \color_group_end: }
32209     }

```

(End definition for `\vbox_to_ht:nn` and others. These functions are documented on page 276.)

`\vbox_set:Nn` Storing material in a vertical box with a natural height.
`\vbox_set:cn` 32210 `\cs_new_protected:Npn \vbox_set:Nn #1#2`
`\vbox_gset:Nn` 32211 `{`
`\vbox_gset:cn` 32212 `\tex_setbox:D #1 \tex_vbox:D`
32213 `{ \color_group_begin: #2 \par \color_group_end: }`
32214 `}`
32215 `\cs_new_protected:Npn \vbox_gset:Nn #1#2`
32216 `{`
32217 `\tex_global:D \tex_setbox:D #1 \tex_vbox:D`
32218 `{ \color_group_begin: #2 \par \color_group_end: }`
32219 `}`
32220 `\cs_generate_variant:Nn \vbox_set:Nn { c }`
32221 `\cs_generate_variant:Nn \vbox_gset:Nn { c }`

(End definition for `\vbox_set:Nn` and `\vbox_gset:Nn`. These functions are documented on page 276.)

`\vbox_set_top:Nn` Storing material in a vertical box with a natural height and reference point at the baseline
`\vbox_set_top:cn` of the first object in the box.
`\vbox_gset_top:Nn` 32222 `\cs_new_protected:Npn \vbox_set_top:Nn #1#2`
`\vbox_gset_top:cn` 32223 `{`
32224 `\tex_setbox:D #1 \tex_vtop:D`
32225 `{ \color_group_begin: #2 \par \color_group_end: }`
32226 `}`
32227 `\cs_new_protected:Npn \vbox_gset_top:Nn #1#2`
32228 `{`
32229 `\tex_global:D \tex_setbox:D #1 \tex_vtop:D`
32230 `{ \color_group_begin: #2 \par \color_group_end: }`
32231 `}`
32232 `\cs_generate_variant:Nn \vbox_set_top:Nn { c }`
32233 `\cs_generate_variant:Nn \vbox_gset_top:Nn { c }`

(End definition for `\vbox_set_top:Nn` and `\vbox_gset_top:Nn`. These functions are documented on page 276.)

`\vbox_set_to_ht:Nnn` Storing material in a vertical box with a specified height.
`\vbox_set_to_ht:cn` 32234 `\cs_new_protected:Npn \vbox_set_to_ht:Nnn #1#2#3`
`\vbox_gset_to_ht:Nnn` 32235 `{`
`\vbox_gset_to_ht:cn` 32236 `\tex_setbox:D #1 \tex_vbox:D to _box_dim_eval:n {#2}`
32237 `{ \color_group_begin: #3 \par \color_group_end: }`
32238 `}`
32239 `\cs_new_protected:Npn \vbox_gset_to_ht:Nnn #1#2#3`
32240 `{`
32241 `\tex_global:D \tex_setbox:D #1 \tex_vbox:D to _box_dim_eval:n {#2}`
32242 `{ \color_group_begin: #3 \par \color_group_end: }`
32243 `}`
32244 `\cs_generate_variant:Nn \vbox_set_to_ht:Nnn { c }`
32245 `\cs_generate_variant:Nn \vbox_gset_to_ht:Nnn { c }`

(End definition for `\vbox_set_to_ht:Nnn` and `\vbox_gset_to_ht:Nnn`. These functions are documented on page 276.)

`\vbox_set:Nw` Storing material in a vertical box. This type is useful in environment definitions.
`\vbox_set:cw`
`\vbox_gset:Nw`
`\vbox_gset:cw`
`\vbox_set_end:`
`\vbox_gset_end:`

```

32246 \cs_new_protected:Npn \vbox_set:Nw #1
32247 {
32248   \tex_setbox:D #1 \tex_vbox:D
32249   \c_group_begin_token
32250   \color_group_begin:
32251 }
32252 \cs_new_protected:Npn \vbox_gset:Nw #1
32253 {
32254   \tex_global:D \tex_setbox:D #1 \tex_vbox:D
32255   \c_group_begin_token
32256   \color_group_begin:
32257 }
32258 \cs_generate_variant:Nn \vbox_set:Nw { c }
32259 \cs_generate_variant:Nn \vbox_gset:Nw { c }
32260 \cs_new_protected:Npn \vbox_set_end:
32261 {
32262   \par
32263   \color_group_end:
32264   \c_group_end_token
32265 }
32266 \cs_new_eq:NN \vbox_gset_end: \vbox_set_end:

```

(End definition for `\vbox_set:Nw` and others. These functions are documented on page 276.)

`\vbox_set_to_ht:Nnw` A combination of the above ideas.
`\vbox_set_to_ht:cnw`
`\vbox_gset_to_ht:Nnw`
`\vbox_gset_to_ht:cnw`

```

32267 \cs_new_protected:Npn \vbox_set_to_ht:Nnw #1#2
32268 {
32269   \tex_setbox:D #1 \tex_vbox:D to \_box_dim_eval:n {#2}
32270   \c_group_begin_token
32271   \color_group_begin:
32272 }
32273 \cs_new_protected:Npn \vbox_gset_to_ht:Nnw #1#2
32274 {
32275   \tex_global:D \tex_setbox:D #1 \tex_vbox:D to \_box_dim_eval:n {#2}
32276   \c_group_begin_token
32277   \color_group_begin:
32278 }
32279 \cs_generate_variant:Nn \vbox_set_to_ht:Nnw { c }
32280 \cs_generate_variant:Nn \vbox_gset_to_ht:Nnw { c }

```

(End definition for `\vbox_set_to_ht:Nnw` and `\vbox_gset_to_ht:Nnw`. These functions are documented on page 277.)

`\vbox_unpack:N` Unpacking a box and if requested also clear it.
`\vbox_unpack:c`
`\vbox_unpack_drop:N`
`\vbox_unpack_drop:c`

```

32281 \cs_new_eq:NN \vbox_unpack:N \tex_unvcopy:D
32282 \cs_new_eq:NN \vbox_unpack_drop:N \tex_unvbox:D
32283 \cs_generate_variant:Nn \vbox_unpack:N { c }
32284 \cs_generate_variant:Nn \vbox_unpack_drop:N { c }

```

(End definition for `\vbox_unpack:N` and `\vbox_unpack_drop:N`. These functions are documented on page 277.)

```

\ vbox_set_split_to_ht:NNn Splitting a vertical box in two.
\ vbox_set_split_to_ht:cNn
\ vbox_set_split_to_ht:Ncn
\ vbox_set_split_to_ht:ccn
\ vbox_gset_split_to_ht:NNn
\ vbox_gset_split_to_ht:cNn
\ vbox_gset_split_to_ht:Ncn
\ vbox_gset_split_to_ht:ccn
32285 \cs_new_protected:Npn \vbox_set_split_to_ht:NNn #1#2#3
32286 { \tex_setbox:D #1 \tex_vsplit:D #2 to \_box_dim_eval:n {#3} }
32287 \cs_generate_variant:Nn \vbox_set_split_to_ht:NNn { c , Nc , cc }
32288 \cs_new_protected:Npn \vbox_gset_split_to_ht:NNn #1#2#3
32289 {
32290 \tex_global:D \tex_setbox:D #1
32291 \tex_vsplit:D #2 to \_box_dim_eval:n {#3}
32292 }
32293 \cs_generate_variant:Nn \vbox_gset_split_to_ht:NNn { c , Nc , cc }

```

(End definition for `\vbox_set_split_to_ht:NNn` and `\vbox_gset_split_to_ht:NNn`. These functions are documented on page [277](#).)

84.12 Affine transformations

`\l__box_angle_fp` When rotating boxes, the angle itself may be needed by the engine-dependent code. This is done using the `fp` module so that the value is tidied up properly.

```
32294 \fp_new:N \l__box_angle_fp
```

(End definition for `\l__box_angle_fp`.)

`\l__box_cos_fp` These are used to hold the calculated sine and cosine values while carrying out a rotation.

```
\l__box_sin_fp
32295 \fp_new:N \l__box_cos_fp
```

```
32296 \fp_new:N \l__box_sin_fp
```

(End definition for `\l__box_cos_fp` and `\l__box_sin_fp`.)

`\l__box_top_dim` These are the positions of the four edges of a box before manipulation.

```

\l__box_bottom_dim
\l__box_left_dim
\l__box_right_dim
32297 \dim_new:N \l__box_top_dim
32298 \dim_new:N \l__box_bottom_dim
32299 \dim_new:N \l__box_left_dim
32300 \dim_new:N \l__box_right_dim

```

(End definition for `\l__box_top_dim` and others.)

`\l__box_top_new_dim` These are the positions of the four edges of a box after manipulation.

```

\l__box_bottom_new_dim
\l__box_left_new_dim
\l__box_right_new_dim
32301 \dim_new:N \l__box_top_new_dim
32302 \dim_new:N \l__box_bottom_new_dim
32303 \dim_new:N \l__box_left_new_dim
32304 \dim_new:N \l__box_right_new_dim

```

(End definition for `\l__box_top_new_dim` and others.)

`\l__box_internal_box` Scratch space, but also needed by some parts of the driver.

```
32305 \box_new:N \l__box_internal_box
```

(End definition for `\l__box_internal_box`.)

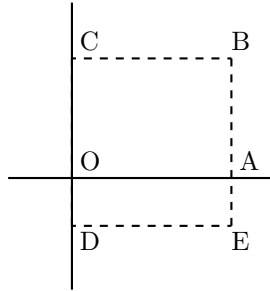


Figure 1: Co-ordinates of a box prior to rotation.

\box_rotate:Nn Rotation of a box starts with working out the relevant sine and cosine. The actual
\box_rotate:cn rotation is in an auxiliary to keep the flow slightly clearer

\box_grotate:Nn 32306 \cs_new_protected:Npn \box_rotate:Nn #1#2

\box_grotate:cn 32307 { __box_rotate:NnN #1 {#2} \hbox_set:Nn }

__box_rotate:NnN 32308 \cs_generate_variant:Nn \box_rotate:Nn { c }

__box_rotate:N 32309 \cs_new_protected:Npn \box_grotate:Nn #1#2

__box_rotate_xdir:nnN 32310 { __box_rotate:NnN #1 {#2} \hbox_gset:Nn }

__box_rotate_ydir:nnN 32311 \cs_generate_variant:Nn \box_grotate:Nn { c }

__box_rotate_quadrant_one: 32312 \cs_new_protected:Npn __box_rotate:NnN #1#2#3

__box_rotate_quadrant_two: 32313 {

__box_rotate_quadrant_three: 32314 #3 #1

__box_rotate_quadrant_four: 32315 {

\fp_set:Nn \l__box_angle_fp {#2}

\fp_set:Nn \l__box_sin_fp { sind (\l__box_angle_fp) }

\fp_set:Nn \l__box_cos_fp { cosd (\l__box_angle_fp) }

__box_rotate:N #1

}

The edges of the box are then recorded: the left edge is always at zero. Rotation of the four edges then takes place: this is most efficiently done on a quadrant by quadrant basis.

32322 \cs_new_protected:Npn __box_rotate:N #1

32323 {

32324 \dim_set:Nn \l__box_top_dim { \box_ht:N #1 }

32325 \dim_set:Nn \l__box_bottom_dim { -\box_dp:N #1 }

32326 \dim_set:Nn \l__box_right_dim { \box_wd:N #1 }

32327 \dim_zero:N \l__box_left_dim

The next step is to work out the x and y coordinates of vertices of the rotated box in relation to its original coordinates. The box can be visualized with vertices B , C , D and E is illustrated (Figure 1). The vertex O is the reference point on the baseline, and in this implementation is also the centre of rotation. The formulae are, for a point P and angle α :

$$\begin{aligned} P'_x &= P_x - O_x \\ P'_y &= P_y - O_y \\ P''_x &= (P'_x \cos(\alpha)) - (P'_y \sin(\alpha)) \\ P''_y &= (P'_x \sin(\alpha)) + (P'_y \cos(\alpha)) \\ P'''_x &= P''_x + O_x + L_x \\ P'''_y &= P''_y + O_y \end{aligned}$$

The “extra” horizontal translation L_x at the end is calculated so that the leftmost point of the resulting box has x -coordinate 0. This is desirable as \TeX boxes must have the reference point at the left edge of the box. (As O is always $(0,0)$, this part of the calculation is omitted here.)

```

32328 \fp_compare:nNnTF \l__box_sin_fp > \c_zero_fp
32329 {
32330     \fp_compare:nNnTF \l__box_cos_fp > \c_zero_fp
32331     { \__box_rotate_quadrant_one: }
32332     { \__box_rotate_quadrant_two: }
32333 }
32334 {
32335     \fp_compare:nNnTF \l__box_cos_fp < \c_zero_fp
32336     { \__box_rotate_quadrant_three: }
32337     { \__box_rotate_quadrant_four: }
32338 }

```

The position of the box edges are now known, but the box at this stage be misplaced relative to the current \TeX reference point. So the content of the box is moved such that the reference point of the rotated box is in the same place as the original.

```

32339 \hbox_set:Nn \l__box_internal_box { \box_use:N #1 }
32340 \hbox_set:Nn \l__box_internal_box
32341 {
32342     \__kernel_kern:n { -\l__box_left_new_dim }
32343     \hbox:n
32344     {
32345         \__box_backend_rotate:Nn
32346         \l__box_internal_box
32347         \l__box_angle_fp
32348     }
32349 }

```

Tidy up the size of the box so that the material is actually inside the bounding box. The result can then be used to reset the original box.

```

32350 \box_set_ht:Nn \l__box_internal_box { \l__box_top_new_dim }
32351 \box_set_dp:Nn \l__box_internal_box { -\l__box_bottom_new_dim }
32352 \box_set_wd:Nn \l__box_internal_box
32353 { \l__box_right_new_dim - \l__box_left_new_dim }
32354 \box_use_drop:N \l__box_internal_box
32355 }

```

These functions take a general point $(\#1,\#2)$ and rotate its location about the origin, using the previously-set sine and cosine values. Each function gives only one component of the location of the updated point. This is because for rotation of a box each step needs only one value, and so performance is gained by avoiding working out both x' and y' at the same time. Contrast this with the equivalent function in the `l3coffins` module, where both parts are needed.

```

32356 \cs_new_protected:Npn \__box_rotate_xdir:nnN #1#2#3
32357 {
32358     \dim_set:Nn #3
32359     {
32360         \fp_to_dim:n
32361         {
32362             \l__box_cos_fp * \dim_to_fp:n {#1}
32363             - \l__box_sin_fp * \dim_to_fp:n {#2}

```

```

32364     }
32365   }
32366 }
32367 \cs_new_protected:Npn \__box_rotate_ydir:nnN #1#2#3
32368 {
32369   \dim_set:Nn #3
32370   {
32371     \fp_to_dim:n
32372     {
32373       \l__box_sin_fp * \dim_to_fp:n {#1}
32374       + \l__box_cos_fp * \dim_to_fp:n {#2}
32375     }
32376   }
32377 }

```

Rotation of the edges is done using a different formula for each quadrant. In every case, the top and bottom edges only need the resulting y -values, whereas the left and right edges need the x -values. Each case is a question of picking out which corner ends up at with the maximum top, bottom, left and right value. Doing this by hand means a lot less calculating and avoids lots of comparisons.

```

32378 \cs_new_protected:Npn \__box_rotate_quadrant_one:
32379 {
32380   \__box_rotate_ydir:nnN \l__box_right_dim \l__box_top_dim
32381   \l__box_top_new_dim
32382   \__box_rotate_ydir:nnN \l__box_left_dim \l__box_bottom_dim
32383   \l__box_bottom_new_dim
32384   \__box_rotate_xdir:nnN \l__box_left_dim \l__box_top_dim
32385   \l__box_left_new_dim
32386   \__box_rotate_xdir:nnN \l__box_right_dim \l__box_bottom_dim
32387   \l__box_right_new_dim
32388 }
32389 \cs_new_protected:Npn \__box_rotate_quadrant_two:
32390 {
32391   \__box_rotate_ydir:nnN \l__box_right_dim \l__box_bottom_dim
32392   \l__box_top_new_dim
32393   \__box_rotate_ydir:nnN \l__box_left_dim \l__box_top_dim
32394   \l__box_bottom_new_dim
32395   \__box_rotate_xdir:nnN \l__box_right_dim \l__box_top_dim
32396   \l__box_left_new_dim
32397   \__box_rotate_xdir:nnN \l__box_left_dim \l__box_bottom_dim
32398   \l__box_right_new_dim
32399 }
32400 \cs_new_protected:Npn \__box_rotate_quadrant_three:
32401 {
32402   \__box_rotate_ydir:nnN \l__box_left_dim \l__box_bottom_dim
32403   \l__box_top_new_dim
32404   \__box_rotate_ydir:nnN \l__box_right_dim \l__box_top_dim
32405   \l__box_bottom_new_dim
32406   \__box_rotate_xdir:nnN \l__box_right_dim \l__box_bottom_dim
32407   \l__box_left_new_dim
32408   \__box_rotate_xdir:nnN \l__box_left_dim \l__box_top_dim
32409   \l__box_right_new_dim
32410 }
32411 \cs_new_protected:Npn \__box_rotate_quadrant_four:

```

```

32412 {
32413   \__box_rotate_ydir:nnN \l__box_left_dim \l__box_top_dim
32414   \l__box_top_new_dim
32415   \__box_rotate_ydir:nnN \l__box_right_dim \l__box_bottom_dim
32416   \l__box_bottom_new_dim
32417   \__box_rotate_xdir:nnN \l__box_left_dim \l__box_bottom_dim
32418   \l__box_left_new_dim
32419   \__box_rotate_xdir:nnN \l__box_right_dim \l__box_top_dim
32420   \l__box_right_new_dim
32421 }

```

(End definition for `\box_rotate:Nn` and others. These functions are documented on page 281.)

`\l__box_scale_x_fp` Scaling is potentially-different in the two axes.

```

\l__box_scale_y_fp 32422 \fp_new:N \l__box_scale_x_fp
32423 \fp_new:N \l__box_scale_y_fp

```

(End definition for `\l__box_scale_x_fp` and `\l__box_scale_y_fp`.)

`\box_resize_to_wd_and_ht_plus_dp:Nnn` Resizing a box starts by working out the various dimensions of the existing box.

```

\box_resize_to_wd_and_ht_plus_dp:cnm 32424 \cs_new_protected:Npn \box_resize_to_wd_and_ht_plus_dp:Nnn #1#2#3
\box_gresize_to_wd_and_ht_plus_dp:Nnn 32425 {
\box_gresize_to_wd_and_ht_plus_dp:cnm 32426   \__box_resize_to_wd_and_ht_plus_dp:NnnN #1 {#2} {#3}
\__box_resize_to_wd_and_ht_plus_dp:NnnN 32427   \hbox_set:Nn
32428 }
\__box_resize_set_corners:N 32429 \cs_generate_variant:Nn \box_resize_to_wd_and_ht_plus_dp:Nnn { c }
\__box_resize:N 32430 \cs_new_protected:Npn \box_gresize_to_wd_and_ht_plus_dp:Nnn #1#2#3
\__box_resize:NNN 32431 {
32432   \__box_resize_to_wd_and_ht_plus_dp:NnnN #1 {#2} {#3}
32433   \hbox_gset:Nn
32434 }
32435 \cs_generate_variant:Nn \box_gresize_to_wd_and_ht_plus_dp:Nnn { c }
32436 \cs_new_protected:Npn \__box_resize_to_wd_and_ht_plus_dp:NnnN #1#2#3#4
32437 {
32438   #4 #1
32439   {
32440     \__box_resize_set_corners:N #1

```

The x -scaling and resulting box size is easy enough to work out: the dimension is that given as #2, and the scale is simply the new width divided by the old one.

```

32441   \fp_set:Nn \l__box_scale_x_fp
32442   { \dim_to_fp:n {#2} / \dim_to_fp:n { \l__box_right_dim } }

```

The y -scaling needs both the height and the depth of the current box.

```

32443   \fp_set:Nn \l__box_scale_y_fp
32444   {
32445     \dim_to_fp:n {#3}
32446     / \dim_to_fp:n { \l__box_top_dim - \l__box_bottom_dim }
32447   }

```

Hand off to the auxiliary which does the rest of the work.

```

32448   \__box_resize:N #1
32449 }
32450 }
32451 \cs_new_protected:Npn \__box_resize_set_corners:N #1
32452 {

```

```

32453 \dim_set:Nn \l__box_top_dim { \box_ht:N #1 }
32454 \dim_set:Nn \l__box_bottom_dim { -\box_dp:N #1 }
32455 \dim_set:Nn \l__box_right_dim { \box_wd:N #1 }
32456 \dim_zero:N \l__box_left_dim
32457 }

```

With at least one real scaling to do, the next phase is to find the new edge co-ordinates. In the x direction this is relatively easy: just scale the right edge. In the y direction, both dimensions have to be scaled, and this again needs the absolute scale value. Once that is all done, the common resize/rescale code can be employed.

```

32458 \cs_new_protected:Npn \__box_resize:N #1
32459 {
32460   \__box_resize:NNN \l__box_right_new_dim
32461   \l__box_scale_x_fp \l__box_right_dim
32462   \__box_resize:NNN \l__box_bottom_new_dim
32463   \l__box_scale_y_fp \l__box_bottom_dim
32464   \__box_resize:NNN \l__box_top_new_dim
32465   \l__box_scale_y_fp \l__box_top_dim
32466   \__box_resize_common:N #1
32467 }
32468 \cs_new_protected:Npn \__box_resize:NNN #1#2#3
32469 {
32470   \dim_set:Nn #1
32471   { \fp_to_dim:n { \fp_abs:n { #2 } * \dim_to_fp:n { #3 } } }
32472 }

```

(End definition for `\box_resize_to_wd_and_ht_plus_dp:Nnn` and others. These functions are documented on page 280.)

<pre> \box_resize_to_ht:Nn \box_resize_to_ht:cn \box_gresize_to_ht:Nn \box_gresize_to_ht:cn __box_resize_to_ht:Nnn \box_resize_to_ht_plus_dp:Nn \box_resize_to_ht_plus_dp:cn \box_gresize_to_ht_plus_dp:Nn \box_gresize_to_ht_plus_dp:cn __box_resize_to_ht_plus_dp:Nnn \box_resize_to_wd:Nn \box_resize_to_wd:cn \box_gresize_to_wd:Nn \box_gresize_to_wd:cn __box_resize_to_wd:Nnn \box_resize_to_wd_and_ht:Nnn \box_resize_to_wd_and_ht:cn \box_gresize_to_wd_and_ht:Nnn \box_gresize_to_wd_and_ht:cn __box_resize_to_wd_ht:NnnN </pre>	<pre> 32473 \cs_new_protected:Npn \box_resize_to_ht:Nn #1#2 32474 { __box_resize_to_ht:Nnn #1 {#2} \hbox_set:Nn } 32475 \cs_generate_variant:Nn \box_resize_to_ht:Nn { c } 32476 \cs_new_protected:Npn \box_gresize_to_ht:Nn #1#2 32477 { __box_resize_to_ht:Nnn #1 {#2} \hbox_gset:Nn } 32478 \cs_generate_variant:Nn \box_gresize_to_ht:Nn { c } 32479 \cs_new_protected:Npn __box_resize_to_ht:Nnn #1#2#3 32480 { 32481 #3 #1 32482 { 32483 __box_resize_set_corners:N #1 32484 \fp_set:Nn \l__box_scale_y_fp 32485 { 32486 \dim_to_fp:n {#2} 32487 / \dim_to_fp:n { \l__box_top_dim } 32488 } 32489 \fp_set_eq:NN \l__box_scale_x_fp \l__box_scale_y_fp 32490 __box_resize:N #1 32491 } 32492 } 32493 \cs_new_protected:Npn \box_resize_to_ht_plus_dp:Nn #1#2 </pre>
--	---

```

32494 { \_box_resize_to_ht_plus_dp:NnN #1 {#2} \hbox_set:Nn }
32495 \cs_generate_variant:Nn \box_resize_to_ht_plus_dp:Nn { c }
32496 \cs_new_protected:Npn \box_gresize_to_ht_plus_dp:Nn #1#2
32497 { \_box_resize_to_ht_plus_dp:NnN #1 {#2} \hbox_gset:Nn }
32498 \cs_generate_variant:Nn \box_gresize_to_ht_plus_dp:Nn { c }
32499 \cs_new_protected:Npn \_box_resize_to_ht_plus_dp:NnN #1#2#3
32500 {
32501   #3 #1
32502   {
32503     \_box_resize_set_corners:N #1
32504     \fp_set:Nn \l__box_scale_y_fp
32505     {
32506       \dim_to_fp:n {#2}
32507       / \dim_to_fp:n { \l__box_top_dim - \l__box_bottom_dim }
32508     }
32509     \fp_set_eq:NN \l__box_scale_x_fp \l__box_scale_y_fp
32510     \_box_resize:N #1
32511   }
32512 }
32513 \cs_new_protected:Npn \box_resize_to_wd:Nn #1#2
32514 { \_box_resize_to_wd:NnN #1 {#2} \hbox_set:Nn }
32515 \cs_generate_variant:Nn \box_resize_to_wd:Nn { c }
32516 \cs_new_protected:Npn \box_gresize_to_wd:Nn #1#2
32517 { \_box_resize_to_wd:NnN #1 {#2} \hbox_gset:Nn }
32518 \cs_generate_variant:Nn \box_gresize_to_wd:Nn { c }
32519 \cs_new_protected:Npn \_box_resize_to_wd:NnN #1#2#3
32520 {
32521   #3 #1
32522   {
32523     \_box_resize_set_corners:N #1
32524     \fp_set:Nn \l__box_scale_x_fp
32525     { \dim_to_fp:n {#2} / \dim_to_fp:n { \l__box_right_dim } }
32526     \fp_set_eq:NN \l__box_scale_y_fp \l__box_scale_x_fp
32527     \_box_resize:N #1
32528   }
32529 }
32530 \cs_new_protected:Npn \box_resize_to_wd_and_ht:Nnn #1#2#3
32531 { \_box_resize_to_wd_and_ht:NnnN #1 {#2} {#3} \hbox_set:Nn }
32532 \cs_generate_variant:Nn \box_resize_to_wd_and_ht:Nnn { c }
32533 \cs_new_protected:Npn \box_gresize_to_wd_and_ht:Nnn #1#2#3
32534 { \_box_resize_to_wd_and_ht:NnnN #1 {#2} {#3} \hbox_gset:Nn }
32535 \cs_generate_variant:Nn \box_gresize_to_wd_and_ht:Nnn { c }
32536 \cs_new_protected:Npn \_box_resize_to_wd_and_ht:NnnN #1#2#3#4
32537 {
32538   #4 #1
32539   {
32540     \_box_resize_set_corners:N #1
32541     \fp_set:Nn \l__box_scale_x_fp
32542     { \dim_to_fp:n {#2} / \dim_to_fp:n { \l__box_right_dim } }
32543     \fp_set:Nn \l__box_scale_y_fp
32544     {
32545       \dim_to_fp:n {#3}
32546       / \dim_to_fp:n { \l__box_top_dim }
32547     }

```



```

32548         \_box_resize:N #1
32549     }
32550 }

```

(End definition for `\box_resize_to_ht:Nn` and others. These functions are documented on page 279.)

\box_scale:Nnn When scaling a box, setting the scaling itself is easy enough. The new dimensions are also relatively easy to find, allowing only for the need to keep them positive in all cases. **\box_scale:cnn** Once that is done then after a check for the trivial scaling a hand-off can be made to the common code. **\box_gscale:Nnn** The code here is split into two as this allows sharing with the auto-resizing functions. **\box_gscale:cnn**

```

\__box_scale:NnnN
\__box_scale:N
32551 \cs_new_protected:Npn \box_scale:Nnn #1#2#3
32552 { \__box_scale:NnnN #1 {#2} {#3} \hbox_set:Nn }
32553 \cs_generate_variant:Nn \box_scale:Nnn { c }
32554 \cs_new_protected:Npn \box_gscale:Nnn #1#2#3
32555 { \__box_scale:NnnN #1 {#2} {#3} \hbox_gset:Nn }
32556 \cs_generate_variant:Nn \box_gscale:Nnn { c }
32557 \cs_new_protected:Npn \__box_scale:NnnN #1#2#3#4
32558 {
32559     #4 #1
32560     {
32561         \fp_set:Nn \l__box_scale_x_fp {#2}
32562         \fp_set:Nn \l__box_scale_y_fp {#3}
32563         \__box_scale:N #1
32564     }
32565 }
32566 \cs_new_protected:Npn \__box_scale:N #1
32567 {
32568     \dim_set:Nn \l__box_top_dim { \box_ht:N #1 }
32569     \dim_set:Nn \l__box_bottom_dim { -\box_dp:N #1 }
32570     \dim_set:Nn \l__box_right_dim { \box_wd:N #1 }
32571     \dim_zero:N \l__box_left_dim
32572     \dim_set:Nn \l__box_top_new_dim
32573     { \fp_abs:n { \l__box_scale_y_fp } \l__box_top_dim }
32574     \dim_set:Nn \l__box_bottom_new_dim
32575     { \fp_abs:n { \l__box_scale_y_fp } \l__box_bottom_dim }
32576     \dim_set:Nn \l__box_right_new_dim
32577     { \fp_abs:n { \l__box_scale_x_fp } \l__box_right_dim }
32578     \__box_resize_common:N #1
32579 }

```

(End definition for `\box_scale:Nnn` and others. These functions are documented on page 281.)

\box_autosize_to_wd_and_ht:Nnn Although autosizing a box uses dimensions, it has more in common in implementation with scaling. As such, most of the real work here is done elsewhere. **\box_autosize_to_wd_and_ht:cnn**

```

\box_gautosize_to_wd_and_ht:Nnn
\box_gautosize_to_wd_and_ht:cnn
32580 \cs_new_protected:Npn \box_autosize_to_wd_and_ht:Nnn #1#2#3
32581 { \__box_autosize:NnnN #1 {#2} {#3} { \box_ht:N #1 } \hbox_set:Nn }
32582 \cs_generate_variant:Nn \box_autosize_to_wd_and_ht:Nnn { c }
32583 \cs_new_protected:Npn \box_gautosize_to_wd_and_ht:Nnn #1#2#3
32584 { \__box_autosize:NnnN #1 {#2} {#3} { \box_ht:N #1 } \hbox_gset:Nn }
32585 \cs_generate_variant:Nn \box_gautosize_to_wd_and_ht:Nnn { c }
32586 \cs_new_protected:Npn \box_autosize_to_wd_and_ht_plus_dp:Nnn #1#2#3
32587 {
32588     \__box_autosize:NnnN #1 {#2} {#3} { \box_ht:N #1 + \box_dp:N #1 }

```

```

32589     \hbox_set:Nn
32590   }
32591   \cs_generate_variant:Nn \box_autosize_to_wd_and_ht_plus_dp:Nnn { c }
32592   \cs_new_protected:Npn \box_gautosize_to_wd_and_ht_plus_dp:Nnn #1#2#3
32593   {
32594     \__box_autosize:NnnN #1 {#2} {#3} { \box_ht:N #1 + \box_dp:N #1 }
32595     \hbox_gset:Nn
32596   }
32597   \cs_generate_variant:Nn \box_gautosize_to_wd_and_ht_plus_dp:Nnn { c }
32598   \cs_new_protected:Npn \__box_autosize:NnnN #1#2#3#4#5
32599   {
32600     #5 #1
32601     {
32602       \fp_set:Nn \l__box_scale_x_fp { ( #2 ) / \box_wd:N #1 }
32603       \fp_set:Nn \l__box_scale_y_fp { ( #3 ) / ( #4 ) }
32604       \fp_compare:nNnTF \l__box_scale_x_fp > \l__box_scale_y_fp
32605       { \fp_set_eq:NN \l__box_scale_x_fp \l__box_scale_y_fp }
32606       { \fp_set_eq:NN \l__box_scale_y_fp \l__box_scale_x_fp }
32607       \__box_scale:N #1
32608     }
32609   }

```

(End definition for `\box_autosize_to_wd_and_ht:Nnn` and others. These functions are documented on page 279.)

`__box_resize_common:N` The main resize function places its input into a box which start off with zero width, and includes the handles for engine rescaling.

```

32610   \cs_new_protected:Npn \__box_resize_common:N #1
32611   {
32612     \hbox_set:Nn \l__box_internal_box
32613     {
32614       \__box_backend_scale:Nnn
32615       #1
32616       \l__box_scale_x_fp
32617       \l__box_scale_y_fp
32618     }

```

The new height and depth can be applied directly.

```

32619     \fp_compare:nNnTF \l__box_scale_y_fp > \c_zero_fp
32620     {
32621       \box_set_ht:Nn \l__box_internal_box { \l__box_top_new_dim }
32622       \box_set_dp:Nn \l__box_internal_box { -\l__box_bottom_new_dim }
32623     }
32624     {
32625       \box_set_dp:Nn \l__box_internal_box { \l__box_top_new_dim }
32626       \box_set_ht:Nn \l__box_internal_box { -\l__box_bottom_new_dim }
32627     }

```

Things are not quite as obvious for the width, as the reference point needs to remain unchanged. For positive scaling factors resizing the box is all that is needed. However, for case of a negative scaling the material must be shifted such that the reference point ends up in the right place.

```

32628     \fp_compare:nNnTF \l__box_scale_x_fp < \c_zero_fp
32629     {
32630       \hbox_to_wd:nn { \l__box_right_new_dim }

```

```

32631         {
32632             \__kernel_kern:n { \l__box_right_new_dim }
32633             \box_use_drop:N \l__box_internal_box
32634             \tex_hss:D
32635         }
32636     }
32637     {
32638         \box_set_wd:Nn \l__box_internal_box { \l__box_right_new_dim }
32639         \hbox:n
32640         {
32641             \__kernel_kern:n { Opt }
32642             \box_use_drop:N \l__box_internal_box
32643             \tex_hss:D
32644         }
32645     }
32646 }

```

(End definition for __box_resize_common:N.)

```

32647 \endpackage

```

Chapter 85

l3coffins Implementation

```
32648 <*package>
32649 <@@=coffin>
```

85.1 Coffins: data structures and general variables

`\l__coffin_internal_box` Scratch variables.

```
\l__coffin_internal_dim 32650 \box_new:N \l__coffin_internal_box
\l__coffin_internal_tl 32651 \dim_new:N \l__coffin_internal_dim
32652 \tl_new:N \l__coffin_internal_tl
```

(End definition for `\l__coffin_internal_box`, `\l__coffin_internal_dim`, and `\l__coffin_internal_tl`.)

`\c__coffin_corners_prop` The “corners”; of a coffin define the real content, as opposed to the TeX bounding box. They all start off in the same place, of course.

```
32653 \prop_const_from_keyval:Nn \c__coffin_corners_prop
32654 {
32655   tl = { Opt } { Opt } ,
32656   tr = { Opt } { Opt } ,
32657   bl = { Opt } { Opt } ,
32658   br = { Opt } { Opt } ,
32659 }
```

(End definition for `\c__coffin_corners_prop`.)

`\c__coffin_poles_prop` Pole positions are given for horizontal, vertical and reference-point based values.

```
32660 \prop_const_from_keyval:Nn \c__coffin_poles_prop
32661 {
32662   l  = { Opt } { Opt } { Opt } { 1000pt } ,
32663   hc = { Opt } { Opt } { Opt } { 1000pt } ,
32664   r  = { Opt } { Opt } { Opt } { 1000pt } ,
32665   b  = { Opt } { Opt } { 1000pt } { Opt } ,
32666   vc = { Opt } { Opt } { 1000pt } { Opt } ,
32667   t  = { Opt } { Opt } { 1000pt } { Opt } ,
32668   B  = { Opt } { Opt } { 1000pt } { Opt } ,
32669   H  = { Opt } { Opt } { 1000pt } { Opt } ,
32670   T  = { Opt } { Opt } { 1000pt } { Opt } ,
32671 }
```

(End definition for `\c__coffin_poles_prop`.)

`\l__coffin_slope_A_fp` Used for calculations of intersections.

`\l__coffin_slope_B_fp` 32672 `\fp_new:N \l__coffin_slope_A_fp`
32673 `\fp_new:N \l__coffin_slope_B_fp`

(End definition for `\l__coffin_slope_A_fp` and `\l__coffin_slope_B_fp`.)

`\l__coffin_error_bool` For propagating errors so that parts of the code can work around them.

32674 `\bool_new:N \l__coffin_error_bool`

(End definition for `\l__coffin_error_bool`.)

`\l__coffin_offset_x_dim` The offset between two sets of coffin handles when typesetting. These values are corrected from those requested in an alignment for the positions of the handles.

`\l__coffin_offset_y_dim` 32675 `\dim_new:N \l__coffin_offset_x_dim`
32676 `\dim_new:N \l__coffin_offset_y_dim`

(End definition for `\l__coffin_offset_x_dim` and `\l__coffin_offset_y_dim`.)

`\l__coffin_pole_a_tl` Needed for finding the intersection of two poles.

`\l__coffin_pole_b_tl` 32677 `\tl_new:N \l__coffin_pole_a_tl`
32678 `\tl_new:N \l__coffin_pole_b_tl`

(End definition for `\l__coffin_pole_a_tl` and `\l__coffin_pole_b_tl`.)

`\l__coffin_x_dim` For calculating intersections and so forth.

`\l__coffin_y_dim` 32679 `\dim_new:N \l__coffin_x_dim`
`\l__coffin_x_prime_dim` 32680 `\dim_new:N \l__coffin_y_dim`
`\l__coffin_y_prime_dim` 32681 `\dim_new:N \l__coffin_x_prime_dim`
32682 `\dim_new:N \l__coffin_y_prime_dim`

(End definition for `\l__coffin_x_dim` and others.)

85.2 Basic coffin functions

There are a number of basic functions needed for creating coffins and placing material in them. This all relies on the following data structures.

`__coffin_to_value:N` Coffins are a two-part structure and we rely on the internal nature of box allocation to make everything work. As such, we need an interface to turn coffin identifiers into numbers. For the purposes here, the signature allowed is N despite the nature of the underlying primitive.

32683 `\cs_new_eq:NN __coffin_to_value:N \tex_number:D`

(End definition for `__coffin_to_value:N`.)

`\coffin_if_exist_p:N` Several of the higher-level coffin functions would give multiple errors if the coffin does not exist. A cleaner way to handle this is provided here: both the box and the coffin structure are checked.

```

\coffin_if_exist_p:c
\coffin_if_exist:N $\overline{TF}$ 
\coffin_if_exist:c $\overline{TF}$ 
32684 \prg_new_conditional:Npnn \coffin_if_exist:N #1 { p , T , F , TF }
32685 {
32686   \cs_if_exist:NTF #1
32687   {
32688     \cs_if_exist:cTF { coffin ~ \__coffin_to_value:N #1 ~ poles }
32689     { \prg_return_true: }
32690     { \prg_return_false: }
32691   }
32692   { \prg_return_false: }
32693 }
32694 \prg_generate_conditional_variant:Nnn \coffin_if_exist:N
32695 { c } { p , T , F , TF }

```

(End definition for `\coffin_if_exist:NTF`. This function is documented on page 282.)

`__coffin_if_exist:NT` Several of the higher-level coffin functions would give multiple errors if the coffin does not exist. So a wrapper is provided to deal with this correctly, issuing an error on erroneous use.

```

32696 \cs_new_protected:Npn \__coffin_if_exist:NT #1#2
32697 {
32698   \coffin_if_exist:NTF #1
32699   { #2 }
32700   {
32701     \msg_error:nnx { coffin } { unknown }
32702     { \token_to_str:N #1 }
32703   }
32704 }

```

(End definition for `__coffin_if_exist:NT`.)

`\coffin_clear:N` Clearing coffins means emptying the box and resetting all of the structures.

```

\coffin_clear:c
\coffin_gclear:N
\coffin_gclear:c
32705 \cs_new_protected:Npn \coffin_clear:N #1
32706 {
32707   \__coffin_if_exist:NT #1
32708   {
32709     \box_clear:N #1
32710     \__coffin_reset_structure:N #1
32711   }
32712 }
32713 \cs_generate_variant:Nn \coffin_clear:N { c }
32714 \cs_new_protected:Npn \coffin_gclear:N #1
32715 {
32716   \__coffin_if_exist:NT #1
32717   {
32718     \box_gclear:N #1
32719     \__coffin_greset_structure:N #1
32720   }
32721 }
32722 \cs_generate_variant:Nn \coffin_gclear:N { c }

```

(End definition for `\coffin_clear:N` and `\coffin_gclear:N`. These functions are documented on page 282.)

\coffin_new:N Creating a new coffin means making the underlying box and adding the data structures. The **\debug_suspend:** and **\debug_resume:** functions prevent **\prop_gclear_new:c** from writing useless information to the log file.

```

32723 \cs_new_protected:Npn \coffin_new:N #1
32724 {
32725   \box_new:N #1
32726   \debug_suspend:
32727   \prop_gclear_new:c { coffin ~ \__coffin_to_value:N #1 ~ corners }
32728   \prop_gclear_new:c { coffin ~ \__coffin_to_value:N #1 ~ poles }
32729   \prop_gset_eq:cN { coffin ~ \__coffin_to_value:N #1 ~ corners }
32730     \c__coffin_corners_prop
32731   \prop_gset_eq:cN { coffin ~ \__coffin_to_value:N #1 ~ poles }
32732     \c__coffin_poles_prop
32733   \debug_resume:
32734 }
32735 \cs_generate_variant:Nn \coffin_new:N { c }

```

(End definition for **\coffin_new:N**. This function is documented on page 282.)

\hcoffin_set:Nn Horizontal coffins are relatively easy: set the appropriate box, reset the structures then
\hcoffin_set:cn update the handle positions.

```

\hcoffin_gset:Nn
\hcoffin_gset:cn
32736 \cs_new_protected:Npn \hcoffin_set:Nn #1#2
32737 {
32738   \__coffin_if_exist:NT #1
32739   {
32740     \hbox_set:Nn #1
32741     {
32742       \color_ensure_current:
32743       #2
32744     }
32745     \__coffin_update:N #1
32746   }
32747 }
32748 \cs_generate_variant:Nn \hcoffin_set:Nn { c }
32749 \cs_new_protected:Npn \hcoffin_gset:Nn #1#2
32750 {
32751   \__coffin_if_exist:NT #1
32752   {
32753     \hbox_gset:Nn #1
32754     {
32755       \color_ensure_current:
32756       #2
32757     }
32758     \__coffin_gupdate:N #1
32759   }
32760 }
32761 \cs_generate_variant:Nn \hcoffin_gset:Nn { c }

```

(End definition for **\hcoffin_set:Nn** and **\hcoffin_gset:Nn**. These functions are documented on page 283.)

\vcoffin_set:Nnn Setting vertical coffins is more complex. First, the material is typeset with a given width.
\vcoffin_set:cnn The default handles and poles are set as for a horizontal coffin, before finding the top
\vcoffin_gset:Nnn baseline using a temporary box. No **\color_ensure_current:** here as that would add a
\vcoffin_gset:cnn

```

\__coffin_set_vertical:NnnNN
\__coffin_set_vertical_aux:

```

whatsit to the start of the vertical box and mess up the location of the T pole (see *TeX by Topic* for discussion of the `\vtop` primitive, used to do the measuring).

```

32762 \cs_new_protected:Npn \vcoffin_set:Nnn #1#2#3
32763 {
32764   \__coffin_set_vertical:NnnNN #1 {#2} {#3}
32765   \vbox_set:Nn \__coffin_update:N
32766 }
32767 \cs_generate_variant:Nn \vcoffin_set:Nnn { c }
32768 \cs_new_protected:Npn \vcoffin_gset:Nnn #1#2#3
32769 {
32770   \__coffin_set_vertical:NnnNN #1 {#2} {#3}
32771   \vbox_gset:Nn \__coffin_gupdate:N
32772 }
32773 \cs_generate_variant:Nn \vcoffin_gset:Nnn { c }
32774 \cs_new_protected:Npn \__coffin_set_vertical:NnnNN #1#2#3#4#5
32775 {
32776   \__coffin_if_exist:NT #1
32777   {
32778     #4 #1
32779     {
32780       \dim_set:Nn \tex_hsize:D {#2}
32781       \__coffin_set_vertical_aux:
32782       #3
32783     }
32784     #5 #1
32785     \vbox_set_top:Nn \l__coffin_internal_box { \vbox_unpack:N #1 }
32786     \__coffin_set_pole:Nnx #1 { T }
32787     {
32788       { Opt }
32789       {
32790         \dim_eval:n
32791         { \box_ht:N #1 - \box_ht:N \l__coffin_internal_box }
32792       }
32793       { 1000pt }
32794       { Opt }
32795     }
32796     \box_clear:N \l__coffin_internal_box
32797   }
32798 }
32799 \cs_new_protected:Npx \__coffin_set_vertical_aux:
32800 {
32801   \bool_lazy_and:nnT
32802   { \cs_if_exist_p:N \fmtname }
32803   { \str_if_eq_p:Vn \fmtname { LaTeX2e } }
32804   {
32805     \dim_set_eq:NN \exp_not:N \linewidth \tex_hsize:D
32806     \dim_set_eq:NN \exp_not:N \columnwidth \tex_hsize:D
32807   }
32808 }

```

(End definition for `\vcoffin_set:Nnn` and others. These functions are documented on page 283.)

`\hcoffin_set:Nw` These are the “begin”/“end” versions of the above: watch the grouping!

`\hcoffin_set:cw` 32809 `\cs_new_protected:Npn \hcoffin_set:Nw #1`

`\hcoffin_gset:Nw`

`\hcoffin_gset:cw`

`\hcoffin_set_end:`

`\hcoffin_gset_end:`


```

32810 {
32811   \__coffin_if_exist:NT #1
32812   {
32813     \hbox_set:Nw #1 \color_ensure_current:
32814     \cs_set_protected:Npn \hcoffin_set_end:
32815     {
32816       \hbox_set_end:
32817       \__coffin_update:N #1
32818     }
32819   }
32820 }
32821 \cs_generate_variant:Nn \hcoffin_set:Nw { c }
32822 \cs_new_protected:Npn \hcoffin_gset:Nw #1
32823 {
32824   \__coffin_if_exist:NT #1
32825   {
32826     \hbox_gset:Nw #1 \color_ensure_current:
32827     \cs_set_protected:Npn \hcoffin_gset_end:
32828     {
32829       \hbox_gset_end:
32830       \__coffin_gupdate:N #1
32831     }
32832   }
32833 }
32834 \cs_generate_variant:Nn \hcoffin_gset:Nw { c }
32835 \cs_new_protected:Npn \hcoffin_set_end: { }
32836 \cs_new_protected:Npn \hcoffin_gset_end: { }

```

(End definition for `\hcoffin_set:Nw` and others. These functions are documented on page 283.)

```

\vcoffin_set:Nnw The same for vertical coffins.
\vcoffin_set:cnw
\vcoffin_gset:Nnw
\vcoffin_gset:cnw
\__coffin_set_vertical:NnNNNw
\vcoffin_set_end:
\vcoffin_gset_end:
32837 \cs_new_protected:Npn \vcoffin_set:Nnw #1#2
32838 {
32839   \__coffin_set_vertical:NnNNNw #1 {#2} \vbox_set:Nw
32840   \vcoffin_set_end:
32841   \vbox_set_end: \__coffin_update:N
32842 }
32843 \cs_generate_variant:Nn \vcoffin_set:Nnw { c }
32844 \cs_new_protected:Npn \vcoffin_gset:Nnw #1#2
32845 {
32846   \__coffin_set_vertical:NnNNNw #1 {#2} \vbox_gset:Nw
32847   \vcoffin_gset_end:
32848   \vbox_gset_end: \__coffin_gupdate:N
32849 }
32850 \cs_generate_variant:Nn \vcoffin_gset:Nnw { c }
32851 \cs_new_protected:Npn \__coffin_set_vertical:NnNNNw #1#2#3#4#5#6
32852 {
32853   \__coffin_if_exist:NT #1
32854   {
32855     #3 #1
32856     \dim_set:Nn \tex_hsize:D {#2}
32857     \__coffin_set_vertical_aux:
32858     \cs_set_protected:Npn #4
32859     {

```

```

32860         #5
32861         #6 #1
32862         \vbox_set_top:Nn \l__coffin_internal_box { \vbox_unpack:N #1 }
32863         \__coffin_set_pole:Nnx #1 { T }
32864         {
32865             { Opt }
32866             {
32867                 \dim_eval:n
32868                 { \box_ht:N #1 - \box_ht:N \l__coffin_internal_box }
32869             }
32870             { 1000pt }
32871             { Opt }
32872         }
32873         \box_clear:N \l__coffin_internal_box
32874     }
32875 }
32876 }
32877 \cs_new_protected:Npn \vcoffin_set_end: { }
32878 \cs_new_protected:Npn \vcoffin_gset_end: { }

```

(End definition for `\vcoffin_set:Nnw` and others. These functions are documented on page 283.)

```

\coffin_set_eq:NN Setting two coffins equal is just a wrapper around other functions.
\coffin_set_eq:Nc
\coffin_set_eq:cN
\coffin_set_eq:cc
\coffin_gset_eq:NN
\coffin_gset_eq:Nc
\coffin_gset_eq:cN
\coffin_gset_eq:cc
32879 \cs_new_protected:Npn \coffin_set_eq:NN #1#2
32880 {
32881     \__coffin_if_exist:NT #1
32882     {
32883         \box_set_eq:NN #1 #2
32884         \prop_set_eq:cc { coffin ~ \__coffin_to_value:N #1 ~ corners }
32885         { coffin ~ \__coffin_to_value:N #2 ~ corners }
32886         \prop_set_eq:cc { coffin ~ \__coffin_to_value:N #1 ~ poles }
32887         { coffin ~ \__coffin_to_value:N #2 ~ poles }
32888     }
32889 }
32890 \cs_generate_variant:Nn \coffin_set_eq:NN { c , Nc , cc }
32891 \cs_new_protected:Npn \coffin_gset_eq:NN #1#2
32892 {
32893     \__coffin_if_exist:NT #1
32894     {
32895         \box_gset_eq:NN #1 #2
32896         \prop_gset_eq:cc { coffin ~ \__coffin_to_value:N #1 ~ corners }
32897         { coffin ~ \__coffin_to_value:N #2 ~ corners }
32898         \prop_gset_eq:cc { coffin ~ \__coffin_to_value:N #1 ~ poles }
32899         { coffin ~ \__coffin_to_value:N #2 ~ poles }
32900     }
32901 }
32902 \cs_generate_variant:Nn \coffin_gset_eq:NN { c , Nc , cc }

```

(End definition for `\coffin_set_eq:NN` and `\coffin_gset_eq:NN`. These functions are documented on page 282.)

\c_empty_coffin Special coffins: these cannot be set up earlier as they need `\coffin_new:N`. The empty coffin is set as a box as the full coffin-setting system needs some material which is not yet available. The empty coffin is created entirely by hand: not everything is in place yet.

```

\l__coffin_aligned_coffin
\l__coffin_aligned_internal_coffin

```

```

32903 \coffin_new:N \c_empty_coffin
32904 \coffin_new:N \l__coffin_aligned_coffin
32905 \coffin_new:N \l__coffin_aligned_internal_coffin

```

(End definition for `\c_empty_coffin`, `\l__coffin_aligned_coffin`, and `\l__coffin_aligned_internal_coffin`. This variable is documented on page 286.)

```

\l_tmpa_coffin The usual scratch space.
\l_tmpb_coffin 32906 \coffin_new:N \l_tmpa_coffin
\g_tmpa_coffin 32907 \coffin_new:N \l_tmpb_coffin
\g_tmpb_coffin 32908 \coffin_new:N \g_tmpa_coffin
               32909 \coffin_new:N \g_tmpb_coffin

```

(End definition for `\l_tmpa_coffin` and others. These variables are documented on page 286.)

85.3 Measuring coffins

`\coffin_dp:N` Coffins are just boxes when it comes to measurement. However, semantically a separate set of functions are required.

```

\coffin_dp:c 32910 \cs_new_eq:NN \coffin_dp:N \box_dp:N
\coffin_ht:N 32911 \cs_new_eq:NN \coffin_dp:c \box_dp:c
\coffin_ht:c 32912 \cs_new_eq:NN \coffin_ht:N \box_ht:N
\coffin_wd:N 32913 \cs_new_eq:NN \coffin_ht:c \box_ht:c
\coffin_wd:c 32914 \cs_new_eq:NN \coffin_wd:N \box_wd:N
               32915 \cs_new_eq:NN \coffin_wd:c \box_wd:c

```

(End definition for `\coffin_dp:N`, `\coffin_ht:N`, and `\coffin_wd:N`. These functions are documented on page 285.)

85.4 Coffins: handle and pole management

`__coffin_get_pole:NnN` A simple wrapper around the recovery of a coffin pole, with some error checking and recovery built-in.

```

32916 \cs_new_protected:Npn \__coffin_get_pole:NnN #1#2#3
32917 {
32918   \prop_get:cnNF
32919     { coffin ~ \__coffin_to_value:N #1 ~ poles } {#2} #3
32920   {
32921     \msg_error:nxxx { coffin } { unknown-pole }
32922     { \exp_not:n {#2} } { \token_to_str:N #1 }
32923     \tl_set:Nn #3 { { Opt } { Opt } { Opt } { Opt } }
32924   }
32925 }

```

(End definition for `__coffin_get_pole:NnN`.)

`__coffin_reset_structure:N` Resetting the structure is a simple copy job.

```

\__coffin_greset_structure:N 32926 \cs_new_protected:Npn \__coffin_reset_structure:N #1
                             32927 {
                             32928   \prop_set_eq:cN { coffin ~ \__coffin_to_value:N #1 ~ corners }
                             32929   \c__coffin_corners_prop
                             32930   \prop_set_eq:cN { coffin ~ \__coffin_to_value:N #1 ~ poles }
                             32931   \c__coffin_poles_prop

```

```

32932 }
32933 \cs_new_protected:Npn \__coffin_greset_structure:N #1
32934 {
32935   \prop_gset_eq:cN { coffin ~ \__coffin_to_value:N #1 ~ corners }
32936   \c__coffin_corners_prop
32937   \prop_gset_eq:cN { coffin ~ \__coffin_to_value:N #1 ~ poles }
32938   \c__coffin_poles_prop
32939 }

```

(End definition for __coffin_reset_structure:N and __coffin_greset_structure:N.)

\coffin_set_horizontal_pole:Nnn
\coffin_set_horizontal_pole:cnm
\coffin_gset_horizontal_pole:Nnn
\coffin_gset_horizontal_pole:cnm
__coffin_set_horizontal_pole:NnnN
\coffin_set_vertical_pole:Nnn
\coffin_set_vertical_pole:cnm
\coffin_gset_vertical_pole:Nnn
\coffin_gset_vertical_pole:cnm
__coffin_set_vertical_pole:NnnN
__coffin_set_pole:Nnn
__coffin_set_pole:Nnx

Setting the pole of a coffin at the user/designer level requires a bit more care. The idea here is to provide a reasonable interface to the system, then to do the setting with full expansion. The three-argument version is used internally to do a direct setting.

```

32940 \cs_new_protected:Npn \coffin_set_horizontal_pole:Nnn #1#2#3
32941 { \__coffin_set_horizontal_pole:NnnN #1 {#2} {#3} \prop_put:cnx }
32942 \cs_generate_variant:Nn \coffin_set_horizontal_pole:Nnn { c }
32943 \cs_new_protected:Npn \coffin_gset_horizontal_pole:Nnn #1#2#3
32944 { \__coffin_set_horizontal_pole:NnnN #1 {#2} {#3} \prop_gput:cnx }
32945 \cs_generate_variant:Nn \coffin_gset_horizontal_pole:Nnn { c }
32946 \cs_new_protected:Npn \__coffin_set_horizontal_pole:NnnN #1#2#3#4
32947 {
32948   \__coffin_if_exist:NT #1
32949   {
32950     #4 { coffin ~ \__coffin_to_value:N #1 ~ poles }
32951     {#2}
32952     {
32953       { Opt } { \dim_eval:n {#3} }
32954       { 1000pt } { Opt }
32955     }
32956   }
32957 }
32958 \cs_new_protected:Npn \coffin_set_vertical_pole:Nnn #1#2#3
32959 { \__coffin_set_vertical_pole:NnnN #1 {#2} {#3} \prop_put:cnx }
32960 \cs_generate_variant:Nn \coffin_set_vertical_pole:Nnn { c }
32961 \cs_new_protected:Npn \coffin_gset_vertical_pole:Nnn #1#2#3
32962 { \__coffin_set_vertical_pole:NnnN #1 {#2} {#3} \prop_gput:cnx }
32963 \cs_generate_variant:Nn \coffin_gset_vertical_pole:Nnn { c }
32964 \cs_new_protected:Npn \__coffin_set_vertical_pole:NnnN #1#2#3#4
32965 {
32966   \__coffin_if_exist:NT #1
32967   {
32968     #4 { coffin ~ \__coffin_to_value:N #1 ~ poles }
32969     {#2}
32970     {
32971       { \dim_eval:n {#3} } { Opt }
32972       { Opt } { 1000pt }
32973     }
32974   }
32975 }
32976 \cs_new_protected:Npn \__coffin_set_pole:Nnn #1#2#3
32977 {
32978   \prop_put:cnm { coffin ~ \__coffin_to_value:N #1 ~ poles }
32979   {#2} {#3}

```

```

32980 }
32981 \cs_generate_variant:Nn \__coffin_set_pole:Nnn { Nnx }

```

(End definition for `\coffin_set_horizontal_pole:Nnn` and others. These functions are documented on page 283.)

`__coffin_update:N` Simple shortcuts.
`__coffin_gupdate:N`

```

32982 \cs_new_protected:Npn \__coffin_update:N #1
32983 {
32984   \__coffin_reset_structure:N #1
32985   \__coffin_update_corners:N #1
32986   \__coffin_update_poles:N #1
32987 }
32988 \cs_new_protected:Npn \__coffin_gupdate:N #1
32989 {
32990   \__coffin_greset_structure:N #1
32991   \__coffin_gupdate_corners:N #1
32992   \__coffin_gupdate_poles:N #1
32993 }

```

(End definition for `__coffin_update:N` and `__coffin_gupdate:N`.)

`__coffin_update_corners:N` Updating the corners of a coffin is straight-forward as at this stage there can be no
`__coffin_gupdate_corners:N` rotation. So the corners of the content are just those of the underlying \TeX box.
`__coffin_update_corners:NN`
`__coffin_update_corners:NNN`

```

32994 \cs_new_protected:Npn \__coffin_update_corners:N #1
32995 { \__coffin_update_corners:NN #1 \prop_put:Nnx }
32996 \cs_new_protected:Npn \__coffin_gupdate_corners:N #1
32997 { \__coffin_update_corners:NN #1 \prop_gput:Nnx }
32998 \cs_new_protected:Npn \__coffin_update_corners:NN #1#2
32999 {
33000   \exp_args:Nc \__coffin_update_corners:NNN
33001   { coffin ~ \__coffin_to_value:N #1 ~ corners }
33002   #1 #2
33003 }
33004 \cs_new_protected:Npn \__coffin_update_corners:NNN #1#2#3
33005 {
33006   #3 #1
33007   { tl }
33008   { { Opt } { \dim_eval:n { \box_ht:N #2 } } } }
33009   #3 #1
33010   { tr }
33011   {
33012     { \dim_eval:n { \box_wd:N #2 } }
33013     { \dim_eval:n { \box_ht:N #2 } }
33014   }
33015   #3 #1
33016   { bl }
33017   { { Opt } { \dim_eval:n { -\box_dp:N #2 } } } }
33018   #3 #1
33019   { br }
33020   {
33021     { \dim_eval:n { \box_wd:N #2 } }
33022     { \dim_eval:n { -\box_dp:N #2 } }
33023   }
33024 }

```

(End definition for `_coffin_update_corners:N` and others.)

`_coffin_update_poles:N`
`_coffin_gupdate_poles:N`
`_coffin_update_poles:NN`
`_coffin_update_poles:NNN`

This function is called when a coffin is set, and updates the poles to reflect the nature of size of the box. Thus this function only alters poles where the default position is dependent on the size of the box. It also does not set poles which are relevant only to vertical coffins.

```

33025 \cs_new_protected:Npn \_coffin_update_poles:N #1
33026   { \_coffin_update_poles:NN #1 \prop_put:Nnx }
33027 \cs_new_protected:Npn \_coffin_gupdate_poles:N #1
33028   { \_coffin_update_poles:NN #1 \prop_gput:Nnx }
33029 \cs_new_protected:Npn \_coffin_update_poles:NN #1#2
33030   {
33031     \exp_args:Nc \_coffin_update_poles:NNN
33032     { coffin ~ \_coffin_to_value:N #1 ~ poles }
33033     #1 #2
33034   }
33035 \cs_new_protected:Npn \_coffin_update_poles:NNN #1#2#3
33036   {
33037     #3 #1 { hc }
33038     {
33039       { \dim_eval:n { 0.5 \box_wd:N #2 } }
33040       { 0pt } { 0pt } { 1000pt }
33041     }
33042     #3 #1 { r }
33043     {
33044       { \dim_eval:n { \box_wd:N #2 } }
33045       { 0pt } { 0pt } { 1000pt }
33046     }
33047     #3 #1 { vc }
33048     {
33049       { 0pt }
33050       { \dim_eval:n { ( \box_ht:N #2 - \box_dp:N #2 ) / 2 } }
33051       { 1000pt }
33052       { 0pt }
33053     }
33054     #3 #1 { t }
33055     {
33056       { 0pt }
33057       { \dim_eval:n { \box_ht:N #2 } }
33058       { 1000pt }
33059       { 0pt }
33060     }
33061     #3 #1 { b }
33062     {
33063       { 0pt }
33064       { \dim_eval:n { -\box_dp:N #2 } }
33065       { 1000pt }
33066       { 0pt }
33067     }
33068   }

```

(End definition for `_coffin_update_poles:N` and others.)

85.5 Coffins: calculation of pole intersections

The lead off in finding intersections is to recover the two poles and then hand off to the auxiliary for the actual calculation. There may of course not be an intersection, for which an error trap is needed.

```

33069 \cs_new_protected:Npn \__coffin_calculate_intersection:Nnn #1#2#3
33070 {
33071   \__coffin_get_pole:NnN #1 {#2} \l__coffin_pole_a_tl
33072   \__coffin_get_pole:NnN #1 {#3} \l__coffin_pole_b_tl
33073   \bool_set_false:N \l__coffin_error_bool
33074   \exp_last_two_unbraced:Noo
33075   \__coffin_calculate_intersection:nnnnnnnn
33076   \l__coffin_pole_a_tl \l__coffin_pole_b_tl
33077   \bool_if:NT \l__coffin_error_bool
33078   {
33079     \msg_error:nn { coffin } { no-pole-intersection }
33080     \dim_zero:N \l__coffin_x_dim
33081     \dim_zero:N \l__coffin_y_dim
33082   }
33083 }
```

The two poles passed here each have four values (as dimensions), (a, b, c, d) and (a', b', c', d') . These are arguments 1–4 and 5–8, respectively. In both cases a and b are the co-ordinates of a point on the pole and c and d define the direction of the pole. Finding the intersection depends on the directions of the poles, which are given by d/c and d'/c' . However, if one of the poles is either horizontal or vertical then one or more of c, d, c' and d' are zero and a special case is needed.

```

33084 \cs_new_protected:Npn \__coffin_calculate_intersection:nnnnnnnn
33085   #1#2#3#4#5#6#7#8
33086   {
33087     \dim_compare:nNnTF {#3} = \c_zero_dim
```

The case where the first pole is vertical. So the x -component of the interaction is at a . There is then a test on the second pole: if it is also vertical then there is an error.

```

33088   {
33089     \dim_set:Nn \l__coffin_x_dim {#1}
33090     \dim_compare:nNnTF {#7} = \c_zero_dim
33091     { \bool_set_true:N \l__coffin_error_bool }
```

The second pole may still be horizontal, in which case the y -component of the intersection is b' . If not,

$$y = \frac{d'}{c'}(a - a') + b'$$

with the x -component already known to be $\#1$.

```

33092   {
33093     \dim_set:Nn \l__coffin_y_dim
33094     {
33095       \dim_compare:nNnTF {#8} = \c_zero_dim
33096       {#6}
33097       {
33098         \fp_to_dim:n
33099         {
33100           ( \dim_to_fp:n {#8} / \dim_to_fp:n {#7} )
33101           * ( \dim_to_fp:n {#1} - \dim_to_fp:n {#5} )
```

```

33102         + \dim_to_fp:n {#6}
33103     }
33104 }
33105 }
33106 }
33107 }

```

If the first pole is not vertical then it may be horizontal. If so, then the procedure is essentially the same as that already done but with the x - and y -components interchanged.

```

33108 {
33109     \dim_compare:nNnTF {#4} = \c_zero_dim
33110     {
33111         \dim_set:Nn \l__coffin_y_dim {#2}
33112         \dim_compare:nNnTF {#8} = { \c_zero_dim }
33113         { \bool_set_true:N \l__coffin_error_bool }
33114     }

```

Now we deal with the case where the second pole may be vertical, or if not we have

$$x = \frac{c'}{d'} (b - b') + a'$$

which is again handled by the same auxiliary.

```

33115     \dim_set:Nn \l__coffin_x_dim
33116     {
33117         \dim_compare:nNnTF {#7} = \c_zero_dim
33118         {#5}
33119         {
33120             \fp_to_dim:n
33121             {
33122                 ( \dim_to_fp:n {#7} / \dim_to_fp:n {#8} )
33123                 * ( \dim_to_fp:n {#4} - \dim_to_fp:n {#6} )
33124                 + \dim_to_fp:n {#5}
33125             }
33126         }
33127     }
33128 }
33129 }

```

The first pole is neither horizontal nor vertical. To avoid even more complexity, we now work out both slopes and pass to an auxiliary.

```

33130 {
33131     \use:x
33132     {
33133         \__coffin_calculate_intersection:nnnnnn
33134         { \dim_to_fp:n {#4} / \dim_to_fp:n {#3} }
33135         { \dim_to_fp:n {#8} / \dim_to_fp:n {#7} }
33136     }
33137     {#1} {#2} {#5} {#6}
33138 }
33139 }
33140 }

```

Assuming the two poles are not parallel, then the intersection point is found in two steps. First we find the x -value with

$$x = \frac{sa - s'a' - b + b'}{s - s'}$$

and then finding the y -value with

$$y = s(x - a) + b$$

```

33141 \cs_set_protected:Npn \__coffin_calculate_intersection:nnnnnn #1#2#3#4#5#6
33142 {
33143   \fp_compare:nNnTF {#1} = {#2}
33144   { \bool_set_true:N \l__coffin_error_bool }
33145   {
33146     \dim_set:Nn \l__coffin_x_dim
33147     {
33148       \fp_to_dim:n
33149       {
33150         (
33151           #1 * \dim_to_fp:n {#3}
33152           - #2 * \dim_to_fp:n {#5}
33153           - \dim_to_fp:n {#4}
33154           + \dim_to_fp:n {#6}
33155         )
33156         /
33157         ( #1 - #2 )
33158       }
33159     }
33160     \dim_set:Nn \l__coffin_y_dim
33161     {
33162       \fp_to_dim:n
33163       {
33164         #1 * ( \l__coffin_x_dim - \dim_to_fp:n {#3} )
33165         + \dim_to_fp:n {#4}
33166       }
33167     }
33168   }
33169 }

```

(End definition for `__coffin_calculate_intersection:Nnn`, `__coffin_calculate_intersection:nnnnnnnn`, and `__coffin_calculate_intersection:nnnnnn`.)

85.6 Affine transformations

`\l__coffin_sin_fp` Used for rotations to get the sine and cosine values.
`\l__coffin_cos_fp`

```

33170 \fp_new:N \l__coffin_sin_fp
33171 \fp_new:N \l__coffin_cos_fp

```

(End definition for `\l__coffin_sin_fp` and `\l__coffin_cos_fp`.)

`\l__coffin_bounding_prop` A property list for the bounding box of a coffin. This is only needed during the rotation, so there is just the one.

```

33172 \prop_new:N \l__coffin_bounding_prop

```

(End definition for `\l__coffin_bounding_prop`.)

`\l__coffin_corners_prop` Used to avoid needing to track scope for intermediate steps.

```

\l__coffin_poles_prop
33173 \prop_new:N \l__coffin_corners_prop
33174 \prop_new:N \l__coffin_poles_prop

```

(End definition for \l__coffin_corners_prop and \l__coffin_poles_prop.)

\l__coffin_bounding_shift_dim The shift of the bounding box of a coffin from the real content.

```
33175 \dim_new:N \l__coffin_bounding_shift_dim
```

(End definition for \l__coffin_bounding_shift_dim.)

\l__coffin_left_corner_dim \l__coffin_right_corner_dim \l__coffin_bottom_corner_dim \l__coffin_top_corner_dim These are used to hold maxima for the various corner values: these thus define the minimum size of the bounding box after rotation.

```
33176 \dim_new:N \l__coffin_left_corner_dim
```

```
33177 \dim_new:N \l__coffin_right_corner_dim
```

```
33178 \dim_new:N \l__coffin_bottom_corner_dim
```

```
33179 \dim_new:N \l__coffin_top_corner_dim
```

(End definition for \l__coffin_left_corner_dim and others.)

\coffin_rotate:Nn Rotating a coffin requires several steps which can be conveniently run together. The sine and cosine of the angle in degrees are computed. This is then used to set \l__coffin_sin_fp and \l__coffin_cos_fp, which are carried through unchanged for the rest of the procedure.

\coffin_rotate:cn

\coffin_grotate:Nn

\coffin_grotate:cn

__coffin_rotate:NnNNN

```
33180 \cs_new_protected:Npn \coffin_rotate:Nn #1#2
```

```
33181 { \__coffin_rotate:NnNNN #1 {#2} \box_rotate:Nn \prop_set_eq:cn \hbox_set:Nn }
```

```
33182 \cs_generate_variant:Nn \coffin_rotate:Nn { c }
```

```
33183 \cs_new_protected:Npn \coffin_grotate:Nn #1#2
```

```
33184 { \__coffin_rotate:NnNNN #1 {#2} \box_grotate:Nn \prop_gset_eq:cn \hbox_gset:Nn }
```

```
33185 \cs_generate_variant:Nn \coffin_grotate:Nn { c }
```

```
33186 \cs_new_protected:Npn \__coffin_rotate:NnNNN #1#2#3#4#5
```

```
33187 {
```

```
33188   \fp_set:Nn \l__coffin_sin_fp { sind ( #2 ) }
```

```
33189   \fp_set:Nn \l__coffin_cos_fp { cosd ( #2 ) }
```

Use a local copy of the property lists to avoid needing to pass the name and scope around.

```
33190   \prop_set_eq:Nc \l__coffin_corners_prop
```

```
33191   { coffin ~ \__coffin_to_value:N #1 ~ corners }
```

```
33192   \prop_set_eq:Nc \l__coffin_poles_prop
```

```
33193   { coffin ~ \__coffin_to_value:N #1 ~ poles }
```

The corners and poles of the coffin can now be rotated around the origin. This is best achieved using mapping functions.

```
33194   \prop_map_inline:Nn \l__coffin_corners_prop
```

```
33195   { \__coffin_rotate_corner:Nnnn #1 {##1} ##2 }
```

```
33196   \prop_map_inline:Nn \l__coffin_poles_prop
```

```
33197   { \__coffin_rotate_pole:Nnnnnn #1 {##1} ##2 }
```

The bounding box of the coffin needs to be rotated, and to do this the corners have to be found first. They are then rotated in the same way as the corners of the coffin material itself.

```
33198   \__coffin_set_bounding:N #1
```

```
33199   \prop_map_inline:Nn \l__coffin_bounding_prop
```

```
33200   { \__coffin_rotate_bounding:nnn {##1} ##2 }
```

At this stage, there needs to be a calculation to find where the corners of the content and the box itself will end up.

```
33201   \__coffin_find_corner_maxima:N #1
```

```
33202   \__coffin_find_bounding_shift:
```

```
33203   #3 #1 {#2}
```

The correction of the box position itself takes place here. The idea is that the bounding box for a coffin is tight up to the content, and has the reference point at the bottom-left. The x -direction is handled by moving the content by the difference in the positions of the bounding box and the content left edge. The y -direction is dealt with by moving the box down by any depth it has acquired. The internal box is used here to allow for the next step.

```

33204     \hbox_set:Nn \l__coffin_internal_box
33205     {
33206         \__kernel_kern:n
33207         { \l__coffin_bounding_shift_dim - \l__coffin_left_corner_dim }
33208         \box_move_down:nn { \l__coffin_bottom_corner_dim }
33209         { \box_use:N #1 }
33210     }

```

If there have been any previous rotations then the size of the bounding box will be bigger than the contents. This can be corrected easily by setting the size of the box to the height and width of the content. As this operation requires setting box dimensions and these transcend grouping, the safe way to do this is to use the internal box and to reset the result into the target box.

```

33211     \box_set_ht:Nn \l__coffin_internal_box
33212     { \l__coffin_top_corner_dim - \l__coffin_bottom_corner_dim }
33213     \box_set_dp:Nn \l__coffin_internal_box { Opt }
33214     \box_set_wd:Nn \l__coffin_internal_box
33215     { \l__coffin_right_corner_dim - \l__coffin_left_corner_dim }
33216     #5 #1 { \box_use_drop:N \l__coffin_internal_box }

```

The final task is to move the poles and corners such that they are back in alignment with the box reference point.

```

33217     \prop_map_inline:Nn \l__coffin_corners_prop
33218     { \__coffin_shift_corner:Nnnn #1 {##1} ##2 }
33219     \prop_map_inline:Nn \l__coffin_poles_prop
33220     { \__coffin_shift_pole:Nnnnn #1 {##1} ##2 }

```

Update the coffin data.

```

33221     #4 { coffin ~ \__coffin_to_value:N #1 ~ corners }
33222     \l__coffin_corners_prop
33223     #4 { coffin ~ \__coffin_to_value:N #1 ~ poles }
33224     \l__coffin_poles_prop
33225 }

```

(End definition for \coffin_rotate:Nn, \coffin_grotate:Nn, and __coffin_rotate:NnNNN. These functions are documented on page 284.)

`__coffin_set_bounding:N` The bounding box corners for a coffin are easy enough to find: this is the same code as for the corners of the material itself, but using a dedicated property list.

```

33226     \cs_new_protected:Npn \__coffin_set_bounding:N #1
33227     {
33228         \prop_put:Nnx \l__coffin_bounding_prop { tl }
33229         { { Opt } { \dim_eval:n { \box_ht:N #1 } } }
33230         \prop_put:Nnx \l__coffin_bounding_prop { tr }
33231         {
33232             { \dim_eval:n { \box_wd:N #1 } }
33233             { \dim_eval:n { \box_ht:N #1 } }
33234         }

```

```

33235 \dim_set:Nn \l__coffin_internal_dim { -\box_dp:N #1 }
33236 \prop_put:Nnx \l__coffin_bounding_prop { bl }
33237 { { Opt } { \dim_use:N \l__coffin_internal_dim } }
33238 \prop_put:Nnx \l__coffin_bounding_prop { br }
33239 {
33240 { \dim_eval:n { \box_wd:N #1 } }
33241 { \dim_use:N \l__coffin_internal_dim }
33242 }
33243 }

```

(End definition for __coffin_set_bounding:N.)

__coffin_rotate_bounding:nnn

Rotating the position of the corner of the coffin is just a case of treating this as a vector from the reference point. The same treatment is used for the corners of the material itself and the bounding box.

```

33244 \cs_new_protected:Npn \__coffin_rotate_bounding:nnn #1#2#3
33245 {
33246 \__coffin_rotate_vector:nnNN {#2} {#3} \l__coffin_x_dim \l__coffin_y_dim
33247 \prop_put:Nnx \l__coffin_bounding_prop {#1}
33248 { { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim } }
33249 }
33250 \cs_new_protected:Npn \__coffin_rotate_corner:Nnnn #1#2#3#4
33251 {
33252 \__coffin_rotate_vector:nnNN {#3} {#4} \l__coffin_x_dim \l__coffin_y_dim
33253 \prop_put:Nnx \l__coffin_corners_prop {#2}
33254 { { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim } }
33255 }

```

(End definition for __coffin_rotate_bounding:nnn and __coffin_rotate_corner:Nnnn.)

__coffin_rotate_pole:Nnnnnn

Rotating a single pole simply means shifting the co-ordinate of the pole and its direction. The rotation here is about the bottom-left corner of the coffin.

```

33256 \cs_new_protected:Npn \__coffin_rotate_pole:Nnnnnn #1#2#3#4#5#6
33257 {
33258 \__coffin_rotate_vector:nnNN {#3} {#4} \l__coffin_x_dim \l__coffin_y_dim
33259 \__coffin_rotate_vector:nnNN {#5} {#6}
33260 \l__coffin_x_prime_dim \l__coffin_y_prime_dim
33261 \prop_put:Nnx \l__coffin_poles_prop {#2}
33262 {
33263 { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim }
33264 { \dim_use:N \l__coffin_x_prime_dim }
33265 { \dim_use:N \l__coffin_y_prime_dim }
33266 }
33267 }

```

(End definition for __coffin_rotate_pole:Nnnnnn.)

__coffin_rotate_vector:nnNN

A rotation function, which needs only an input vector (as dimensions) and an output space. The values \l__coffin_cos_fp and \l__coffin_sin_fp should previously have been set up correctly. Working this way means that the floating point work is kept to a minimum: for any given rotation the sin and cosine values do no change, after all.

```

33268 \cs_new_protected:Npn \__coffin_rotate_vector:nnNN #1#2#3#4
33269 {
33270 \dim_set:Nn #3

```

```

33271     {
33272         \fp_to_dim:n
33273         {
33274             \dim_to_fp:n {#1} * \l__coffin_cos_fp
33275             - \dim_to_fp:n {#2} * \l__coffin_sin_fp
33276         }
33277     }
33278     \dim_set:Nn #4
33279     {
33280         \fp_to_dim:n
33281         {
33282             \dim_to_fp:n {#1} * \l__coffin_sin_fp
33283             + \dim_to_fp:n {#2} * \l__coffin_cos_fp
33284         }
33285     }
33286 }

```

(End definition for __coffin_rotate_vector:nnNN.)

__coffin_find_corner_maxima:N
__coffin_find_corner_maxima_aux:nn

The idea here is to find the extremities of the content of the coffin. This is done by looking for the smallest values for the bottom and left corners, and the largest values for the top and right corners. The values start at the maximum dimensions so that the case where all are positive or all are negative works out correctly.

```

33287 \cs_new_protected:Npn \__coffin_find_corner_maxima:N #1
33288 {
33289     \dim_set:Nn \l__coffin_top_corner_dim { -\c_max_dim }
33290     \dim_set:Nn \l__coffin_right_corner_dim { -\c_max_dim }
33291     \dim_set:Nn \l__coffin_bottom_corner_dim { \c_max_dim }
33292     \dim_set:Nn \l__coffin_left_corner_dim { \c_max_dim }
33293     \prop_map_inline:Nn \l__coffin_corners_prop
33294     { \__coffin_find_corner_maxima_aux:nn ##2 }
33295 }
33296 \cs_new_protected:Npn \__coffin_find_corner_maxima_aux:nn #1#2
33297 {
33298     \dim_set:Nn \l__coffin_left_corner_dim
33299     { \dim_min:nn { \l__coffin_left_corner_dim } {#1} }
33300     \dim_set:Nn \l__coffin_right_corner_dim
33301     { \dim_max:nn { \l__coffin_right_corner_dim } {#1} }
33302     \dim_set:Nn \l__coffin_bottom_corner_dim
33303     { \dim_min:nn { \l__coffin_bottom_corner_dim } {#2} }
33304     \dim_set:Nn \l__coffin_top_corner_dim
33305     { \dim_max:nn { \l__coffin_top_corner_dim } {#2} }
33306 }

```

(End definition for __coffin_find_corner_maxima:N and __coffin_find_corner_maxima_aux:nn.)

__coffin_find_bounding_shift:
__coffin_find_bounding_shift_aux:nn

The approach to finding the shift for the bounding box is similar to that for the corners. However, there is only one value needed here and a fixed input property list, so things are a bit clearer.

```

33307 \cs_new_protected:Npn \__coffin_find_bounding_shift:
33308 {
33309     \dim_set:Nn \l__coffin_bounding_shift_dim { \c_max_dim }
33310     \prop_map_inline:Nn \l__coffin_bounding_prop
33311     { \__coffin_find_bounding_shift_aux:nn ##2 }

```

```

33312 }
33313 \cs_new_protected:Npn \__coffin_find_bounding_shift_aux:nn #1#2
33314 {
33315   \dim_set:Nn \l__coffin_bounding_shift_dim
33316   { \dim_min:nn { \l__coffin_bounding_shift_dim } {#1} }
33317 }

```

(End definition for __coffin_find_bounding_shift: and __coffin_find_bounding_shift_aux:nn.)

__coffin_shift_corner:Nnnn Shifting the corners and poles of a coffin means subtracting the appropriate values from the x - and y -components. For the poles, this means that the direction vector is unchanged.

__coffin_shift_pole:Nnnnnn

```

33318 \cs_new_protected:Npn \__coffin_shift_corner:Nnnn #1#2#3#4
33319 {
33320   \prop_put:Nnx \l__coffin_corners_prop {#2}
33321   {
33322     { \dim_eval:n { #3 - \l__coffin_left_corner_dim } }
33323     { \dim_eval:n { #4 - \l__coffin_bottom_corner_dim } }
33324   }
33325 }
33326 \cs_new_protected:Npn \__coffin_shift_pole:Nnnnnn #1#2#3#4#5#6
33327 {
33328   \prop_put:Nnx \l__coffin_poles_prop {#2}
33329   {
33330     { \dim_eval:n { #3 - \l__coffin_left_corner_dim } }
33331     { \dim_eval:n { #4 - \l__coffin_bottom_corner_dim } }
33332     {#5} {#6}
33333   }
33334 }

```

(End definition for __coffin_shift_corner:Nnnn and __coffin_shift_pole:Nnnnnn.)

\l__coffin_scale_x_fp Storage for the scaling factors in x and y , respectively.

\l__coffin_scale_y_fp

```

33335 \fp_new:N \l__coffin_scale_x_fp
33336 \fp_new:N \l__coffin_scale_y_fp

```

(End definition for \l__coffin_scale_x_fp and \l__coffin_scale_y_fp.)

\l__coffin_scaled_total_height_dim When scaling, the values given have to be turned into absolute values.

\l__coffin_scaled_width_dim

```

33337 \dim_new:N \l__coffin_scaled_total_height_dim
33338 \dim_new:N \l__coffin_scaled_width_dim

```

(End definition for \l__coffin_scaled_total_height_dim and \l__coffin_scaled_width_dim.)

\coffin_resize:Nnn Resizing a coffin begins by setting up the user-friendly names for the dimensions of the coffin box. The new sizes are then turned into scale factor. This is the same operation as takes place for the underlying box, but that operation is grouped and so the same calculation is done here.

\coffin_resize:cnn

\coffin_gresize:Nnn

\coffin_gresize:cnn

__coffin_resize:NnnNN

```

33339 \cs_new_protected:Npn \coffin_resize:Nnn #1#2#3
33340 {
33341   \__coffin_resize:NnnNN #1 {#2} {#3}
33342   \box_resize_to_wd_and_ht_plus_dp:Nnn
33343   \prop_set_eq:cN
33344 }

```

```

33345 \cs_generate_variant:Nn \coffin_resize:Nnn { c }
33346 \cs_new_protected:Npn \coffin_gresize:Nnn #1#2#3
33347 {
33348   \__coffin_resize:NnnNN #1 {#2} {#3}
33349   \box_gresize_to_wd_and_ht_plus_dp:Nnn
33350   \prop_gset_eq:cN
33351 }
33352 \cs_generate_variant:Nn \coffin_gresize:Nnn { c }
33353 \cs_new_protected:Npn \__coffin_resize:NnnNN #1#2#3#4#5
33354 {
33355   \fp_set:Nn \l__coffin_scale_x_fp
33356   { \dim_to_fp:n {#2} / \dim_to_fp:n { \coffin_wd:N #1 } }
33357   \fp_set:Nn \l__coffin_scale_y_fp
33358   {
33359     \dim_to_fp:n {#3}
33360     / \dim_to_fp:n { \coffin_ht:N #1 + \coffin_dp:N #1 }
33361   }
33362   #4 #1 {#2} {#3}
33363   \__coffin_resize_common:NnnN #1 {#2} {#3} #5
33364 }

```

(End definition for `\coffin_resize:Nnn`, `\coffin_gresize:Nnn`, and `__coffin_resize:NnnNN`. These functions are documented on page 284.)

`__coffin_resize_common:NnnN` The poles and corners of the coffin are scaled to the appropriate places before actually resizing the underlying box.

```

33365 \cs_new_protected:Npn \__coffin_resize_common:NnnN #1#2#3#4
33366 {
33367   \prop_set_eq:Nc \l__coffin_corners_prop
33368   { coffin ~ \__coffin_to_value:N #1 ~ corners }
33369   \prop_set_eq:Nc \l__coffin_poles_prop
33370   { coffin ~ \__coffin_to_value:N #1 ~ poles }
33371   \prop_map_inline:Nn \l__coffin_corners_prop
33372   { \__coffin_scale_corner:Nnnn #1 {##1} ##2 }
33373   \prop_map_inline:Nn \l__coffin_poles_prop
33374   { \__coffin_scale_pole:Nnnnnn #1 {##1} ##2 }

```

Negative x -scaling values place the poles in the wrong location: this is corrected here.

```

33375   \fp_compare:nNnT \l__coffin_scale_x_fp < \c_zero_fp
33376   {
33377     \prop_map_inline:Nn \l__coffin_corners_prop
33378     { \__coffin_x_shift_corner:Nnnn #1 {##1} ##2 }
33379     \prop_map_inline:Nn \l__coffin_poles_prop
33380     { \__coffin_x_shift_pole:Nnnnnn #1 {##1} ##2 }
33381   }
33382   #4 { coffin ~ \__coffin_to_value:N #1 ~ corners }
33383   \l__coffin_corners_prop
33384   #4 { coffin ~ \__coffin_to_value:N #1 ~ poles }
33385   \l__coffin_poles_prop
33386 }

```

(End definition for `__coffin_resize_common:NnnN`.)

`\coffin_scale:Nnn` For scaling, the opposite calculation is done to find the new dimensions for the coffin.
`\coffin_scale:cnn` Only the total height is needed, as this is the shift required for corners and poles. The
`\coffin_gscale:Nnn`
`\coffin_gscale:cnn`
`\coffin_scale:NnnNN`

scaling is done the T_EX way as this works properly with floating point values without needing to use the fp module.

```

33387 \cs_new_protected:Npn \coffin_scale:Nnn #1#2#3
33388 { \__coffin_scale:NnnNN #1 {#2} {#3} \box_scale:Nnn \prop_set_eq:cN }
33389 \cs_generate_variant:Nn \coffin_scale:Nnn { c }
33390 \cs_new_protected:Npn \coffin_gscale:Nnn #1#2#3
33391 { \__coffin_scale:NnnNN #1 {#2} {#3} \box_gscale:Nnn \prop_gset_eq:cN }
33392 \cs_generate_variant:Nn \coffin_gscale:Nnn { c }
33393 \cs_new_protected:Npn \__coffin_scale:NnnNN #1#2#3#4#5
33394 {
33395   \fp_set:Nn \l__coffin_scale_x_fp {#2}
33396   \fp_set:Nn \l__coffin_scale_y_fp {#3}
33397   #4 #1 { \l__coffin_scale_x_fp } { \l__coffin_scale_y_fp }
33398   \dim_set:Nn \l__coffin_internal_dim
33399     { \coffin_ht:N #1 + \coffin_dp:N #1 }
33400   \dim_set:Nn \l__coffin_scaled_total_height_dim
33401     { \fp_abs:n { \l__coffin_scale_y_fp } \l__coffin_internal_dim }
33402   \dim_set:Nn \l__coffin_scaled_width_dim
33403     { -\fp_abs:n { \l__coffin_scale_x_fp } \coffin_wd:N #1 }
33404   \__coffin_resize_common:NnnN #1
33405     { \l__coffin_scaled_width_dim } { \l__coffin_scaled_total_height_dim }
33406   #5
33407 }

```

(End definition for `\coffin_scale:Nnn`, `\coffin_gscale:Nnn`, and `\coffin_scale:NnnNN`. These functions are documented on page 284.)

`__coffin_scale_vector:nnNN` This function scales a vector from the origin using the pre-set scale factors in x and y . This is a much less complex operation than rotation, and as a result the code is a lot clearer.

```

33408 \cs_new_protected:Npn \__coffin_scale_vector:nnNN #1#2#3#4
33409 {
33410   \dim_set:Nn #3
33411     { \fp_to_dim:n { \dim_to_fp:n {#1} * \l__coffin_scale_x_fp } }
33412   \dim_set:Nn #4
33413     { \fp_to_dim:n { \dim_to_fp:n {#2} * \l__coffin_scale_y_fp } }
33414 }

```

(End definition for `__coffin_scale_vector:nnNN`.)

`__coffin_scale_corner:Nnnn` `__coffin_scale_pole:Nnnnnn` Scaling both corners and poles is a simple calculation using the preceding vector scaling.

```

33415 \cs_new_protected:Npn \__coffin_scale_corner:Nnnn #1#2#3#4
33416 {
33417   \__coffin_scale_vector:nnNN {#3} {#4} \l__coffin_x_dim \l__coffin_y_dim
33418   \prop_put:Nnx \l__coffin_corners_prop {#2}
33419     { { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim } }
33420 }
33421 \cs_new_protected:Npn \__coffin_scale_pole:Nnnnnn #1#2#3#4#5#6
33422 {
33423   \__coffin_scale_vector:nnNN {#3} {#4} \l__coffin_x_dim \l__coffin_y_dim
33424   \prop_put:Nnx \l__coffin_poles_prop {#2}
33425   {
33426     { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim }
33427     {#5} {#6}

```



```

33428     }
33429 }

```

(End definition for `_coffin_scale_corner:Nnnn` and `_coffin_scale_pole:Nnnnnn`.)

```

\_coffin_x_shift_corner:Nnnn
\_coffin_x_shift_pole:Nnnnnn

```

These functions correct for the x displacement that takes place with a negative horizontal scaling.

```

33430 \cs_new_protected:Npn \_coffin_x_shift_corner:Nnnn #1#2#3#4
33431 {
33432   \prop_put:Nnx \l__coffin_corners_prop {#2}
33433   {
33434     { \dim_eval:n { #3 + \box_wd:N #1 } } {#4}
33435   }
33436 }
33437 \cs_new_protected:Npn \_coffin_x_shift_pole:Nnnnnn #1#2#3#4#5#6
33438 {
33439   \prop_put:Nnx \l__coffin_poles_prop {#2}
33440   {
33441     { \dim_eval:n { #3 + \box_wd:N #1 } } {#4}
33442     {#5} {#6}
33443   }
33444 }

```

(End definition for `_coffin_x_shift_corner:Nnnn` and `_coffin_x_shift_pole:Nnnnnn`.)

85.7 Aligning and typesetting of coffins

```

\coffin_join:NnnNnnnn
\coffin_join:cnnNnnnn
\coffin_join:Nnnncnnnn
\coffin_join:cnncnnnn

```

This command joins two coffins, using a horizontal and vertical pole from each coffin and making an offset between the two. The result is stored as the as a third coffin, which has all of its handles reset to standard values. First, the more basic alignment function is used to get things started.

```

\coffin_gjoin:NnnNnnnn
\coffin_gjoin:cnnNnnnn
\coffin_gjoin:Nnnncnnnn
\coffin_gjoin:cnncnnnn
\_coffin_join:NnnNnnnnN

```

```

33445 \cs_new_protected:Npn \coffin_join:NnnNnnnn #1#2#3#4#5#6#7#8
33446 {
33447   \_coffin_join:NnnNnnnnN #1 {#2} {#3} #4 {#5} {#6} {#7} {#8}
33448   \coffin_set_eq:NN
33449 }
33450 \cs_generate_variant:Nn \coffin_join:NnnNnnnn { c , Nnnc , cnnc }
33451 \cs_new_protected:Npn \coffin_gjoin:NnnNnnnn #1#2#3#4#5#6#7#8
33452 {
33453   \_coffin_join:NnnNnnnnN #1 {#2} {#3} #4 {#5} {#6} {#7} {#8}
33454   \coffin_gset_eq:NN
33455 }
33456 \cs_generate_variant:Nn \coffin_gjoin:NnnNnnnn { c , Nnnc , cnnc }
33457 \cs_new_protected:Npn \_coffin_join:NnnNnnnnN #1#2#3#4#5#6#7#8#9
33458 {
33459   \_coffin_align:NnnNnnnnN
33460   #1 {#2} {#3} #4 {#5} {#6} {#7} {#8} \l__coffin_aligned_coffin

```

Correct the placement of the reference point. If the x -offset is negative then the reference point of the second box is to the left of that of the first, which is corrected using a kern. On the right side the first box might stick out, which would show up if it is wider than the sum of the x -offset and the width of the second box. So a second kern may be needed.

```

33461   \hbox_set:Nn \l__coffin_aligned_coffin

```

```

33462     {
33463         \dim_compare:nNnT { \l__coffin_offset_x_dim } < \c_zero_dim
33464         { \__kernel_kern:n { -\l__coffin_offset_x_dim } }
33465         \hbox_unpack:N \l__coffin_aligned_coffin
33466         \dim_set:Nn \l__coffin_internal_dim
33467             { \l__coffin_offset_x_dim - \box_wd:N #1 + \box_wd:N #4 }
33468         \dim_compare:nNnT \l__coffin_internal_dim < \c_zero_dim
33469         { \__kernel_kern:n { -\l__coffin_internal_dim } }
33470     }

```

The coffin structure is reset, and the corners are cleared: only those from the two parent coffins are needed.

```

33471     \__coffin_reset_structure:N \l__coffin_aligned_coffin
33472     \prop_clear:c
33473     {
33474         coffin ~ \__coffin_to_value:N \l__coffin_aligned_coffin
33475         \c_space_tl corners
33476     }
33477     \__coffin_update_poles:N \l__coffin_aligned_coffin

```

The structures of the parent coffins are now transferred to the new coffin, which requires that the appropriate offsets are applied. That then depends on whether any shift was needed.

```

33478     \dim_compare:nNnTF \l__coffin_offset_x_dim < \c_zero_dim
33479     {
33480         \__coffin_offset_poles:Nnn #1 { -\l__coffin_offset_x_dim } { Opt }
33481         \__coffin_offset_poles:Nnn #4 { Opt } { \l__coffin_offset_y_dim }
33482         \__coffin_offset_corners:Nnn #1 { -\l__coffin_offset_x_dim } { Opt }
33483         \__coffin_offset_corners:Nnn #4 { Opt } { \l__coffin_offset_y_dim }
33484     }
33485     {
33486         \__coffin_offset_poles:Nnn #1 { Opt } { Opt }
33487         \__coffin_offset_poles:Nnn #4
33488             { \l__coffin_offset_x_dim } { \l__coffin_offset_y_dim }
33489         \__coffin_offset_corners:Nnn #1 { Opt } { Opt }
33490         \__coffin_offset_corners:Nnn #4
33491             { \l__coffin_offset_x_dim } { \l__coffin_offset_y_dim }
33492     }
33493     \__coffin_update_vertical_poles:NNN #1 #4 \l__coffin_aligned_coffin
33494     #9 #1 \l__coffin_aligned_coffin
33495 }

```

(End definition for \coffin_join:NnnNnnnn, \coffin_gjoin:NnnNnnnn, and __coffin_join:NnnNnnnnN. These functions are documented on page 285.)

```

\coffin_attach:NnnNnnnn
\coffin_attach:cnnNnnnn
\coffin_attach:Nnncnnnn
\coffin_attach:cnncnnnn
\coffin_gattach:NnnNnnnn
\coffin_gattach:cnnNnnnn
\coffin_gattach:Nnncnnnn
\coffin_gattach:cnncnnnn
\__coffin_attach:NnnNnnnnN
    \__coffin_attach_mark:NnnNnnnn

```

A more simple version of the above, as it simply uses the size of the first coffin for the new one. This means that the work here is rather simplified compared to the above code. The function used when marking a position is hear also as it is similar but without the structure updates.

```

33496 \cs_new_protected:Npn \coffin_attach:NnnNnnnn #1#2#3#4#5#6#7#8
33497 {
33498     \__coffin_attach:NnnNnnnnN #1 {#2} {#3} #4 {#5} {#6} {#7} {#8}
33499     \coffin_set_eq:NN
33500 }
33501 \cs_generate_variant:Nn \coffin_attach:NnnNnnnn { c , Nnnc , cnnc }

```

```

33502 \cs_new_protected:Npn \coffin_gattach:NnnNnnnn #1#2#3#4#5#6#7#8
33503 {
33504   \__coffin_attach:NnnNnnnnN #1 {#2} {#3} #4 {#5} {#6} {#7} {#8}
33505   \coffin_gset_eq:NN
33506 }
33507 \cs_generate_variant:Nn \coffin_gattach:NnnNnnnn { c , Nnnc , cnnc }
33508 \cs_new_protected:Npn \__coffin_attach:NnnNnnnnN #1#2#3#4#5#6#7#8#9
33509 {
33510   \__coffin_align:NnnNnnnnN
33511   #1 {#2} {#3} #4 {#5} {#6} {#7} {#8} \l__coffin_aligned_coffin
33512   \box_set_ht:Nn \l__coffin_aligned_coffin { \box_ht:N #1 }
33513   \box_set_dp:Nn \l__coffin_aligned_coffin { \box_dp:N #1 }
33514   \box_set_wd:Nn \l__coffin_aligned_coffin { \box_wd:N #1 }
33515   \__coffin_reset_structure:N \l__coffin_aligned_coffin
33516   \prop_set_eq:cc
33517   {
33518     coffin ~ \__coffin_to_value:N \l__coffin_aligned_coffin
33519     \c_space_tl corners
33520   }
33521   { coffin ~ \__coffin_to_value:N #1 ~ corners }
33522   \__coffin_update_poles:N \l__coffin_aligned_coffin
33523   \__coffin_offset_poles:Nnn #1 { Opt } { Opt }
33524   \__coffin_offset_poles:Nnn #4
33525   { \l__coffin_offset_x_dim } { \l__coffin_offset_y_dim }
33526   \__coffin_update_vertical_poles:NNN #1 #4 \l__coffin_aligned_coffin
33527   #9 #1 \l__coffin_aligned_coffin
33528 }
33529 \cs_new_protected:Npn \__coffin_attach_mark:NnnNnnnn #1#2#3#4#5#6#7#8
33530 {
33531   \__coffin_align:NnnNnnnnN
33532   #1 {#2} {#3} #4 {#5} {#6} {#7} {#8} \l__coffin_aligned_coffin
33533   \box_set_ht:Nn \l__coffin_aligned_coffin { \box_ht:N #1 }
33534   \box_set_dp:Nn \l__coffin_aligned_coffin { \box_dp:N #1 }
33535   \box_set_wd:Nn \l__coffin_aligned_coffin { \box_wd:N #1 }
33536   \box_set_eq:NN #1 \l__coffin_aligned_coffin
33537 }

```

(End definition for \coffin_attach:NnnNnnnn and others. These functions are documented on page 284.)

__coffin_align:NnnNnnnnN

The internal function aligns the two coffins into a third one, but performs no corrections on the resulting coffin poles. The process begins by finding the points of intersection for the poles for each of the input coffins. Those for the first coffin are worked out after those for the second coffin, as this allows the ‘primed’ storage area to be used for the second coffin. The ‘real’ box offsets are then calculated, before using these to re-box the input coffins. The default poles are then set up, but the final result depends on how the bounding box is being handled.

```

33538 \cs_new_protected:Npn \__coffin_align:NnnNnnnnN #1#2#3#4#5#6#7#8#9
33539 {
33540   \__coffin_calculate_intersection:Nnn #4 {#5} {#6}
33541   \dim_set:Nn \l__coffin_x_prime_dim { \l__coffin_x_dim }
33542   \dim_set:Nn \l__coffin_y_prime_dim { \l__coffin_y_dim }
33543   \__coffin_calculate_intersection:Nnn #1 {#2} {#3}
33544   \dim_set:Nn \l__coffin_offset_x_dim

```

```

33545     { \l__coffin_x_dim - \l__coffin_x_prime_dim + #7 }
33546 \dim_set:Nn \l__coffin_offset_y_dim
33547     { \l__coffin_y_dim - \l__coffin_y_prime_dim + #8 }
33548 \hbox_set:Nn \l__coffin_aligned_internal_coffin
33549     {
33550     \box_use:N #1
33551     \__kernel_kern:n { -\box_wd:N #1 }
33552     \__kernel_kern:n { \l__coffin_offset_x_dim }
33553     \box_move_up:nn { \l__coffin_offset_y_dim } { \box_use:N #4 }
33554     }
33555 \coffin_set_eq:NN #9 \l__coffin_aligned_internal_coffin
33556 }

```

(End definition for __coffin_align:NnnNnnnnN.)

__coffin_offset_poles:Nnn
 __coffin_offset_pole:Nnnnnnn

Transferring structures from one coffin to another requires that the positions are updated by the offset between the two coffins. This is done by mapping over the property list of the source coffins, moving as appropriate and saving to the new coffin data structures. The test for a - means that the structures from the parent coffins are uniquely labelled and do not depend on the order of alignment. The pay off for this is that - should not be used in coffin pole or handle names, and that multiple alignments do not result in a whole set of values.

```

33557 \cs_new_protected:Npn \__coffin_offset_poles:Nnn #1#2#3
33558 {
33559     \prop_map_inline:cn { coffin ~ \__coffin_to_value:N #1 ~ poles }
33560     { \__coffin_offset_pole:Nnnnnnn #1 {##1} ##2 {#2} {#3} }
33561 }
33562 \cs_new_protected:Npn \__coffin_offset_pole:Nnnnnnn #1#2#3#4#5#6#7#8
33563 {
33564     \dim_set:Nn \l__coffin_x_dim { #3 + #7 }
33565     \dim_set:Nn \l__coffin_y_dim { #4 + #8 }
33566     \tl_if_in:nnTF {#2} { - }
33567     { \tl_set:Nn \l__coffin_internal_tl { {#2} } }
33568     { \tl_set:Nn \l__coffin_internal_tl { { #1 - #2 } } }
33569     \exp_last_unbraced:NNo \__coffin_set_pole:Nnx \l__coffin_aligned_coffin
33570     { \l__coffin_internal_tl }
33571     {
33572     { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim }
33573     {#5} {#6}
33574     }
33575 }

```

(End definition for __coffin_offset_poles:Nnn and __coffin_offset_pole:Nnnnnnn.)

__coffin_offset_corners:Nnn
 __coffin_offset_corner:Nnnnn

Saving the offset corners of a coffin is very similar, except that there is no need to worry about naming: every corner can be saved here as order is unimportant.

```

33576 \cs_new_protected:Npn \__coffin_offset_corners:Nnn #1#2#3
33577 {
33578     \prop_map_inline:cn { coffin ~ \__coffin_to_value:N #1 ~ corners }
33579     { \__coffin_offset_corner:Nnnnn #1 {##1} ##2 {#2} {#3} }
33580 }
33581 \cs_new_protected:Npn \__coffin_offset_corner:Nnnnn #1#2#3#4#5#6
33582 {
33583     \prop_put:cnx

```

```

33584     {
33585         coffin ~ \_coffin_to_value:N \l__coffin_aligned_coffin
33586         \c_space_tl corners
33587     }
33588     { #1 - #2 }
33589     {
33590         { \dim_eval:n { #3 + #5 } }
33591         { \dim_eval:n { #4 + #6 } }
33592     }
33593 }

```

(End definition for _coffin_offset_corners:Nnn and _coffin_offset_corner:Nnnnn.)

```

\_coffin_update_vertical_poles:NNN
\_coffin_update_T:nnnnnnnnN
\_coffin_update_B:nnnnnnnnN

```

The T and B poles need to be recalculated after alignment. These functions find the larger absolute value for the poles, but this is of course only logical when the poles are horizontal.

```

33594 \cs_new_protected:Npn \_coffin_update_vertical_poles:NNN #1#2#3
33595 {
33596     \_coffin_get_pole:NnN #3 { #1 -T } \l__coffin_pole_a_tl
33597     \_coffin_get_pole:NnN #3 { #2 -T } \l__coffin_pole_b_tl
33598     \exp_last_two_unbraced:Noo \_coffin_update_T:nnnnnnnnN
33599     \l__coffin_pole_a_tl \l__coffin_pole_b_tl #3
33600     \_coffin_get_pole:NnN #3 { #1 -B } \l__coffin_pole_a_tl
33601     \_coffin_get_pole:NnN #3 { #2 -B } \l__coffin_pole_b_tl
33602     \exp_last_two_unbraced:Noo \_coffin_update_B:nnnnnnnnN
33603     \l__coffin_pole_a_tl \l__coffin_pole_b_tl #3
33604 }
33605 \cs_new_protected:Npn \_coffin_update_T:nnnnnnnnN #1#2#3#4#5#6#7#8#9
33606 {
33607     \dim_compare:nNnTF {#2} < {#6}
33608     {
33609         \_coffin_set_pole:Nnx #9 { T }
33610         { { Opt } {#6} { 1000pt } { Opt } }
33611     }
33612     {
33613         \_coffin_set_pole:Nnx #9 { T }
33614         { { Opt } {#2} { 1000pt } { Opt } }
33615     }
33616 }
33617 \cs_new_protected:Npn \_coffin_update_B:nnnnnnnnN #1#2#3#4#5#6#7#8#9
33618 {
33619     \dim_compare:nNnTF {#2} < {#6}
33620     {
33621         \_coffin_set_pole:Nnx #9 { B }
33622         { { Opt } {#2} { 1000pt } { Opt } }
33623     }
33624     {
33625         \_coffin_set_pole:Nnx #9 { B }
33626         { { Opt } {#6} { 1000pt } { Opt } }
33627     }
33628 }

```

(End definition for _coffin_update_vertical_poles:NNN, _coffin_update_T:nnnnnnnnN, and _coffin_update_B:nnnnnnnnN.)

`\c__coffin_empty_coffin` An empty-but-horizontal coffin.

```

33629 \coffin_new:N \c__coffin_empty_coffin
33630 \tex_setbox:D \c__coffin_empty_coffin = \tex_hbox:D { }

(End definition for \c__coffin_empty_coffin.)

```

`\coffin_typeset:Nnnnn` Typesetting a coffin means aligning it with the current position, which is done using a coffin with no content at all. As well as aligning to the empty coffin, there is also a need to leave vertical mode, if necessary.

`\coffin_typeset:cnnnn`

```

33631 \cs_new_protected:Npn \coffin_typeset:Nnnnn #1#2#3#4#5
33632 {
33633     \mode_leave_vertical:
33634     \__coffin_align:NnnNnnnnN \c__coffin_empty_coffin { H } { 1 }
33635     #1 {#2} {#3} {#4} {#5} \l__coffin_aligned_coffin
33636     \box_use_drop:N \l__coffin_aligned_coffin
33637 }
33638 \cs_generate_variant:Nn \coffin_typeset:Nnnnn { c }

(End definition for \coffin_typeset:Nnnnn. This function is documented on page 285.)

```

85.8 Coffin diagnostics

`\l__coffin_display_coffin` Used for printing coffins with data structures attached.

```

\l__coffin_display_coord_coffin
\l__coffin_display_pole_coffin

```

```

33639 \coffin_new:N \l__coffin_display_coffin
33640 \coffin_new:N \l__coffin_display_coord_coffin
33641 \coffin_new:N \l__coffin_display_pole_coffin

```

(End definition for `\l__coffin_display_coffin`, `\l__coffin_display_coord_coffin`, and `\l__coffin_display_pole_coffin`.)

`\l__coffin_display_handles_prop` This property list is used to print coffin handles at suitable positions. The offsets are expressed as multiples of the basic offset value, which therefore acts as a scale-factor.

```

33642 \prop_new:N \l__coffin_display_handles_prop
33643 \prop_put:Nnn \l__coffin_display_handles_prop { tl }
33644 { { b } { r } { -1 } { 1 } }
33645 \prop_put:Nnn \l__coffin_display_handles_prop { thc }
33646 { { b } { hc } { 0 } { 1 } }
33647 \prop_put:Nnn \l__coffin_display_handles_prop { tr }
33648 { { b } { l } { 1 } { 1 } }
33649 \prop_put:Nnn \l__coffin_display_handles_prop { vcl }
33650 { { vc } { r } { -1 } { 0 } }
33651 \prop_put:Nnn \l__coffin_display_handles_prop { vhc }
33652 { { vc } { hc } { 0 } { 0 } }
33653 \prop_put:Nnn \l__coffin_display_handles_prop { vcr }
33654 { { vc } { l } { 1 } { 0 } }
33655 \prop_put:Nnn \l__coffin_display_handles_prop { bl }
33656 { { t } { r } { -1 } { -1 } }
33657 \prop_put:Nnn \l__coffin_display_handles_prop { bhc }
33658 { { t } { hc } { 0 } { -1 } }
33659 \prop_put:Nnn \l__coffin_display_handles_prop { br }
33660 { { t } { l } { 1 } { -1 } }
33661 \prop_put:Nnn \l__coffin_display_handles_prop { Tl }
33662 { { t } { r } { -1 } { -1 } }

```

```

33663 \prop_put:Nnn \l__coffin_display_handles_prop { Thc }
33664   { { t } { hc } { 0 } { -1 } }
33665 \prop_put:Nnn \l__coffin_display_handles_prop { Tr }
33666   { { t } { l } { 1 } { -1 } }
33667 \prop_put:Nnn \l__coffin_display_handles_prop { Hl }
33668   { { vc } { r } { -1 } { 1 } }
33669 \prop_put:Nnn \l__coffin_display_handles_prop { Hhc }
33670   { { vc } { hc } { 0 } { 1 } }
33671 \prop_put:Nnn \l__coffin_display_handles_prop { Hr }
33672   { { vc } { l } { 1 } { 1 } }
33673 \prop_put:Nnn \l__coffin_display_handles_prop { Bl }
33674   { { b } { r } { -1 } { -1 } }
33675 \prop_put:Nnn \l__coffin_display_handles_prop { Bhc }
33676   { { b } { hc } { 0 } { -1 } }
33677 \prop_put:Nnn \l__coffin_display_handles_prop { Br }
33678   { { b } { l } { 1 } { -1 } }

```

(End definition for \l__coffin_display_handles_prop.)

\l__coffin_display_offset_dim The standard offset for the label from the handle position when displaying handles.

```

33679 \dim_new:N \l__coffin_display_offset_dim
33680 \dim_set:Nn \l__coffin_display_offset_dim { 2pt }

```

(End definition for \l__coffin_display_offset_dim.)

\l__coffin_display_x_dim \l__coffin_display_y_dim As the intersections of poles have to be calculated to find which ones to print, there is a need to avoid repetition. This is done by saving the intersection into two dedicated values.

```

33681 \dim_new:N \l__coffin_display_x_dim
33682 \dim_new:N \l__coffin_display_y_dim

```

(End definition for \l__coffin_display_x_dim and \l__coffin_display_y_dim.)

\l__coffin_display_poles_prop A property list for printing poles: various things need to be deleted from this to get a “nice” output.

```

33683 \prop_new:N \l__coffin_display_poles_prop

```

(End definition for \l__coffin_display_poles_prop.)

\l__coffin_display_font_tl Stores the settings used to print coffin data: this keeps things flexible.

```

33684 \tl_new:N \l__coffin_display_font_tl
33685 \bool_lazy_and:nnT
33686   { \cs_if_exist_p:N \fmtname }
33687   { \str_if_eq_p:Vn \fmtname { LaTeX2e } }
33688   {
33689     \tl_set:Nn \l__coffin_display_font_tl
33690       { \sffamily \tiny }
33691   }

```

(End definition for \l__coffin_display_font_tl.)

__coffin_rule:nn Abstract out creation of rules here until there is a higher-level interface.

```

33692 \cs_new_protected:Npn \__coffin_rule:nn #1#2
33693   {
33694     \mode_leave_vertical:
33695     \hbox:n { \tex_vrule:D width #1 height #2 \scan_stop: }
33696   }

```

(End definition for _coffin_rule:nn.)

\coffin_mark_handle:Nnnn Marking a single handle is relatively easy. The standard attachment function is used, meaning that there are two calculations for the location. However, this is likely to be okay given the load expected. Contrast with the more optimised version for showing all handles which comes next.

\coffin_mark_handle:cnnn
_coffin_mark_handle_aux:nnnnNnn

```

33697 \cs_new_protected:Npn \coffin_mark_handle:Nnnn #1#2#3#4
33698 {
33699   \hcoffin_set:Nn \l__coffin_display_pole_coffin
33700   {
33701     \color_select:n {#4}
33702     \_coffin_rule:nn { 1pt } { 1pt }
33703   }
33704   \_coffin_attach_mark:NnnNnnnn #1 {#2} {#3}
33705   \l__coffin_display_pole_coffin { hc } { vc } { Opt } { Opt }
33706   \hcoffin_set:Nn \l__coffin_display_coord_coffin
33707   {
33708     \color_select:n {#4}
33709     \l__coffin_display_font_tl
33710     ( \tl_to_str:n { #2 , #3 } )
33711   }
33712   \prop_get:NnN \l__coffin_display_handles_prop
33713   { #2 #3 } \l__coffin_internal_tl
33714   \quark_if_no_value:NTF \l__coffin_internal_tl
33715   {
33716     \prop_get:NnN \l__coffin_display_handles_prop
33717     { #3 #2 } \l__coffin_internal_tl
33718     \quark_if_no_value:NTF \l__coffin_internal_tl
33719     {
33720       \_coffin_attach_mark:NnnNnnnn #1 {#2} {#3}
33721       \l__coffin_display_coord_coffin { l } { vc }
33722       { 1pt } { Opt }
33723     }
33724     {
33725       \exp_last_unbraced:No \_coffin_mark_handle_aux:nnnnNnn
33726       \l__coffin_internal_tl #1 {#2} {#3}
33727     }
33728   }
33729   {
33730     \exp_last_unbraced:No \_coffin_mark_handle_aux:nnnnNnn
33731     \l__coffin_internal_tl #1 {#2} {#3}
33732   }
33733 }
33734 \cs_new_protected:Npn \_coffin_mark_handle_aux:nnnnNnn #1#2#3#4#5#6#7
33735 {
33736   \_coffin_attach_mark:NnnNnnnn #5 {#6} {#7}
33737   \l__coffin_display_coord_coffin {#1} {#2}
33738   { #3 \l__coffin_display_offset_dim }
33739   { #4 \l__coffin_display_offset_dim }
33740 }
33741 \cs_generate_variant:Nn \coffin_mark_handle:Nnnn { c }

```

(End definition for \coffin_mark_handle:Nnnn and _coffin_mark_handle_aux:nnnnNnn. This function is documented on page [286](#).)


```

\coffin_display_handles:Nn
\coffin_display_handles:cn
  \_coffin_display_handles_aux:nnnnnn
  \_coffin_display_handles_aux:nnnn
  \_coffin_display_attach:Nnnnn

```

Printing the poles starts by removing any duplicates, for which the H poles is used as the definitive version for the baseline and bottom. Two loops are then used to find the combinations of handles for all of these poles. This is done such that poles are removed during the loops to avoid duplication.

```

33742 \cs_new_protected:Npn \coffin_display_handles:Nn #1#2
33743 {
33744   \hcoffin_set:Nn \l__coffin_display_pole_coffin
33745   {
33746     \color_select:n {#2}
33747     \_coffin_rule:nn { 1pt } { 1pt }
33748   }
33749   \prop_set_eq:Nc \l__coffin_display_poles_prop
33750   { coffin ~ \_coffin_to_value:N #1 ~ poles }
33751   \_coffin_get_pole:NnN #1 { H } \l__coffin_pole_a_tl
33752   \_coffin_get_pole:NnN #1 { T } \l__coffin_pole_b_tl
33753   \tl_if_eq:NNT \l__coffin_pole_a_tl \l__coffin_pole_b_tl
33754   { \prop_remove:Nn \l__coffin_display_poles_prop { T } }
33755   \_coffin_get_pole:NnN #1 { B } \l__coffin_pole_b_tl
33756   \tl_if_eq:NNT \l__coffin_pole_a_tl \l__coffin_pole_b_tl
33757   { \prop_remove:Nn \l__coffin_display_poles_prop { B } }
33758   \coffin_set_eq:NN \l__coffin_display_coffin #1
33759   \prop_map_inline:Nn \l__coffin_display_poles_prop
33760   {
33761     \prop_remove:Nn \l__coffin_display_poles_prop {##1}
33762     \_coffin_display_handles_aux:nnnnnn {##1} ##2 {#2}
33763   }
33764   \box_use_drop:N \l__coffin_display_coffin
33765 }

```

For each pole there is a check for an intersection, which here does not give an error if none is found. The successful values are stored and used to align the pole coffin with the main coffin for output. The positions are recovered from the preset list if available.

```

33766 \cs_new_protected:Npn \_coffin_display_handles_aux:nnnnnn #1#2#3#4#5#6
33767 {
33768   \prop_map_inline:Nn \l__coffin_display_poles_prop
33769   {
33770     \bool_set_false:N \l__coffin_error_bool
33771     \_coffin_calculate_intersection:nnnnnnnn {#2} {#3} {#4} {#5} ##2
33772     \bool_if:NF \l__coffin_error_bool
33773     {
33774       \dim_set:Nn \l__coffin_display_x_dim { \l__coffin_x_dim }
33775       \dim_set:Nn \l__coffin_display_y_dim { \l__coffin_y_dim }
33776       \_coffin_display_attach:Nnnnn
33777       \l__coffin_display_pole_coffin { hc } { vc }
33778       { Opt } { Opt }
33779       \hcoffin_set:Nn \l__coffin_display_coord_coffin
33780       {
33781         \color_select:n {#6}
33782         \l__coffin_display_font_tl
33783         ( \tl_to_str:n { #1 , ##1 } )
33784       }
33785       \prop_get:NnN \l__coffin_display_handles_prop
33786       { #1 ##1 } \l__coffin_internal_tl
33787       \quark_if_no_value:NTF \l__coffin_internal_tl

```

```

33788         {
33789             \prop_get:NnN \l__coffin_display_handles_prop
33790             { ##1 #1 } \l__coffin_internal_tl
33791             \quark_if_no_value:NTF \l__coffin_internal_tl
33792             {
33793                 \__coffin_display_attach:Nnnnn
33794                 \l__coffin_display_coord_coffin { 1 } { vc }
33795                 { 1pt } { Opt }
33796             }
33797             {
33798                 \exp_last_unbraced:No
33799                 \__coffin_display_handles_aux:nnnn
33800                 \l__coffin_internal_tl
33801             }
33802         }
33803         {
33804             \exp_last_unbraced:No \__coffin_display_handles_aux:nnnn
33805             \l__coffin_internal_tl
33806         }
33807     }
33808 }
33809 }
33810 \cs_new_protected:Npn \__coffin_display_handles_aux:nnnn #1#2#3#4
33811 {
33812     \__coffin_display_attach:Nnnnn
33813     \l__coffin_display_coord_coffin {#1} {#2}
33814     { #3 \l__coffin_display_offset_dim }
33815     { #4 \l__coffin_display_offset_dim }
33816 }
33817 \cs_generate_variant:Nn \coffin_display_handles:Nn { c }

```

This is a dedicated version of `\coffin_attach:NnnNnnnn` with a hard-wired first coffin. As the intersection is already known and stored for the display coffin the code simply uses it directly, with no calculation.

```

33818 \cs_new_protected:Npn \__coffin_display_attach:Nnnnn #1#2#3#4#5
33819 {
33820     \__coffin_calculate_intersection:Nnn #1 {#2} {#3}
33821     \dim_set:Nn \l__coffin_x_prime_dim { \l__coffin_x_dim }
33822     \dim_set:Nn \l__coffin_y_prime_dim { \l__coffin_y_dim }
33823     \dim_set:Nn \l__coffin_offset_x_dim
33824         { \l__coffin_display_x_dim - \l__coffin_x_prime_dim + #4 }
33825     \dim_set:Nn \l__coffin_offset_y_dim
33826         { \l__coffin_display_y_dim - \l__coffin_y_prime_dim + #5 }
33827     \hbox_set:Nn \l__coffin_aligned_coffin
33828     {
33829         \box_use:N \l__coffin_display_coffin
33830         \__kernel_kern:n { -\box_wd:N \l__coffin_display_coffin }
33831         \__kernel_kern:n { \l__coffin_offset_x_dim }
33832         \box_move_up:nn { \l__coffin_offset_y_dim } { \box_use:N #1 }
33833     }
33834     \box_set_ht:Nn \l__coffin_aligned_coffin
33835         { \box_ht:N \l__coffin_display_coffin }
33836     \box_set_dp:Nn \l__coffin_aligned_coffin
33837         { \box_dp:N \l__coffin_display_coffin }

```

```

33838     \box_set_wd:Nn \l__coffin_aligned_coffin
33839     { \box_wd:N \l__coffin_display_coffin }
33840     \box_set_eq:NN \l__coffin_display_coffin \l__coffin_aligned_coffin
33841 }

```

(End definition for `\coffin_display_handles:Nn` and others. This function is documented on page 285.)

`\coffin_show_structure:N` For showing the various internal structures attached to a coffin in a way that keeps things relatively readable. If there is no apparent structure then the code complains.

```

\coffin_show_structure:c
\coffin_log_structure:N
\coffin_log_structure:c
\__coffin_show_structure:NN
33842 \cs_new_protected:Npn \coffin_show_structure:N
33843 { \__coffin_show_structure:NN \msg_show:nnxxxx }
33844 \cs_generate_variant:Nn \coffin_show_structure:N { c }
33845 \cs_new_protected:Npn \coffin_log_structure:N
33846 { \__coffin_show_structure:NN \msg_log:nnxxxx }
33847 \cs_generate_variant:Nn \coffin_log_structure:N { c }
33848 \cs_new_protected:Npn \__coffin_show_structure:NN #1#2
33849 {
33850   \__coffin_if_exist:NT #2
33851   {
33852     #1 { coffin } { show }
33853     { \token_to_str:N #2 }
33854     {
33855       \iow_newline: >~ ht ~~~ \dim_eval:n { \coffin_ht:N #2 }
33856       \iow_newline: >~ dp ~~~ \dim_eval:n { \coffin_dp:N #2 }
33857       \iow_newline: >~ wd ~~~ \dim_eval:n { \coffin_wd:N #2 }
33858     }
33859     {
33860       \prop_map_function:cN
33861       { coffin ~ \__coffin_to_value:N #2 ~ poles }
33862       \msg_show_item_unbraced:nn
33863     }
33864     { }
33865   }
33866 }

```

(End definition for `\coffin_show_structure:N`, `\coffin_log_structure:N`, and `__coffin_show_structure:NN`. These functions are documented on page 286.)

`\coffin_show:N` Essentially a combination of `\coffin_show_structure:N` and `\box_show:Nnn`, but we need to avoid having two prompts, so we use `\msg_term:nnxxxx` instead of `\msg_show:nnxxxx` in the show case.

```

\coffin_show:c
\coffin_log:N
\coffin_log:c
\coffin_show:Nnn
\coffin_show:cnn
\coffin_log:Nnn
\coffin_log:cnn
\__coffin_show:NNNnn
33867 \cs_new_protected:Npn \coffin_show:N #1
33868 { \coffin_show:Nnn #1 \c_max_int \c_max_int }
33869 \cs_generate_variant:Nn \coffin_show:N { c }
33870 \cs_new_protected:Npn \coffin_log:N #1
33871 { \coffin_log:Nnn #1 \c_max_int \c_max_int }
33872 \cs_generate_variant:Nn \coffin_log:N { c }
33873 \cs_new_protected:Npn \coffin_show:NNn
33874 { \__coffin_show:NNNnn \msg_term:nnxxxx \box_show:Nnn }
33875 \cs_generate_variant:Nn \coffin_show:NNn { c }
33876 \cs_new_protected:Npn \coffin_log:NNn
33877 { \__coffin_show:NNNnn \msg_log:nnxxxx \box_show:Nnn }
33878 \cs_generate_variant:Nn \coffin_log:NNn { c }
33879 \cs_new_protected:Npn \__coffin_show:NNNnn #1#2#3#4#5

```

```

33880 {
33881   \__coffin_if_exist:NT #3
33882   {
33883     \__coffin_show_structure:NN #1 #3
33884     #2 #3 {#4} {#5}
33885   }
33886 }

```

(End definition for `\coffin_show:N` and others. These functions are documented on page 286.)

85.9 Messages

```

33887 \msg_new:nnnn { coffin } { no-pole-intersection }
33888 { No~intersection~between~coffin~poles. }
33889 {
33890   LaTeX~was~asked~to~find~the~intersection~between~two~poles,~
33891   but~they~do~not~have~a~unique~meeting~point:~
33892   the~value~(Opt,~Opt)~will~be~used.
33893 }
33894 \msg_new:nnnn { coffin } { unknown }
33895 { Unknown~coffin~'#1'. }
33896 { The~coffin~'#1'~was~never~defined. }
33897 \msg_new:nnnn { coffin } { unknown-pole }
33898 { Pole~'#1'~unknown~for~coffin~'#2'. }
33899 {
33900   LaTeX~was~asked~to~find~a~typesetting~pole~for~a~coffin,~
33901   but~either~the~coffin~does~not~exist~or~the~pole~name~is~wrong.
33902 }
33903 \msg_new:nnn { coffin } { show }
33904 {
33905   Size~of~coffin~#1 : #2 \\
33906   Poles~of~coffin~#1 : #3 .
33907 }
33908 </package>

```

Chapter 86

l3color Implementation

```
33909 <*package>
```

```
33910 <@@=color>
```

86.1 Basics

`\l__color_current_tl` The color currently active for foreground (text, *etc.*) material. This is stored in the form of a color model followed by one or more values. There are four pre-defined models, three of which take numerical values in the range [0, 1]:

- `gray` `<gray>` Grayscale color with the `<gray>` value running from 0 (fully black) to 1 (fully white)
- `cmyk` `<cyan>` `<magenta>` `<yellow>` `<black>`
- `rgb` `<red>` `<green>` `<blue>`

Notice that the value are separated by spaces. There is a fourth pre-defined model using a string value and a numerical one:

- `spot` `<name>` `<tint>` A pre-defined spot color, where the `<name>` should be a pre-defined string color name and the `<tint>` should be in the range [0, 1].

Additional models may be created to allow mixing of spot colors. The number of data entries these require will depend on the number of colors to be mixed.

T_EXhackers note: The content of `\l__color_current_tl` comprises two brace groups, the first containing the color model and the second containing the value(s) applicable in that model.

(End definition for `\l__color_current_tl`.)

`\color_group_begin:` Grouping for color is the same as using the basic `\group_begin:` and `\group_end:` functions. However, for semantic reasons, they are renamed here.

```
33911 \cs_new_eq:NN \color_group_begin: \group_begin:
```

```
33912 \cs_new_eq:NN \color_group_end: \group_end:
```

(End definition for `\color_group_begin:` and `\color_group_end:`. These functions are documented on page 288.)

\color_ensure_current: A driver-independent wrapper for setting the foreground color to the current color “now”.

```

33913 \cs_new_protected:Npn \color_ensure_current:
33914 {
33915     \__color_backend_pickup:N \l__color_current_tl
33916     \__color_select:N \l__color_current_tl
33917 }

```

(End definition for \color_ensure_current:. This function is documented on page 288.)

\s__color_stop Internal scan marks.

```

33918 \scan_new:N \s__color_stop

```

(End definition for \s__color_stop.)

__color_select:N Take an internal color specification and pass it to the driver. This code is needed to ensure the current color but will also be used by the higher-level material.

```

\__color_select_math:N
\__color_select:nn
33919 \cs_new_protected:Npn \__color_select:N #1
33920 {
33921     \exp_after:wN \__color_select:nn #1
33922     \group_insert_after:N \__color_backend_reset:
33923 }
33924 \cs_new_protected:Npn \__color_select_math:N #1
33925 { \exp_after:wN \__color_select:nn #1 }
33926 \cs_new_protected:Npn \__color_select:nn #1#2
33927 { \use:c { __color_backend_select_ #1 :n } {#2} }

```

(End definition for __color_select:N, __color_select_math:N, and __color_select:nn.)

\l__color_current_tl The current color, with the model and

```

33928 \tl_new:N \l__color_current_tl
33929 \tl_set:Nn \l__color_current_tl { { gray } { 0 } }

```

(End definition for \l__color_current_tl.)

86.2 Predefined color names

The ability to predefine colors with a name is a key part of this module and means there has to be a method for storing the results. At first sight, it seems natural to follow the usual `expl3` model and create a `color` variable type for the process. That would then allow both local and global colors, constant colors and the like. However, these names need to be accessible in some form at the user level, for selection of colors either simply by name or as part of a more complex expression. This does not require that the full name is exposed but does require that they can be looked up in a predictable way. As such, it is more useful to expose just the color names as part of the interface, with the result that only local color names can be created. (This is also seen for example in key creation in `l3keys`.) As a result, color names are declarative (no `new` functions).

Since there is no need to manipulate colors *en masse*, each is stored in a two-part structure: a `prop` for the colors themselves, and a `tl` for the default model for each color.

86.3 Setup

```

\l__color_internal_int
\l__color_internal_tl
33930 \int_new:N \l__color_internal_int
33931 \tl_new:N \l__color_internal_tl

(End definition for \l__color_internal_int and \l__color_internal_tl.)

\s__color_mark Internal scan marks. \s__color_stop is already defined in l3color-base.
33932 \scan_new:N \s__color_mark

(End definition for \s__color_mark.)

```

86.4 Utility functions

```

\__color_if_defined:nTF A simple wrapper to avoid needing to have the lookup repeated in too many places. To
guard against a color created in a group, we need to test for entries in the prop.
33933 \prg_new_conditional:Npnn \__color_if_defined:n #1 { T, F, TF }
33934 {
33935   \prop_if_exist:cTF { l__color_named_ #1 _prop }
33936   {
33937     \prop_if_empty:cTF { l__color_named_ #1 _prop }
33938     \prg_return_false:
33939     \prg_return_true:
33940   }
33941   \prg_return_false:
33942 }

(End definition for \__color_if_defined:nTF.)

\__color_model:N Simple abstractions.
\__color_values:N
33943 \cs_new:Npn \__color_model:N #1 { \exp_after:wN \use_i:nn #1 }
33944 \cs_new:Npn \__color_values:N #1 { \exp_after:wN \use_ii:nn #1 }

(End definition for \__color_model:N and \__color_values:N.)

\__color_extract:nNN Recover the values for the standard model for a color.
\__color_extract:VNN
33945 \cs_new_protected:Npn \__color_extract:nNN #1#2#3
33946 {
33947   \tl_set_eq:Nc #2 { l__color_named_ #1 _tl }
33948   \prop_get:cVN { l__color_named_ #1 _prop } #2 #3
33949 }
33950 \cs_generate_variant:Nn \__color_extract:nNN { V }

(End definition for \__color_extract:nNN.)

```

86.5 Model conversion

Model conversion is carried out using standard formulae for base models, as described in the manual for xcolor (see also the *PostScript Language Reference Manual*). For other models direct conversion might not be defined, so we go through the fallback models if necessary.

```

__color_convert:nnN
__color_convert:VVN
__color_convert:nnnN
__color_convert:nVnN
__color_convert:nnVN
t_rgb_rgb:w_____
__color_convert_rgb_cmyk:w
__color_convert_rgb_cmyk:nnnn

33951 \cs_new_protected:Npn \__color_convert:nnN #1#2#3
33952 { \__color_convert:nnVN {#1} {#2} #3 #3 }
33953 \cs_generate_variant:Nn \__color_convert:nnN { VV }
33954 \cs_generate_variant:Nn \exp_last_unbraced:Nf { c }
33955 \cs_new_protected:Npn \__color_convert:nnnN #1#2#3#4
33956 {
33957   \tl_set:Nx #4
33958   {
33959     \cs_if_exist_use:cTF { __color_convert_ #1 _ #2 :w }
33960     { #3 \s__color_stop }
33961     {
33962       \cs_if_exist_use:cTF { __color_convert_ \use:c { c__color_fallback_ #1 _tl } _ #2 :
33963       {
33964         \exp_last_unbraced:cf
33965         { __color_convert_ \use:c { c__color_fallback_ #1 _tl } _ #2 :w }
33966         { \use:c { __color_convert_ #1 _ \use:c { c__color_fallback_ #1 _tl } :w
33967         \s__color_stop
33968       }
33969     }
33970     \exp_last_unbraced:cf
33971     { __color_convert_ \use:c { c__color_fallback_ #2 _tl } _ #2 :w }
33972     {
33973       \cs_if_exist_use:cTF { __color_convert_ #1 _ \use:c { c__color_fallback
33974       { #3 \s__color_stop }
33975       {
33976         \exp_last_unbraced:cf
33977         { __color_convert_ \use:c { c__color_fallback_ #1 _tl } _ \use:c
33978         { \use:c { __color_convert_ #1 _ \use:c { c__color_fallback_ #1 _
33979         \s__color_stop
33980       }
33981     }
33982     \s__color_stop
33983   }
33984 }
33985 }
33986 }
33987 \cs_generate_variant:Nn \__color_convert:nnnN { nV , nnV }
33988 \cs_new:Npn \__color_convert_gray_gray:w #1 \s__color_stop
33989 { #1 }
33990 \cs_new:Npn \__color_convert_gray_rgb:w #1 \s__color_stop
33991 { #1 ~ #1 ~ #1 }
33992 \cs_new:Npn \__color_convert_gray_cmyk:w #1 \s__color_stop
33993 { 0 ~ 0 ~ 0 ~ \fp_eval:n { 1 - #1 } }

```

These rather odd values are based on NTSC television: the set are used for the cmyk conversion.

```

33994 \cs_new:Npn \__color_convert_rgb_gray:w #1 ~ #2 ~ #3 \s__color_stop
33995 { \fp_eval:n { 0.3 * #1 + 0.59 * #2 + 0.11 * #3 } }

```



```

33996 \cs_new:Npn \__color_convert_rgb_rgb:w #1 \s__color_stop
33997 { #1 }

```

The conversion from `rgb` to `cmk` is the most complex: a two-step procedure which requires *black generation* and *undercolor removal* functions. The PostScript reference describes them as device-dependent, but following `xcolor` we assume they are linear. Moreover, as the likelihood of anyone using a non-unitary matrix here is tiny, we simplify and treat those two concepts as no-ops.

```

33998 \cs_new:Npn \__color_convert_rgb_cmyk:w #1 ~ #2 ~ #3 \s__color_stop
33999 {
34000   \exp_args:Nf \__color_convert_rgb_cmyk:nnnn
34001   { \fp_eval:n { min ( 1 - #1 , 1 - #2 , 1 - #3 ) } } {#1} {#2} {#3}
34002 }
34003 \cs_new:Npn \__color_convert_rgb_cmyk:nnnn #1#2#3#4
34004 {
34005   \fp_eval:n { min ( 1 , max ( 0 , 1 - #2 - #1 ) ) } \c_space_tl
34006   \fp_eval:n { min ( 1 , max ( 0 , 1 - #3 - #1 ) ) } \c_space_tl
34007   \fp_eval:n { min ( 1 , max ( 0 , 1 - #4 - #1 ) ) } \c_space_tl
34008   #1
34009 }
34010 \cs_new:Npn \__color_convert_cmyk_gray:w #1 ~ #2 ~ #3 ~ #4 \s__color_stop
34011 { \fp_eval:n { 1 - min ( 1 , 0.3 * #1 + 0.59 * #2 + 0.11 * #3 + #4 ) } }
34012 \cs_new:Npn \__color_convert_cmyk_rgb:w #1 ~ #2 ~ #3 ~ #4 \s__color_stop
34013 {
34014   \fp_eval:n { 1 - min ( 1 , #1 + #4 ) } \c_space_tl
34015   \fp_eval:n { 1 - min ( 1 , #2 + #4 ) } \c_space_tl
34016   \fp_eval:n { 1 - min ( 1 , #3 + #4 ) }
34017 }
34018 \cs_new:Npn \__color_convert_cmyk_cmyk:w #1 \s__color_stop
34019 { #1 }

```

(End definition for `__color_convert:nnN` and others.)

86.6 Color expressions

```

\l__color_model_tl
\l__color_value_tl
\l__color_next_model_tl
\l__color_next_value_tl

```

Working space to store the color data whilst doing calculations: keeping it on the stack is attractive but gets tricky (return is non-trivial).

```

34020 \tl_new:N \l__color_model_tl
34021 \tl_new:N \l__color_value_tl
34022 \tl_new:N \l__color_next_model_tl
34023 \tl_new:N \l__color_next_value_tl

```

(End definition for `\l__color_model_tl` and others.)

```

\__color_parse:nN
\__color_parse_aux:nN
\__color_parse_eq:Nn
\__color_parse_eq:nNn
\__color_parse:Nw
\__color_parse_loop_init:Nnn
\__color_parse_loop:w
\__color_parse_loop:nn
\__color_parse_gray:n
\__color_parse_std:n
\__color_parse_break:w
\__color_parse_end:
\__color_parse_mix:Nnnn
\__color_parse_mix:NVVn
\__color_parse_mix:nNnn
\__color_parse_mix_gray:nw
\__color_parse_mix_rgb:nw
\__color_parse_mix_cmyk:nw

```

The main function for parsing color expressions removes actives but otherwise expands, then starts working through the expression itself. At the end, we apply the payload.

```

34024 \cs_new_protected:Npx \__color_parse:nN #1#2
34025 {
34026   \exp_not:N \__color_backend_pickup:N \exp_not:N \l__color_current_tl
34027   \tl_set:Nx \exp_not:c { l__color_named_ . _tl }
34028   { \exp_not:N \__color_model:N \exp_not:N \l__color_current_tl }
34029   \prop_put:NVx \exp_not:c { l__color_named_ . _prop }
34030   \exp_not:c { l__color_named_ . _tl }

```

```

34031     { \exp_not:N \__color_values:N \exp_not:N \l__color_current_tl }
34032 \exp_not:N \exp_args:Ne \exp_not:N \__color_parse_aux:nN
34033     { \exp_not:N \tl_to_str:n {#1} } #2
34034 }

```

Before going to all of the effort of parsing an expression, these two precursor functions look for a pre-defined name, either on its own or with a trailing ! (which is the same thing).

```

34035 \cs_new_protected:Npn \__color_parse_aux:nN #1#2
34036 {
34037     \__color_if_defined:nTF {#1}
34038     { \__color_parse_set_eq:Nn #2 {#1} }
34039     { \__color_parse:Nw #2#1 ! \s__color_stop }
34040     \__color_check_model:N #2
34041 }
34042 \cs_new_protected:Npn \__color_parse_set_eq:Nn #1#2
34043 {
34044     \tl_if_empty:NTF \l_color_fixed_model_tl
34045     { \exp_args:Nv \__color_parse_set_eq:nNn { l__color_named_ #2 _tl } }
34046     { \exp_args:Nv \__color_parse_set_eq:nNn \l_color_fixed_model_tl }
34047     #1 {#2}
34048 }

```

Here, we have to allow for the case where there is a fixed model: that can't be swept up by generic conversion as we are dealing with a named color.

```

34049 \cs_new_protected:Npn \__color_parse_set_eq:nNn #1#2#3
34050 {
34051     \prop_get:cnTF
34052     { l__color_named_ #3 _prop } {#1}
34053     \l__color_value_tl
34054     { \tl_set:Nx #2 { {#1} { \l__color_value_tl } } }
34055     {
34056         \tl_set_eq:Nc \l__color_model_tl { l__color_named_ #3 _tl }
34057         \prop_get:cVN { l__color_named_ #3 _prop } \l__color_model_tl
34058         \l__color_value_tl
34059         \__color_convert:nnN
34060         \l__color_model_tl {#1} \l__color_value_tl
34061         \tl_set:Nx #2
34062         {
34063             {#1}
34064             { \l__color_value_tl }
34065         }
34066     }
34067 }
34068 \cs_new_protected:Npn \__color_parse:Nw #1#2 ! #3 \s__color_stop
34069 {
34070     \__color_if_defined:nTF {#2}
34071     {
34072         \tl_if_blank:nTF {#3}
34073         { \__color_parse_set_eq:Nn #1 {#2} }
34074         { \__color_parse_loop_init:Nnn #1 {#2} {#3} }
34075     }
34076     {
34077         \msg_error:nnn { color } { unknown-color } {#2}
34078         \tl_set:Nn \l__color_current_tl { { gray } { 0 } }

```

```

34079     }
34080 }

```

Once we establish that a full parse is needed, the next job is to get the detail of the first color. That will determine the model we use for the calculation: splitting here makes checking that a bit easier.

```

34081 \cs_new_protected:Npn \__color_parse_loop_init:Nnn #1#2#3
34082 {
34083   \group_begin:
34084     \__color_extract:nNN {#2} \l__color_model_tl \l__color_value_tl
34085     \__color_parse_loop:w #3 ! ! ! \s__color_stop
34086     \tl_set:Nx \l__color_internal_tl
34087       { { \l__color_model_tl } { \l__color_value_tl } }
34088     \exp_args:NNNV \group_end:
34089     \tl_set:Nn #1 \l__color_internal_tl
34090 }

```

This is the loop proper: there can be an open-ended set of colors to parse, separated by ! tokens. There are a few cases to look out for. At the end of the expression and with we find a mix of 100 then we simply skip the next color entirely (we can't stop the loop as there might be a further valid color to mix in). On the other hand, if we get a mix of 0 then drop everything so far and start again. There is also a trailing **white** to “read in” if the final explicit data is a mix. Those conditions are separate from actually looping, which is therefore sorted out by checking if we have further data to process: in contrast to **xcolor**, we don't allow !! so the test can be simplified.

```

34091 \cs_new_protected:Npn \__color_parse_loop:w #1 ! #2 ! #3 ! #4 ! #5 \s__color_stop
34092 {
34093   \bool_lazy_or:nnF
34094     { \tl_if_blank_p:n {#1} }
34095     { \int_compare_p:nNn {#1} = { 100 } }
34096     {
34097       \int_compare:nNnTF {#1} = { 0 }
34098       {
34099         \tl_if_blank:nTF {#2}
34100           { \__color_extract:nNN { white } }
34101           { \__color_extract:nNN {#2} }
34102           \l__color_model_tl \l__color_value_tl
34103       }
34104       {
34105         \use:x
34106         {
34107           \__color_parse_loop:nn {#1}
34108           { \tl_if_blank:nTF {#2} { white } {#2} }
34109         }
34110       }
34111     }
34112   \tl_if_blank:nF {#3}
34113   { \__color_parse_loop:w #3 ! #4 ! #5 \s__color_stop }
34114   \__color_parse_end:
34115 }

```

The “payload” of calculation in the loop first. If the model for the upcoming color is different from that of the existing (partial) color, convert the model. For **gray** the two are flipped round so that the outcome is something with “real” color. We are then in

a position to do the actual calculation itself. The two auxiliaries here give us a way to break the loop should an invalid name be found.

```

34116 \cs_new_protected:Npn \__color_parse_loop:nn #1#2
34117 {
34118   \__color_if_defined:nTF {#2}
34119   {
34120     \__color_extract:nNN {#2} \l__color_next_model_tl \l__color_next_value_tl
34121     \tl_if_eq:NNF \l__color_model_tl \l__color_next_model_tl
34122     {
34123       \str_if_eq:VnTF \l__color_model_tl { gray }
34124       { \__color_parse_gray:n {#2} }
34125       { \__color_parse_std:n {#2} }
34126     }
34127     \tl_set:Nx \l__color_value_tl
34128     {
34129       \__color_parse_mix:NVVn
34130       \l__color_model_tl \l__color_value_tl \l__color_next_value_tl {#1}
34131     }
34132   }
34133   {
34134     \msg_error:nnn { color } { unknown-color } {#2}
34135     \__color_extract:nNN { black } \l__color_model_tl \l__color_value_tl
34136     \__color_parse_break:w
34137   }
34138 }

```

The `gray` model needs special handling: the models need to be swapped: we do that using a dedicated function.

```

34139 \cs_new_protected:Npn \__color_parse_gray:n #1
34140 {
34141   \tl_set_eq:NN \l__color_model_tl \l__color_next_model_tl
34142   \tl_set:Nn \l__color_next_model_tl { gray }
34143   \exp_args:NnV \__color_convert:nnN { gray } \l__color_model_tl
34144   \l__color_value_tl
34145   \prop_get:cVN { l__color_named_ #1 _prop } \l__color_model_tl
34146   \l__color_next_value_tl
34147 }
34148 \cs_new_protected:Npn \__color_parse_std:n #1
34149 {
34150   \prop_get:cVNF { l__color_named_ #1 _prop }
34151   \l__color_model_tl
34152   \l__color_next_value_tl
34153   {
34154     \__color_convert:VVN
34155     \l__color_next_model_tl
34156     \l__color_model_tl
34157     \l__color_next_value_tl
34158   }
34159 }
34160 \cs_new_protected:Npn \__color_parse_break:w #1 \__color_parse_end: { }
34161 \cs_new_protected:Npn \__color_parse_end: { }

```

Do the vector arithmetic: mainly a question of shuffling input, along with one pre-calculation to keep down the use of division.

```

34162 \cs_new:Npn \__color_parse_mix:Nnnn #1#2#3#4
34163 {
34164   \exp_args:Nf \__color_parse_mix:nNnn
34165     { \fp_eval:n { #4 / 100 } }
34166     #1 {#2} {#3}
34167 }
34168 \cs_generate_variant:Nn \__color_parse_mix:Nnnn { NVV }
34169 \cs_new:Npn \__color_parse_mix:nNnn #1#2#3#4
34170 {
34171   \use:c { __color_parse_mix_ #2 :nw } {#1}
34172   #3 \s__color_mark #4 \s__color_stop
34173 }
34174 \cs_new:Npn \__color_parse_mix_gray:nw #1#2 \s__color_mark #3 \s__color_stop
34175 { \fp_eval:n { #2 * #1 + #3 * ( 1 - #1 ) } }
34176 \cs_new:Npn \__color_parse_mix_rgb:nw
34177   #1#2 ~ #3 ~ #4 \s__color_mark #5 ~ #6 ~ #7 \s__color_stop
34178 {
34179   \fp_eval:n { #2 * #1 + #5 * ( 1 - #1 ) } \c_space_tl
34180   \fp_eval:n { #3 * #1 + #6 * ( 1 - #1 ) } \c_space_tl
34181   \fp_eval:n { #4 * #1 + #7 * ( 1 - #1 ) }
34182 }
34183 \cs_new:Npn \__color_parse_mix_cmyk:nw
34184   #1#2 ~ #3 ~ #4 ~ #5 \s__color_mark #6 ~ #7 ~ #8 ~ #9 \s__color_stop
34185 {
34186   \fp_eval:n { #2 * #1 + #6 * ( 1 - #1 ) } \c_space_tl
34187   \fp_eval:n { #3 * #1 + #7 * ( 1 - #1 ) } \c_space_tl
34188   \fp_eval:n { #4 * #1 + #8 * ( 1 - #1 ) } \c_space_tl
34189   \fp_eval:n { #5 * #1 + #9 * ( 1 - #1 ) }
34190 }

```

(End definition for __color_parse:nN and others.)

```

\__color_parse_model_gray:w Turn the input into internal form, also tidying up the number quickly.
\__color_parse_model_rgb:w
\__color_parse_model_cmyk:w
\__color_parse_number:n
\__color_parse_number:w
34191 \cs_new:Npn \__color_parse_model_gray:w #1 , #2 \s__color_stop
34192 { { gray } { \__color_parse_number:n {#1} } }
34193 \cs_new:Npn \__color_parse_model_rgb:w #1 , #2 , #3 , #4 \s__color_stop
34194 {
34195   { rgb }
34196   {
34197     \__color_parse_number:n {#1} ~
34198     \__color_parse_number:n {#2} ~
34199     \__color_parse_number:n {#3}
34200   }
34201 }
34202 \cs_new:Npn \__color_parse_model_cmyk:w #1 , #2 , #3 , #4 , #5 \s__color_stop
34203 {
34204   { cmyk }
34205   {
34206     \__color_parse_number:n {#1} ~
34207     \__color_parse_number:n {#2} ~
34208     \__color_parse_number:n {#3} ~
34209     \__color_parse_number:n {#4}
34210   }
34211 }

```

```

34212 \cs_new:Npn \__color_parse_number:n #1
34213 { \__color_parse_number:w #1 . 0 . \s__color_stop }
34214 \cs_new:Npn \__color_parse_number:w #1 . #2 . #3 \s__color_stop
34215 { \tl_if_blank:nTF {#1} { 0 } {#1} . #2 }

```

(End definition for __color_parse_model_gray:w and others.)

```

\__color_parse_model_Gray:w
\__color_parse_model_hsb:w
\__color_parse_model_Hsb:w
\__color_parse_model_HSB:w
\__color_parse_model_HTML:w
\__color_parse_model_RGB:w
\__color_parse_model_hsb:nnn
  \__color_parse_model_hsb_aux:nnn
  \__color_parse_model_hsb:nnnn
  \__color_parse_model_hsb:nnnnn
  \__color_parse_model_hsb_0:nnnn
  \__color_parse_model_hsb_1:nnnn
  \__color_parse_model_hsb_2:nnnn
  \__color_parse_model_hsb_3:nnnn
  \__color_parse_model_hsb_4:nnnn
  \__color_parse_model_hsb_5:nnnn
\__color_parse_model_wave:w
  \__color_parse_model_wave_auxi:nn
  \__color_parse_model_wave_auxii:nn
  \__color_parse_model_wave_rho:n
34216 \cs_new:Npn \__color_parse_model_Gray:w #1 , #2 \s__color_stop
34217 { { gray } { \fp_eval:n { #1 / 15 } } }
34218 \cs_new:Npn \__color_parse_model_hsb:w #1 , #2 , #3 , #4 \s__color_stop
34219 { \__color_parse_model_hsb:nnn {#1} {#2} {#3} }
34220 \cs_new:Npn \__color_parse_model_Hsb:w #1 , #2 , #3 , #4 \s__color_stop
34221 {
34222   \exp_args:Ne \__color_parse_model_hsb:nnn { \fp_eval:n { #1 / 360 } }
34223   {#2} {#3}
34224 }
34225 \cs_new:Npn \__color_parse_model_hsb:nnn #1#2#3
34226 {
34227   { rgb }
34228   {
34229     \exp_args:Ne \__color_parse_model_hsb_aux:nnn
34230     { \fp_eval:n { 6 * #1 } } {#2} {#3}
34231   }
34232 }
34233 \cs_new:Npn \__color_parse_model_hsb_aux:nnn #1#2#3
34234 {
34235   \exp_args:Nee \__color_parse_model_hsb_aux:nnnn
34236   { \fp_eval:n { floor(#1) } } { \fp_eval:n { #1 - floor(#1) } }
34237   {#2} {#3}
34238 }
34239 \cs_new:Npn \__color_parse_model_hsb_aux:nnnn #1#2#3#4
34240 {
34241   \use:e
34242   {
34243     \exp_not:N \__color_parse_model_hsb_aux:nnnnn
34244     { \__color_parse_number:n {#4} }
34245     { \fp_eval:n { round(#4 * (1 - #3),5) } }
34246     { \fp_eval:n { round(#4 * (1 - #3 * #2),5) } }
34247     { \fp_eval:n { round(#4 * (1 - #3 * (1 - #2)),5) } }
34248     {#1}
34249   }
34250 }
34251 \cs_new:Npn \__color_parse_model_hsb_aux:nnnnn #1#2#3#4#5
34252 { \use:c { __color_parse_model_hsb_#5 :nnnn } {#1} {#2} {#3} {#4} }
34253 \cs_new:cpn { __color_parse_model_hsb_0:nnnn } #1#2#3#4 { #1 ~ #4 ~ #2 }
34254 \cs_new:cpn { __color_parse_model_hsb_1:nnnn } #1#2#3#4 { #3 ~ #1 ~ #2 }
34255 \cs_new:cpn { __color_parse_model_hsb_2:nnnn } #1#2#3#4 { #2 ~ #1 ~ #4 }
34256 \cs_new:cpn { __color_parse_model_hsb_3:nnnn } #1#2#3#4 { #2 ~ #3 ~ #1 }
34257 \cs_new:cpn { __color_parse_model_hsb_4:nnnn } #1#2#3#4 { #4 ~ #2 ~ #1 }
34258 \cs_new:cpn { __color_parse_model_hsb_5:nnnn } #1#2#3#4 { #1 ~ #2 ~ #3 }

```

The conversion here is non-trivial but is described at length in the xcolor manual. For ease, we calculate the integer and fractional parts of the hue first, then use them to work out the possible values for r , g and b before putting them in the correct places.

```

34259 \cs_new:cpn { __color_parse_model_hsb_6:nnnn } #1#2#3#4 { #1 ~ #2 ~ #2 }
34260 \cs_new:Npn \__color_parse_model_HSB:w #1 , #2 , #3 , #4 \s__color_stop
34261 {
34262   \exp_args:Neee \__color_parse_model_hsb:nnn
34263   { \fp_eval:n {#1 / 240} }
34264   { \fp_eval:n {#2 / 240} }
34265   { \fp_eval:n {#3 / 240} }
34266 }
34267 \cs_new:Npn \__color_parse_model_HTML:w #1 , #2 \s__color_stop
34268 { \__color_parse_model_HTML_aux:w #1 0 0 0 0 0 \s__color_stop }
34269 \cs_new:Npn \__color_parse_model_HTML_aux:w #1#2#3#4#5#6#7 \s__color_stop
34270 {
34271   { rgb }
34272   {
34273     \fp_eval:n { round(\int_from_hex:n {#1#2} / 255,5) } ~
34274     \fp_eval:n { round(\int_from_hex:n {#3#4} / 255,5) } ~
34275     \fp_eval:n { round(\int_from_hex:n {#5#6} / 255,5) }
34276   }
34277 }
34278 \cs_new:Npn \__color_parse_model_RGB:w #1 , #2 , #3 , #4 \s__color_stop
34279 {
34280   { rgb }
34281   {
34282     \fp_eval:n { round(#1 / 255,5) } ~
34283     \fp_eval:n { round(#2 / 255,5) } ~
34284     \fp_eval:n { round(#3 / 255,5) }
34285   }
34286 }

```

Following the description in the xcolor manual. As we always use rgb, there is no need to find the sixth, we just pass the information straight to the hsb auxiliary defined earlier.

```

34287 \cs_new:Npn \__color_parse_model_wave:w #1 , #2 \s__color_stop
34288 {
34289   { rgb }
34290   {
34291     \fp_compare:nNnTF {#1} < { 420 }
34292     { \__color_parse_model_wave_auxi:nn {#1} { 0.3 + 0.7 * (#1 - 380) / 40 }
34293     }
34294     {
34295       \fp_compare:nNnTF {#1} > { 700 }
34296       { \__color_parse_model_wave_auxi:nn {#1} { 0.3 + 0.7 * (#1 - 780) / -80 } }
34297       { \__color_parse_model_wave_auxi:nn {#1} { 1 } }
34298     }
34299   }
34300 }
34301 \cs_new:Npn \__color_parse_model_wave_auxi:nn #1#2
34302 {
34303   \fp_compare:nNnTF {#1} < { 440 }
34304   {
34305     \__color_parse_model_wave_auxii:nn
34306     { 4 + \__color_parse_model_wave_rho:n { (#1 - 440) / -60 } }
34307     {#2}
34308   }
34309   {

```

```

34310 \fp_compare:nNnTF {#1} < { 490 }
34311 {
34312   \__color_parse_model_wave_auxii:nn
34313   { 4 - \__color_parse_model_wave_rho:n { (#1 - 440) / 50 } }
34314   {#2}
34315 }
34316 {
34317   \fp_compare:nNnTF {#1} < { 510 }
34318   {
34319     \__color_parse_model_wave_auxii:nn
34320     { 2 + \__color_parse_model_wave_rho:n { (#1 - 510) / -20 } }
34321     {#2}
34322   }
34323   {
34324     \fp_compare:nNnTF {#1} < { 580 }
34325     {
34326       \__color_parse_model_wave_auxii:nn
34327       { 2 - \__color_parse_model_wave_rho:n { (#1 - 510) / 70 } }
34328       {#2}
34329     }
34330     {
34331       \fp_compare:nNnTF {#1} < { 645 }
34332       {
34333         \__color_parse_model_wave_auxii:nn
34334         { \__color_parse_model_wave_rho:n { (#1 - 645) / -65 } }
34335         {#2}
34336       }
34337       { \__color_parse_model_wave_auxii:nn { 0 } {#2} }
34338     }
34339   }
34340 }
34341 }
34342 }
34343 \cs_new:Npn \__color_parse_model_wave_auxii:nn #1#2
34344 {
34345   \exp_args:Neee \__color_parse_model_hsb_aux:nnn
34346   { \fp_eval:n {#1} }
34347   { 1 }
34348   { \__color_parse_model_wave_rho:n {#2} }
34349 }
34350 \cs_new:Npn \__color_parse_model_wave_rho:n #1
34351 { \fp_eval:n { min(1, max(0,#1) ) } }

```

(End definition for `__color_parse_model_Gray:w` and others.)

86.7 Selecting colors (and color models)

`\l_color_fixed_model_tl` For selecting a single fixed model.

```
34352 \tl_new:N \l_color_fixed_model_tl
```

(End definition for `\l_color_fixed_model_tl`. This variable is documented on page 291.)

`__color_check_model:N` Check that the model in use is the one required.

`__color_check_model:nn`


```

34353 \cs_new_protected:Npn \__color_check_model:N #1
34354 {
34355   \tl_if_empty:NF \l_color_fixed_model_tl
34356   {
34357     \exp_after:wN \__color_check_model:nn #1
34358     \tl_if_eq:NNF \l__color_model_tl \l_color_fixed_model_tl
34359     {
34360       \__color_convert:VVN \l__color_model_tl \l_color_fixed_model_tl
34361       \l__color_value_tl
34362     }
34363     \tl_set:Nx #1
34364     { { \l_color_fixed_model_tl } { \l__color_value_tl } }
34365   }
34366 }
34367 \cs_new_protected:Npn \__color_check_model:nn #1#2
34368 {
34369   \tl_set:Nn \l__color_model_tl {#1}
34370   \tl_set:Nn \l__color_value_tl {#2}
34371 }

```

(End definition for __color_check_model:N and __color_check_model:nn.)

__color_finalise_current: A backend-neutral location for “last minute” manipulations before handing off to the backend code. We set the special . syntax here: this will therefore always be available. The finalisation is separate from the main function so it can also be applied to *e.g.* page color.

```

34372 \cs_new_protected:Npx \__color_finalise_current:
34373 {
34374   \tl_set:Nx \exp_not:c { l__color_named_ . _tl }
34375   { \exp_not:N \__color_model:N \exp_not:N \l__color_current_tl }
34376   \prop_clear:N \exp_not:c { l__color_named_ . _prop }
34377   \prop_put:Nv \exp_not:c { l__color_named_ . _prop }
34378   \exp_not:c { l__color_named_ . _tl }
34379   { \exp_not:N \__color_values:N \exp_not:N \l__color_current_tl }
34380 }

```

(End definition for __color_finalise_current:.)

\color_select:n Parse the input expressions then get the backend to actually activate them. The main complexity here is the need to check through multiple models. That is done “locally” here as the approach is subtly different to when different models are being stored.

```

\color_select:nn
\__color_select_main:Nw
\__color_select_loop:Nw
\__color_select:nnN
\__color_select_swap:Nnn
34381 \cs_new_protected:Npn \color_select:n #1
34382 {
34383   \__color_parse:nN {#1} \l__color_current_tl
34384   \__color_finalise_current:
34385   \__color_select:N \l__color_current_tl
34386 }
34387 \cs_new_protected:Npn \color_select:nn #1#2
34388 {
34389   \__color_select_main:Nw \l__color_current_tl
34390   #1 // \s__color_mark #2 // \s__color_stop
34391   \__color_finalise_current:
34392   \__color_select:N \l__color_current_tl
34393 }

```

If the first color model is the fixed one, or if there is no fixed model, we don't need most of the data: just set up and apply the backend function.

```

34394 \cs_new_protected:Npn \__color_select_main:Nw
34395   #1 #2 / #3 / #4 \s__color_mark #5 / #6 / #7 \s__color_stop
34396   {
34397     \__color_select:nnN {#2} {#5} #1
34398     \bool_lazy_or:nnF
34399       { \tl_if_empty_p:N \l_color_fixed_model_tl }
34400       { \str_if_eq_p:nV {#2} \l_color_fixed_model_tl }
34401       { \__color_select_loop:Nw #1 #3 / #4 \s__color_mark #6 / #7 \s__color_stop }
34402   }

```

If a fixed model applies, we need to check each possible value in order. If there is no hit at all, fall back on the generic formula-based interchange.

```

34403 \cs_new_protected:Npn \__color_select_loop:Nw
34404   #1 #2 / #3 \s__color_mark #4 / #5 \s__color_stop
34405   {
34406     \str_if_eq:nVTF {#2} \l_color_fixed_model_tl
34407     { \__color_select:nnN {#2} {#4} #1 }
34408     {
34409       \tl_if_blank:nTF {#2}
34410       { \exp_after:wN \__color_select_swap:Nnn \exp_after:wN #1 #1 }
34411       { \__color_select_loop:Nw #1 #3 \s__color_mark #5 \s__color_stop }
34412     }
34413   }
34414 \cs_new_protected:Npn \__color_select:nnN #1#2#3
34415   {
34416     \cs_if_exist:cTF { __color_parse_model_ #1 :w }
34417     {
34418       \tl_set:Nx #3
34419       { \use:c { __color_parse_model_ #1 :w } #2 , 0 , 0 , 0 , 0 \s__color_stop }
34420     }
34421     { \msg_error:nnn { color } { unknown-model } {#1} }
34422   }
34423 \cs_new_protected:Npn \__color_select_swap:Nnn #1#2#3
34424   {
34425     \__color_convert:nVnN {#2} \l_color_fixed_model_tl {#3} \l__color_value_tl
34426     \tl_set:Nx #1
34427     { { \l_color_fixed_model_tl } { \l__color_value_tl } }
34428   }

```

(End definition for `\color_select:n` and others. These functions are documented on page 291.)

86.8 Math color

The approach here is the same as for the $\text{\LaTeX 2}_{\epsilon}$ `\mathcolor` command, but as we are working at the `expl3` level we can make some minor changes.

`\l_color_math_active_tl` Tokens representing active sub/superscripts.

```

34429 \tl_new:N \l_color_math_active_tl
34430 \tl_set:Nn \l_color_math_active_tl { ' }

```

(End definition for `\l_color_math_active_tl`. This function is documented on page 292.)

`\g__color_math_seq` Not all engines have multiple color stacks, and at the same time we are not expecting breaking within a colored math fragment. So we track the color stack ourselves.

```
34431 \seq_new:N \g__color_math_seq
```

(End definition for `\g__color_math_seq`.)

`\color_math:nn` The basic set up here is relatively simple: store the current color, parse the new color
`\color_math:nnn` as-normal, then switch color before inserting the tokens we are asked to change. The
`__color_math:nn` tricky part is right at the end, handling the reset.

```
34432 \cs_new_protected:Npn \color_math:nn #1#2
34433 {
34434   \__color_math:nn {#2}
34435   { \__color_parse:nN {#1} \l__color_current_tl }
34436 }
34437 \cs_new_protected:Npn \color_math:nnn #1#2#3
34438 {
34439   \__color_math:nn {#3}
34440   {
34441     \__color_select_main:Nw \l__color_current_tl
34442     #1 / / \s__color_mark #2 / / \s__color_stop
34443   }
34444 }
34445 \cs_new_protected:Npn \__color_math:nn #1#2
34446 {
34447   \seq_gpush:NV \g__color_math_seq \l__color_current_tl
34448   #2
34449   \__color_select_math:N \l__color_current_tl
34450   #1
34451   \__color_math_scan:w
34452 }
```

(End definition for `\color_math:nn`, `\color_math:nnn`, and `__color_math:nn`. These functions are documented on page 292.)

`__color_math_scan:w` The complication when changing the color back is due to the fact that the `\color_`
`__color_math_scan_auxi:` `math:nn(n)` may be followed by `^` or `_` or the hidden superscript (for example `'`) and its
`__color_math_scan_auxii:` argument may end in a `\mathop` in which case the sub- and superscripts may be attached
`__color_math_scan_end:` as `\limits` instead of after the material. All cases need separate treatment. To avoid repeatedly collecting the same token, we first check for an alignment tab: assuming we don't have one of those, we can "recycle" `\l_peek_token` safely.

```
34453 \cs_new_protected:Npn \__color_math_scan:w
34454 {
34455   \peek_remove_filler:n
34456   {
34457     \peek_catcode:NTF \c_alignment_token
34458     { \__color_math_scan_end: }
34459     { \__color_math_scan_auxi: }
34460   }
34461 }
```

Dealing with literal `_` and `^` is easy, and as we have exactly two cases, we can hard-code this. We use a hard-coded list for limits: these are all primitives. The `\use_none:n` herealso removes the test token so it is left just in the right place.

```
34462 \cs_new_protected:Npn \__color_math_scan_auxi:
```

```

34463 {
34464   \token_case_meaning:NnTF \l_peek_token
34465   {
34466     \c_math_subscript_token { }
34467     \c_math_superscript_token { }
34468   }
34469   { \_color_math_scripts:Nw }
34470   {
34471     \token_case_meaning:NnTF \l_peek_token
34472     {
34473       \tex_limits:D { \tex_limits:D }
34474       \tex_nolimits:D { \tex_nolimits:D }
34475       \tex_displaylimits:D { \tex_displaylimits:D }
34476     }
34477     { \_color_math_scan:w \use_none:n }
34478     { \_color_math_scan_auxii: }
34479   }
34480 }

```

The one final case to handle is math-active tokens, most obviously ', as these won't be covered earlier.

```

34481 \cs_new_protected:Npn \_color_math_scan_auxii:
34482 {
34483   \tl_map_inline:Nn \l_color_math_active_tl
34484   {
34485     \token_if_eq_meaning:NNT \l_peek_token ##1
34486     {
34487       \tl_map_break:n
34488       {
34489         \use_i:nn
34490         { \_color_math_scan_auxiii:N ##1 }
34491       }
34492     }
34493     \_color_math_scan_end:
34494   }
34495 }
34496 \cs_new_protected:Npn \_color_math_scan_auxiii:N #1
34497 {
34498   \exp_after:wN \exp_after:wN \exp_after:wN \_color_math_scan:w
34499   \char_generate:nn { '#1 } { 13 }
34500 }
34501 \cs_new_protected:Npn \_color_math_scan_end:
34502 {
34503   \_color_backend_reset:
34504   \seq_gpop:NN \g_color_math_seq \l_color_current_tl
34505 }

```

(End definition for _color_math_scan:w and others.)

```

\_color_math_scripts:Nw
\_color_math_script_aux:N

```

The tricky part of handling sub and superscripts is that we have to reset color to the one that is on the stack but reset it back to what it was before to allow for cases like

```

\[ \color{math:n}{red} { a + \sum } _ { i = 1 } ^ { n } \]

```

Here, \TeX constructs a `\vbox` stacking subscript, summation sign, and superscript. So technically the superscript comes first and the `\sum` that should get colored red is the middle.

The approach here is to set up a brace group immediately after the script token, then to set the color appropriately in that argument. We need an extra group to keep the color contained, and as we need to allow for an explicit closing brace in the source, the inner group also is a brace one rather than `\group_begin:-`-based. At the end of the outer group we need to insert `__color_math_scan:w` to continue the search for a second script token.

Notice that here we *don't* need to use the math-specific color selector as we can allow the `\group_insert_after:N \@@_backend_reset:` to operate normally.

```

34506 \cs_new_protected:Npn \__color_math_scripts:Nw #1
34507 {
34508     #1
34509     \c_group_begin_token
34510     \c_group_begin_token
34511     \seq_get:NN \g__color_math_seq \l__color_current_tl
34512     \__color_select:N \l__color_current_tl
34513     \group_insert_after:N \c_group_end_token
34514     \group_insert_after:N \__color_math_scan:w
34515     \peek_remove_filler:n
34516     {
34517         \peek_catcode_remove:NF \c_group_begin_token
34518         { \__color_math_script_aux:N }
34519     }
34520 }
```

Deal with the case where we do not have an explicit brace pair in the source.

```

34521 \cs_new_protected:Npn \__color_math_script_aux:N #1 { #1 \c_group_end_token }
```

(End definition for `__color_math_scripts:Nw` and `__color_math_script_aux:N`.)

86.9 Fill and stroke color

```

\color_fill:n
\color_stroke:n
\color_fill:nn
\color_stroke:nn
__color_draw:nnn
34522 \cs_new_protected:Npn \color_fill:n #1
34523 {
34524     \__color_parse:nN {#1} \l__color_current_tl
34525     \exp_after:wN \__color_draw:nnn \l__color_current_tl { fill }
34526 }
34527 \cs_new_protected:Npn \color_stroke:n #1
34528 {
34529     \__color_parse:nN {#1} \l__color_current_tl
34530     \exp_after:wN \__color_draw:nnn \l__color_current_tl { stroke }
34531 }
34532 \cs_new_protected:Npn \color_fill:nn #1#2
34533 {
34534     \__color_select_main:Nw \l__color_current_tl
34535     #1 // \s__color_mark #2 // \s__color_stop
34536     \exp_after:wN \__color_draw:nnn \l__color_current_tl { fill }
34537 }
34538 \cs_new_protected:Npn \color_stroke:nn #1#2
```

```

34539 {
34540   \_color_select_main:Nw \l__color_current_tl
34541   #1 / / \s__color_mark #2 / / \s__color_stop
34542   \exp_after:wN \_color_draw:nnn \l__color_current_tl { stroke }
34543 }
34544 \cs_new_protected:Npn \_color_draw:nnn #1#2#3
34545 { \use:c { __color_backend_ #3 _ #1 :n } {#2} }

```

(End definition for `\color_fill:n` and others. These functions are documented on page 291.)

86.10 Defining named colors

`\l__color_named_tl` Space to store the detail of the named color.

```

34546 \tl_new:N \l__color_named_tl

```

(End definition for `\l__color_named_tl`.)

```

\color_set:nn Defining named colors means working through the model list and saving both the “main”
\__color_set:nnn color and any equivalents in other models. Even if there is only one model, we store a
\__color_set:nn prop as well as a tl, as there could be grouping weirdness, etc. When setting using an
\__color_set:nnw expression, we need to avoid any fixed model issues, which is done without a group as in
\color_set:nnn l3keys.
\__color_set_aux:nnn
\__color_set_colon:nnw
\__color_set_loop:nw
\color_set_eq:nn
34547 \cs_new_protected:Npn \color_set:nn #1#2
34548 {
34549   \exp_args:NV \__color_set:nnn
34550   \l_color_fixed_model_tl {#1} {#2}
34551 }
34552 \cs_new_protected:Npn \__color_set:nnn #1#2#3
34553 {
34554   \tl_clear:N \l_color_fixed_model_tl
34555   \__color_set:nn {#2} {#3}
34556   \tl_set:Nn \l_color_fixed_model_tl {#1}
34557 }
34558 \cs_new_protected:Npn \__color_set:nn #1#2
34559 {
34560   \str_if_eq:nnF {#1} { . }
34561   {
34562     \__color_parse:nN {#2} \l__color_named_tl
34563     \tl_clear_new:c { l__color_named_ #1 _tl }
34564     \tl_set:cx { l__color_named_ #1 _tl }
34565     { \__color_model:N \l__color_named_tl }
34566     \prop_clear_new:c { l__color_named_ #1 _prop }
34567     \prop_put:cvx { l__color_named_ #1 _prop } { l__color_named_ #1 _tl }
34568     { \__color_values:N \l__color_named_tl }
34569     \__color_set:nnw {#1} {#2} #2 ! \s__color_stop
34570   }
34571 }

```

When setting an expression-based color, there could be multiple model data available for one or more of the input colors. Where that is true for the *first* named color in an expression, we re-parse the expression when they are also parameter-based: only `cmYk`, `gray` and `rgb` make any sense here. There is a bit of a performance hit but this should be rare and taking place during set-up.

```

34572 \cs_new_protected:Npn \__color_set:nnw #1#2#3 ! #4 \s__color_stop
34573 {
34574   \clist_map_inline:nn { cmyk , gray , rgb }
34575   {
34576     \prop_get:cnNT { l__color_named_ #3 _prop } {##1} \l__color_internal_tl
34577     {
34578       \prop_if_in:cnF { l__color_named_ #1 _prop } {##1}
34579       {
34580         \group_begin:
34581         \tl_set:cn { l__color_named_ #3 _tl } {##1}
34582         \__color_parse:nN {#2} \l__color_internal_tl
34583         \exp_args:NNNV \group_end:
34584         \tl_set:Nn \l__color_internal_tl \l__color_internal_tl
34585         \prop_put:cxx { l__color_named_ #1 _prop }
34586         { \__color_model:N \l__color_internal_tl }
34587         { \__color_values:N \l__color_internal_tl }
34588       }
34589     }
34590   }
34591 }
34592 \cs_new_protected:Npn \color_set:nnn #1#2#3
34593 {
34594   \str_if_eq:nnF {#1} { . }
34595   {
34596     \tl_clear_new:c { l__color_named_ #1 _tl }
34597     \prop_clear_new:c { l__color_named_ #1 _prop }
34598     \exp_args:Ne \__color_set_aux:nnn { \tl_to_str:n {#2} }
34599     {#1} {#3}
34600   }
34601 }
34602 \cs_new_protected:Npx \__color_set_aux:nnn #1#2#3
34603 {
34604   \exp_not:N \__color_set_colon:nnw {#2} {#3}
34605   #1 \c_colon_str \c_colon_str \exp_not:N \s__color_stop
34606 }
34607 \use:x
34608 {
34609   \cs_new_protected:Npn \exp_not:N \__color_set_colon:nnw
34610   ##1##2 ##3 \c_colon_str ##4 \c_colon_str
34611   ##5 \exp_not:N \s__color_stop
34612 }
34613 {
34614   \tl_if_blank:nTF {#4}
34615   { \__color_set_loop:nw {#1} #3 }
34616   { \__color_set_loop:nw {#1} #4 }
34617   // \s__color_mark #2 // \s__color_stop
34618 }
34619 \cs_new_protected:Npn \__color_set_loop:nw
34620 #1#2 / #3 \s__color_mark #4 / #5 \s__color_stop
34621 {
34622   \tl_if_blank:nF {#2}
34623   {
34624     \__color_select:nnN {#2} {#4} \l__color_named_tl
34625     \tl_set:Nx \l__color_internal_tl { \__color_model:N \l__color_named_tl }

```

```

34626     \tl_if_empty:cT { l__color_named_ #1 _tl }
34627     { \tl_set_eq:cN { l__color_named_ #1 _tl } \l__color_internal_tl }
34628     \prop_put:cVx { l__color_named_ #1 _prop } \l__color_internal_tl
34629     { \__color_values:N \l__color_named_tl }
34630     \__color_set_loop:nw {#1} #3 \s__color_mark #5 \s__color_stop
34631   }
34632 }
34633 \cs_new_protected:Npn \color_set_eq:nn #1#2
34634 {
34635   \__color_if_defined:nTF {#2}
34636   {
34637     \tl_clear_new:c { l__color_named_ #1 _tl }
34638     \prop_clear_new:c { l__color_named_ #1 _prop }
34639     \str_if_eq:nnTF {#2} { . }
34640     {
34641       \tl_set:cx { l__color_named_ #1 _tl }
34642       { \__color_model:N \l__color_current_tl }
34643       \prop_put:cvx { l__color_named_ #1 _prop } { l__color_named_ #1 _tl }
34644       { \__color_values:N \l__color_current_tl }
34645     }
34646     {
34647       \tl_set_eq:cc { l__color_named_ #1 _tl } { l__color_named_ #2 _tl }
34648       \prop_set_eq:cc { l__color_named_ #1 _prop } { l__color_named_ #2 _prop }
34649     }
34650   }
34651   {
34652     \msg_error:nnn { color } { unknown-color } {#2}
34653   }
34654 }

```

(End definition for `\color_set:nn` and others. These functions are documented on page 290.)

A small set of colors are always defined.

```

34655 \color_set:nnn { black } { gray } { 0 }
34656 \color_set:nnn { white } { gray } { 1 }
34657 \color_set:nnn { cyan } { cmyk } { 1 , 0 , 0 , 0 }
34658 \color_set:nnn { magenta } { cmyk } { 0 , 1 , 0 , 0 }
34659 \color_set:nnn { yellow } { cmyk } { 0 , 0 , 1 , 0 }
34660 \color_set:nnn { red } { rgb } { 1 , 0 , 0 }
34661 \color_set:nnn { green } { rgb } { 0 , 1 , 0 }
34662 \color_set:nnn { blue } { rgb } { 0 , 0 , 1 }

```

`\l__color_named_._prop` A special named color: this is always defined though not fixed in definition.

```

\l__color_named_._tl
34663 \prop_new:c { l__color_named_._prop }
34664 \tl_new:c { l__color_named_._tl }
34665 \tl_set:cx { l__color_named_._tl } { \__color_model:N \l__color_current_tl }

```

(End definition for `\l__color_named_._prop` and `\l__color_named_._tl`.)

86.11 Exporting colors

```

\color_export:nnN
\color_export:nnnN
\__color_export:nN
\__color_export:nnnN
34666 \cs_new_protected:Npn \color_export:nnN #1#2#3
34667 {

```



```

34668 \group_begin:
34669 \tl_if_exist:cT { c__color_export_ #2 _tl }
34670 { \tl_set_eq:Nc \l_color_fixed_model_tl { c__color_export_ #2 _tl } }
34671 \__color_parse:nN {#1} #3
34672 \__color_export:nN {#2} #3
34673 \exp_args:NNNV \group_end:
34674 \tl_set:Nn #3 #3
34675 }
34676 \cs_new_protected:Npn \color_export:nnnN #1#2#3#4
34677 {
34678 \__color_select_main:Nw #4
34679 #1 / / \s__color_mark #2 / / \s__color_stop
34680 \__color_export:nN {#3} #4
34681 }
34682 \cs_new_protected:Npn \__color_export:nN #1#2
34683 { \exp_after:wN \__color_export:nnnN #2 {#1} #2 }
34684 \cs_new:Npn \__color_export:nnnN #1#2#3#4
34685 {
34686 \cs_if_exist_use:cF { __color_export_format_ #3 :nnN }
34687 {
34688 \msg_error:nnn { color } { unknown-export-format } {#3}
34689 \use_none:nnn
34690 }
34691 {#1} {#2} #4
34692 }

```

(End definition for `\color_export:nnN` and others. These functions are documented on page 293.)

`__color_export_format_backend:nnN` Simple.

```

34693 \cs_new_protected:Npn \__color_export_format_backend:nnN #1#2#3
34694 { \tl_set:Nn #3 { {#1} {#2} } }

```

(End definition for `__color_export_format_backend:nnN`.)

`__color_export:nnnNN` A generic auxiliary for cases where only one model is appropriate.

```

34695 \cs_new_protected:Npn \__color_export:nnnNN #1#2#3#4#5
34696 {
34697 \str_if_eq:nnTF {#2} {#1}
34698 { #5 #4 #3 \s__color_stop }
34699 {
34700 \__color_convert:nnnN {#2} {#1} {#3} #4
34701 \exp_after:wN #5 \exp_after:wN #4
34702 #4 \s__color_stop
34703 }
34704 }

```

(End definition for `__color_export:nnnNN`.)

```

\c_color_export_comma-sep-cmyk_tl
\c_color_export_comma-sep-rgb_tl
\c__color_export_HTML_tl
\c_color_export_space-sep-cmyk_tl
\c_color_export_space-sep-rgb_tl
34705 \tl_const:cn { c__color_export_comma-sep-cmyk_tl } { cmyk }
34706 \tl_const:cn { c__color_export_comma-sep-rgb_tl } { rgb }
34707 \tl_const:Nn \c__color_export_HTML_tl { rgb }
34708 \tl_const:cn { c__color_export_space-sep-cmyk_tl } { cmyk }
34709 \tl_const:cn { c__color_export_space-sep-rgb_tl } { rgb }

```

(End definition for \c__color_export_comma-sep-cmyk_tl and others.)

```

\__color_export_format_comma-sep-cmyk:nnN
\__color_export_format_comma-sep-rgb:nnN
\__color_export_format_space-sep-cmyk:nnN
\__color_export_format_space-sep-rgb:nnN
34710 \group_begin:
34711 \cs_set_protected:Npn \__color_tmp:w #1#2
34712 {
34713   \cs_new_protected:cpx { __color_export_format_ #1 :nnN } ##1##2##3
34714   {
34715     \exp_not:N \__color_export:nnnNN {#2} {##1} {##2} ##3
34716     \exp_not:c { __color_export_ #1 :Nw }
34717   }
34718 }
34719 \__color_tmp:w { comma-sep-cmyk } { cmyk }
34720 \__color_tmp:w { comma-sep-rgb } { rgb }
34721 \__color_tmp:w { HTML } { rgb }
34722 \__color_tmp:w { space-sep-cmyk } { cmyk }
34723 \__color_tmp:w { space-sep-rgb } { rgb }
34724
34725 \group_end:

```

(End definition for __color_export_format_comma-sep-cmyk:nnN and others.)

```

\__color_export_space-sep-cmyk:Nw
\__color_export_comma-sep-cmyk:Nw
34726 \cs_new_protected:cpn { __color_export_comma-sep-cmyk:Nw }
34727   #1#2 ~ #3 ~ #4 ~ #5 \s__color_stop
34728   { \tl_set:Nn #1 { #2 , #3 , #4 , #5 } }
34729 \cs_new_protected:cpn { __color_export_space-sep-cmyk:Nw } #1#2 \s__color_stop
34730   { \tl_set:Nn #1 {#2} }

```

(End definition for __color_export_space-sep-cmyk:Nw and __color_export_comma-sep-cmyk:Nw.)

```

\__color_export_comma-sep-rgb:Nw HTML values must be given in rgb: we force conversion if required, then do some simple
\__color_export_HTML:Nw maths.
\__color_export_space-sep-rgb:Nw
\__color_export_HTML:Nw
34731 \cs_new_protected:cpn { __color_export_comma-sep-rgb:Nw } #1#2 ~ #3 ~ #4 \s__color_stop
34732   { \tl_set:Nx #1 { #2 , #3 , #4 } }
34733 \cs_new_protected:Npn \__color_export_HTML:Nw #1#2 ~ #3 ~ #4 \s__color_stop
34734   {
34735     \tl_set:Nx #1
34736     {
34737       \__color_export_HTML:n {#2}
34738       \__color_export_HTML:n {#3}
34739       \__color_export_HTML:n {#4}
34740     }
34741   }
34742 \cs_new:Npn \__color_export_HTML:n #1
34743   {
34744     \fp_compare:nNnTF {#1} = { 0 }
34745     { 00 }
34746     {
34747       \fp_compare:nNnT { #1 * 255 } < { 16 } { 0 }
34748       \int_to_Hex:n { \fp_to_int:n { #1 * 255 } }
34749     }
34750   }
34751 \cs_new_protected:cpn { __color_export_space-sep-rgb:Nw } #1#2 \s__color_stop
34752   { \tl_set:Nn #1 {#2} }

```

(End definition for `_color_export_comma-sep-rgb:Nw` and others.)

86.12 Additional color models

`\l__color_internal_prop`

34753 `\prop_new:N \l__color_internal_prop`

(End definition for `\l__color_internal_prop`.)

`\g__color_model_int`

A tracker for the total number of new models.

34754 `\int_new:N \g__color_model_int`

(End definition for `\g__color_model_int`.)

`\c__color_fallback_cmyk_tl`
`\c__color_fallback_gray_tl`
`\c__color_fallback_rgb_tl`

For every colorspace, we define one of the base colorspace as a fallback. The base colorspace themselves are their own fallback.

34755 `\tl_const:Nn \c__color_fallback_cmyk_tl { cmyk }`

34756 `\tl_const:Nn \c__color_fallback_gray_tl { gray }`

34757 `\tl_const:Nn \c__color_fallback_rgb_tl { rgb }`

(End definition for `\c__color_fallback_cmyk_tl`, `\c__color_fallback_gray_tl`, and `\c__color_fallback_rgb_tl`.)

`\g__color_colorants_prop`

Mapping from names to colorants.

34758 `\prop_new:N \g__color_colorants_prop`

34759 `\prop_gput:Nnn \g__color_colorants_prop { black } { Black }`

34760 `\prop_gput:Nnn \g__color_colorants_prop { blue } { Blue }`

34761 `\prop_gput:Nnn \g__color_colorants_prop { cyan } { Cyan }`

34762 `\prop_gput:Nnn \g__color_colorants_prop { green } { Green }`

34763 `\prop_gput:Nnn \g__color_colorants_prop { magenta } { Magenta }`

34764 `\prop_gput:Nnn \g__color_colorants_prop { none } { None }`

34765 `\prop_gput:Nnn \g__color_colorants_prop { red } { Red }`

34766 `\prop_gput:Nnn \g__color_colorants_prop { yellow } { Yellow }`

(End definition for `\g__color_colorants_prop`.)

`\c__color_model_whitepoint_CIELAB_a_tl`

Whitepoint data for the CIELAB profiles.

`\c__color_model_whitepoint_CIELAB_b_tl`

34767 `\tl_const:Nn \c__color_model_whitepoint_CIELAB_a_tl { 1.0985 ~ 1 ~ 0.3558 }`

`\c__color_model_whitepoint_CIELAB_e_tl`

34768 `\tl_const:Nn \c__color_model_whitepoint_CIELAB_b_tl { 0.9807 ~ 1 ~ 1.1822 }`

`\c__color_model_whitepoint_CIELAB_d50_tl`

34769 `\tl_const:Nn \c__color_model_whitepoint_CIELAB_e_tl { 1 ~ 1 ~ 1 }`

`\c__color_model_whitepoint_CIELAB_d55_tl`

34770 `\tl_const:cn { c__color_model_whitepoint_CIELAB_d50_tl } { 0.9642 ~ 1 ~ 0.8251 }`

`\c__color_model_whitepoint_CIELAB_d65_tl`

34771 `\tl_const:cn { c__color_model_whitepoint_CIELAB_d55_tl } { 0.9568 ~ 1 ~ 0.9214 }`

`\c__color_model_whitepoint_CIELAB_d75_tl`

34772 `\tl_const:cn { c__color_model_whitepoint_CIELAB_d65_tl } { 0.9504 ~ 1 ~ 1.0888 }`

34773 `\tl_const:cn { c__color_model_whitepoint_CIELAB_d75_tl } { 0.9497 ~ 1 ~ 1.2261 }`

(End definition for `\c__color_model_whitepoint_CIELAB_a_tl` and others.)

`\c__color_model_range_CIELAB_tl`

The range for CIELAB color spaces.

34774 `\tl_const:Nn \c__color_model_range_CIELAB_tl { 0 ~ 100 ~ -128 ~ 127 ~ -128 ~ 127 }`

(End definition for `\c__color_model_range_CIELAB_tl`.)

`\g_color_alternative_model_prop` For tracking the alternative model set up for separations, etc.

```

34775 \prop_new:N \g_color_alternative_model_prop
34776 \clist_map_inline:nn { cyan , magenta , yellow , black }
34777   { \prop_gput:Nnn \g_color_alternative_model_prop {#1} { cmyk } }
34778 \clist_map_inline:nn { red , green , blue }
34779   { \prop_gput:Nnn \g_color_alternative_model_prop {#1} { rgb } }

```

(End definition for `\g_color_alternative_model_prop`.)

`\g_color_alternative_values_prop` Same for the values: a bit more involved.

```

34780 \prop_new:N \g_color_alternative_values_prop
34781 \prop_gput:Nnn \g_color_alternative_values_prop { cyan } { 1 , 0 , 0 , 0 }
34782 \prop_gput:Nnn \g_color_alternative_values_prop { magenta } { 0 , 1 , 0 , 0 }
34783 \prop_gput:Nnn \g_color_alternative_values_prop { yellow } { 0 , 0 , 1 , 0 }
34784 \prop_gput:Nnn \g_color_alternative_values_prop { black } { 0 , 0 , 0 , 1 }
34785 \prop_gput:Nnn \g_color_alternative_values_prop { red } { 1 , 0 , 0 , 0 }
34786 \prop_gput:Nnn \g_color_alternative_values_prop { green } { 0 , 1 , 0 , 0 }
34787 \prop_gput:Nnn \g_color_alternative_values_prop { blue } { 0 , 0 , 1 , 0 }

```

(End definition for `\g_color_alternative_values_prop`.)

`\color_model_new:nnn` Set up a new model: in general this has to be handled by a family-dependent function.
`__color_model_new:nnn` To avoid some “interesting” questions with casing, we fold the case of the family name. The key–value list should always be present, so we convert it up-front to a `prop`, then deal with the detail on a per-family basis.

```

34788 \cs_new_protected:Npn \color_model_new:nnn #1#2#3
34789 {
34790   \exp_args:Nee \__color_model_new:nnn
34791     { \tl_to_str:n {#1} }
34792     { \str_foldcase:n {#2} } {#3}
34793 }
34794 \cs_new_protected:Npn \__color_model_new:nnn #1#2#3
34795 {
34796   \cs_if_exist:cTF { __color_parse_model_ #1 :w }
34797   {
34798     \msg_error:nnn { color } { model-already-defined } {#1}
34799   }
34800   {
34801     \cs_if_exist:cTF { __color_model_ #2 :n }
34802     {
34803       \prop_set_from_keyval:Nn \l__color_internal_prop {#3}
34804       \use:c { __color_model_ #2 :n } {#1}
34805     }
34806     {
34807       \msg_error:nnn { color } { unknown-model-type } {#2}
34808     }
34809   }
34810 }

```

(End definition for `\color_model_new:nnn` and `__color_model_new:nnn`. This function is documented on page 294.)

`__color_model_init:nnnn` A shared auxiliary to do the basics of setting up a new model: reserve a number, create
`__color_model_init:nnnx` a white-equivalent, set up links to the backend.

```

34811 \cs_new_protected:Npn \__color_model_init:nnnn #1#2#3#4
34812 {
34813   \int_gincr:N \g__color_model_int
34814   \clist_map_inline:nn { fill , stroke , select }
34815   {
34816     \cs_new_protected:cpx { __color_backend_ ##1 _ #1 :n } ####1
34817     {
34818       \exp_not:c { __color_backend_ ##1 _ #3 :nn }
34819       { color \int_use:N \g__color_model_int } {####1}
34820     }
34821   }
34822   \cs_new_protected:cpx { __color_model_ #1 _white: }
34823   {
34824     \prop_put:Nnn \exp_not:N \l__color_named_white_prop {#1}
34825     { \exp_not:n {#4} }
34826     \exp_not:N \int_compare:nNnF { \tex_currentgrouplevel:D } = 0
34827     { \group_insert_after:N \exp_not:c { __color_model_ #1 _ white: } }
34828   }
34829   \use:c { __color_model_ #1 _white: }
34830 }
34831 \cs_generate_variant:Nn \__color_model_init:nnnn { nnnx }

```

(End definition for __color_model_init:nnnn.)

```

\__color_model_separation:n
\__color_model_separation:nn
  \__color_model_separation:nnn
\__color_model_separation:w
  \__color_model_separation_cmyk:nnnnnn
  \__color_model_separation_gray:nnnnnn
  \__color_model_separation_rgb:nnnnnn
\__color_model_convert:nnn
  \__color_model_separation_CIELAB:nnnnnn
\__color_model_separation_CIELAB:nnnnnnn

```

Separations must have a “real” name, which is pretty easy to find.

```

34832 \cs_new_protected:Npn \__color_model_separation:n #1
34833 {
34834   \prop_get:NnNTF \l__color_internal_prop { name }
34835   \l__color_internal_tl
34836   {
34837     \exp_args:NV \__color_model_separation:nn
34838     \l__color_internal_tl {#1}
34839   }
34840   {
34841     \msg_error:nnn { color }
34842     { separation-requires-name } {#1}
34843   }
34844 }

```

We have two keys to find at this stage: the alternative space model and linked values.

```

34845 \cs_new_protected:Npn \__color_model_separation:nn #1#2
34846 {
34847   \prop_get:NnNTF \l__color_internal_prop { alternative-model }
34848   \l__color_internal_tl
34849   {
34850     \exp_args:NV \__color_model_separation:nnn
34851     \l__color_internal_tl {#2} {#1}
34852   }
34853   {
34854     \msg_error:nnn { color }
34855     { separation-alternative-model } {#2}
34856   }
34857 }
34858 \cs_new_protected:Npn \__color_model_separation:nnn #1#2#3
34859 {

```

```

34860 \cs_if_exist:cTF { __color_model_separation_ #1 :nnnnnn }
34861 {
34862   \prop_get:NnNTF \l__color_internal_prop { alternative-values }
34863   \l__color_internal_tl
34864   {
34865     \exp_after:wN \__color_model_separation:w \l__color_internal_tl
34866     , 0 , 0 , 0 , 0 \s__color_stop {#2} {#3} {#1}
34867   }
34868   {
34869     \msg_error:nnn { color }
34870     { separation-alternative-values } {#2}
34871   }
34872 }
34873 {
34874   \msg_error:nnn { color }
34875   { unknown-alternative-model } {#1}
34876 }
34877 }

```

As each alternative space leads to a different requirement for conversion, and as there are only a small number of choices, we manually split the data and then set up. Notice that mixing tints is really just the same as mixing gray. The **white** color is special, as it allows tints to be adjusted without an additional color space. To make sure the data is set for that at all group levels, we need to work on a per-level basis. Within the output, only the set-up needs the “real” name of the colorspace: we use a simple tracking number for general usage as this is a clear namespace without issues of escaping chars.

```

34878 \cs_new_protected:Npn \__color_model_separation:w
34879 #1 , #2 , #3 , #4 , #5 \s__color_stop #6#7#8
34880 {
34881   \__color_model_init:nnnn {#6} { 1 } { separation } { 0 }
34882   \cs_new_eq:cN { __color_parse_mix_ #6 :nw } \__color_parse_mix_gray:nw
34883   \cs_new:cpn { __color_parse_model_ #6 :w } ##1 , ##2 \s__color_stop
34884   { {#6} { __color_parse_number:n {##1} } }
34885   \use:c { __color_model_separation_ #8 :nnnnnn }
34886   {#6} {#7} {#1} {#2} {#3} {#4}
34887   \prop_gput:Nnn \g__color_alternative_model_prop {#6} {#8}
34888   \prop_gput:Nnx \g__color_colorants_prop {#6}
34889   { \str_convert_pdfname:n {#7} }
34890 }
34891 \cs_new_protected:Npn \__color_model_separation_cmyk:nnnnnn #1#2#3#4#5#6
34892 {
34893   \tl_const:cn { c__color_fallback_ #1 _tl } { cmyk }
34894   \cs_new:cpn { __color_convert_ #1 _cmyk:w } ##1 \s__color_stop
34895   {
34896     \fp_eval:n {##1 * #3} ~
34897     \fp_eval:n {##1 * #4} ~
34898     \fp_eval:n {##1 * #5} ~
34899     \fp_eval:n {##1 * #6}
34900   }
34901   \cs_new:cpn { __color_convert_cmyk_ #1 :w } ##1 \s__color_stop { 1 }
34902   \prop_gput:Nnn \g__color_alternative_values_prop {#1} { #3 , #4 , #5 , #6 }
34903   \__color_backend_separation_init:nnnn {#2} { /DeviceCMYK } { }
34904   { 0 ~ 0 ~ 0 ~ 0 } { #3 ~ #4 ~ #5 ~ #6 }
34905 }

```

```

34906 \cs_new_protected:Npn \__color_model_separation_rgb:nnnnnn #1#2#3#4#5#6
34907 {
34908   \tl_const:cn { c__color_fallback_ #1 _tl } { rgb }
34909   \cs_new:cpn { __color_convert_ #1 _rgb:w } ##1 \s__color_stop
34910   {
34911     \fp_eval:n {##1 * #3} ~
34912     \fp_eval:n {##1 * #4} ~
34913     \fp_eval:n {##1 * #5}
34914   }
34915   \cs_new:cpn { __color_convert_rgb_ #1 :w } ##1 \s__color_stop { 1 }
34916   \prop_gput:Nnn \g__color_alternative_values_prop {#1} { #3 , #4 , #5 }
34917   \__color_backend_separation_init:nnnnn {#2} { /DeviceRGB } { }
34918   { 0 ~ 0 ~ 0 } { #3 ~ #4 ~ #5 }
34919 }
34920 \cs_new_protected:Npn \__color_model_separation_gray:nnnnnn #1#2#3#4#5#6
34921 {
34922   \tl_const:cn { c__color_fallback_ #1 _tl } { gray }
34923   \cs_new:cpn { __color_convert_ #1 _gray:w } ##1 \s__color_stop
34924   { \fp_eval:n {##1 * #3} }
34925   \cs_new:cpn { __color_convert_gray_ #1 :w } ##1 \s__color_stop { 1 }
34926   \prop_gput:Nnn \g__color_alternative_values_prop {#1} {#3}
34927   \__color_backend_separation_init:nnnnn {#2} { /DeviceGray } { } { 0 } {#3}
34928 }

```

Generic model conversion *via* an alternative intermediate.

```

34929 \cs_new_protected:Npn \__color_model_convert:nnn #1#2#3
34930 {
34931   \cs_new:cpx { __color_convert_ #1 _ #3 :w } ##1 \s__color_stop
34932   {
34933     \exp_not:N \exp_args:NNe \exp_not:N \use:nn
34934     \exp_not:c { __color_convert_ #2 _ #3 :w }
34935     { \exp_not:c { __color_convert_ #1 _ #2 :w } ##1 \s__color_stop }
34936     \c_space_tl \exp_not:N \s__color_stop
34937   }
34938 }

```

Setting up for CIELAB needs a bit more work: there is the illuminant and the need for an appropriate object.

```

34939 \cs_new_protected:Npn \__color_model_separation_CIELAB:nnnnnn #1#2#3#4#5#6
34940 {
34941   \prop_get:NnNF \l__color_internal_prop { illuminant }
34942   \l__color_internal_tl
34943   {
34944     \msg_error:nnn { color }
34945     { CIELAB-requires-illuminant } {#1}
34946     \tl_set:Nn \l__color_internal_tl { d50 }
34947   }
34948   \exp_args:NV \__color_model_separation_CIELAB:nnnnnnn
34949   \l__color_internal_tl {#1} {#2} {#3} {#4} {#5} {#6}
34950 }

```

If a CIELAB space is being set up, we need the illuminant, then create the appropriate set up. At present, this doesn't include `BlackPoint` or `Range` data, but that may be added later. As CIELAB colors cannot be converted to anything else, we fallback to

producing black in the gray colorspace: the user should set up a second model for colors set up this way.

```

34951 \cs_new_protected:Npn \__color_model_separation_CIELAB:nnnnnnn #1#2#3#4#5#6#7
34952 {
34953   \tl_if_exist:cTF { c__color_model_whitepoint_CIELAB_ #1 _tl }
34954   {
34955     \__color_backend_separation_init_CIELAB:nnn {#1} {#3} { #4 ~ #5 ~ #6 }
34956     \tl_const:cn { c__color_fallback_ #2 _tl } { gray }
34957     \cs_new:cpn { __color_convert_ #2 _gray:w } ##1 \s__color_stop
34958     { 0 }
34959     \cs_new:cpn { __color_convert_gray_ #2 :w } ##1 \s__color_stop
34960     { 1 }
34961   }
34962   {
34963     \msg_error:nnn { color }
34964     { unknown-CIELAB-illuminant } {#1}
34965   }
34966 }

```

(End definition for `__color_model_separation:n` and others.)

```

\__color_model_devicen:n
\__color_model_devicen:nn
\__color_model_devicen:nnn
\__color_model_devicen:nnnn
  \__color_model_devicen_parse_1:nn
  \__color_model_devicen_parse_2:nn
  \__color_model_devicen_parse_3:nn
  \__color_model_devicen_parse_4:nn
\__color_model_devicen_parse_generic:nn
  \__color_model_devicen_parse:nw
  \__color_model_devicen_mix:nw
  \__color_model_devicen_init:nnn
  \__color_model_devicen_init:nnnn
  \__color_model_devicen_tranform:w
\__color_model_devicen_tranform_1:nnnnn
\__color_model_devicen_tranform_3:nnnnn
\__color_model_devicen_tranform_4:nnnnn
  \__color_model_devicen_tranform:nnn
  \__color_model_devicen_colorant:n
  \__color_model_devicen_convert:nnn
  \__color_model_devicen_convert_cmyk:n
  \__color_model_devicen_convert_gray:n
  \__color_model_devicen_convert_rgb:n
  \__color_model_devicen_convert:nnnn
  \__color_model_devicen_convert:n
  \__color_model_devicen_convert_aux:n
  \__color_model_devicen_convert:w
  \__color_convert_devicen_cmyk:nnnnw
\__color_convert_devicen_cmyk:nnnnnnnn
\__color_convert_devicen_cmyk_aux:nnnnw
  \__color_convert_devicen_gray:nw
  \__color_convert_devicen_gray:nnn
  \__color_convert_devicen_gray_aux:nw
  \__color_convert_devicen_rgb:nnnn
  \__color_convert_devicen_rgb:nnnnnnn
  \__color_convert_devicen_rgb_aux:nnnnw

```

We require a list of component names here: one might call them colorants, but it's convenient to use TeX names instead so we slightly adjust the terminology.

```

34967 \cs_new_protected:Npn \__color_model_devicen:n #1
34968 {
34969   \prop_get:NnNTF \l__color_internal_prop { names }
34970   \l__color_internal_tl
34971   {
34972     \exp_args:NV \__color_model_devicen:nn
34973     \l__color_internal_tl {#1}
34974   }
34975   {
34976     \msg_error:nnn { color }
34977     { DeviceN-requires-names } {#1}
34978   }
34979 }

```

All valid models will have an alternative listed, either hard-coded for the core device ones, or dynamically added for Separations, etc.

```

34980 \cs_new_protected:Npn \__color_model_devicen:nn #1#2
34981 {
34982   \tl_clear:N \l__color_model_tl
34983   \clist_map_inline:nn {#1}
34984   {
34985     \prop_get:NnNTF \g__color_alternative_model_prop {##1}
34986     \l__color_internal_tl
34987     {
34988       \tl_if_empty:NTF \l__color_model_tl
34989       { \tl_set_eq:NN \l__color_model_tl \l__color_internal_tl }
34990       {
34991         \str_if_eq:VVF \l__color_model_tl \l__color_internal_tl
34992         {
34993           \msg_error:nnn { color }
34994           { DeviceN-inconsistent-alternative }

```



```

34995         {#2}
34996         \clist_map_break:n { \use_none:nnnn }
34997     }
34998 }
34999 }
35000 {
35001     \str_if_eq:nnF {##1} { none }
35002     {
35003         \msg_error:nnn { color }
35004         { DeviceN-no-alternative }
35005         {#2}
35006     }
35007 }
35008 }
35009 \tl_if_empty:NTF \l__color_model_tl
35010 {
35011     \msg_error:nnn { color }
35012     { DeviceN-no-alternative } {#2}
35013 }
35014 { \exp_args:NV \__color_model_devicen:nnn \l__color_model_tl {#1} {#2} }
35015 }

```

We now complete the data we require by first finding out how many colorants there are, then moving on to begin constructing the function required to map to the alternative color space.

```

35016 \cs_new_protected:Npn \__color_model_devicen:nnn #1#2#3
35017 {
35018     \exp_args:Nx \__color_model_devicen:nnnn
35019     { \clist_count:n {#2} } {#1} {#2} {#3}
35020 }

```

At this stage, we have checked everything is in place, so we can set up the T_EX and backend data structures. As for separations, it’s not really possible in general to have a fallback, so we simply provide “black” for each element.

```

35021 \cs_new_protected:Npn \__color_model_devicen:nnnn #1#2#3#4
35022 {
35023     \__color_model_init:nnnx {#4} {#1} { devicen }
35024     {
35025         0 \prg_replicate:nn { #1 - 1 } { ~ 0 }
35026     }
35027     \cs_if_exist_use:cF { __color_model_devicen_parse_ #1 :nn }
35028     { \__color_model_devicen_parse_generic:nn }
35029     {#4} {#1}
35030     \__color_model_devicen_init:nnn {#1} {#2} {#3}
35031     \__color_model_devicen_convert:nnnx {#4} {#2} {#3}
35032     {
35033         1 \prg_replicate:nn { #1 - 1 } { ~ 1 }
35034     }
35035 }

```

For short lists of DeviceN colors, we can use hand-tuned parsing. This lines up with other models, where we allow for up to four components. For larger spaces, rather than limit artificially, we use a somewhat slow approach based on open-ended commas-lists.

```

35036 \cs_new_protected:cpn { __color_model_devicen_parse_1:nn } #1#2
35037 {

```

```

35038 \cs_new:cpn { __color_parse_model_ #1 :w } ##1 , ##2 \s__color_stop
35039 { {#1} { \__color_parse_number:n {##1} } }
35040 \cs_new_eq:cN { __color_parse_mix_ #1 :nw } \__color_parse_mix_gray:nw
35041 }
35042 \cs_new_protected:cpn { __color_model_devicen_parse_2:nn } #1#2
35043 {
35044 \cs_new:cpn { __color_parse_model_ #1 :w } ##1 , ##2 , ##3 \s__color_stop
35045 { {#1} { \__color_parse_number:n {##1} ~ \__color_parse_number:n {##2} } }
35046 \cs_new:cpn { __color_parse_mix_ #1 :nw }
35047 ##1##2 ~ ##3 \s__color_mark ##4 ~ ##5 \s__color_stop
35048 {
35049 \fp_eval:n { ##2 * ##1 + ##4 * ( 1 - ##1 ) } \c_space_tl
35050 \fp_eval:n { ##3 * ##1 + ##5 * ( 1 - ##1 ) }
35051 }
35052 }
35053 \cs_new_protected:cpn { __color_model_devicen_parse_3:nn } #1#2
35054 {
35055 \cs_new:cpn { __color_parse_model_ #1 :w } ##1 , ##2 , ##3 , ##4 \s__color_stop
35056 {
35057 {#1}
35058 {
35059 \__color_parse_number:n {##1} ~
35060 \__color_parse_number:n {##2} ~
35061 \__color_parse_number:n {##3}
35062 }
35063 }
35064 \cs_new_eq:cN { __color_parse_mix_ #1 :nw } \__color_parse_mix_rgb:nw
35065 }
35066 \cs_new_protected:cpn { __color_model_devicen_parse_4:nn } #1#2
35067 {
35068 \cs_new:cpn { __color_parse_model_ #1 :w }
35069 ##1 , ##2 , ##3 , ##4 , ##5 \s__color_stop
35070 {
35071 {#1}
35072 {
35073 \__color_parse_number:n {##1} ~
35074 \__color_parse_number:n {##2} ~
35075 \__color_parse_number:n {##3} ~
35076 \__color_parse_number:n {##4}
35077 }
35078 }
35079 \cs_new_eq:cN { __color_parse_mix_ #1 :nw } \__color_parse_mix_cmyk:nw
35080 }
35081 \cs_new_protected:Npn \__color_model_devicen_parse_generic:nn #1#2
35082 {
35083 \cs_new:cpn { __color_parse_model_ #1 :w } ##1 , ##2 \s__color_stop
35084 {
35085 {#1}
35086 { \__color_model_devicen_parse:nw {#2} ##1 , ##2 , \q_nil , \s__color_stop }
35087 }
35088 \cs_new:cpx { __color_parse_mix_ #1 :nw }
35089 ##1 ##2 \s__color_mark ##3 \s__color_stop
35090 {
35091 \exp_not:N \__color_model_devicen_mix:nw {##1}

```

```

35092         ##2 \c_space_tl \exp_not:N \q_nil \c_space_tl \exp_not:N \s__color_mark
35093         ##3 \c_space_tl \exp_not:N \q_nil \c_space_tl \exp_not:N \s__color_stop
35094     }
35095 }
35096 \cs_new:Npn \__color_model_devicen_parse:nw #1#2 , #3 \s__color_stop
35097 {
35098     \int_compare:nNtT {#1} > 0
35099     {
35100         \quark_if_nil:nTF {#2}
35101         { \prg_replicate:nn {#1} { 0 ~ } }
35102         {
35103             \__color_parse_number:n {#2}
35104             \int_compare:nNtT {#1} > 1 { ~ }
35105             \exp_args:Nf \__color_model_devicen_parse:nw
35106                 { \int_eval:n { #1 - 1 } } #3 \s__color_stop
35107         }
35108     }
35109 }
35110 \cs_new:Npn \__color_model_devicen_mix:nw #1#2 ~ #3 \s__color_mark #4 ~ #5 \s__color_stop
35111 {
35112     \fp_eval:n { #2 * #1 + #4 * ( 1 - #1 ) }
35113     \quark_if_nil:oF { \tl_head:w #3 \q_stop }
35114     {
35115         \c_space_tl
35116         \__color_model_devicen_mix:nw {#1} #3 \s__color_mark #5 \s__color_stop
35117     }
35118 }

```

To construct the tint transformation, we have to use PostScript. The aim is to have the final tint for each device colorant as

$$1 - \prod_n (1 - X_n D_{X_n})$$

where X is a DeviceN colorant and D is the amount of device colorant that the DeviceN colorant maps to. At the start of the process, the PostScript stack will contain the X_n values, whilst we have the D values on a per-DeviceN colorant basis. The more convenient approach for us is therefore to take each DeviceN colorant in turn and find the value $1 - X_n D_{X_n}$, multiplying as we go, and finalise with the subtraction. That contrasts to `colorspace`: it splits the process up by process color, which works better when you have a fixed list of colorants. (`colorspace` only supports up to 4 DeviceN colors, and only `cmymk` as the alternative space.) To set this up, we first need to know the number of values in the target color space: this is easily handled as there are a very small range of possibilities. Once we have that information, it's relatively easy to build the required PostScript using some generic code.

```

35119 \cs_new_protected:Npn \__color_model_devicen_init:nnn #1#2#3
35120 {
35121     \exp_args:Ne \__color_model_devicen_init:nnnn
35122     {
35123         \str_case:nn {#2}
35124         {
35125             { cmyk } { 4 }
35126             { gray } { 1 }
35127             { rgb } { 3 }

```

```

35128     }
35129   }
35130   {#1} {#2} {#3}
35131 }

```

As we always need to split the alternative values into parts, we use a shared auxiliary and only use a minimal difference between code paths. Construction of the tint transformation is as far as possible done using loops, which means there are some inefficiencies for device colors in the DeviceN space: we roll the stack one-at-a-time even if there is a potential shortcut. However, that way there is nothing to special-case. Once this is sorted, we can write the tint transform object, which will remain as the last object until we sort out the final step: the colorant list.

```

35132 \cs_new_protected:Npn \__color_model_devicen_init:nnnn #1#2#3#4
35133 {
35134   \tl_set:Nx \l__color_internal_tl
35135     { \prg_replicate:nn {#1} { 1.0 ~ } }
35136   \int_zero:N \l__color_internal_int
35137   \clist_map_inline:nn {#4}
35138     {
35139       \int_incr:N \l__color_internal_int
35140       \prop_get:NnN \g__color_alternative_values_prop {##1}
35141         \l__color_value_tl
35142       \exp_after:wN \__color_model_devicen_transform:w
35143         \l__color_value_tl , 0 , 0 , 0 , \s__color_stop {#1} {#2}
35144     }
35145   \tl_put_right:Nx \l__color_internal_tl
35146     {
35147       \prg_replicate:nn {#1}
35148         { neg ~ 1.0 ~ add ~ #1 ~ -1 ~ roll ~ }
35149       \int_eval:n { #2 + #1 } ~ #1 ~ roll
35150       \prg_replicate:nn {#2} { ~ pop } ~
35151       #1 ~ 1 ~ roll
35152     }
35153   \use:x
35154     {
35155       \__color_backend_devicen_init:nnn
35156       {
35157         \clist_map_function:nN {#4}
35158           \__color_model_devicen_colorant:n
35159       }
35160       {
35161         \str_case:nn {#3}
35162           {
35163             { cmyk } { /DeviceCMYK }
35164             { gray } { /DeviceGray }
35165             { rgb } { /DeviceRGB }
35166           }
35167       }
35168       { \exp_not:V \l__color_internal_tl }
35169     }
35170 }
35171 \cs_new_protected:Npn \__color_model_devicen_transform:w
35172   #1 , #2 , #3 , #4 , #5 \s__color_stop #6#7
35173 {

```

```

35174 \use:c { __color_model_devicen_transform_ #6 :nnnnn }
35175 { #1 } { #2 } { #3 } { #4 } { #7 }
35176 }
35177 \cs_new_protected:cpn { __color_model_devicen_transform_1:nnnnn } #1#2#3#4#5
35178 { \__color_model_devicen_transform:nnn { #5 } { 1 } { #1 } }
35179 \cs_new_protected:cpn { __color_model_devicen_transform_3:nnnnn } #1#2#3#4#5
35180 {
35181 \clist_map_inline:nn { #1 , #2 , #3 }
35182 { \__color_model_devicen_transform:nnn { #5 } { 3 } { ##1 } }
35183 }
35184 \cs_new_protected:cpn { __color_model_devicen_transform_4:nnnnn } #1#2#3#4#5
35185 {
35186 \clist_map_inline:nn { #1 , #2 , #3 , #4 }
35187 { \__color_model_devicen_transform:nnn { #5 } { 4 } { ##1 } }
35188 }
35189 \cs_new_protected:Npn \__color_model_devicen_transform:nnn #1#2#3
35190 {
35191 \tl_put_right:Nx \l__color_internal_tl
35192 {
35193 \fp_compare:nNnF { #3 } = \c_zero_fp
35194 {
35195 \int_eval:n { #1 - \l__color_internal_int + #2 } ~ index ~
35196 -#3 ~ mul ~ 1.0 ~ add ~ mul ~
35197 }
35198 #2 ~ -1 ~ roll ~
35199 }
35200 }
35201 \cs_new:Npn \__color_model_devicen_colorant:n #1
35202 {
35203 / \prop_item:Nn \g__color_colorants_prop { #1 } ~
35204 }

```

Here we need to set up conversion from the DeviceN space to the alternative at the \TeX level. This also means supplying methods for inter-converting to other parameter-based spaces. Essentially the approach is exactly the same as the PostScript, just expressed in \TeX terms.

```

35205 \cs_new_protected:Npn \__color_model_devicen_convert:nnnn #1#2#3
35206 {
35207 \use:c { __color_model_devicen_convert_ #2 :nnn } { #1 } { #3 }
35208 }
35209 \cs_generate_variant:Nn \__color_model_devicen_convert:nnnn { nnnx }
35210 \cs_new_protected:Npn \__color_model_devicen_convert_cmyk:nnn #1#2
35211 {
35212 \tl_const:cn { c__color_fallback_ #1 _tl } { cmyk }
35213 \__color_model_devicen_convert:nnnn { #1 } { cmyk } { 4 } { #2 }
35214 }
35215 \cs_new_protected:Npn \__color_model_devicen_convert_gray:nnn #1#2
35216 {
35217 \tl_const:cn { c__color_fallback_ #1 _tl } { gray }
35218 \__color_model_devicen_convert:nnnn { #1 } { gray } { 1 } { #2 }
35219 }
35220 \cs_new_protected:Npn \__color_model_devicen_convert_rgb:nnn #1#2
35221 {
35222 \tl_const:cn { c__color_fallback_ #1 _tl } { rgb }

```

```

35223     \_color_model_devicen_convert:nnnnn {#1} { rgb } { 3 } {#2}
35224   }
35225 \cs_new_protected:Npn \_color_model_devicen_convert:nnnnn #1#2#3#4#5
35226 {
35227   \cs_new:cpn { \_color_convert_ #2 _ #1 :w } ##1 \s_color_stop {#5}
35228   \cs_new:cpx { \_color_convert_ #1 _ #2 :w } ##1 \s_color_stop
35229   {
35230     \exp_not:c { \_color_convert_devicen_ #2 : \prg_replicate:nn {#3} { n } w }
35231     \prg_replicate:nn {#3} { { 1 } }
35232     ##1 ~ \exp_not:N \s_color_mark
35233     \clist_map_function:nN {#4} \_color_model_devicen_convert:n
35234     {}
35235     \exp_not:N \s_color_stop
35236   }
35237 }
35238 \cs_new:Npn \_color_model_devicen_convert:n #1
35239 {
35240   {
35241     \exp_args:Ne \_color_model_devicen_convert_aux:n
35242     { \prop_item:Nn \g_color_alternative_values_prop {#1} }
35243   }
35244 }
35245 \cs_new:Npn \_color_model_devicen_convert_aux:n #1
35246 { \_color_model_devicen_convert_aux:w #1 , , , , \s_color_stop }
35247 \cs_new:Npn \_color_model_devicen_convert_aux:w #1 , #2 , #3 , #4 , #5 \s_color_stop
35248 {
35249   {#1}
35250   \tl_if_blank:nF {#2}
35251   {
35252     {#2}
35253     \tl_if_blank:nF {#3}
35254     {
35255       {#3}
35256       \tl_if_blank:nF {#4} { {#4} }
35257     }
35258   }
35259 }
35260 \cs_new:Npn \_color_convert_devicen_cmyk:nnnnw
35261 #1#2#3#4#5 ~ #6 \s_color_mark #7#8 \s_color_stop
35262 {
35263   \_color_convert_devicen_cmyk:nnnnnnnn {#5} {#1} {#2} {#3} {#4} #7
35264   #6 \s_color_mark #8 \s_color_stop
35265 }
35266 \cs_new:Npn \_color_convert_devicen_cmyk:nnnnnnnn #1#2#3#4#5#6#7#8#9
35267 {
35268   \use:e
35269   {
35270     \exp_not:N \_color_convert_devicen_cmyk_aux:nnnnw
35271     { \fp_eval:n { #2 * (1 - (#1 * #6)) } }
35272     { \fp_eval:n { #3 * (1 - (#1 * #7)) } }
35273     { \fp_eval:n { #4 * (1 - (#1 * #8)) } }
35274     { \fp_eval:n { #5 * (1 - (#1 * #9)) } }
35275   }
35276 }

```

```

35277 \cs_new:Npn \__color_convert_devicen_cmyk_aux:nnnnw
35278   #1#2#3#4 #5 \s__color_mark #6 \s__color_stop
35279   {
35280     \tl_if_blank:nTF {#5}
35281     {
35282       \fp_eval:n { 1 - #1 } ~
35283       \fp_eval:n { 1 - #2 } ~
35284       \fp_eval:n { 1 - #3 } ~
35285       \fp_eval:n { 1 - #4 }
35286     }
35287     {
35288       \__color_convert_devicen_cmyk:nnnnw {#1} {#2} {#3} {#4}
35289       #5 \s__color_mark #6 \s__color_stop
35290     }
35291   }
35292 \cs_new:Npn \__color_convert_devicen_gray:nw
35293   #1#2 ~ #3 \s__color_mark #4#5 \s__color_stop
35294   {
35295     \__color_convert_devicen_gray:nnn {#2} {#1} #4
35296     #3 \s__color_mark #5 \s__color_stop
35297   }
35298 \cs_new:Npn \__color_convert_devicen_gray:nnn #1#2#3
35299   {
35300     \exp_args:Ne \__color_convert_devicen_gray_aux:nw
35301     { \fp_eval:n { #2 * (1 - (#1 * #3)) } }
35302   }
35303 \cs_new:Npn \__color_convert_devicen_gray_aux:nw
35304   #1 #2 \s__color_mark #3 \s__color_stop
35305   {
35306     \tl_if_blank:nTF {#2}
35307     { \fp_eval:n { 1 - #1 } }
35308     {
35309       \__color_convert_devicen_gray:nw {#1}
35310       #2 \s__color_mark #3 \s__color_stop
35311     }
35312   }
35313 \cs_new:Npn \__color_convert_devicen_rgb:nnnw
35314   #1#2#3#4 ~ #5 \s__color_mark #6#7 \s__color_stop
35315   {
35316     \__color_convert_devicen_rgb:nnnnnnn {#4} {#1} {#2} {#3} #6
35317     #5 \s__color_mark #7 \s__color_stop
35318   }
35319 \cs_new:Npn \__color_convert_devicen_rgb:nnnnnnn #1#2#3#4#5#6#7
35320   {
35321     \use:e
35322     {
35323       \exp_not:N \__color_convert_devicen_rgb_aux:nnnw
35324       { \fp_eval:n { #2 * (1 - (#1 * #5)) } }
35325       { \fp_eval:n { #3 * (1 - (#1 * #6)) } }
35326       { \fp_eval:n { #4 * (1 - (#1 * #7)) } }
35327     }
35328   }
35329 \cs_new:Npn \__color_convert_devicen_rgb_aux:nnnw
35330   #1#2#3 #4 \s__color_mark #5 \s__color_stop

```

```

35331 {
35332   \tl_if_blank:nTF {#4}
35333   {
35334     \fp_eval:n { 1 - #1 } ~
35335     \fp_eval:n { 1 - #2 } ~
35336     \fp_eval:n { 1 - #3 }
35337   }
35338   {
35339     \__color_convert_devicen_rgb:nnnw {#1} {#2} {#3}
35340     #4 \s__color_mark #5 \s__color_stop
35341   }
35342 }

```

(End definition for __color_model_devicen:n and others.)

\c__color_icc_colorspace_signatures_prop

The signatures in the ICC file header indicating the underlying colorspace. We map it to three values: The number of components, the values corresponding to white, and the range.

```

35343 \prop_const_from_keyval:Nn \c__color_icc_colorspace_signatures_prop
35344 {
35345   % Gray
35346   47524159 = {1} {1} {0} {},
35347   % RGB
35348   52474220 = {3} {0~0~0} {1~1~1} {},
35349   % CMYK
35350   434D594B = {4} {0~0~0~1} {0~0~0~0} {},
35351   % Lab
35352   4C616220 = {3} {0~0~0} {100~0~0} {0~100~-128~127~-128~127}
35353 }

```

(End definition for \c__color_icc_colorspace_signatures_prop.)

__color_model_iccbased:n
 __color_model_iccbased:nn
 __color_model_iccbased:nnn
 __color_model_iccbased_aux:nnn

For an ICC profile, we need a file name and a number of components. The file name is processed here so the backend can treat it as a string.

```

35354 \cs_new_protected:Npn \__color_model_iccbased:n #1
35355 {
35356   \prop_get:NnNTF \l__color_internal_prop { file }
35357   \l__color_internal_tl
35358   {
35359     \exp_args:NV \__color_model_iccbased:nn
35360     \l__color_internal_tl {#1}
35361   }
35362   {
35363     \msg_error:nnn { color }
35364     { ICCBased-requires-file } {#1}
35365   }
35366 }
35367 \cs_new_protected:Npn \__color_model_iccbased:nn #1#2
35368 {
35369   \exp_args:NNx \prop_get:NnNTF \c__color_icc_colorspace_signatures_prop
35370   { \file_hex_dump:nnn { #1 } { 17 } { 20 } } \l__color_internal_tl
35371   {
35372     \exp_last_unbraced:NV \__color_model_iccbased_aux:nnnnnn
35373     \l__color_internal_tl { #2 } { #1 }

```



```

35374     }
35375     {
35376         \msg_error:nnn { color }
35377         { ICCBased-unsupported-colorspace } {#2}
35378     }
35379 }

```

Here, we can use the same internals as for DeviceN approach as we know the number of components. No conversion is possible, so there is no need to worry about that at all.

```

35380 \cs_new_protected:Npn \__color_model_iccbased_aux:nnnnnn #1#2#3#4#5#6
35381 {
35382     \__color_model_init:nnnn {#5} {#1} { iccbased } {#3}
35383     \tl_const:cn { c__color_fallback_ #5 _tl } { gray }
35384     \cs_new:cpn { __color_convert_ #5 _gray:w } ##1 \s__color_stop { 0 }
35385     \cs_new:cpn { __color_convert_gray_ #5 :w } ##1 \s__color_stop { #2 }
35386     \use:c { __color_model_devicen_parse_ #1 :nn } {#5} {#1}
35387     \exp_args:Nx \__color_backend_iccbased_init:nnn
35388         { \file_full_name:n {#6} } {#1} {#4}
35389 }

```

(End definition for __color_model_iccbased:n and others.)

86.13 Applying profiles

With a limited range of outcomes, this is largely about getting data to the backend.

```

\color_profile_apply:nn
\__color_profile_apply:nn
    \__color_profile_apply_gray:n
\__color_profile_apply_rgb:n
    \__color_profile_apply_cmyk:n

35390 \cs_new_protected:Npn \color_profile_apply:nn #1#2
35391 {
35392     \exp_args:Ne \__color_profile_apply:nn
35393         { \file_full_name:n {#1} } {#2}
35394 }
35395 \cs_new_protected:Npn \__color_profile_apply:nn #1#2
35396 {
35397     \cs_if_exist_use:cF { __color_profile_apply_ \tl_to_str:n {#2} :n }
35398     {
35399         \msg_error:nnn { color } { ICC-Device-unknown } {#2}
35400         \use_none:n
35401     }
35402     {#1}
35403 }
35404 \cs_new_protected:Npn \__color_profile_apply_gray:n #1
35405 {
35406     \int_gincr:N \g__color_model_int
35407     \__color_backend_iccbased_device:nnn {#1} { Gray } { 1 }
35408 }
35409 \cs_new_protected:Npn \__color_profile_apply_rgb:n #1
35410 {
35411     \int_gincr:N \g__color_model_int
35412     \__color_backend_iccbased_device:nnn {#1} { RGB } { 3 }
35413 }
35414 \cs_new_protected:Npn \__color_profile_apply_cmyk:n #1
35415 {
35416     \int_gincr:N \g__color_model_int
35417     \__color_backend_iccbased_device:nnn {#1} { CMYK } { 4 }
35418 }

```

(End definition for `\color_profile_apply:n` and others. This function is documented on page 294.)

86.14 Diagnostics

```

\color_show:n Extract the information about a color and format for the user: the approach is similar
\color_log:n to the keys module here.
\__color_show:Nn 35419 \cs_new_protected:Npn \color_show:n
\__color_show:n 35420 { \__color_show:Nn \msg_show:nnxxxx }
35421 \cs_new_protected:Npn \color_log:n
35422 { \__color_show:Nn \msg_log:nnxxxx }
35423 \cs_new_protected:Npn \__color_show:Nn #1#2
35424 {
35425   #1 { color } { show }
35426   {#2}
35427   {
35428     \__color_if_defined:nT {#2}
35429     {
35430       \exp_args:Nv \__color_show:n { l__color_named_ #2 _tl }
35431       \prop_map_function:cN
35432       { l__color_named_ #2 _prop }
35433       \msg_show_item_unbraced:nn
35434     }
35435   }
35436   { }
35437   { }
35438 }
35439 \cs_new:Npn \__color_show:n #1
35440 {
35441   \msg_show_item_unbraced:nn { model } {#1}
35442 }

```

(End definition for `\color_show:n` and others. These functions are documented on page 291.)

86.15 Messages

```

35443 \msg_new:nnnn { color } { CIELAB-requires-illuminant }
35444 { CIELAB~color~space~'#1'~require~an~illuminant. }
35445 {
35446   LaTeX~has~been~asked~to~create~a~separation~color~space~using~
35447   CIELAB~specifications,~but~no~\\ \\
35448   \iow_indent:n { illuminant~=<basis> }
35449   \\ \\
35450   key~was~given~with~the~correct~information.~LaTeX~will~use~illuminant~
35451   'd50'~for~recovery.
35452 }
35453 \msg_new:nnnn { color } { conversion-not-available }
35454 { No~model~conversion~available~from~'#1'~to~'#2'. }
35455 {
35456   LaTeX~has~been~asked~to~convert~a~color~from~model~'#1'~
35457   to~model~'#2',~but~there~is~no~method~available~to~do~that.
35458 }
35459 \msg_new:nnnn { color } { DeviceN-inconsistent-alternative }

```

```

35460 { DeviceN~color~spaces~require~a~single~alternative~space. }
35461 {
35462   LaTeX~has~been~asked~to~create~a~DeviceN~color~space~'#1',~
35463   but~the~constituent~colors~do~not~have~a~common~alternative~
35464   color.
35465 }
35466 \msg_new:nnnn { color } { DeviceN-no-alternative }
35467 { DeviceN~color~spaces~require~an~alternative~space. }
35468 {
35469   LaTeX~has~been~asked~to~create~a~DeviceN~color~space~'#1',~
35470   but~the~constituent~colors~do~not~all~have~a~device~based~alternative.
35471 }
35472 \msg_new:nnnn { color } { DeviceN-requires-names }
35473 { DeviceN~color~space~'#1'~require~a~list~of~names. }
35474 {
35475   LaTeX~has~been~asked~to~create~a~DeviceN~color~space,~
35476   but~no~\\ \\
35477   \iow_indent:n { names~=<names> }
35478   \\ \\
35479   key~was~given~with~the~correct~information.
35480 }
35481 \msg_new:nnnn { color } { ICC-Device-unknown }
35482 { Unknown~device~color~space~'#1'. }
35483 {
35484   LaTeX~has~been~asked~to~apply~an~ICC~profile~but~the~device~color~space~
35485   '#1'~is~unknown.
35486 }
35487 \msg_new:nnnn { color } { ICCBased-unsupported-colorspace }
35488 { ICCBased~color~space~'#1'~uses~an~unsupported~data~color~space. }
35489 {
35490   LaTeX~has~been~asked~to~create~a~ICCBased~colorspace,~but~the~
35491   used~data~colorspace~is~not~supported.~ICC~profiles~used~for~
35492   defining~a~ICCBased~colorspace~should~use~a~Lab,~RGB,~or~
35493   CMYK~data~colorspace.~LaTeX~will~ignore~this~request.
35494 }
35495 \msg_new:nnnn { color } { ICCBased-requires-file }
35496 { ICCBased~color~space~'#1'~require~an~file. }
35497 {
35498   LaTeX~has~been~asked~to~create~an~ICCBased~color~space,~but~no~\\ \\
35499   \iow_indent:n { file~=<name> }
35500   \\ \\
35501   key~was~given~with~the~correct~information.~LaTeX~will~ignore~this~
35502   request.
35503 }
35504 \msg_new:nnnn { color } { model-already-defined }
35505 { Color~model~'#1'~already~defined. }
35506 {
35507   LaTeX~was~asked~to~define~a~new~color~model~called~'#1',~but~
35508   this~color~model~already~exists.
35509 }
35510 \msg_new:nnnn { color } { separation-alternative-model }
35511 { Separation~color~space~'#1'~require~an~alternative~model. }
35512 {
35513   LaTeX~has~been~asked~to~create~a~separation~color~space,~

```

```

35514     but-no~\\ \\
35515     \iow_indent:n { alternative-model~=<model> }
35516     \\ \\
35517     key-was-given-with-the-correct-information.
35518 }
35519 \msg_new:nnnn { color } { separation-alternative-values }
35520 { Separation~color~space~'#1'~require-values-for-the-alternative-space. }
35521 {
35522     LaTeX-has-been-asked-to-create-a-separation-color-space,~
35523     but-no~\\ \\
35524     \iow_indent:n { alternative-values~=<model> }
35525     \\ \\
35526     key-was-given-with-the-correct-information.
35527 }
35528 \msg_new:nnnn { color } { separation-requires-name }
35529 { Separation~color~space~'#1'~require-a-formal-name. }
35530 {
35531     LaTeX-has-been-asked-to-create-a-separation-color-space,~
35532     but-no~\\ \\
35533     \iow_indent:n { name~=<formal-name> }
35534     \\ \\
35535     key-was-given-with-the-correct-information.
35536 }
35537 \msg_new:nnnn { color } { unknown-color }
35538 { Unknown~color~'#1'. }
35539 {
35540     LaTeX-has-been-asked-to-use-a-color-named~'#1',~
35541     but-this-has-never-been-defined.
35542 }
35543 \msg_new:nnnn { color } { unknown-alternative-model }
35544 { Separation~color~space~'#1'~require-an-valid-alternative-space. }
35545 {
35546     LaTeX-has-been-asked-to-create-a-separation-color-space,~
35547     but-the-model-given-as\\ \\
35548     \iow_indent:n { alternative-model~=<model> }
35549     \\ \\
35550     is-unknown.
35551 }
35552 \msg_new:nnnn { color } { unknown-export-format }
35553 { Unknown~export~format~'#1'. }
35554 {
35555     LaTeX-has-been-asked-to-export-a-color-in-format~'#1',~
35556     but-this-has-never-been-defined.
35557 }
35558 \msg_new:nnnn { color } { unknown-CIELAB-illuminant }
35559 { Unknown~illuminant~model~'#1'. }
35560 {
35561     LaTeX-has-been-asked-to-use-create-a-color-space-using-CIELAB-
35562     illuminant~'#1',~but-this-does-not-exist.
35563 }
35564 \msg_new:nnnn { color } { unknown-model }
35565 { Unknown~color~model~'#1'. }
35566 {
35567     LaTeX-has-been-asked-to-use-a-color-model-called~'#1',~

```

```

35568     but~this~model~is~not~set~up.
35569   }
35570   \msg_new:nnnn { color } { unknown-model-type }
35571   { Unknown~color~model~type~'#1'. }
35572   {
35573     LaTeX~has~been~asked~to~create~a~new~color~model~called~'#1',~
35574     but~this~type~of~model~was~never~set~up.
35575   }
35576   \prop_gput:Nnn \g_msg_module_name_prop { color } { LaTeX3 }
35577   \prop_gput:Nnn \g_msg_module_type_prop { color } { }
35578   \msg_new:nnn { color } { show }
35579   {
35580     The~color~#1~
35581     \tl_if_empty:nTF {#2}
35582       { is~undefined. }
35583       { has~the~properties: #2 }
35584   }
35585 \end{package}

```

Chapter 87

l3pdf implementation

```
35586 (*package)
35587 (@@=pdf)

\s__pdf_stop Internal scan marks.
35588 \scan_new:N \s__pdf_stop

(End definition for \s__pdf_stop.)

\g__pdf_init_bool A flag so we have some chance of avoiding setting things we are not allowed to. As we
are potentially early in the format, we have to work a bit harder than ideal.
35589 \bool_new:N \g__pdf_init_bool
35590 \bool_lazy_and:nnT
35591 { \str_if_eq_p:Vn \fmtname { LaTeX2e } }
35592 { \tl_if_exist_p:N \@expl@finalise@setup@@ }
35593 {
35594   \tl_gput_right:Nn \@expl@finalise@setup@@
35595   {
35596     \tl_gput_right:Nn \@kernel@after@begindocument
35597     { \bool_gset_true:N \g__pdf_init_bool }
35598   }
35599 }
```

(End definition for \g__pdf_init_bool.)

87.1 Compression

\pdf_uncompress: Simple to do.

```
35600 \cs_new_protected:Npn \pdf_uncompress:
35601 {
35602   \bool_if:NF \g__pdf_init_bool
35603   {
35604     \__pdf_backend_compresslevel:n { 0 }
35605     \__pdf_backend_compress_objects:n { \c_false_bool }
35606   }
35607 }
```

(End definition for \pdf_uncompress:. This function is documented on page [297](#).)

87.2 Objects

Simple to do.

```

\pdf_object_new:nn
\pdf_object_write:nn
\pdf_object_write:nx
\pdf_object_if_exist_p:n
\pdf_object_if_exist:nTF
\pdf_object_ref:n
\pdf_object_unnamed_write:nn
\pdf_object_unnamed_write:nx
\pdf_object_ref_last:

35608 \cs_new_protected:Npn \pdf_object_new:nn #1#2
35609 { \__pdf_backend_object_new:nn {#1} {#2} }
35610 \prg_new_conditional:Npnn \pdf_object_if_exist:n #1 { p , T , F , TF }
35611 {
35612   \int_if_exist:cTF { c__pdf_backend_object_ \tl_to_str:n {#1} _int }
35613   { \prg_return_true: }
35614   { \prg_return_false: }
35615 }
35616 \cs_new_protected:Npn \pdf_object_write:nn #1#2
35617 {
35618   \__pdf_backend_object_write:nn {#1} {#2}
35619   \bool_gset_true:N \g__pdf_init_bool
35620 }
35621 \cs_generate_variant:Nn \pdf_object_write:nn { nx }
35622 \cs_new:Npn \pdf_object_ref:n #1 { \__pdf_backend_object_ref:n {#1} }
35623 \cs_new_protected:Npn \pdf_object_unnamed_write:nn #1#2
35624 {
35625   \__pdf_backend_object_now:nn {#1} {#2}
35626   \bool_gset_true:N \g__pdf_init_bool
35627 }
35628 \cs_generate_variant:Nn \pdf_object_unnamed_write:nn { nx }
35629 \cs_new:Npn \pdf_object_ref_last: { \__pdf_backend_object_last: }

```

(End definition for `\pdf_object_new:nn` and others. These functions are documented on page 295.)

`\pdf_pageobject_ref:n`

```

35630 \cs_new:Npn \pdf_pageobject_ref:n #1
35631 { \__pdf_backend_pageobject_ref:n {#1} }

```

(End definition for `\pdf_pageobject_ref:n`. This function is documented on page 296.)

87.3 Version

To compare version, we need to split the given value then deal with both major and minor version

```

\pdf_version_compare:Nn
__pdf_version_compare_=:w
__pdf_version_compare_<:w
__pdf_version_compare_>:w

35632 \prg_new_conditional:Npnn \pdf_version_compare:Nn #1#2 { p , T , F , TF }
35633 { \use:c { __pdf_version_compare_ #1 :w } #2 . . \s__pdf_stop }
35634 \cs_new:cpn { __pdf_version_compare_=:w } #1 . #2 . #3 \s__pdf_stop
35635 {
35636   \bool_lazy_and:nnTF
35637   { \int_compare_p:nNn \__pdf_backend_version_major: = {#1} }
35638   { \int_compare_p:nNn \__pdf_backend_version_minor: = {#2} }
35639   { \prg_return_true: }
35640   { \prg_return_false: }
35641 }
35642 \cs_new:cpn { __pdf_version_compare_<:w } #1 . #2 . #3 \s__pdf_stop
35643 {
35644   \bool_lazy_or:nnTF
35645   { \int_compare_p:nNn \__pdf_backend_version_major: < {#1} }
35646   {

```

```

35647     \bool_lazy_and_p:nn
35648     { \int_compare_p:nNn \__pdf_backend_version_major: = {#1} }
35649     { \int_compare_p:nNn \__pdf_backend_version_minor: < {#2} }
35650   }
35651   { \prg_return_true: }
35652   { \prg_return_false: }
35653 }
35654 \cs_new:cpn { __pdf_version_compare_>:w } #1 . #2 . #3 \s__pdf_stop
35655 {
35656   \bool_lazy_or:nnTF
35657   { \int_compare_p:nNn \__pdf_backend_version_major: > {#1} }
35658   {
35659     \bool_lazy_and_p:nn
35660     { \int_compare_p:nNn \__pdf_backend_version_major: = {#1} }
35661     { \int_compare_p:nNn \__pdf_backend_version_minor: > {#2} }
35662   }
35663   { \prg_return_true: }
35664   { \prg_return_false: }
35665 }

```

(End definition for \pdf_version_compare:Nn and others. This function is documented on page ??.)

```

\pdf_version_gset:n Split the version and set.
\pdf_version_min_gset:n
\__pdf_version_gset:w
35666 \cs_new_protected:Npn \pdf_version_gset:n #1
35667 { \__pdf_version_gset:w #1 . . \s__pdf_stop }
35668 \cs_new_protected:Npn \pdf_version_min_gset:n #1
35669 {
35670   \pdf_version_compare:NnT < {#1}
35671   { \__pdf_version_gset:w #1 . . \s__pdf_stop }
35672 }
35673 \cs_new_protected:Npn \__pdf_version_gset:w #1 . #2 . #3 \s__pdf_stop
35674 {
35675   \bool_if:NF \g__pdf_init_bool
35676   {
35677     \__pdf_backend_version_major_gset:n {#1}
35678     \__pdf_backend_version_minor_gset:n {#2}
35679   }
35680 }

```

(End definition for \pdf_version_gset:n, \pdf_version_min_gset:n, and __pdf_version_gset:w. These functions are documented on page 296.)

```

\pdf_version: Wrappers.
\pdf_version_major:
\pdf_version_minor:
35681 \cs_new:Npn \pdf_version:
35682 { \__pdf_backend_version_major: . \__pdf_backend_version_minor: }
35683 \cs_new:Npn \pdf_version_major: { \__pdf_backend_version_major: }
35684 \cs_new:Npn \pdf_version_minor: { \__pdf_backend_version_minor: }

```

(End definition for \pdf_version:, \pdf_version_major:, and \pdf_version_minor:. These functions are documented on page 296.)

87.4 Destinations

`\pdf_destination:nn`

```
35685 \cs_new_protected:Npn \pdf_destination:nn #1#2
35686 { \__pdf_backend_destination:nn {#1} {#2} }
```

(End definition for \pdf_destination:nn. This function is documented on page 297.)

`\pdf_destination:nnnn`

```
35687 \cs_new_protected:Npn \pdf_destination:nnnn #1#2#3#4
35688 {
35689   \hbox_to_zero:n
35690   { \__pdf_backend_destination:nnnn {#1} {#2} {#3} {#4} }
35691 }
```

(End definition for \pdf_destination:nnnn. This function is documented on page 298.)

```
35692 \endpackage
```

Chapter 88

l3candidates Implementation

35693 $\langle *package \rangle$

88.1 Additions to l3box

35694 $\langle @@=box \rangle$

88.1.1 Viewing part of a box

```
\box_clip:N A wrapper around the driver-dependent code.
\box_clip:c
\box_gclip:N
\box_gclip:c
35695 \cs_new_protected:Npn \box_clip:N #1
35696 { \hbox_set:Nn #1 { \__box_backend_clip:N #1 } }
35697 \cs_generate_variant:Nn \box_clip:N { c }
35698 \cs_new_protected:Npn \box_gclip:N #1
35699 { \hbox_gset:Nn #1 { \__box_backend_clip:N #1 } }
35700 \cs_generate_variant:Nn \box_gclip:N { c }
```

(End definition for `\box_clip:N` and `\box_gclip:N`. These functions are documented on page 301.)

```
\box_set_trim:Nnnnn Trimming from the left- and right-hand edges of the box is easy: kern the appropriate
\box_set_trim:cnnnn parts off each side.
\box_gset_trim:Nnnnn
\box_gset_trim:cnnnn
\__box_set_trim:NnnnnN
35701 \cs_new_protected:Npn \box_set_trim:Nnnnn #1#2#3#4#5
35702 { \__box_set_trim:NnnnnN #1 {#2} {#3} {#4} {#5} \box_set_eq:NN }
35703 \cs_generate_variant:Nn \box_set_trim:Nnnnn { c }
35704 \cs_new_protected:Npn \box_gset_trim:Nnnnn #1#2#3#4#5
35705 { \__box_set_trim:NnnnnN #1 {#2} {#3} {#4} {#5} \box_gset_eq:NN }
35706 \cs_generate_variant:Nn \box_gset_trim:Nnnnn { c }
35707 \cs_new_protected:Npn \__box_set_trim:NnnnnN #1#2#3#4#5#6
35708 {
35709   \hbox_set:Nn \l__box_internal_box
35710   {
35711     \__kernel_kern:n { -#2 }
35712     \box_use:N #1
35713     \__kernel_kern:n { -#4 }
35714   }
```

For the height and depth, there is a need to watch the baseline is respected. Material always has to stay on the correct side, so trimming has to check that there is enough material to trim. First, the bottom edge. If there is enough depth, simply set the depth, or if not move down so the result is zero depth. `\box_move_down:nn` is used in both

cases so the resulting box always contains a \lower primitive. The internal box is used here as it allows safe use of \box_set_dp:Nn.

```

35715 \dim_compare:nNnTF { \box_dp:N #1 } > {#3}
35716 {
35717   \hbox_set:Nn \l__box_internal_box
35718   {
35719     \box_move_down:nn \c_zero_dim
35720     { \box_use_drop:N \l__box_internal_box }
35721   }
35722   \box_set_dp:Nn \l__box_internal_box { \box_dp:N #1 - (#3) }
35723 }
35724 {
35725   \hbox_set:Nn \l__box_internal_box
35726   {
35727     \box_move_down:nn { (#3) - \box_dp:N #1 }
35728     { \box_use_drop:N \l__box_internal_box }
35729   }
35730   \box_set_dp:Nn \l__box_internal_box \c_zero_dim
35731 }

```

Same thing, this time from the top of the box.

```

35732 \dim_compare:nNnTF { \box_ht:N \l__box_internal_box } > {#5}
35733 {
35734   \hbox_set:Nn \l__box_internal_box
35735   {
35736     \box_move_up:nn \c_zero_dim
35737     { \box_use_drop:N \l__box_internal_box }
35738   }
35739   \box_set_ht:Nn \l__box_internal_box
35740   { \box_ht:N \l__box_internal_box - (#5) }
35741 }
35742 {
35743   \hbox_set:Nn \l__box_internal_box
35744   {
35745     \box_move_up:nn { (#5) - \box_ht:N \l__box_internal_box }
35746     { \box_use_drop:N \l__box_internal_box }
35747   }
35748   \box_set_ht:Nn \l__box_internal_box \c_zero_dim
35749 }
35750 #6 #1 \l__box_internal_box
35751 }

```

(End definition for \box_set_trim:Nnnnn, \box_gset_trim:Nnnnn, and __box_set_trim:NnnnnN. These functions are documented on page 301.)

\box_set_viewport:Nnnnn
\box_set_viewport:cnnnn
\box_gset_viewport:Nnnnn
\box_gset_viewport:cnnnn
__box_viewport:NnnnnN

The same general logic as for the trim operation, but with absolute dimensions. As a result, there are some things to watch out for in the vertical direction.

```

35752 \cs_new_protected:Npn \box_set_viewport:Nnnnn #1#2#3#4#5
35753 { \__box_set_viewport:NnnnnN #1 {#2} {#3} {#4} {#5} \box_set_eq:NN }
35754 \cs_generate_variant:Nn \box_set_viewport:Nnnnn { c }
35755 \cs_new_protected:Npn \box_gset_viewport:Nnnnn #1#2#3#4#5
35756 { \__box_set_viewport:NnnnnN #1 {#2} {#3} {#4} {#5} \box_gset_eq:NN }
35757 \cs_generate_variant:Nn \box_gset_viewport:Nnnnn { c }
35758 \cs_new_protected:Npn \__box_set_viewport:NnnnnN #1#2#3#4#5#6

```

```

35759 {
35760   \hbox_set:Nn \l__box_internal_box
35761   {
35762     \__kernel_kern:n { -#2 }
35763     \box_use:N #1
35764     \__kernel_kern:n { #4 - \box_wd:N #1 }
35765   }
35766   \dim_compare:nNnTF {#3} < \c_zero_dim
35767   {
35768     \hbox_set:Nn \l__box_internal_box
35769     {
35770       \box_move_down:nn \c_zero_dim
35771       { \box_use_drop:N \l__box_internal_box }
35772     }
35773     \box_set_dp:Nn \l__box_internal_box { - \__box_dim_eval:n {#3} }
35774   }
35775   {
35776     \hbox_set:Nn \l__box_internal_box
35777     { \box_move_down:nn {#3} { \box_use_drop:N \l__box_internal_box } }
35778     \box_set_dp:Nn \l__box_internal_box \c_zero_dim
35779   }
35780   \dim_compare:nNnTF {#5} > \c_zero_dim
35781   {
35782     \hbox_set:Nn \l__box_internal_box
35783     {
35784       \box_move_up:nn \c_zero_dim
35785       { \box_use_drop:N \l__box_internal_box }
35786     }
35787     \box_set_ht:Nn \l__box_internal_box
35788     {
35789       (#5)
35790       \dim_compare:nNnT {#3} > \c_zero_dim
35791       { - (#3) }
35792     }
35793   }
35794   {
35795     \hbox_set:Nn \l__box_internal_box
35796     {
35797       \box_move_up:nn { - \__box_dim_eval:n {#5} }
35798       { \box_use_drop:N \l__box_internal_box }
35799     }
35800     \box_set_ht:Nn \l__box_internal_box \c_zero_dim
35801   }
35802   #6 #1 \l__box_internal_box
35803 }

```

(End definition for `\box_set_viewport:Nnnnn`, `\box_gset_viewport:Nnnnn`, and `__box_viewport:Nnnnn`.
These functions are documented on page [301](#).)

88.2 Additions to l3flag

```

35804 <@@=flag>

```

`\flag_raise_if_clear:n` It might be faster to just call the “trap” function in all cases but conceptually the function name suggests we should only run it if the flag is zero in case the “trap” made customizable in the future.

```

35805 \cs_new:Npn \flag_raise_if_clear:n #1
35806 {
35807   \if_cs_exist:w flag-#1-0 \cs_end:
35808   \else:
35809     \cs:w flag-#1 \cs_end: 0 ;
35810   \fi:
35811 }

```

(End definition for `\flag_raise_if_clear:n`. This function is documented on page 302.)

88.3 Additions to `l3msg`

```

35812 (@@=msg)

```

`\msg_show_eval:Nn` A short-hand used for `\int_show:n` and similar functions that passes to `\tl_show:n` the result of applying #1 (a function such as `\int_eval:n`) to the expression #2. The use of f-expansion ensures that #1 is expanded in the scope in which the show command is called, rather than in the group created by `\iow_wrap:nnnN`. This is only important for expressions involving the `\currentgrouplevel` or `\currentgrouptype`. On the other hand we want the expression to be converted to a string with the usual escape character, hence within the wrapping code.

```

35813 \cs_new_protected:Npn \msg_show_eval:Nn #1#2
35814 { \exp_args:Nf \_msg_show_eval:nnN { #1 {#2} } {#2} \tl_show:n }
35815 \cs_new_protected:Npn \msg_log_eval:Nn #1#2
35816 { \exp_args:Nf \_msg_show_eval:nnN { #1 {#2} } {#2} \tl_log:n }
35817 \cs_new_protected:Npn \_msg_show_eval:nnN #1#2#3 { #3 { #2 = #1 } }

```

(End definition for `\msg_show_eval:Nn`, `\msg_log_eval:Nn`, and `_msg_show_eval:nnN`. These functions are documented on page 303.)

`\msg_show_item:n` Each item in the variable is formatted using one of the following functions. We cannot use `\` and so on because these short-hands cannot be used inside the arguments of messages, only when defining the messages.

```

\msg_show_item_unbraced:n
\msg_show_item:nn
\msg_show_item_unbraced:nn
35818 \cs_new:Npx \msg_show_item:n #1
35819 { \iow_newline: > ~ \c_space_tl \exp_not:N \tl_to_str:n { {#1} } }
35820 \cs_new:Npx \msg_show_item_unbraced:n #1
35821 { \iow_newline: > ~ \c_space_tl \exp_not:N \tl_to_str:n {#1} }
35822 \cs_new:Npx \msg_show_item:nn #1#2
35823 {
35824   \iow_newline: > \use:nn { ~ } { ~ }
35825   \exp_not:N \tl_to_str:n { {#1} }
35826   \use:nn { ~ } { ~ } => \use:nn { ~ } { ~ }
35827   \exp_not:N \tl_to_str:n { {#2} }
35828 }
35829 \cs_new:Npx \msg_show_item_unbraced:nn #1#2
35830 {
35831   \iow_newline: > \use:nn { ~ } { ~ }
35832   \exp_not:N \tl_to_str:n {#1}
35833   \use:nn { ~ } { ~ } => \use:nn { ~ } { ~ }
35834   \exp_not:N \tl_to_str:n {#2}
35835 }

```

(End definition for \msg_show_item:n and others. These functions are documented on page 303.)

88.4 Additions to l3prg

35836 <@@=bool>

\bool_set_inverse:N Set to false or true locally or globally.
 \bool_set_inverse:c 35837 \cs_new_protected:Npn \bool_set_inverse:N #1
 \bool_gset_inverse:N 35838 { \bool_if:NTF #1 { \bool_set_false:N } { \bool_set_true:N } #1 }
 \bool_gset_inverse:c 35839 \cs_generate_variant:Nn \bool_set_inverse:N { c }
 35840 \cs_new_protected:Npn \bool_gset_inverse:N #1
 35841 { \bool_if:NTF #1 { \bool_gset_false:N } { \bool_gset_true:N } #1 }
 35842 \cs_generate_variant:Nn \bool_gset_inverse:N { c }

(End definition for \bool_set_inverse:N and \bool_gset_inverse:N. These functions are documented on page 303.)

\s__bool_mark Internal scan marks.
 \s__bool_stop 35843 \scan_new:N \s__bool_mark
 35844 \scan_new:N \s__bool_stop

(End definition for \s__bool_mark and \s__bool_stop.)

\bool_case_true:n For boolean cases the overall idea is the same as for \tl_case:nnTF as described in l3tl.
 \bool_case_true:nTF 35845 \cs_new:Npn \bool_case_true:nTF
 \bool_case_false:n 35846 { \exp:w __bool_case:NnTF \c_true_bool }
 \bool_case_false:nTF 35847 \cs_new:Npn \bool_case_true:nT #1#2
 __bool_case:NnTF 35848 { \exp:w __bool_case:NnTF \c_true_bool {#1} {#2} { } }
 __bool_case_true:w 35849 \cs_new:Npn \bool_case_true:nF #1
 __bool_case_false:w 35850 { \exp:w __bool_case:NnTF \c_true_bool {#1} { } }
 __bool_case_end:nw 35851 \cs_new:Npn \bool_case_true:n #1
 35852 { \exp:w __bool_case:NnTF \c_true_bool {#1} { } { } }
 35853 \cs_new:Npn \bool_case_false:nTF
 35854 { \exp:w __bool_case:NnTF \c_false_bool }
 35855 \cs_new:Npn \bool_case_false:nT #1#2
 35856 { \exp:w __bool_case:NnTF \c_false_bool {#1} {#2} { } }
 35857 \cs_new:Npn \bool_case_false:nF #1
 35858 { \exp:w __bool_case:NnTF \c_false_bool {#1} { } }
 35859 \cs_new:Npn \bool_case_false:n #1
 35860 { \exp:w __bool_case:NnTF \c_false_bool {#1} { } { } }
 35861 \cs_new:Npn __bool_case:NnTF #1#2#3#4
 35862 {
 35863 \bool_if:NTF #1 __bool_case_true:w __bool_case_false:w
 35864 #2 #1 { } \s__bool_mark {#3} \s__bool_mark {#4} \s__bool_stop
 35865 }
 35866 \cs_new:Npn __bool_case_true:w #1#2
 35867 {
 35868 \bool_if:NTF {#1}
 35869 { __bool_case_end:nw {#2} }
 35870 { __bool_case_true:w }
 35871 }
 35872 \cs_new:Npn __bool_case_false:w #1#2
 35873 {
 35874 \bool_if:NTF {#1}

```

35875         { \_bool\_case\_false:w }
35876         { \_bool\_case\_end:nw {#2} }
35877     }
35878 \cs\_new:Npn \_bool\_case\_end:nw #1#2#3 \s\_bool\_mark #4#5 \s\_bool\_stop
35879     { \exp\_end: #1 #4 }

```

(End definition for `\bool_case_true:nTF` and others. These functions are documented on page 304.)

88.5 Additions to `l3prop`

```

35880 <@@=prop>

```

`_prop_use_i_delimit_by_s_stop:nw` Functions to gobble up to a scan mark.

```

35881 \cs\_new:Npn \_prop\_use\_i\_delimit\_by\_s\_stop:nw #1 #2 \s\_prop\_stop {#1}

```

(End definition for `_prop_use_i_delimit_by_s_stop:nw`.)

```

\prop\_rand\_key\_value:N
\prop\_rand\_key\_value:c
\_prop\_rand\_item:w

```

Contrarily to `clist`, `seq` and `tl`, there is no function to get an item of a `prop` given an integer between 1 and the number of items, so we write the appropriate code. There is no bounds checking because `\int_rand:nn` is always within bounds. The initial `\int_value:w` is stopped by the first `\s_prop` in #1.

```

35882 \cs\_new:Npn \prop\_rand\_key\_value:N #1
35883 {
35884     \prop\_if\_empty:NF #1
35885     {
35886         \exp\_after:wN \_prop\_rand\_item:w
35887         \int\_value:w \int\_rand:nn { 1 } { \prop\_count:N #1 }
35888         #1 \s\_prop\_stop
35889     }
35890 }
35891 \cs\_generate\_variant:Nn \prop\_rand\_key\_value:N { c }
35892 \cs\_new:Npn \_prop\_rand\_item:w #1 \s\_prop \_prop\_pair:wn #2 \s\_prop #3
35893 {
35894     \int\_compare:nNnF {#1} > 1
35895     { \_prop\_use\_i\_delimit\_by\_s\_stop:nw { \exp\_not:n { {#2} {#3} } } }
35896     \exp\_after:wN \_prop\_rand\_item:w
35897     \int\_value:w \int\_eval:n { #1 - 1 } \s\_prop
35898 }

```

(End definition for `\prop_rand_key_value:N` and `_prop_rand_item:w`. This function is documented on page 304.)

88.6 Additions to `l3seq`

```

35899 <@@=seq>

```

```

\seq\_mapthread\_function:NNN
\seq\_mapthread\_function:NcN
\seq\_mapthread\_function:cNN
\seq\_mapthread\_function:ccN
\_seq\_mapthread\_function:wNN
\_seq\_mapthread\_function:wNw
\_seq\_mapthread\_function:Nnnwnn

```

The idea is to first expand both sequences, adding the usual `{ ? \prg_break: } { }` to the end of each one. This is most conveniently done in two steps using an auxiliary function. The mapping then throws away the first tokens of #2 and #5, which for items in both sequences are `\s_seq` `_seq_item:n`. The function to be mapped are then be applied to the two entries. When the code hits the end of one of the sequences, the break material stops the entire loop and tidy up. This avoids needing to find the count of the two sequences, or worrying about which is longer.

```

35900 \cs_new:Npn \seq_mapthread_function:NNN #1#2#3
35901 { \exp_after:wN \__seq_mapthread_function:wNN #2 \s__seq_stop #1 #3 }
35902 \cs_new:Npn \__seq_mapthread_function:wNN \s__seq #1 \s__seq_stop #2#3
35903 {
35904   \exp_after:wN \__seq_mapthread_function:wNw #2 \s__seq_stop #3
35905   #1 { ? \prg_break: } { }
35906   \prg_break_point:
35907 }
35908 \cs_new:Npn \__seq_mapthread_function:wNw \s__seq #1 \s__seq_stop #2
35909 {
35910   \__seq_mapthread_function:Nnnwnn #2
35911   #1 { ? \prg_break: } { }
35912   \s__seq_stop
35913 }
35914 \cs_new:Npn \__seq_mapthread_function:Nnnwnn #1#2#3#4 \s__seq_stop #5#6
35915 {
35916   \use_none:n #2
35917   \use_none:n #5
35918   #1 {#3} {#6}
35919   \__seq_mapthread_function:Nnnwnn #1 #4 \s__seq_stop
35920 }
35921 \cs_generate_variant:Nn \seq_mapthread_function:NNN { Nc , c , cc }

```

(End definition for `\seq_mapthread_function:NNN` and others. This function is documented on page 305.)

`\seq_set_filter:NNn` Similar to `\seq_map_inline:Nn`, without a `\prg_break_point:` because the user's code is performed within the evaluation of a boolean expression, and skipping out of that would break horribly. The `__seq_wrap_item:n` function inserts the relevant `__seq_item:n` without expansion in the input stream, hence in the x-expanding assignment.

`\seq_gset_filter:NNn`

`__seq_set_filter:NNNn`

```

35922 \cs_new_protected:Npn \seq_set_filter:NNn
35923 { \__seq_set_filter:NNNn \__kernel_tl_set:Nx }
35924 \cs_new_protected:Npn \seq_gset_filter:NNn
35925 { \__seq_set_filter:NNNn \__kernel_tl_gset:Nx }
35926 \cs_new_protected:Npn \__seq_set_filter:NNNn #1#2#3#4
35927 {
35928   \__seq_push_item_def:n { \bool_if:nT {#4} { \__seq_wrap_item:n {##1} } }
35929   #1 #2 { #3 }
35930   \__seq_pop_item_def:
35931 }

```

(End definition for `\seq_set_filter:NNn`, `\seq_gset_filter:NNn`, and `__seq_set_filter:NNNn`. These functions are documented on page 305.)

`\seq_set_from_inline_x:Nnn` Set `__seq_item:n` then map it using the loop code.

`\seq_gset_from_inline_x:Nnn`

`__seq_set_from_inline_x:NNnn`

```

35932 \cs_new_protected:Npn \seq_set_from_inline_x:Nnn
35933 { \__seq_set_from_inline_x:NNnn \__kernel_tl_set:Nx }
35934 \cs_new_protected:Npn \seq_gset_from_inline_x:Nnn
35935 { \__seq_set_from_inline_x:NNnn \__kernel_tl_gset:Nx }
35936 \cs_new_protected:Npn \__seq_set_from_inline_x:NNnn #1#2#3#4
35937 {
35938   \__seq_push_item_def:n { \exp_not:N \__seq_item:n {#4} }
35939   #1 #2 { \s__seq #3 \__seq_item:n }
35940   \__seq_pop_item_def:
35941 }

```


(End definition for `\seq_set_from_inline_x:Nnn`, `\seq_gset_from_inline_x:Nnn`, and `__seq_set_from_inline_x:Nnn`. These functions are documented on page 305.)

`\seq_set_from_function:Nnn`
`\seq_gset_from_function:Nnn`

Reuse `\seq_set_from_inline_x:Nnn`.

```
35942 \cs_new_protected:Npn \seq_set_from_function:Nnn #1#2#3
35943   { \seq_set_from_inline_x:Nnn #1 {#2} { #3 {##1} } }
35944 \cs_new_protected:Npn \seq_gset_from_function:Nnn #1#2#3
35945   { \seq_gset_from_inline_x:Nnn #1 {#2} { #3 {##1} } }
```

(End definition for `\seq_set_from_function:Nnn` and `\seq_gset_from_function:Nnn`. These functions are documented on page 305.)

`__seq_int_eval:w` Useful to more quickly go through items.

```
35946 \cs_new_eq:NN \__seq_int_eval:w \tex_numexpr:D
```

(End definition for `__seq_int_eval:w`.)

`\seq_set_item:Nnn`
`\seq_set_item:cnn`

The conditionals are distinguished from the `Nnn` versions by the last argument `\use_ii:nn` vs `\use_i:nn`.

`\seq_set_item:NnnTF`

`\seq_set_item:cnnTF`

`\seq_gset_item:Nnn`

`\seq_gset_item:cnn`

`\seq_gset_item:NnnTF`

`\seq_gset_item:cnnTF`

`__seq_set_item:NnnNN`

`__seq_set_item:nnNNNN`

`__seq_set_item_false:nnNNNN`

`__seq_set_item:nNnnNNNN`

`__seq_set_item:wn`

`__seq_set_item_end:w`

```
35947 \cs_new_protected:Npn \seq_set_item:Nnn #1#2#3
35948   { \__seq_set_item:NnnNN #1 {#2} {#3} \__kernel_tl_set:Nx \use_i:nn }
35949 \cs_new_protected:Npn \seq_gset_item:Nnn #1#2#3
35950   { \__seq_set_item:NnnNN #1 {#2} {#3} \__kernel_tl_gset:Nx \use_i:nn }
35951 \cs_generate_variant:Nn \seq_set_item:Nnn { c }
35952 \cs_generate_variant:Nn \seq_gset_item:Nnn { c }
35953 \prg_new_protected_conditional:Npnn \seq_set_item:Nnn #1#2#3 { TF , T , F }
35954   { \__seq_set_item:NnnNN #1 {#2} {#3} \__kernel_tl_set:Nx \use_ii:nn }
35955 \prg_new_protected_conditional:Npnn \seq_gset_item:Nnn #1#2#3 { TF , T , F }
35956   { \__seq_set_item:NnnNN #1 {#2} {#3} \__kernel_tl_gset:Nx \use_ii:nn }
35957 \prg_generate_conditional_variant:Nnn \seq_set_item:Nnn { c } { TF , T , F }
35958 \prg_generate_conditional_variant:Nnn \seq_gset_item:Nnn { c } { TF , T , F }
```

Save the item to be stored and evaluate the position and the sequence length only once. Then depending on the sign of the position, check that it is not bigger than the length (in absolute value) nor zero.

```
35959 \cs_new_protected:Npn \__seq_set_item:NnnNN #1#2#3
35960   {
35961     \tl_set:Nn \l__seq_internal_a_tl { \__seq_item:n {#3} }
35962     \exp_args:Nff \__seq_set_item:nnNNNN
35963       { \int_eval:n {#2} } { \seq_count:N #1 } #1 \use_none:nn
35964   }
35965 \cs_new_protected:Npn \__seq_set_item:nnNNNN #1#2
35966   {
35967     \int_compare:nNnTF {#1} > 0
35968     { \int_compare:nNnF {#1} > {#2} { \__seq_set_item:nNnnNNNN { #1 - 1 } } }
35969     {
35970       \int_compare:nNnF {#1} < {-#2}
35971       {
35972         \int_compare:nNnF {#1} = 0
35973         { \__seq_set_item:nNnnNNNN { #2 + #1 } }
35974       }
35975     }
35976     \__seq_set_item_false:nnNNNN {#1} {#2}
35977   }
```

If the position is not ok, `__seq_set_item_false:nnNNNN` calls an error or returns `false` (depending on the `\use_i:nn` vs `\use_ii:nn` argument mentioned above).

```

35978 \cs_new_protected:Npn \__seq_set_item_false:nnNNNN #1#2#3#4#5#6
35979 {
35980     #6
35981     {
35982         \msg_error:nnxxx { seq } { item-too-large }
35983         { \token_to_str:N #3 } {#2} {#1}
35984     }
35985     { \prg_return_false: }
35986 }
35987 \msg_new:nnnn { seq } { item-too-large }
35988 { Sequence~'~#1'~does-not-have-an~item~#3 }
35989 {
35990     An~attempt~was~made~to~push~or~pop~the~item~at~position~#3~
35991     of~'~#1'~,~but~this~
35992     \int_compare:nTF { #3 = 0 }
35993     { position~does~not~exist. }
35994     { sequence~only~has~#2~item \int_compare:nF { #2 = 1 } {s}. }
35995 }

```

If the position is ok, `__seq_set_item:nNnnNNNN` makes the assignment and returns `true` (in the case of conditionals). Here `#1` is an integer expression (position minus one), it needs to be evaluated. The sequence `#5` starts with `\s__seq` (even if empty), which stops the integer expression and is absorbed by it. The `\if_meaning:w` test is slightly faster than an integer test (but only works when testing against zero, hence the offset we chose in the position). When we are done skipping items, insert the saved item `\l__seq_internal_a_tl`. For put functions the last argument of `__seq_set_item_end:w` is `\use_none:nn` and it absorbs the item `#2` that we are removing: this is only useful for the `pop` functions.

```

35996 \cs_new_protected:Npn \__seq_set_item:nNnnNNNN #1#2#3#4#5#6#7#8
35997 {
35998     #7 #5
35999     {
36000         \s__seq
36001         \exp_after:wN \__seq_set_item:wn
36002         \int_value:w \__seq_int_eval:w #1
36003         #5 \s__seq_stop #6
36004     }
36005     #8 { } { \prg_return_true: }
36006 }
36007 \cs_new:Npn \__seq_set_item:wn #1 \__seq_item:n #2
36008 {
36009     \if_meaning:w 0 #1 \__seq_set_item_end:w \fi:
36010     \exp_not:n { \__seq_item:n {#2} }
36011     \exp_after:wN \__seq_set_item:wn
36012     \int_value:w \__seq_int_eval:w #1 - 1 \s__seq
36013 }
36014 \cs_new:Npn \__seq_set_item_end:w #1 \exp_not:n #2 #3 \s__seq #4 \s__seq_stop #5
36015 {
36016     #1
36017     \exp_not:o \l__seq_internal_a_tl
36018     \exp_not:n {#4}

```

```

36019     #5 #2
36020   }

```

(End definition for `\seq_set_item:NnnTF` and others. These functions are documented on page 306.)

```

\seq_pop_item:NnN The NnN versions simply call the conditionals, for which we will rely on the internals of
\seq_pop_item:cnN \seq_set_item:Nnn.
\seq_pop_item:NnNTF 36021 \cs_new_protected:Npn \seq_pop_item:NnN #1#2#3
\seq_pop_item:cnNTF 36022 { \seq_pop_item:NnNTF #1 {#2} #3 { } { } }
\seq_gpop_item:NnN 36023 \cs_new_protected:Npn \seq_gpop_item:NnN #1#2#3
\seq_gpop_item:cnN 36024 { \seq_gpop_item:NnNTF #1 {#2} #3 { } { } }
\seq_gpop_item:NnNTF 36025 \cs_generate_variant:Nn \seq_pop_item:NnN { c }
\seq_gpop_item:cnNTF 36026 \cs_generate_variant:Nn \seq_gpop_item:NnN { c }
\__seq_pop_item:NnNNN 36027 \prg_new_protected_conditional:Npnn \seq_pop_item:NnN #1#2#3 { TF , T , F }
36028 { \__seq_pop_item:NnNN #1 {#2} #3 \__kernel_tl_set:Nx }
36029 \prg_new_protected_conditional:Npnn \seq_gpop_item:NnN #1#2#3 { TF , T , F }
36030 { \__seq_pop_item:NnNN #1 {#2} #3 \__kernel_tl_gset:Nx }
36031 \prg_generate_conditional_variant:Nnn \seq_pop_item:NnN { c } { TF , T , F }
36032 \prg_generate_conditional_variant:Nnn \seq_gpop_item:NnN { c } { TF , T , F }

```

Save in `\l__seq_internal_b_tl` the token list variable #3 in which we will store the item. The `__seq_set_item:nnNNNN` auxiliary eventually inserts `\l__seq_internal_a_tl` in place of the item found in the sequence, so we empty that. Instead of the last argument `\use_i:nn` or `\use_ii:nn` used for `put` functions, we introduce `__seq_pop_item:nn`, which stores `\q_no_value` before calling its second argument (`\prg_return_true:/false:`) to end the conditional. The item found is passed to `__seq_pop_item_aux:w`, which interrupts the x-expanding sequence assignment and stores the item using the assignment function in `\l__seq_internal_b_tl`.

```

36033 \cs_new_protected:Npn \__seq_pop_item:NnNN #1#2#3#4
36034 {
36035   \tl_clear:N \l__seq_internal_a_tl
36036   \tl_set:Nn \l__seq_internal_b_tl { \__kernel_tl_set:Nx #3 }
36037   \exp_args:Nff \__seq_set_item:nnNNNN
36038     { \int_eval:n {#2} } { \seq_count:N #1 }
36039     #1 \__seq_pop_item_aux:w #4 \__seq_pop_item:nn
36040 }
36041 \cs_new_protected:Npn \__seq_pop_item:nn #1#2
36042 {
36043   \if_meaning:w \prg_return_false: #2
36044     \l__seq_internal_b_tl { \exp_not:N \q_no_value }
36045   \fi:
36046   #2
36047 }
36048 \cs_new:Npn \__seq_pop_item_aux:w \__seq_item:n #1
36049 {
36050   \if_false: { \fi: }
36051   \l__seq_internal_b_tl { \if_false: } \fi: \exp_not:n {#1}
36052 }

```

(End definition for `\seq_pop_item:NnNTF`, `\seq_gpop_item:NnNTF`, and `__seq_pop_item:NnNNN`. These functions are documented on page 306.)

88.7 Additions to l3sys

```

36053 <@@=sys>
\c_sys_engine_version_str Various different engines, various different ways to extract the data!
36054 \str_const:Nx \c_sys_engine_version_str
36055 {
36056   \str_case:on \c_sys_engine_str
36057   {
36058     { pdftex }
36059     {
36060       \fp_eval:n { round(\int_use:N \tex_pdftexversion:D / 100 , 2) }
36061       .
36062       \tex_pdftexrevision:D
36063     }
36064     { ptex }
36065     {
36066       \cs_if_exist:NT \tex_ptexversion:D
36067       {
36068         p
36069         \int_use:N \tex_ptexversion:D
36070         .
36071         \int_use:N \tex_ptexminorversion:D
36072         \tex_ptexrevision:D
36073         -
36074         \int_use:N \tex_epTeXversion:D
36075       }
36076     }
36077     { luatex }
36078     {
36079       \fp_eval:n { round(\int_use:N \tex_luatexversion:D / 100, 2) }
36080       .
36081       \tex_luatexrevision:D
36082     }
36083     { uptex }
36084     {
36085       \cs_if_exist:NT \tex_ptexversion:D
36086       {
36087         p
36088         \int_use:N \tex_ptexversion:D
36089         .
36090         \int_use:N \tex_ptexminorversion:D
36091         \tex_ptexrevision:D
36092         -
36093         u
36094         \int_use:N \tex_uptexversion:D
36095         \tex_uptexrevision:D
36096         -
36097         \int_use:N \tex_epTeXversion:D
36098       }
36099     }
36100     { xetex }
36101     {
36102       \int_use:N \tex_XeTeXversion:D

```

```

36103         \tex_XeTeXrevision:D
36104     }
36105 }
36106 }

```

(End definition for `\c_sys_engine_version_str`. This variable is documented on page 306.)

88.8 Additions to l3file

```

36107 <@@=ior>
\ior_shell_open:Nn \__ior_shell_open:nN

```

Actually much easier than either the standard `open` or `input` versions! When calling `__kernel_ior_open:Nn` the file the pipe is added to signal a shell command, but the quotes are not added yet—they are added later by `__kernel_file_name_quote:n`.

```

36108 \cs_new_protected:Npn \ior_shell_open:Nn #1#2
36109 {
36110     \sys_if_shell:TF
36111     { \exp_args:No \__ior_shell_open:nN { \tl_to_str:n {#2} } #1 }
36112     { \msg_error:nn { ior } { pipe-failed } }
36113 }
36114 \cs_new_protected:Npn \__ior_shell_open:nN #1#2
36115 {
36116     \tl_if_in:nnTF {#1} { " }
36117     {
36118         \msg_error:nnx
36119         { ior } { quote-in-shell } {#1}
36120     }
36121     { \__kernel_ior_open:Nn #2 { |#1 } }
36122 }
36123 \msg_new:nnnn { ior } { pipe-failed }
36124 { Cannot~run~piped~system~commands. }
36125 {
36126     LaTeX~tried~to~call~a~system~process~but~this~was~not~possible.\\
36127     Try~the~"--shell-escape"~(or~"--enable-pipes")~option.
36128 }

```

(End definition for `\ior_shell_open:Nn` and `__ior_shell_open:nN`. This function is documented on page 302.)

88.9 Additions to l3tl

88.9.1 Building a token list

```

36129 <@@=tl>

```

Between `\tl_build_begin:N <tl var>` and `\tl_build_end:N <tl var>`, the `<tl var>` has the structure

```

\exp_end: ... \exp_end: \__tl_build_last:NNn <assignment> <next tl>
{\<left>} <right>

```

where `<right>` is not braced. The “data” it represents is `<left>` followed by the “data” of `<next tl>` followed by `<right>`. The `<next tl>` is a token list variable whose name is that of `<tl var>` followed by `'`. There are between 0 and 4 `\exp_end:` to keep track of when `<left>`

and $\langle right \rangle$ should be put into the $\langle next tl \rangle$. The $\langle assignment \rangle$ is `\cs_set_nopar:Npx` if the variable is local, and `\cs_gset_nopar:Npx` if it is global.

`\tl_build_begin:N` First construct the $\langle next tl \rangle$: using a prime here conflicts with the usual `expl3` convention
`\tl_build_gbegin:N` but we need a name that can be derived from `#1` without any external data such as a
`__tl_build_begin:NN` counter. Empty that $\langle next tl \rangle$ and setup the structure. The local and global versions
`__tl_build_begin:NNN` only differ by a single function `\cs_(g)set_nopar:Npx` used for all assignments: this is
important because only that function is stored in the $\langle tl var \rangle$ and $\langle next tl \rangle$ for subsequent
assignments. In principle `__tl_build_begin:NNN` could use `\tl_(g)clear_new:N` to
empty `#1` and make sure it is defined, but logging the definition does not seem useful so
we just do `#3 #1 { }` to clear it locally or globally as appropriate.

```

36130 \cs_new_protected:Npn \tl_build_begin:N #1
36131 { \__tl_build_begin:NN \cs_set_nopar:Npx #1 }
36132 \cs_new_protected:Npn \tl_build_gbegin:N #1
36133 { \__tl_build_begin:NN \cs_gset_nopar:Npx #1 }
36134 \cs_new_protected:Npn \__tl_build_begin:NN #1#2
36135 { \exp_args:Nc \__tl_build_begin:NNN { \cs_to_str:N #2 ' } #2 #1 }
36136 \cs_new_protected:Npn \__tl_build_begin:NNN #1#2#3
36137 {
36138   #3 #1 { }
36139   #3 #2
36140   {
36141     \exp_not:n { \exp_end: \exp_end: \exp_end: \exp_end: }
36142     \exp_not:n { \__tl_build_last:NNn #3 #1 { } }
36143   }
36144 }
```

(End definition for `\tl_build_begin:N` and others. These functions are documented on page 307.)

`\tl_build_clear:N` The `begin` and `gbegin` functions already clear enough to make the token list variable
`\tl_build_gclear:N` effectively empty. Eventually the `begin` and `gbegin` functions should check that `#1'` is
empty or undefined, while the `clear` and `gclear` functions ought to empty `#1'`, `#1''`
and so on, similar to `\tl_build_end:N`. This only affects memory usage.

```

36145 \cs_new_eq:NN \tl_build_clear:N \tl_build_begin:N
36146 \cs_new_eq:NN \tl_build_gclear:N \tl_build_gbegin:N
```

(End definition for `\tl_build_clear:N` and `\tl_build_gclear:N`. These functions are documented on page 307.)

`\tl_build_put_right:Nn` Similar to `\tl_put_right:Nn`, but apply `\exp:w` to `#1`. Most of the time this just removes
`\tl_build_put_right:Nx` one `\exp_end:.` When there are none left, `__tl_build_last:NNn` is expanded instead.
`\tl_build_gput_right:Nn` It resets the definition of the $\langle tl var \rangle$ by ending the `\exp_not:n` and the definition early.
`\tl_build_gput_right:Nx` Then it makes sure the $\langle next tl \rangle$ (its argument `#1`) is set-up and starts a new definition.
`__tl_build_last:NNn` Then `__tl_build_put:nn` and `__tl_build_put:nw` place the $\langle left \rangle$ part of the original
`__tl_build_put:nn` $\langle tl var \rangle$ as appropriate for the definition of the $\langle next tl \rangle$ (the $\langle right \rangle$ part is left in the right
`__tl_build_put:nw` place without ever becoming a macro argument). We use `\exp_after:wN` rather than
some `\exp_args:No` to avoid reading arguments that are likely very long token lists. We
use `\cs_(g)set_nopar:Npx` rather than `\tl_(g)set:Nx` partly for the same reason and
partly because the assignments are interrupted by brace tricks, which implies that the
assignment does not simply set the token list to an x-expansion of the second argument.

```

36147 \cs_new_protected:Npn \tl_build_put_right:Nn #1#2
36148 {
36149   \cs_set_nopar:Npx #1
```

```

36150     { \exp_after:wN \exp_not:n \exp_after:wN { \exp:w #1 #2 } }
36151   }
36152 \cs_new_protected:Npn \tl_build_put_right:Nx #1#2
36153 {
36154   \cs_set_nopar:Npx #1
36155   { \exp_after:wN \exp_not:n \exp_after:wN { \exp:w #1 } #2 }
36156 }
36157 \cs_new_protected:Npn \tl_build_gput_right:Nn #1#2
36158 {
36159   \cs_gset_nopar:Npx #1
36160   { \exp_after:wN \exp_not:n \exp_after:wN { \exp:w #1 #2 } }
36161 }
36162 \cs_new_protected:Npn \tl_build_gput_right:Nx #1#2
36163 {
36164   \cs_gset_nopar:Npx #1
36165   { \exp_after:wN \exp_not:n \exp_after:wN { \exp:w #1 } #2 }
36166 }
36167 \cs_new_protected:Npn \__tl_build_last:NNn #1#2
36168 {
36169   \if_false: { { \fi:
36170     \exp_end: \exp_end: \exp_end: \exp_end: \exp_end:
36171     \__tl_build_last:NNn #1 #2 { }
36172   }
36173 }
36174 \if_meaning:w \c_empty_tl #2
36175   \__tl_build_begin:NN #1 #2
36176 \fi:
36177 #1 #2
36178 {
36179   \exp_after:wN \exp_not:n \exp_after:wN
36180   {
36181     \exp:w \if_false: } } \fi:
36182     \exp_after:wN \__tl_build_put:nn \exp_after:wN {#2}
36183 }
36184 \cs_new_protected:Npn \__tl_build_put:nn #1#2 { \__tl_build_put:nw {#2} #1 }
36185 \cs_new_protected:Npn \__tl_build_put:nw #1#2 \__tl_build_last:NNn #3#4#5
36186 { #2 \__tl_build_last:NNn #3 #4 { #1 #5 } }

```

(End definition for `\tl_build_put_right:Nn` and others. These functions are documented on page 308.)

<code>\tl_build_put_left:Nn</code> <code>\tl_build_put_left:Nx</code> <code>\tl_build_gput_left:Nn</code> <code>\tl_build_gput_left:Nx</code> <code>__tl_build_put_left:NNn</code>	<p>See <code>\tl_build_put_right:Nn</code> for all the machinery. We could easily provide <code>\tl_build_put_left_right:NNn</code>, by just add the <code>\right</code> material after the <code>{\left}</code> in the x-expanding assignment.</p> <pre> 36187 \cs_new_protected:Npn \tl_build_put_left:Nn #1 36188 { __tl_build_put_left:NNn \cs_set_nopar:Npx #1 } 36189 \cs_generate_variant:Nn \tl_build_put_left:Nn { Nx } 36190 \cs_new_protected:Npn \tl_build_gput_left:Nn #1 36191 { __tl_build_put_left:NNn \cs_gset_nopar:Npx #1 } 36192 \cs_generate_variant:Nn \tl_build_gput_left:Nn { Nx } 36193 \cs_new_protected:Npn __tl_build_put_left:NNn #1#2#3 36194 { 36195 #1 #2 36196 { 36197 \exp_after:wN \exp_not:n \exp_after:wN </pre>
---	--

```

36198         {
36199             \exp:w \exp_after:wN \__tl_build_put:nn
36200             \exp_after:wN {#2} {#3}
36201         }
36202     }
36203 }

```

(End definition for `\tl_build_put_left:Nn`, `\tl_build_gput_left:Nn`, and `__tl_build_put_left:NNn`. These functions are documented on page 308.)

`\tl_build_get:NN` The idea is to expand the $\langle tl\ var \rangle$ then the $\langle next\ tl \rangle$ and so on, all within an x-expanding assignment, and wrap as appropriate in `\exp_not:n`. The various $\langle left \rangle$ parts are left in the assignment as we go, which enables us to expand the $\langle next\ tl \rangle$ at the right place. The various $\langle right \rangle$ parts are eventually picked up in one last `\exp_not:n`, with a brace trick to wrap all the $\langle right \rangle$ parts together.

```

36204 \cs_new_protected:Npn \tl_build_get:NN
36205 { \__tl_build_get:NNN \__kernel_tl_set:Nx }
36206 \cs_new_protected:Npn \__tl_build_get:NNN #1#2#3
36207 { #1 #3 { \if_false: { \fi: \exp_after:wN \__tl_build_get:w #2 } } }
36208 \cs_new:Npn \__tl_build_get:w #1 \__tl_build_last:NNn #2#3#4
36209 {
36210     \exp_not:n {#4}
36211     \if_meaning:w \c_empty_tl #3
36212         \exp_after:wN \__tl_build_get_end:w
36213     \fi:
36214     \exp_after:wN \__tl_build_get:w #3
36215 }
36216 \cs_new:Npn \__tl_build_get_end:w #1#2#3
36217 { \exp_after:wN \exp_not:n \exp_after:wN { \if_false: } \fi: }

```

(End definition for `\tl_build_get:NN` and others. This function is documented on page 308.)

`\tl_build_end:N` Get the data then clear the $\langle next\ tl \rangle$ recursively until finding an empty one. It is perhaps wasteful to repeatedly use `\cs_to_sr:N`. The local/global scope is checked by `\tl_set:Nx` or `\tl_gset:Nx`.

```

36218 \cs_new_protected:Npn \tl_build_end:N #1
36219 {
36220     \__tl_build_get:NNN \__kernel_tl_set:Nx #1 #1
36221     \exp_args:Nc \__tl_build_end_loop:NN { \cs_to_str:N #1 ' } \tl_clear:N
36222 }
36223 \cs_new_protected:Npn \tl_build_gend:N #1
36224 {
36225     \__tl_build_get:NNN \__kernel_tl_gset:Nx #1 #1
36226     \exp_args:Nc \__tl_build_end_loop:NN { \cs_to_str:N #1 ' } \tl_gclear:N
36227 }
36228 \cs_new_protected:Npn \__tl_build_end_loop:NN #1#2
36229 {
36230     \if_meaning:w \c_empty_tl #1
36231         \exp_after:wN \use_none:nnnnnn
36232     \fi:
36233     #2 #1
36234     \exp_args:Nc \__tl_build_end_loop:NN { \cs_to_str:N #1 ' } #2
36235 }

```

(End definition for `\tl_build_end:N`, `\tl_build_gend:N`, and `__tl_build_end_loop:NN`. These functions are documented on page 308.)

88.9.2 Other additions to l3tl

`\tl_range_braced:Nnn` For the braced version `__tl_range_braced:w` sets up `__tl_range_collect_braced:w` which stores items one by one in an argument after the semicolon. The unbraced version is almost identical. The version preserving braces and spaces starts by deleting spaces before the argument to avoid collecting them, and sets up `__tl_range_collect:nn` with a first argument of the form `{ {⟨collected⟩} ⟨tokens⟩ }`, whose head is the collected tokens and whose tail is what remains of the original token list. This form makes it easier to move tokens to the `⟨collected⟩` tokens.

```

\tl_range_braced:cnn
\tl_range_braced:nnn
\tl_range_unbraced:Nnn
\tl_range_unbraced:cnn
\tl_range_unbraced:nnn
\__tl_range_braced:w
\__tl_range_collect_braced:w
\__tl_range_unbraced:w
\tl_range_collect_unbraced:w
36236 \cs_new:Npn \tl_range_braced:Nnn { \exp_args:No \tl_range_braced:nnn }
36237 \cs_generate_variant:Nn \tl_range_braced:Nnn { c }
36238 \cs_new:Npn \tl_range_braced:nnn { \__tl_range:Nnnn \__tl_range_braced:w }
36239 \cs_new:Npn \tl_range_unbraced:Nnn
36240   { \exp_args:No \tl_range_unbraced:nnn }
36241 \cs_generate_variant:Nn \tl_range_unbraced:Nnn { c }
36242 \cs_new:Npn \tl_range_unbraced:nnn
36243   { \__tl_range:Nnnn \__tl_range_unbraced:w }
36244 \cs_new:Npn \__tl_range_braced:w #1 ; #2
36245   { \__tl_range_collect_braced:w #1 ; { } #2 }
36246 \cs_new:Npn \__tl_range_unbraced:w #1 ; #2
36247   { \__tl_range_collect_unbraced:w #1 ; { } #2 }
36248 \cs_new:Npn \__tl_range_collect_braced:w #1 ; #2#3
36249   {
36250     \if_int_compare:w #1 > \c_one_int
36251       \exp_after:wN \__tl_range_collect_braced:w
36252       \int_value:w \int_eval:n { #1 - 1 } \exp_after:wN ;
36253     \fi:
36254     { #2 {#3} }
36255   }
36256 \cs_new:Npn \__tl_range_collect_unbraced:w #1 ; #2#3
36257   {
36258     \if_int_compare:w #1 > \c_one_int
36259       \exp_after:wN \__tl_range_collect_unbraced:w
36260       \int_value:w \int_eval:n { #1 - 1 } \exp_after:wN ;
36261     \fi:
36262     { #2 #3 }
36263   }

```

(End definition for `\tl_range_braced:Nnn` and others. These functions are documented on page 307.)

88.10 Additions to l3token

`\c_catcode_active_space_tl` While `\char_generate:nn` can produce active characters in some engines it cannot in general. It would be possible to simply change the catcode of space but then the code would need to avoid all spaces, making it quite unreadable. Instead we use the primitive `\tex_lowercase:D` trick.

```

36264 \group_begin:
36265   \char_set_catcode_active:N *
36266   \char_set_lccode:nn { '*' } { '\ }
36267   \tex_lowercase:D { \tl_const:Nn \c_catcode_active_space_tl { * } }
36268 \group_end:

```

(End definition for \c_catcode_active_space_tl. This variable is documented on page 308.)

36269 <@@=peek>

\l__peek_collect_tl

36270 \tl_new:N \l__peek_collect_tl

(End definition for \l__peek_collect_tl.)

\peek_catcode_collect_inline:Nn
\peek_charcode_collect_inline:Nn
\peek_meaning_collect_inline:Nn
__peek_collect:NNn
__peek_collect_true:w
__peek_collect_remove:nw
__peek_collect:N

Most of the work is done by __peek_execute_branches_...:, which calls either __peek_true:w or __peek_false:w according to whether the next token \l__peek_token matches the search token (stored in \l__peek_search_token and \l__peek_search_tl). Here, in the true case we run __peek_collect_true:w, which generally calls __peek_collect:N to store the peeked token into \l__peek_collect_tl, except in special non-N-type cases (begin-group, end-group, or space), where a frozen token is stored. The true branch calls __peek_execute_branches_...: to fetch more matching tokens. Once there are no more, __peek_false_aux:n closes the safe-align group and runs the user's inline code.

36271 \cs_new_protected:Npn \peek_catcode_collect_inline:Nn
36272 { __peek_collect:NNn __peek_execute_branches_catcode: }
36273 \cs_new_protected:Npn \peek_charcode_collect_inline:Nn
36274 { __peek_collect:NNn __peek_execute_branches_charcode: }
36275 \cs_new_protected:Npn \peek_meaning_collect_inline:Nn
36276 { __peek_collect:NNn __peek_execute_branches_meaning: }
36277 \cs_new_protected:Npn __peek_collect:NNn #1#2#3
36278 {
36279 \group_align_safe_begin:
36280 \cs_set_eq:NN \l__peek_search_token #2
36281 \tl_set:Nn \l__peek_search_tl {#2}
36282 \tl_clear:N \l__peek_collect_tl
36283 \cs_set:Npn __peek_false:w
36284 { \exp_args:No __peek_false_aux:n \l__peek_collect_tl }
36285 \cs_set:Npn __peek_false_aux:n ##1
36286 {
36287 \group_align_safe_end:
36288 #3
36289 }
36290 \cs_set_eq:NN __peek_true:w __peek_collect_true:w
36291 \cs_set:Npn __peek_true_aux:w { \peek_after:Nw #1 }
36292 __peek_true_aux:w
36293 }
36294 \cs_new_protected:Npn __peek_collect_true:w
36295 {
36296 \if_case:w
36297 \if_catcode:w \exp_not:N \l__peek_token { 1 \exp_stop_f: \fi:
36298 \if_catcode:w \exp_not:N \l__peek_token } 2 \exp_stop_f: \fi:
36299 \if_meaning:w \l__peek_token \c_space_token 3 \exp_stop_f: \fi:
36300 0 \exp_stop_f:
36301 \exp_after:wN __peek_collect:N
36302 \or: __peek_collect_remove:nw { \c_group_begin_token }
36303 \or: __peek_collect_remove:nw { \c_group_end_token }
36304 \or: __peek_collect_remove:nw { ~ }
36305 \fi:
36306 }

```

36307 \cs_new_protected:Npn \__peek_collect:N #1
36308 {
36309   \tl_put_right:Nn \l__peek_collect_tl {#1}
36310   \__peek_true_aux:w
36311 }
36312 \cs_new_protected:Npn \__peek_collect_remove:nw #1
36313 {
36314   \tl_put_right:Nn \l__peek_collect_tl {#1}
36315   \exp_after:wN \__peek_true_remove:w
36316 }

```

(End definition for \peek_catcode_collect_inline:Nn and others. These functions are documented on page 309.)

```

36317 \endpackage

```

Chapter 89

l3deprecation implementation

```
36318 <*package>
36319 <@@=deprecation>
```

89.1 Patching definitions to deprecate

```
\__kernel_patch_deprecation:nnNNpn {<date>} {<replacement>} <definition>
<function> <parameters> {<code>}
```

defines the *<function>* to produce a warning and run its *<code>*, or to produce an error and not run any *<code>*, depending on the `expl3` date.

- If the `expl3` date is less than the *<date>* (plus 6 months in case `undo-recent-deprecations` is used) then we define the *<function>* to produce a warning and run its code. The warning is actually suppressed in two cases:
 - if neither `undo-recent-deprecations` nor `enable-debug` are in effect we may be in an end-user’s document so it is suppressed;
 - if the command is expandable then we cannot produce a warning.
- Otherwise, we define the *<function>* to produce an error.

In both cases we additionally make `\debug_on:n {deprecation}` turn the *<function>* into an `\outer` error, and `\debug_off:n {deprecation}` restore whatever the behaviour was without `\debug_on:n {deprecation}`.

In later sections we use the `l3doc` key `deprecated` with a date equal to that *<date>* plus 6 months, so that `l3doc` will complain if we forget to remove the stale *<parameters>* and *{<code>}*.

In the explanations below, *<definition>* *<function>* *<parameters>* *{<code>}* or assignments that only differ in the scope of the *<definition>* will be called “the standard definition”.

```
\__kernel_patch_deprecation:nnNNpn (The parameter text is grabbed using #5#.) The arguments of \__kernel_deprecation_
\__deprecation_patch_aux:nnNNnn code:nn are run upon \debug_on:n {deprecation} and \debug_off:n {deprecation},
\__deprecation_warn_once:nnNnn respectively. In both scenarios we the <function> may be \outer so we undefine it with
\__deprecation_patch_aux:Nn \tex_let:D before redefining it, with \__kernel_deprecation_error:Nnn or with some
\__deprecation_just_error:nnNN code added shortly.
```

```

36320 \cs_new_protected:Npn \__kernel_patch_deprecation:nnNNpn #1#2#3#4#5#
36321 { \__deprecation_patch_aux:nnNNnn {#1} {#2} #3 #4 {#5} }
36322 \cs_new_protected:Npn \__deprecation_patch_aux:nnNNnn #1#2#3#4#5#6
36323 {
36324   \__kernel_deprecation_code:nn
36325   {
36326     \tex_let:D #4 \scan_stop:
36327     \__kernel_deprecation_error:Nnn #4 {#2} {#1}
36328   }
36329   { \tex_let:D #4 \scan_stop: }
36330   \cs_if_eq:NNTF #3 \cs_gset_protected:Npn
36331   { \__deprecation_warn_once:nnNnn {#1} {#2} #4 {#5} {#6} }
36332   { \__deprecation_patch_aux:Nn #3 { #4 #5 {#6} } }
36333 }

```

In case we want a warning, the *function* is defined to produce such a warning without grabbing any argument, then redefine itself to the standard definition that the *function* should have, with arguments, and call that definition. The `x-type` expansion and `\exp_not:n` avoid needing to double the #, which we could not do anyways. We then deal with the code for `\debug_off:n {deprecation}`: presumably someone doing that does not need the warning so we simply do the standard definition.

```

36334 \cs_new_protected:Npn \__deprecation_warn_once:nnNnn #1#2#3#4#5
36335 {
36336   \cs_gset_protected:Npx #3
36337   {
36338     \__kernel_if_debug:TF
36339     {
36340       \exp_not:N \msg_warning:nnxxx
36341       { deprecation } { deprecated-command }
36342       {#1}
36343       { \token_to_str:N #3 }
36344       { \tl_to_str:n {#2} }
36345     }
36346   { }
36347   \exp_not:n { \cs_gset_protected:Npn #3 #4 {#5} }
36348   \exp_not:N #3
36349 }
36350 \__kernel_deprecation_code:nn { }
36351 { \cs_set_protected:Npn #3 #4 {#5} }
36352 }

```

In case we want neither warning nor error, the *function* is given its standard definition. Here #1 is `\cs_new:Npn` or `\cs_new_protected:Npn` and #2 is *function* *parameters* *{code}*, so #1#2 performs the assignment. For `\debug_off:n {deprecation}` we want to use the same assignment but with a different scope, hence the `\cs_if_eq:NNTF` test.

```

36353 \cs_new_protected:Npn \__deprecation_patch_aux:Nn #1#2
36354 {
36355   #1 #2
36356   \cs_if_eq:NNTF #1 \cs_gset_protected:Npn
36357   { \__kernel_deprecation_code:nn { } { \cs_set_protected:Npn #2 } }
36358   { \__kernel_deprecation_code:nn { } { \cs_set:Npn #2 } }
36359 }

```

(End definition for `__kernel_patch_deprecation:nnNNpn` and others.)

`__kernel_deprecation_error:Nnn` The `\outer` definition here ensures the command cannot appear in an argument. Use this auxiliary on all commands that have been removed since 2015.

```

36360 \cs_new_protected:Npn \__kernel_deprecation_error:Nnn #1#2#3
36361 {
36362   \tex_protected:D \tex_outer:D \tex_edef:D #1
36363   {
36364     \exp_not:N \msg_expandable_error:nnnnn
36365     { deprecation } { deprecated-command }
36366     { \tl_to_str:n {#3} } { \token_to_str:N #1 } { \tl_to_str:n {#2} }
36367     \exp_not:N \msg_error:nnxxx
36368     { deprecation } { deprecated-command }
36369     { \tl_to_str:n {#3} } { \token_to_str:N #1 } { \tl_to_str:n {#2} }
36370   }
36371 }

```

(End definition for `__kernel_deprecation_error:Nnn`.)

```

36372 \msg_new:nnn { deprecation } { deprecated-command }
36373 {
36374   \tl_if_blank:nF {#3} { Use~ \tl_trim_spaces:n {#3} ~not~ }
36375   #2~deprecated~on~#1.
36376 }

```

89.2 Removed functions

`__deprecation_old_protected:Nnn` Short-hands for old commands whose definition does not matter any more as they were removed.

```

\__deprecation_old:Nnn
36377 \cs_new_protected:Npn \__deprecation_old_protected:Nnn #1#2#3
36378 {
36379   \__kernel_patch_deprecation:nnNNpn {#3} {#2}
36380   \cs_gset_protected:Npn #1 { }
36381 }
36382 \cs_new_protected:Npn \__deprecation_old:Nnn #1#2#3
36383 {
36384   \__kernel_patch_deprecation:nnNNpn {#3} {#2}
36385   \cs_gset:Npn #1 { }
36386 }
36387 \__deprecation_old_protected:Nnn \box_gset_eq_clear:NN
36388 { \box_gset_eq_drop:NN } { 2021-07-01 }
36389 \__deprecation_old_protected:Nnn \box_set_eq_clear:NN
36390 { \box_set_eq_drop:NN } { 2021-07-01 }
36391 \__deprecation_old_protected:Nnn \box_resize:Nnn
36392 { \box_resize_to_wd_and_ht_plus_dp:Nnn } { 2019-01-01 }
36393 \__deprecation_old_protected:Nnn \box_use_clear:N
36394 { \box_use_drop:N } { 2019-01-01 }
36395 \__deprecation_old:Nnn \c_job_name_tl
36396 { \c_sys_jobname_str } { 2017-01-01 }
36397 \__deprecation_old:Nnn \c_minus_one
36398 { -1 } { 2019-01-01 }
36399 \__deprecation_old:Nnn \c_zero
36400 { 0 } { 2020-01-01 }
36401 \__deprecation_old:Nnn \c_one
36402 { 1 } { 2020-01-01 }

```

```

36403 \__deprecation_old:Nnn \c_two
36404 { 2 } { 2020-01-01 }
36405 \__deprecation_old:Nnn \c_three
36406 { 3 } { 2020-01-01 }
36407 \__deprecation_old:Nnn \c_four
36408 { 4 } { 2020-01-01 }
36409 \__deprecation_old:Nnn \c_five
36410 { 5 } { 2020-01-01 }
36411 \__deprecation_old:Nnn \c_six
36412 { 6 } { 2020-01-01 }
36413 \__deprecation_old:Nnn \c_seven
36414 { 7 } { 2020-01-01 }
36415 \__deprecation_old:Nnn \c_eight
36416 { 8 } { 2020-01-01 }
36417 \__deprecation_old:Nnn \c_nine
36418 { 9 } { 2020-01-01 }
36419 \__deprecation_old:Nnn \c_ten
36420 { 10 } { 2020-01-01 }
36421 \__deprecation_old:Nnn \c_eleven
36422 { 11 } { 2020-01-01 }
36423 \__deprecation_old:Nnn \c_twelve
36424 { 12 } { 2020-01-01 }
36425 \__deprecation_old:Nnn \c_thirteen
36426 { 13 } { 2020-01-01 }
36427 \__deprecation_old:Nnn \c_fourteen
36428 { 14 } { 2020-01-01 }
36429 \__deprecation_old:Nnn \c_fifteen
36430 { 15 } { 2020-01-01 }
36431 \__deprecation_old:Nnn \c_sixteen
36432 { 16 } { 2020-01-01 }
36433 \__deprecation_old:Nnn \c_thirty_two
36434 { 32 } { 2020-01-01 }
36435 \__deprecation_old:Nnn \c_one_hundred
36436 { 100 } { 2020-01-01 }
36437 \__deprecation_old:Nnn \c_two_hundred_fifty_five
36438 { 255 } { 2020-01-01 }
36439 \__deprecation_old:Nnn \c_two_hundred_fifty_six
36440 { 256 } { 2020-01-01 }
36441 \__deprecation_old:Nnn \c_one_thousand
36442 { 1000 } { 2020-01-01 }
36443 \__deprecation_old:Nnn \c_ten_thousand
36444 { 10000 } { 2020-01-01 }
36445 \__deprecation_old:Nnn \c_term_ior
36446 { -1 } { 2021-07-01 }
36447 \__deprecation_old:Nnn \dim_case:nnn
36448 { \dim_case:nnF } { 2015-07-14 }
36449 \__deprecation_old_protected:Nnn \file_add_path:nN
36450 { \file_get_full_name:nN } { 2019-01-01 }
36451 \__deprecation_old_protected:Nnn \file_if_exist_input:nT
36452 { \file_if_exist:nT and~ \file_input:n } { 2018-03-05 }
36453 \__deprecation_old_protected:Nnn \file_if_exist_input:nTF
36454 { \file_if_exist:nT and~ \file_input:n } { 2018-03-05 }
36455 \__deprecation_old_protected:Nnn \file_list:
36456 { \file_log_list: } { 2019-01-01 }

```

```

36457 \__deprecation_old:Nnn \file_path_include:n
36458 { \seq_put_right:Nn \l_file_search_path_seq } { 2019-01-01 }
36459 \__deprecation_old_protected:Nnn \file_path_remove:n
36460 { \seq_remove_all:Nn \l_file_search_path_seq } { 2019-01-01 }
36461 \__deprecation_old:Nnn \g_file_current_name_tl
36462 { \g_file_curr_name_str } { 2019-01-01 }
36463 \__deprecation_old_protected:Nnn \hbox_unpack_clear:N
36464 { \hbox_unpack_drop:N } { 2021-07-01 }
36465 \__deprecation_old:Nnn \int_case:nnn
36466 { \int_case:nnF } { 2015-07-14 }
36467 \__deprecation_old:Nnn \int_from_binary:n
36468 { \int_from_bin:n } { 2016-01-05 }
36469 \__deprecation_old:Nnn \int_from_hexadecimal:n
36470 { \int_from_hex:n } { 2016-01-05 }
36471 \__deprecation_old:Nnn \int_from_octal:n
36472 { \int_from_oct:n } { 2016-01-05 }
36473 \__deprecation_old:Nnn \int_to_binary:n
36474 { \int_to_bin:n } { 2016-01-05 }
36475 \__deprecation_old:Nnn \int_to_hexadecimal:n
36476 { \int_to_hex:n } { 2016-01-05 }
36477 \__deprecation_old:Nnn \int_to_octal:n
36478 { \int_to_oct:n } { 2016-01-05 }
36479 \__deprecation_old_protected:Nnn \ior_get_str:NN
36480 { \ior_str_get:NN } { 2018-03-05 }
36481 \__deprecation_old_protected:Nnn \ior_list_streams:
36482 { \ior_show_list: } { 2019-01-01 }
36483 \__deprecation_old_protected:Nnn \ior_log_streams:
36484 { \ior_log_list: } { 2019-01-01 }
36485 \__deprecation_old_protected:Nnn \iow_list_streams:
36486 { \iow_show_list: } { 2019-01-01 }
36487 \__deprecation_old_protected:Nnn \iow_log_streams:
36488 { \iow_log_list: } { 2019-01-01 }
36489 \__deprecation_old:Nnn \lua_escape_x:n
36490 { \lua_escape:e } { 2020-01-01 }
36491 \__deprecation_old:Nnn \lua_now_x:n
36492 { \lua_now:e } { 2020-01-01 }
36493 \__deprecation_old_protected:Nnn \lua_shipout_x:n
36494 { \lua_shipout_e:n } { 2020-01-01 }
36495 \__deprecation_old:Nnn \luatex_if_engine_p:
36496 { \sys_if_engine luatex_p: } { 2017-01-01 }
36497 \__deprecation_old:Nnn \luatex_if_engine:F
36498 { \sys_if_engine luatex:F } { 2017-01-01 }
36499 \__deprecation_old:Nnn \luatex_if_engine:T
36500 { \sys_if_engine luatex:T } { 2017-01-01 }
36501 \__deprecation_old:Nnn \luatex_if_engine:TF
36502 { \sys_if_engine luatex:TF } { 2017-01-01 }
36503 \__deprecation_old_protected:Nnn \msg_interrupt:nnn
36504 { [Defined-error-message] } { 2020-01-01 }
36505 \__deprecation_old_protected:Nnn \msg_log:n
36506 { \iow_log:n } { 2020-01-01 }
36507 \__deprecation_old_protected:Nnn \msg_term:n
36508 { \iow_term:n } { 2020-01-01 }
36509 \__deprecation_old:Nnn \pdftex_if_engine_p:
36510 { \sys_if_engine pdftex_p: } { 2017-01-01 }

```



```

36511 \_deprecation_old:Nnn \pdfTeX_if_engine:F
36512 { \sys_if_engine_pdftex:F } { 2017-01-01 }
36513 \_deprecation_old:Nnn \pdfTeX_if_engine:T
36514 { \sys_if_engine_pdftex:T } { 2017-01-01 }
36515 \_deprecation_old:Nnn \pdfTeX_if_engine:TF
36516 { \sys_if_engine_pdftex:TF } { 2017-01-01 }
36517 \_deprecation_old:Nnn \prop_get:cn
36518 { \prop_item:cn } { 2016-01-05 }
36519 \_deprecation_old:Nnn \prop_get:Nn
36520 { \prop_item:Nn } { 2016-01-05 }
36521 \_deprecation_old:Nnn \quark_if_recursion_tail_break:N
36522 { } { 2015-07-14 }
36523 \_deprecation_old:Nnn \quark_if_recursion_tail_break:n
36524 { } { 2015-07-14 }
36525 \_deprecation_old:Nnn \scan_align_safe_stop:
36526 { protected~commands } { 2017-01-01 }
36527 \_deprecation_old:Nnn \sort_ordered:
36528 { \sort_return_same: } { 2019-01-01 }
36529 \_deprecation_old:Nnn \sort_reversed:
36530 { \sort_return_swapped: } { 2019-01-01 }
36531 \_deprecation_old:Nnn \str_case:nnn
36532 { \str_case:nnF } { 2015-07-14 }
36533 \_deprecation_old:Nnn \str_case:onn
36534 { \str_case:onF } { 2015-07-14 }
36535 \_deprecation_old:Nnn \str_case_x:nn
36536 { \str_case_e:nn } { 2020-01-01 }
36537 \_deprecation_old:Nnn \str_case_x:nnn
36538 { \str_case_e:nnF } { 2015-07-14 }
36539 \_deprecation_old:Nnn \str_case_x:nnT
36540 { \str_case_e:nnT } { 2020-01-01 }
36541 \_deprecation_old:Nnn \str_case_x:nnTF
36542 { \str_case_e:nnTF } { 2020-01-01 }
36543 \_deprecation_old:Nnn \str_case_x:nnF
36544 { \str_case_e:nnF } { 2020-01-01 }
36545 \_deprecation_old:Nnn \str_if_eq_x:p:nn
36546 { \str_if_eq_p:ee } { 2020-01-01 }
36547 \_deprecation_old:Nnn \str_if_eq_x:nnT
36548 { \str_if_eq:eeT } { 2020-01-01 }
36549 \_deprecation_old:Nnn \str_if_eq_x:nnF
36550 { \str_if_eq:eeF } { 2020-01-01 }
36551 \_deprecation_old:Nnn \str_if_eq_x:nnTF
36552 { \str_if_eq:eeTF } { 2020-01-01 }
36553 \_deprecation_old_protected:Nnn \tl_show_analysis:N
36554 { \tl_analysis_show:N } { 2020-01-01 }
36555 \_deprecation_old_protected:Nnn \tl_show_analysis:n
36556 { \tl_analysis_show:n } { 2020-01-01 }
36557 \_deprecation_old:Nnn \tl_case:cn
36558 { \tl_case:cnF } { 2015-07-14 }
36559 \_deprecation_old:Nnn \tl_case:Nnn
36560 { \tl_case:NnF } { 2015-07-14 }
36561 \_deprecation_old_protected:Nnn \tl_gset_from_file:Nnn
36562 { \file_get:nnN } { 2021-07-01 }
36563 \_deprecation_old_protected:Nnn \tl_gset_from_file_x:Nnn
36564 { \file_get:nnN } { 2021-07-01 }

```

```

36565 \__deprecation_old_protected:Nnn \tl_set_from_file:Nnn
36566 { \file_get:nnN } { 2021-07-01 }
36567 \__deprecation_old_protected:Nnn \tl_set_from_file_x:Nnn
36568 { \file_get:nnN } { 2021-07-01 }
36569 \__deprecation_old_protected:Nnn \tl_to_lowercase:n
36570 { \tex_lowercase:D } { 2018-03-05 }
36571 \__deprecation_old_protected:Nnn \tl_to_uppercase:n
36572 { \tex_uppercase:D } { 2018-03-05 }
36573 \__deprecation_old:Nnn \token_get_arg_spec:N
36574 { \cs_argument_spec:N } { 2021-07-01 }
36575 \__deprecation_old:Nnn \token_get_prefix_spec:N
36576 { \cs_prefix_spec:N } { 2021-07-01 }
36577 \__deprecation_old:Nnn \token_get_replacement_spec:N
36578 { \cs_replacement_spec:N } { 2021-07-01 }
36579 \__deprecation_old_protected:Nnn \token_new:Nn
36580 { \cs_new_eq:NN } { 2019-01-01 }
36581 \__deprecation_old_protected:Nnn \vbox_unpack_clear:N
36582 { \vbox_unpack_drop:N } { 2021-07-01 }
36583 \__deprecation_old:Nnn \xetex_if_engine_p:
36584 { \sys_if_engine_xetex_p: } { 2017-01-01 }
36585 \__deprecation_old:Nnn \xetex_if_engine:F
36586 { \sys_if_engine_xetex:F } { 2017-01-01 }
36587 \__deprecation_old:Nnn \xetex_if_engine:T
36588 { \sys_if_engine_xetex:T } { 2017-01-01 }
36589 \__deprecation_old:Nnn \xetex_if_engine:TF
36590 { \sys_if_engine_xetex:TF } { 2017-01-01 }

```

(End definition for __deprecation_old_protected:Nnn and __deprecation_old:Nnn.)

89.3 Deprecated l3str functions

```

36591 <@@=str>

\str_lower_case:n
\str_lower_case:f 36592 \__kernel_patch_deprecation:nnNNpn { 2020-01-03 } { \str_lowercase:n }
\str_upper_case:n 36593 \cs_gset:Npn \str_lower_case:n { \str_lowercase:n }
\str_upper_case:f 36594 \__kernel_patch_deprecation:nnNNpn { 2020-01-03 } { \str_lowercase:f }
\str_fold_case:n 36595 \cs_gset:Npn \str_lower_case:f { \str_lowercase:f }
\str_fold_case:V 36596 \__kernel_patch_deprecation:nnNNpn { 2020-01-03 } { \str_uppercase:n }
36597 \cs_gset:Npn \str_upper_case:n { \str_uppercase:n }
36598 \__kernel_patch_deprecation:nnNNpn { 2020-01-03 } { \str_uppercase:f }
36599 \cs_gset:Npn \str_upper_case:f { \str_uppercase:f }
36600 \__kernel_patch_deprecation:nnNNpn { 2020-01-03 } { \str_foldcase:n }
36601 \cs_gset:Npn \str_fold_case:n { \str_foldcase:n }
36602 \__kernel_patch_deprecation:nnNNpn { 2020-01-03 } { \str_foldcase:V }
36603 \cs_gset:Npn \str_fold_case:V { \str_foldcase:V }

```

(End definition for \str_lower_case:n, \str_upper_case:n, and \str_fold_case:n. These functions are documented on page ??.)

\str_declare_eight_bit_encoding:nnn

This command was made internal, with one more argument. There is no easy way to compute a reasonable value for that extra argument so we take a value that is big enough to accommodate all of Unicode.

```

36604 \__kernel_patch_deprecation:nnNNpn { 2020-08-20 } { }

```

```

36605 \cs_gset_protected:Npn \str_declare_eight_bit_encoding:nnn #1
36606 { \__str_declare_eight_bit_encoding:nnnn {#1} { 1114112 } }

```

(End definition for \str_declare_eight_bit_encoding:nnn. This function is documented on page ??.)

89.4 Deprecated l3seq functions

```

\seq_indexed_map_inline:Nn
\seq_indexed_map_function:NN
36607 \__kernel_patch_deprecation:nnNNpn { 2020-06-18 } { \seq_map_indexed_inline:Nn }
36608 \cs_gset_protected:Npn \seq_indexed_map_inline:Nn { \seq_map_indexed_inline:Nn }
36609 \__kernel_patch_deprecation:nnNNpn { 2020-06-18 } { \seq_map_indexed_function:NN }
36610 \cs_gset:Npn \seq_indexed_map_function:NN { \seq_map_indexed_function:NN }

```

(End definition for \seq_indexed_map_inline:Nn and \seq_indexed_map_function:NN. These functions are documented on page ??.)

89.5 Deprecated l3sys functions

```

\sys_load_deprecation:
36611 \__kernel_patch_deprecation:nnNNpn { 2021-01-11 } { (no-longer-required) }
36612 \cs_gset_protected:Npn \sys_load_deprecation:
36613 {
36614   \bool_if:NF \g__str_deprecation_bool
36615   { \__kernel_sys_configuration_load:n { l3deprecation } }
36616   \bool_gset_true:N \g__str_deprecation_bool
36617 }

```

(End definition for \sys_load_deprecation:. This function is documented on page ??.)

89.6 Deprecated l3tl functions

```

36618 <@@=tl>
\tl_lower_case:n
\tl_lower_case:nn
\tl_upper_case:n
\tl_upper_case:nn
\tl_mixed_case:n
\tl_mixed_case:nn
36619 \__kernel_patch_deprecation:nnNNpn { 2020-01-03 } { \text_lowercase:n }
36620 \cs_gset:Npn \tl_lower_case:n #1
36621 { \text_lowercase:n {#1} }
36622 \__kernel_patch_deprecation:nnNNpn { 2020-01-03 } { \text_lowercase:nn }
36623 \cs_gset:Npn \tl_lower_case:nn #1#2
36624 { \text_lowercase:nn {#1} {#2} }
36625 \__kernel_patch_deprecation:nnNNpn { 2020-01-03 } { \text_uppercase:n }
36626 \cs_gset:Npn \tl_upper_case:n #1
36627 { \text_uppercase:n {#1} }
36628 \__kernel_patch_deprecation:nnNNpn { 2020-01-03 } { \text_uppercase:nn }
36629 \cs_gset:Npn \tl_upper_case:nn #1#2
36630 { \text_uppercase:nn {#1} {#2} }
36631 \__kernel_patch_deprecation:nnNNpn { 2020-01-03 } { \text_titlecase:n }
36632 \cs_gset:Npn \tl_mixed_case:n #1
36633 { \text_titlecase:n {#1} }
36634 \__kernel_patch_deprecation:nnNNpn { 2020-01-03 } { \text_titlecase:nn }
36635 \cs_gset:Npn \tl_mixed_case:nn #1#2
36636 { \text_titlecase:nn {#1} {#2} }

```

(End definition for \tl_lower_case:n and others. These functions are documented on page ??.)

89.7 Deprecated l3token functions

```

\char_lower_case:N
\char_upper_case:N
\char_mixed_case:Nn
\char_fold_case:N
\char_str_lower_case:N
\char_str_upper_case:N
\char_str_mixed_case:Nn
\char_str_fold_case:N

36637 \__kernel_patch_deprecation:nnNNpn { 2020-01-03 } { \char_lowercase:N }
36638 \cs_gset:Npn \char_lower_case:N { \char_lowercase:N }
36639 \__kernel_patch_deprecation:nnNNpn { 2020-01-03 } { \char_uppercase:N }
36640 \cs_gset:Npn \char_upper_case:N { \char_uppercase:N }
36641 \__kernel_patch_deprecation:nnNNpn { 2020-01-03 } { \char_titlecase:N }
36642 \cs_gset:Npn \char_mixed_case:N { \char_titlecase:N }
36643 \__kernel_patch_deprecation:nnNNpn { 2020-01-03 } { \char_foldcase:N }
36644 \cs_gset:Npn \char_fold_case:N { \char_foldcase:N }
36645 \__kernel_patch_deprecation:nnNNpn { 2020-01-03 } { \char_str_lowercase:N }
36646 \cs_gset:Npn \char_str_lower_case:N { \char_str_lowercase:N }
36647 \__kernel_patch_deprecation:nnNNpn { 2020-01-03 } { \char_str_uppercase:N }
36648 \cs_gset:Npn \char_str_upper_case:N { \char_str_uppercase:N }
36649 \__kernel_patch_deprecation:nnNNpn { 2020-01-03 } { \char_str_titlecase:N }
36650 \cs_gset:Npn \char_str_mixed_case:N { \char_str_titlecase:N }
36651 \__kernel_patch_deprecation:nnNNpn { 2020-01-03 } { \char_str_foldcase:N }
36652 \cs_gset:Npn \char_str_fold_case:N { \char_str_foldcase:N }

```

(End definition for `\char_lower_case:N` and others. These functions are documented on page ??.)

```

\peek_catcode_ignore_spaces:NTF
\peek_catcode_remove_ignore_spaces:NTF
\peek_charcode_ignore_spaces:NTF
\peek_charcode_remove_ignore_spaces:NTF
\peek_meaning_ignore_spaces:NTF
\peek_meaning_remove_ignore_spaces:NTF

A little extra fun here to deal with the expansion.

36653 \tl_map_inline:nn
36654 {
36655   { catcode } { catcode_remove }
36656   { charcode } { charcode_remove }
36657   { meaning } { meaning_remove }
36658 }
36659 {
36660   \use:x
36661   {
36662     \__kernel_patch_deprecation:nnNNpn { 2022-01-11 } { \peek_remove_spaces:n }
36663     \cs_gset_protected:Npn \exp_not:c { peek_ #1 _ignore_spaces:NTF } #####1####2####3
36664     {
36665       \peek_remove_spaces:n
36666       { \exp_not:c { peek_ #1 :NTF } #####1 {####2} {####3} }
36667     }
36668     \__kernel_patch_deprecation:nnNNpn { 2022-01-11 } { \peek_remove_spaces:n }
36669     \cs_gset_protected:Npn \exp_not:c { peek_ #1 _ignore_spaces:NT } #####1####2
36670     {
36671       \peek_remove_spaces:n
36672       { \exp_not:c { peek_ #1 :NT } #####1 {####2} }
36673     }
36674     \__kernel_patch_deprecation:nnNNpn { 2022-01-11 } { \peek_remove_spaces:n }
36675     \cs_gset_protected:Npn \exp_not:c { peek_ #1 _ignore_spaces:NF } #####1####2
36676     {
36677       \peek_remove_spaces:n
36678       { \exp_not:c { peek_ #1 :NF } #####1 {####2} }
36679     }
36680   }
36681 }

```

(End definition for \peek_catcode_ignore_spaces:NTF and others. These functions are documented on page ??.)

36682 `\endpackage`

Index

The italic numbers denote the pages where the corresponding entry is described, numbers underlined point to the definition, all others indicate the places where it is used.

Symbols	
!	252
\"	18588, 18591, 29389, 31734, 31758, 31764, 31768, 31774, 31778, 31784, 31790, 31797, 31798, 31804, 31808, 31810, 31919
\#	10501, 13709, 18588, 31603
\\$	5384, 13708, 18588, 18591, 31603
\%	10503, 13710, 18588, 31603
\&	9308, 13701, 18588, 18591
&&	251
\'	29389, 31728, 31755, 31762, 31766, 31771, 31776, 31779, 31781, 31788, 31793, 31794, 31801, 31806, 31809, 31817, 31818, 31865, 31866, 31873, 31874, 31885, 31886, 31891, 31892, 31920, 31921, 31943, 31944, 31947, 31948, 31949, 31950
\(11003, 11189, 11264, 29410
\)	29410
*	252
*	13394, 13417, 18965, 18967, 18971, 18979
**	252
+	251, 252
\,	20586, 31615
-	251, 252
\-	189
\.	29389, 31733, 31821, 31822, 31831, 31832, 31841, 31842, 31859, 31871, 31872, 31922, 31923, 31953, 31954, 31957, 31958
/	252
\/	188, 4263
\:	13707
\::	42, 380, 407, 2267, 2268, <u>2269</u> , 2270, 2271, 2272, 2273, 2275, 2279, 2280, 2283, 2286, 2293, 2296, 2299, 2305, 2392, 2468, 2470, 2475, 2482, 2486, 2489, 2494, 2509, 2550, 2551, 2552, 2553, 2564
\::N	42, <u>2271</u> , 2392, 2553
\::V	42, <u>2299</u>
\::V_unbraced	42, <u>2467</u>
\::c	42, <u>2273</u>
\::e	42, <u>2277</u> , 2392
\::e_unbraced	42, <u>2467</u> , 2509
\::error	83
\::f	42, <u>2286</u> , 2552
\::f_unbraced	42, <u>2467</u>
\::n	42, 780, <u>2270</u> , 2550, 2553
\::o	42, <u>2275</u> , 2551
\::o_unbraced	42, <u>2467</u> , 2550, 2551, 2552, 2553
\::p	42, 380, <u>2272</u>
\::v	42, <u>2299</u>
\::v_unbraced	42, <u>2467</u>
\::x	42, <u>2293</u>
\::x_unbraced	42, <u>2467</u> , 2564
<	251
=	251
\=	20587, 29389, 31731, 31811, 31812, 31827, 31828, 31850, 31851, 31852, 31879, 31880, 31905, 31906, 31959, 31960
>	251
?	251
?:	251
\???	609
\\	2185, 3684, 3687, 3688, 3712, 3713, 3720, 3721, 4208, 4398, 4722, 4723, 6100, 6107, 6108, 6109, 6233, 8051, 8055, 8060, 8094, 8103, 8107, 8112, 8132, 8134, 8135, 8137, 8140, 8142, 8147, 8149, 8151, 8156, 8160, 8163, 8167, 8169, 8173, 8175, 8181, 8183, 8187, 8189, 8193, 8198, 8200, 8242, 8244, 8249, 8251, 8257, 8262, 8263, 8267, 8271, 8281, 8284, 8288, 8289, 8293, 8301, 8327, 8372, 9211, 9229, 9231, 9236, 9237, 9261, 9271, 9278, 9293, 9700, 9708, 9715, 9727, 9728, 9743, 9744, 9751, 9765, 9768, 9769, 9801, 9829, 9862, 9863, 9876, 9924, 9925, 9933, 9940, 9941, 9963, 9979, 9983, 9988, 9995, 10506, 11505, 11509, 11510, 11512, 11518, 11520, 11525, 11526, 11528, 11529, 11531, 11533, 11545, 11555, 11557, 11558, 11559, 13703, 14257, 14258, 14261, 14583, 14586, 14587, 14588, 14589, 14594, 14600, 14605, 14612, 14771, 14774, 14775, 14776, 14778,

- 14784, 14789, 14794, 14945, 14952,
18588, 21782, 21794, 21800, 22856,
22859, 22860, 22861, 22868, 22871,
22872, 29056, 29058, 29059, 29062,
29064, 29065, 29068, 29070, 29071,
29072, 29076, 29083, 31601, 33905,
35447, 35449, 35476, 35478, 35498,
35500, 35514, 35516, 35523, 35525,
35532, 35534, 35547, 35549, 36126
- $\backslash\{$ 4671, 8055,
8060, 8107, 8149, 8151, 8163, 8200,
8289, 8293, 10500, 13704, 14262,
18588, 29417, 29418, 29419, 31603
- $\backslash\}$... 85, 8054, 8060, 8164, 8200, 8289,
8293, 10502, 13705, 14262, 18588,
29417, 29418, 29419, 29420, 31603
- $\langle scope \rangle$ commands:
 $\backslash\langle scope \rangle_tmpa_ \langle type \rangle$ 5
 $\backslash\langle scope \rangle_tmpb_ \langle type \rangle$ 5
- $\backslash^$ 80, 1872,
2594, 3786, 3859, 3862, 4617, 4622,
4623, 4624, 4625, 4628, 4639, 4676,
4730, 4732, 4734, 4736, 4738, 4740,
5383, 7275, 7278, 7292, 7295, 7304,
7307, 7310, 7313, 7327, 7330, 8679,
9315, 10418, 10454, 13706, 14376,
14377, 14712, 14713, 14894, 14895,
14896, 18588, 18591, 18593, 18599,
18650, 26742, 29389, 31729, 31756,
31763, 31767, 31772, 31777, 31782,
31789, 31795, 31796, 31802, 31807,
31819, 31820, 31837, 31838, 31845,
31846, 31860, 31861, 31862, 31893,
31894, 31915, 31916, 31917, 31918
- \wedge 252
- $\backslash_$ 13712, 18588, 18591, 31603
- $\backslash^$.. 29389, 31727, 31754, 31761, 31765,
31770, 31775, 31780, 31787, 31791,
31792, 31800, 31805, 31945, 31946
- $\|$ 251
- $\backslash\sim$ 85, 4661, 4665,
4671, 10504, 12159, 13711, 18588,
18591, 29389, 31607, 31730, 31757,
31769, 31773, 31783, 31799, 31803,
31847, 31848, 31849, 31903, 31904
- $\backslash\sqcup$ 85, 87, 90, 92,
187, 1776, 3735, 3903, 4616, 4621,
4665, 4675, 4859, 7324, 9745, 9910,
10507, 11529, 13394, 13417, 14262,
14586, 14587, 14588, 18588, 18956,
29145, 29415, 29416, 31614, 36266
- ## A
- $\backslash A$ 13395, 13418
- $\backslash AA$ 29393, 31292, 31652
- $\backslash aa$ 29393, 31292, 31662
- $\backslash above$ 190
- $\backslash above displays short skip$ 191
- $\backslash above displays skip$ 192
- $\backslash above with delims$ 193
- abs 252
- $\backslash accent$ 194
- $acos$ 254
- $acosd$ 254
- $acot$ 255
- $acotd$ 255
- $acsc$ 254
- $acscd$ 254
- $\backslash adjdemerits$ 195
- $\backslash adjust spacing$ 908
- $\backslash advance$ 196
- $\backslash AE$ 29394, 31293, 31653, 31947
- $\backslash ae$ 29394, 31293, 31663, 31948
- $\backslash after assignment$ 197
- $\backslash after group$ 198
- $\backslash align mark$ 783
- $\backslash align tab$ 784
- $asec$ 254
- $asecd$ 254
- $asin$ 254
- $asind$ 254
- $atan$ 255
- $atand$ 255
- $\backslash AtBeginDocument$ 649, 11413
- $\backslash atop$ 199
- $\backslash atop with delims$ 200
- $\backslash attribute$ 785
- $\backslash attributedef$ 786
- $\backslash automatic discretionary$ 787
- $\backslash automatic hyphen mode$ 789
- $\backslash automatic hyphen penalty$ 790
- $\backslash autospacing$ 1107
- $\backslash autoxspacing$ 1108
- ## B
- $\backslash b$ 29389, 31741
- $\backslash badness$ 201
- $\backslash baselineskip$ 202
- $\backslash batchmode$ 203
- $\backslash begin$ 11499,
11502, 24826, 29406, 29413, 31598
- $\backslash begin cs name$ 792
- $\backslash begin group$ 3,
20, 24, 29, 33, 52, 84, 89, 103, 181, 204
- $\backslash begin L$ 512
- $\backslash begin R$ 513

- \belowdisplayshortskip 205
- \belowdisplayskip 206
- \bfseries 31577
- \binoppenalty 207
- \bodydir 793
- \bodydirection 794
- bool commands:
 - \bool_case_false:n 304, 35845
 - \bool_case_false:nTF
 - 304, 35845, 35855, 35857
 - \bool_case_true:n 304, 35845
 - \bool_case_true:nTF
 - 304, 35845, 35847, 35849
 - \bool_const:Nn 65, 8381
 - \bool_do_until:Nn 68, 8589
 - \bool_do_until:nn 69, 8595
 - \bool_do_while:Nn 69, 8589
 - \bool_do_while:nn 69, 8595
 - .bool_gset:N 223, 21094
 - \bool_gset:Nn 65, 8403
 - \bool_gset_eq:NN . 65, 4607, 6758, 8399
 - \bool_gset_false:N
 - . 65, 6705, 8387, 13916, 13925, 35841
 - .bool_gset_inverse:N 223, 21102
 - \bool_gset_inverse:N 303, 35837
 - \bool_gset_true:N
 - 65, 6771, 8387, 8852, 13906, 35597, 35619, 35626, 35841, 36616
 - \bool_if:NnTF 65, 145, 2053, 5499, 5508, 5949, 6104, 6190, 6208, 6226, 6380, 6599, 6607, 6840, 7443, 7466, 7539, 7761, 7931, 7937, 7978, 8345, 8350, 8420, 8429, 8584, 8586, 8590, 8592, 8850, 10721, 10728, 13920, 13929, 19609, 19617, 20623, 20786, 21015, 21024, 21070, 21286, 21288, 21290, 21336, 21338, 21340, 21378, 21380, 21382, 21398, 21400, 21402, 21441, 21482, 21501, 21503, 21508, 21515, 21579, 21608, 21618, 21646, 30104, 33077, 33772, 35602, 35675, 35838, 35841, 35863, 36614
 - \bool_if:nTF 65, 67, 69, 879, 6193, 8431, 8463, 8534, 8541, 8560, 8567, 8576, 8597, 8606, 8610, 8619, 8692, 11898, 11903, 35868, 35874, 35928
 - \bool_if_exist:NnTF .. 66, 8459, 20809
 - \bool_if_exist_p:N 66, 8459
 - \bool_if_p:N 65, 8420
 - \bool_if_p:n
 - 67, 570, 8384, 8406, 8411, 8463, 8471, 8541, 8567, 8573, 8577
 - \bool_lazy_all:nTF . 67, 68, 5782, 8521
 - \bool_lazy_all_p:n 68, 8521
 - \bool_lazy_and:nnTF . 67, 68, 8538, 8764, 8980, 10303, 11486, 28882, 29668, 29715, 30565, 30618, 30646, 31487, 32801, 33685, 35590, 35636
 - \bool_lazy_and_p:nn
 - ... 67, 68, 8538, 30093, 35647, 35659
 - \bool_lazy_any:nTF
 - 67, 68, 8547, 11101, 13758, 14030, 14054, 14076, 14246, 29376, 29479
 - \bool_lazy_any_p:n 67, 68, 8547, 30568
 - \bool_lazy_or:nnTF 67, 68, 3792, 8564, 8747, 8820, 15030, 18949, 29007, 29033, 29096, 29284, 29496, 29504, 29706, 30032, 30090, 30106, 30111, 30146, 30207, 30243, 30306, 30345, 30449, 30463, 30497, 30504, 30582, 30627, 30660, 30691, 30738, 30761, 30797, 31522, 31616, 31701, 34093, 34398, 35644, 35656
 - \bool_lazy_or_p:nn 68, 8564, 29718, 30348, 30466, 30649, 31490
 - \bool_log:N 66, 8436
 - \bool_log:n 66, 8432
 - \bool_new:N
 - 65, 4464, 4894, 6674, 6675, 6677, 6678, 6679, 8379, 8455, 8456, 8457, 8458, 8847, 10449, 13769, 20450, 20689, 20690, 20697, 20698, 20702, 20809, 29778, 32674, 35589
 - \bool_not_p:n 68, 8573
 - .bool_set:N 223, 21094
 - \bool_set:Nn 65, 562, 8403
 - \bool_set_eq:NN
 - .. 65, 4601, 6919, 8399, 19627, 19629
 - \bool_set_false:N .. 65, 160, 5473, 5678, 6649, 6720, 6734, 6797, 6839, 8387, 10555, 10697, 10705, 10713, 10723, 10730, 20728, 21280, 21281, 21282, 21332, 21333, 21337, 21373, 21381, 21383, 21392, 21393, 21403, 21424, 21489, 33073, 33770, 35838
 - .bool_set_inverse:N 223, 21102
 - \bool_set_inverse:N 303, 35837
 - \bool_set_true:N ... 65, 174, 5478, 5682, 6643, 6837, 6918, 8387, 10683, 20723, 21287, 21289, 21291, 21331, 21339, 21341, 21374, 21375, 21379, 21394, 21399, 21401, 21419, 21496, 29779, 33091, 33113, 33144, 35838
 - \bool_show:N 66, 8436
 - \bool_show:n 66, 8432
 - \bool_to_str:N 65, 8429
 - \bool_to_str:n .. 65, 8429, 8433, 8435
 - \bool_until_do:Nn 69, 8583

- \bool_until_do:nn [69](#), [8595](#)
- \bool_while_do:Nn [69](#), [8583](#)
- \bool_while_do:nn [69](#), [8595](#)
- \bool_xor:nnTF [68](#), [8574](#)
- \bool_xor_p:nn [68](#), [8574](#)
- \c_false_bool [26](#), [64](#), [365](#), [400](#), [549](#),
[563](#), [566–568](#), [1627](#), [1679](#), [1680](#),
[1711](#), [1733](#), [1738](#), [1770](#), [1789](#), [1990](#),
[1997](#), [2927](#), [3201](#), [5159](#), [5177](#), [5371](#),
[5418](#), [5717](#), [5919](#), [5936](#), [5949](#), [6127](#),
[6263](#), [6768](#), [7866](#), [7875](#), [7884](#), [7894](#),
[7956](#), [7964](#), [8379](#), [8390](#), [8394](#), [8447](#),
[8512](#), [8535](#), [8541](#), [8559](#), [8703](#), [19611](#),
[19619](#), [21038](#), [21040](#), [21047](#), [21052](#),
[35605](#), [35854](#), [35856](#), [35858](#), [35860](#)
- \g_tmpa_bool [66](#), [8455](#)
- \l_tmpa_bool [66](#), [8455](#)
- \g_tmpb_bool [66](#), [8455](#)
- \l_tmpb_bool [66](#), [8455](#)
- \c_true_bool
... [26](#), [64](#), [365](#), [563](#), [566–568](#), [678](#),
[1679](#), [1711](#), [1770](#), [1788](#), [2011](#), [4471](#),
[4604](#), [5042](#), [5116](#), [5173](#), [5361](#), [5363](#),
[5365](#), [5367](#), [5369](#), [5379](#), [5417](#), [5424](#),
[5917](#), [5927](#), [5949](#), [5950](#), [6125](#), [6246](#),
[6248](#), [6271](#), [6366](#), [6537](#), [6548](#), [6563](#),
[6724](#), [7490](#), [7607](#), [7720](#), [8388](#), [8392](#),
[8446](#), [8513](#), [8514](#), [8533](#), [8561](#), [8567](#),
[8697](#), [19610](#), [19618](#), [19627](#), [21045](#),
[21054](#), [35846](#), [35848](#), [35850](#), [35852](#)
- bool internal commands:
 - __bool_!:Nw [8492](#)
 - __bool_&_0: [8504](#)
 - __bool_&_1: [8504](#)
 - __bool_&_2: [8504](#)
 - __bool_(:Nw [8497](#)
 - __bool_)_0: [8504](#)
 - __bool_)_1: [8504](#)
 - __bool_)_2: [8504](#)
 - __bool_case:NnTF [35845](#)
 - __bool_case_end:nw [35845](#)
 - __bool_case_false:w [35845](#)
 - __bool_case_true:w [35845](#)
 - __bool_choose:NNN .. [8499](#), [8503](#), [8504](#)
 - __bool_get_next:NN
... [566](#), [567](#), [8479](#), [8482](#), [8494](#), [8500](#),
[8515](#), [8516](#), [8517](#), [8518](#), [8519](#), [8520](#)
 - __bool_if_p:n [8471](#)
 - __bool_if_p_aux:w [566](#), [8471](#)
 - __bool_if_recursion_tail_stop-
do:nn [8419](#), [8533](#), [8559](#)
 - __bool_lazy_all:n [8521](#)
 - __bool_lazy_any:n [8547](#)
 - __bool_p:Nw [8502](#)
 - __bool_show:NN [8436](#)
 - __bool_use_i_delimit_by_q-
recursion_stop:nw [8417](#), [8535](#), [8561](#)
 - __bool_|_0: [8504](#)
 - __bool_|_1: [8504](#)
 - __bool_|_2: [8504](#)
 - \botmark [208](#)
 - \botmarks [514](#)
 - \box [209](#)
 - box commands:
 - \box_autosize_to_wd_and_ht:Nnn ..
..... [279](#), [32580](#)
 - \box_autosize_to_wd_and_ht_plus_-
dp:Nnn [279](#), [32580](#)
 - \box_clear:N [270](#),
[271](#), [31978](#), [31985](#), [32709](#), [32796](#), [32873](#)
 - \box_clear_new:N [271](#), [31984](#)
 - \box_clip:N [301](#), [35695](#)
 - \box_dp:N
... [272](#), [1259](#), [23344](#), [32006](#), [32013](#),
[32018](#), [32022](#), [32325](#), [32454](#), [32569](#),
[32588](#), [32594](#), [32910](#), [32911](#), [33017](#),
[33022](#), [33050](#), [33064](#), [33235](#), [33513](#),
[33534](#), [33837](#), [35715](#), [35722](#), [35727](#)
 - \box_gautosize_to_wd_and_ht:Nnn ..
..... [279](#), [32580](#)
 - \box_gautosize_to_wd_and_ht_-
plus_dp:Nnn [279](#), [32580](#)
 - \box_gclear:N [270](#), [31978](#), [31987](#), [32718](#)
 - \box_gclear_new:N [271](#), [31984](#)
 - \box_gclip:N [301](#), [35695](#)
 - \box_gresize_to_ht:Nn ... [279](#), [32473](#)
 - \box_gresize_to_ht_plus_dp:Nn ...
..... [280](#), [32473](#)
 - \box_gresize_to_wd:Nn ... [280](#), [32473](#)
 - \box_gresize_to_wd_and_ht:Nnn ...
..... [280](#), [32473](#)
 - \box_gresize_to_wd_and_ht_plus_-
dp:Nnn [280](#), [32424](#), [33349](#)
 - \box_grotate:Nn ... [281](#), [32306](#), [33184](#)
 - \box_gscale:Nnn ... [281](#), [32551](#), [33391](#)
 - \box_gset_dp:Nn [272](#), [32015](#)
 - \box_gset_eq:NN
[271](#), [31981](#), [31990](#), [32895](#), [35705](#), [35756](#)
 - \box_gset_eq_clear:NN [36387](#)
 - \box_gset_eq_drop:NN [278](#), [31996](#), [36388](#)
 - \box_gset_ht:Nn [272](#), [32015](#)
 - \box_gset_to_last:N [273](#), [32069](#)
 - \box_gset_trim:Nnnnn ... [301](#), [35701](#)
 - \box_gset_viewport:Nnnnn . [301](#), [35752](#)
 - \box_gset_wd:Nn [272](#), [32015](#)
 - \box_ht:N
... [272](#), [1259](#), [23343](#), [32006](#), [32013](#),
[32027](#), [32031](#), [32324](#), [32453](#), [32568](#),

- 32581, 32584, 32588, 32594, 32791,
- 32868, 32912, 32913, 33008, 33013,
- 33050, 33057, 33229, 33233, 33512,
- 33533, 33835, 35732, 35740, 35745
- \box_ht_plus_dp:N 272, 32012
- \box_if_empty:NTF 273, 32065
- \box_if_empty_p:N 273, 32065
- \box_if_exist:NTF
..... 271, 31985, 31987, 32002, 32100
- \box_if_exist_p:N 271, 32002
- \box_if_horizontal:NTF ... 273, 32057
- \box_if_horizontal_p:N ... 273, 32057
- \box_if_vertical:NTF 273, 32057
- \box_if_vertical_p:N 273, 32057
- \box_log:N 274, 32086
- \box_log:Nnn 274, 32086
- \box_move_down:nn 271, 1356, 32046,
33208, 35719, 35727, 35770, 35777
- \box_move_left:nn 271, 32046
- \box_move_right:nn 271, 32046
- \box_move_up:nn 271, 32046, 33553,
33832, 35736, 35745, 35784, 35797
- \box_new:N 270,
271, 31972, 32075, 32076, 32077,
32078, 32079, 32305, 32650, 32725
- \box_resize:Nnn 36391
- \box_resize_to_ht:Nn 279, 32473
- \box_resize_to_ht_plus_dp:Nn ...
..... 280, 32473
- \box_resize_to_wd:Nn 280, 32473
- \box_resize_to_wd_and_ht:Nnn ...
..... 280, 32473
- \box_resize_to_wd_and_ht_plus_
dp:Nnn ... 280, 32424, 33342, 36392
- \box_rotate:Nn 281, 32306, 33181
- \box_scale:Nnn 281, 32551, 33388
- \box_set_dp:Nn
..... 272, 1357, 32015, 32351,
32622, 32625, 33213, 33513, 33534,
33836, 35722, 35730, 35773, 35778
- \box_set_eq:NN . 271, 31979, 31990,
32883, 33536, 33840, 35702, 35753
- \box_set_eq_clear:NN 36389
- \box_set_eq_drop:NN 278, 31996, 36390
- \box_set_ht:Nn . 272, 32015, 32350,
32621, 32626, 33211, 33512, 33533,
33834, 35739, 35748, 35787, 35800
- \box_set_to_last:N 273, 32069
- \box_set_trim:Nnnnn 301, 35701
- \box_set_viewport:Nnnnn .. 301, 35752
- \box_set_wd:Nn . 272, 32015, 32352,
32638, 33214, 33514, 33535, 33838
- \box_show:N 274, 277, 286, 32080
- \box_show:Nnn
.. 274, 286, 1309, 32080, 33874, 33877
- \box_use:N
.. 271, 32042, 32339, 33209, 33550,
33553, 33829, 33832, 35712, 35763
- \box_use_clear:N 36393
- \box_use_drop:N 278, 32042,
32354, 32633, 32642, 33216, 33636,
33764, 35720, 35728, 35737, 35746,
35771, 35777, 35785, 35798, 36394
- \box_wd:N 272,
23342, 32006, 32036, 32040, 32326,
32455, 32570, 32602, 32914, 32915,
33012, 33021, 33039, 33044, 33232,
33240, 33434, 33441, 33467, 33514,
33535, 33551, 33830, 33839, 35764
- \c_empty_box
..... 270, 273, 31979, 31981, 32075
- \g_tmpa_box 273, 32076
- \l_tmpa_box 273, 32076
- \g_tmpb_box 273, 32076
- \l_tmpb_box 273, 32076
- box internal commands:
- \l__box_angle_fp
.. 32294, 32316, 32317, 32318, 32347
- __box_autosize:NnnnN 32580
- __box_backend_clip:N . 35696, 35699
- __box_backend_rotate:Nn 32345
- __box_backend_scale:Nnn 32614
- \l__box_bottom_dim 32297,
32325, 32382, 32386, 32391, 32397,
32402, 32406, 32415, 32417, 32446,
32454, 32463, 32507, 32569, 32575
- \l__box_bottom_new_dim
32301, 32351, 32383, 32394, 32405,
32416, 32462, 32574, 32622, 32626
- \l__box_cos_fp 32295,
32318, 32330, 32335, 32362, 32374
- __box_dim_eval:n
..... 1259, 31967, 31971,
32013, 32018, 32022, 32027, 32031,
32036, 32040, 32047, 32049, 32051,
32053, 32132, 32137, 32164, 32170,
32178, 32202, 32236, 32241, 32269,
32275, 32286, 32291, 35773, 35797
- __box_dim_eval:w 31967
- \l__box_internal_box 32305, 32339,
32340, 32346, 32350, 32351, 32352,
32354, 32612, 32621, 32622, 32625,
32626, 32633, 32638, 32642, 35709,
35717, 35720, 35722, 35725, 35728,
35730, 35732, 35734, 35737, 35739,
35740, 35743, 35745, 35746, 35748,
35750, 35760, 35768, 35771, 35773,

- 35776, 35777, 35778, 35782, 35785,
 35787, 35795, 35798, 35800, 35802
 \l_box_left_dim ... [32297](#), [32327](#),
 32382, 32384, 32393, 32397, 32402,
 32408, 32413, 32417, 32456, 32571
 \l_box_left_new_dim [32301](#), [32342](#),
 32353, 32385, 32396, 32407, 32418
 _box_log:nNnn ... [32086](#)
 _box_resize:N ...
 .. [32424](#), [32490](#), [32510](#), [32527](#), [32548](#)
 _box_resize:NNN ... [32424](#)
 _box_resize_common:N ...
 .. [32466](#), [32578](#), [32610](#)
 _box_resize_set_corners:N ...
 .. [32424](#), [32483](#), [32503](#), [32523](#), [32540](#)
 _box_resize_to_ht:NnN ... [32473](#)
 _box_resize_to_ht_plus_dp:NnN ...
 .. [32473](#)
 _box_resize_to_wd:NnN ... [32473](#)
 _box_resize_to_wd_and_ht:NnnN ...
 .. [32531](#), [32534](#), [32536](#)
 _box_resize_to_wd_and_ht_plus_
 dp:NnnN ... [32424](#)
 _box_resize_to_wd_ht:NnnN .. [32473](#)
 \l_box_right_dim .. [32297](#), [32326](#),
 32380, 32386, 32391, 32395, 32404,
 32406, 32415, 32419, 32442, 32455,
 32461, 32525, 32542, 32570, 32577
 \l_box_right_new_dim ... [32301](#),
 32353, 32387, 32398, 32409, 32420,
 32460, 32576, 32630, 32632, 32638
 _box_rotate:N ... [32306](#)
 _box_rotate:NnN ... [32306](#)
 _box_rotate_quadrant_four: ...
 .. [32306](#), [32411](#)
 _box_rotate_quadrant_one: ...
 .. [32306](#), [32378](#)
 _box_rotate_quadrant_three: ...
 .. [32306](#), [32400](#)
 _box_rotate_quadrant_two: ...
 .. [32306](#), [32389](#)
 _box_rotate_xdir:nnN ...
 32306, 32356, 32384, 32386, 32395,
 32397, 32406, 32408, 32417, 32419
 _box_rotate_ydir:nnN ...
 32306, 32367, 32380, 32382, 32391,
 32393, 32402, 32404, 32413, 32415
 _box_scale:N ... [32551](#), [32607](#)
 _box_scale:NnnN ... [32551](#)
 \l_box_scale_x_fp ... [32422](#),
 32441, 32461, 32489, 32509, 32524,
 32526, 32541, 32561, 32577, 32602,
 32604, 32605, 32606, 32616, 32628
 \l_box_scale_y_fp ...
 [32422](#), [32443](#), [32463](#), [32465](#),
 32484, 32489, 32504, 32509, 32526,
 32543, 32562, 32573, 32575, 32603,
 32604, 32605, 32606, 32617, 32619
 _box_set_trim:NnnnnN ... [35701](#)
 _box_set_viewport:NnnnnN ...
 .. 35753, 35756, 35758
 _box_show:NnN . [32084](#), [32094](#), [32098](#)
 \l_box_sin_fp ...
 .. [32295](#), [32317](#), [32328](#), [32363](#), [32373](#)
 \l_box_top_dim [32297](#), [32324](#), [32380](#),
 32384, 32393, 32395, 32404, 32408,
 32413, 32419, 32446, 32453, 32465,
 32487, 32507, 32546, 32568, 32573
 \l_box_top_new_dim ...
 32301, 32350, 32381, 32392, 32403,
 32414, 32464, 32572, 32621, 32625
 _box_viewport:NnnnnN ... [35752](#)
 \boxdir ... [795](#)
 \boxdirection ... [796](#)
 \boxmaxdepth ... [210](#)
 bp ... [257](#)
 \breakafterdirmode ... [797](#)
 \brokenpenalty ... [211](#)
- ## C
- \c .. [29389](#), [31739](#), [31760](#), [31786](#), [31843](#),
 31844, 31863, 31864, 31867, 31868,
 31875, 31876, 31887, 31888, 31895,
 31896, 31899, 31900, 31955, 31956
 \catcode ... [87](#), [120](#), [121](#), [122](#), [123](#),
 124, 125, 126, 127, 128, 132, 133,
 134, 135, 136, 137, 138, 139, 140, 212
 \catcodetable ... [798](#)
 cc ... [257](#)
 cctab commands:
 \cctab_begin:N ... [261](#),
 262, 1181, 1182, 1184–1188, 28848
 \cctab_const:Nn ...
 261, 262, 28980, 28987, 28994, 29038
 \cctab_end: ... [261](#), [262](#), [1181](#),
 1182, 1184, 1185, 1187, 1188, 28862
 \cctab_gset:Nn [261](#), [262](#), 28778, 28983
 \cctab_if_exist:NTF [262](#), 28935, 28942
 \cctab_if_exist_p:N ... [262](#), 28935
 \cctab_item:Nn ... [262](#), 28919
 \cctab_new:N ...
 261, 1181, 1182, 28713, 28982, 28986
 \cctab_select:N ...
 [117](#), [118](#), [261](#), [262](#), 13987,
 28783, 28800, 28989, 28996, 29040
 \c_code_cctab ... [262](#), 13987, 28999
 \c_document_cctab ... [262](#), [1184](#), 28999

- \c_initex_cctab ... [262](#), [28783](#), [28986](#)
- \c_other_cctab [262](#), [28986](#)
- cctab internal commands:
 - \g_cctab_allocate_int
..... [28709](#), [28841](#), [28843](#), [28845](#)
 - __cctab_begin_aux:
..... [1185](#), [1186](#), [28829](#), [28853](#)
 - __cctab_chk_group_begin:n
..... [1187](#), [28854](#), [28873](#)
 - __cctab_chk_group_end:n
..... [1187](#), [28867](#), [28873](#)
 - __cctab_chk_if_valid:NTF
..... [28780](#), [28801](#), [28850](#), [28939](#)
 - __cctab_chk_if_valid_aux:NTF . [28939](#)
 - \g_cctab_endlinechar_prop
... [1183](#), [28712](#), [28759](#), [28761](#), [28808](#)
 - \g_cctab_group_seq
..... [28708](#), [28875](#), [28881](#)
 - __cctab_gset:n
..... [28751](#), [28785](#), [28857](#), [29036](#)
 - __cctab_gset_aux:n [28751](#)
 - __cctab_gstore:Nnn [28713](#)
 - \l_cctab_internal_a_tl
... [1185](#), [1186](#), [28710](#), [28808](#), [28809](#),
[28834](#), [28844](#), [28852](#), [28855](#), [28856](#),
[28857](#), [28864](#), [28866](#), [28868](#), [28869](#)
 - \l_cctab_internal_b_tl
..... [28710](#), [28881](#), [28885](#), [28892](#)
 - \g_cctab_internal_cctab [28790](#)
 - __cctab_internal_cctab_name: ...
... [28790](#), [28811](#), [28812](#), [28813](#), [28814](#)
 - __cctab_item:nN . [28920](#), [28923](#), [28927](#)
 - __cctab_nesting_number:N
..... [28855](#), [28868](#), [28897](#)
 - __cctab_nesting_number:w [28897](#)
 - __cctab_new:N . [1182](#), [1185](#), [28713](#),
[28792](#), [28812](#), [28833](#), [28842](#), [29003](#)
 - \g_cctab_next_cctab [28829](#)
 - __cctab_select:N
..... [1185](#), [28800](#), [28858](#), [28869](#)
 - \g_cctab_stack_seq
... [1181](#), [28706](#), [28856](#), [28864](#), [28915](#)
 - \g_cctab_unused_seq
... [1181](#), [1186](#), [28706](#), [28852](#), [28866](#)
- ceil [253](#)
- \char [213](#), [19144](#)
- char commands:
 - \l_char_active_seq ... [85](#), [186](#), [18586](#)
 - \char_fold_case:N [36637](#)
 - \char_foldcase:N
..... [183](#), [18835](#), [36643](#), [36644](#)
 - \char_generate:nn [117](#),
[182](#), [427](#), [445](#), [446](#), [532](#), [672](#), [730](#),
[1371](#), [3738](#), [3739](#), [3740](#), [3741](#), [3742](#),
[3744](#), [3745](#), [3746](#), [4309](#), [4325](#), [4337](#),
[4755](#), [5792](#), [6148](#), [12140](#), [12156](#),
[13830](#), [14085](#), [14101](#), [18613](#), [18828](#),
[18832](#), [18863](#), [18892](#), [18947](#), [18956](#),
[29288](#), [29326](#), [30048](#), [30058](#), [30059](#),
[30216](#), [30309](#), [30310](#), [30481](#), [30492](#),
[30539](#), [30540](#), [30541](#), [30552](#), [30577](#),
[30605](#), [30632](#), [30655](#), [30672](#), [30684](#),
[30696](#), [30701](#), [30726](#), [30744](#), [30773](#),
[30775](#), [30779](#), [30817](#), [30818](#), [30823](#),
[30825](#), [30833](#), [30834](#), [30839](#), [30841](#),
[31061](#), [31062](#), [31067](#), [31069](#), [31098](#),
[31099](#), [31116](#), [31117](#), [31118](#), [31123](#),
[31125](#), [31127](#), [31135](#), [31136](#), [31137](#),
[31142](#), [31144](#), [31146](#), [31258](#), [31259](#),
[31260](#), [31265](#), [31267](#), [31625](#), [31646](#),
[31648](#), [31707](#), [31721](#), [31723](#), [34499](#)
 - \char_gset_active_eq:NN .. [182](#), [18592](#)
 - \char_gset_active_eq:nN .. [182](#), [18592](#)
 - \char_lower_case:N [36637](#)
 - \char_lowercase:N
..... [183](#), [18835](#), [36637](#), [36638](#)
 - \char_mixed_case:N [36642](#)
 - \char_mixed_case:Nn [36637](#)
 - \char_set_active_eq:NN
..... [182](#), [3735](#), [18592](#)
 - \char_set_active_eq:nN
..... [182](#), [4265](#), [4266](#), [18592](#)
 - \char_set_catcode:nn .. [184](#), [149](#),
[150](#), [151](#), [152](#), [153](#), [154](#), [155](#), [156](#),
[157](#), [18492](#), [18499](#), [18501](#), [18503](#),
[18505](#), [18507](#), [18509](#), [18511](#), [18513](#),
[18515](#), [18517](#), [18519](#), [18521](#), [18523](#),
[18525](#), [18527](#), [18529](#), [18531](#), [18533](#),
[18535](#), [18537](#), [18539](#), [18541](#), [18543](#),
[18545](#), [18547](#), [18549](#), [18551](#), [18553](#),
[18555](#), [18557](#), [18559](#), [18561](#), [28822](#)
 - \char_set_catcode_active:N
... [183](#), [3786](#), [7275](#), [9308](#), [18498](#),
[18593](#), [18650](#), [18979](#), [31607](#), [36265](#)
 - \char_set_catcode_active:n
... [184](#), [18530](#), [18701](#), [20586](#), [20587](#),
[29010](#), [29017](#), [29035](#), [29046](#), [29256](#)
 - \char_set_catcode_alignment:N ...
..... [183](#), [7327](#), [18498](#), [18967](#)
 - \char_set_catcode_alignment:n ...
..... [184](#), [167](#), [18530](#), [18685](#), [29023](#)
 - \char_set_catcode_comment:N
..... [183](#), [18498](#)
 - \char_set_catcode_comment:n
..... [184](#), [18530](#), [29022](#)
 - \char_set_catcode_end_line:N ...
..... [183](#), [18498](#)

- \char_set_catcode_end_line:n ...
..... [184](#), [18530](#), [29018](#)
- \char_set_catcode_escape:N [183](#), [18498](#)
- \char_set_catcode_escape:n
..... [184](#), [18530](#), [29025](#)
- \char_set_catcode_group_begin:N .
..... [183](#), [3859](#), [7278](#), [18498](#)
- \char_set_catcode_group_begin:n .
..... [184](#), [18530](#), [18678](#), [29028](#)
- \char_set_catcode_group_end:N ...
..... [183](#), [3862](#), [7295](#), [18498](#)
- \char_set_catcode_group_end:n ...
..... [184](#), [18530](#), [18680](#), [29030](#)
- \char_set_catcode_ignore:N [183](#), [18498](#)
- \char_set_catcode_ignore:n
.. [184](#), [164](#), [165](#), [18530](#), [29015](#), [29019](#)
- \char_set_catcode_invalid:N
..... [183](#), [18498](#)
- \char_set_catcode_invalid:n
.... [184](#), [18530](#), [29006](#), [29009](#), [29032](#)
- \char_set_catcode_letter:N
..... [183](#), [7304](#), [18498](#), [24967](#), [24968](#)
- \char_set_catcode_letter:n
..... [184](#), [168](#), [170](#), [18530](#),
[18697](#), [29012](#), [29014](#), [29024](#), [29027](#)
- \char_set_catcode_math_subscript:N
..... [183](#), [7292](#), [18498](#), [18971](#)
- \char_set_catcode_math_subscript:n
..... [184](#), [18530](#), [18692](#), [29045](#)
- \char_set_catcode_math_superscript:N
..... [183](#), [7330](#), [18498](#)
- \char_set_catcode_math_superscript:n
..... [184](#), [169](#), [18530](#), [18690](#), [29026](#)
- \char_set_catcode_math_toggle:N .
..... [183](#), [7307](#), [18498](#), [18965](#)
- \char_set_catcode_math_toggle:n .
..... [184](#), [18530](#), [18683](#), [29021](#)
- \char_set_catcode_other:N
..... [183](#), [1184](#),
[4019](#), [7310](#), [14376](#), [14377](#), [14712](#),
[14713](#), [14894](#), [14895](#), [14896](#), [18498](#)
- \char_set_catcode_other:n
..... [184](#), [166](#),
[171](#), [18530](#), [18652](#), [18699](#), [28992](#),
[29011](#), [29013](#), [29016](#), [29029](#), [29044](#)
- \char_set_catcode_parameter:N ...
..... [183](#), [7313](#), [18498](#)
- \char_set_catcode_parameter:n ...
..... [184](#), [18530](#), [18688](#), [29020](#)
- \char_set_catcode_space:N [183](#), [18498](#)
- \char_set_catcode_space:n
... [184](#), [172](#), [11431](#), [18530](#), [18695](#),
[28997](#), [29031](#), [29042](#), [29043](#), [29145](#)
- \char_set_lccode:nn
[184](#), [4248](#), [9304](#), [9305](#), [9306](#), [9307](#),
[18562](#), [18599](#), [18705](#), [18706](#), [36266](#)
- \char_set_mathcode:nn ... [185](#), [18562](#)
- \char_set_sfcode:nn [185](#), [18562](#)
- \char_set_uccode:nn [185](#), [18562](#)
- \char_show_value_catcode:n [184](#), [18492](#)
- \char_show_value_lccode:n [185](#), [18562](#)
- \char_show_value_mathcode:n
..... [185](#), [18562](#)
- \char_show_value_sfcode:n [186](#), [18562](#)
- \char_show_value_uccode:n [185](#), [18562](#)
- \l_char_special_seq [186](#), [18586](#)
- \char_str_fold_case:N [36637](#)
- \char_str_foldcase:N
..... [183](#), [18835](#), [36651](#), [36652](#)
- \char_str_lower_case:N [36637](#)
- \char_str_lowercase:N
..... [183](#), [18835](#), [36645](#), [36646](#)
- \char_str_mixed_case:N [36650](#)
- \char_str_mixed_case:Nn [36637](#)
- \char_str_titlecase:N
..... [183](#), [18835](#), [36649](#), [36650](#)
- \char_str_upper_case:N [36637](#)
- \char_str_uppercase:N
..... [183](#), [18835](#), [36647](#), [36648](#)
- \char_titlecase:N
..... [183](#), [18835](#), [36641](#), [36642](#)
- \char_to_nfd:N ... [308](#), [18813](#), [30252](#)
- \char_to_utfviii_bytes:n
..... [308](#), [15043](#), [18735](#),
[30784](#), [30807](#), [30808](#), [31075](#), [31076](#),
[31105](#), [31273](#), [31274](#), [31638](#), [31715](#)
- \char_upper_case:N [36637](#)
- \char_uppercase:N
.. [183](#), [18835](#), [30263](#), [30272](#), [30284](#),
[30288](#), [30299](#), [30316](#), [36639](#), [36640](#)
- \char_value_catcode:n
..... [184](#), [1189](#), [149](#), [150](#),
[151](#), [152](#), [153](#), [154](#), [155](#), [156](#), [157](#),
[12152](#), [12156](#), [18492](#), [18832](#), [28772](#),
[28931](#), [29289](#), [30727](#), [31626](#), [31708](#)
- \char_value_lccode:n . [185](#), [18562](#),
[18836](#), [18849](#), [18926](#), [18936](#), [29160](#)
- \char_value_mathcode:n ... [185](#), [18562](#)
- \char_value_sfcode:n ... [186](#), [18562](#)
- \char_value_uccode:n
..... [185](#), [18562](#), [18838](#), [18928](#)
- char internal commands:
 __char_change_case:NN [18835](#)
 __char_change_case:nN [18835](#)
 __char_change_case:NNN [18835](#)
 __char_change_case:nNN [18835](#)
 __char_change_case:NNNN [18835](#)

- _char_change_case_catcode:N [18828](#), [18835](#)
- _char_change_case_multi:nN . [18835](#)
- _char_change_case_multi:NNNw [18835](#)
- _char_data_auxi:w [29111](#),
[29151](#), [29156](#), [29184](#), [29189](#), [29222](#)
- _char_data_auxii:w
. [29117](#), [29121](#), [29167](#),
[29171](#), [29192](#), [29193](#), [29195](#), [29197](#)
- _char_data_auxiii:w . [29119](#), [29131](#)
- \g_char_data_ior . . [29095](#), [29110](#),
[29146](#), [29154](#), [29155](#), [29181](#), [29187](#),
[29188](#), [29211](#), [29224](#), [29240](#), [29241](#)
- _char_generate:n [29101](#),
[29126](#), [29128](#), [29140](#), [29163](#), [29175](#),
[29176](#), [29178](#), [29204](#), [29205](#), [29207](#)
- _char_generate_aux:nn [18613](#)
- _char_generate_aux:nnw [18613](#)
- _char_generate_aux:w . [18615](#), [18619](#)
- _char_generate_auxii:nnw . . . [18613](#)
- _char_generate_char:n . . [29099](#),
[29124](#), [29139](#), [29162](#), [29173](#), [29202](#)
- _char_generate_invalid_-
catcode: [18613](#)
- _char_int_to_roman:w
. [18612](#), [18710](#), [18729](#)
- _char_quark_if_no_value:NTF
. [18491](#), [18870](#), [18872](#)
- _char_quark_if_no_value_p:N . [18491](#)
- _char_str_change_case:nN . . . [18835](#)
- _char_str_change_case:nNN . . [18835](#)
- _char_tmp:n
. [18703](#), [18714](#), [18717](#), [18719](#)
- _char_tmp:NN . . [29229](#), [29235](#), [29237](#)
- _char_tmp:nN . . [18594](#), [18605](#), [18606](#)
- \l_char_tmpa_tl [18613](#)
- \l_char_tmpa_tl [29134](#),
[29135](#), [29137](#), [29146](#), [29148](#), [29151](#)
- \l_char_tmppb_tl [29136](#), [29137](#)
- _char_to_nfd:n [18813](#)
- _char_to_nfd:Nw [18813](#)
- _char_to_utfviii_bytes_auxi:n [18735](#)
- _char_to_utfviii_bytes_-
auxii:Nnn [18735](#)
- _char_to_utfviii_bytes_-
auxiii:n [18735](#)
- _char_to_utfviii_bytes_end: . [18735](#)
- _char_to_utfviii_bytes_-
output:nnn [18735](#)
- _char_to_utfviii_bytes_-
outputi:nw [18735](#)
- _char_to_utfviii_bytes_-
outputii:nw [18735](#)
- _char_to_utfviii_bytes_-
outputiii:nw [18735](#)
- _char_to_utfviii_bytes_-
outputiv:nw [18735](#)
- \chardef [1189](#), [130](#), [142](#), [214](#)
- choice commands:
.choice: [223](#), [21110](#)
- choices commands:
.choices:nn [223](#), [21112](#)
- \cite [29406](#), [29413](#)
- \cleaders [215](#)
- \clearmarks [799](#)
- clist commands:
\clist_clear:N
[172](#), [17861](#), [17878](#), [18041](#), [21319](#), [21361](#)
- \clist_clear_new:N [172](#), [17865](#)
- \clist_concat:NNN
. [173](#), [17904](#), [17930](#), [17943](#)
- \clist_const:Nn [172](#), [17858](#)
- \clist_count:N [177](#),
[179](#), [18285](#), [18318](#), [18383](#), [18449](#), [18460](#)
- \clist_count:n
[177](#), [18285](#), [18414](#), [18440](#), [18461](#), [35019](#)
- \clist_gclear:N . . . [172](#), [17861](#), [17880](#)
- \clist_gclear_new:N [172](#), [17865](#)
- \clist_gconcat:NNN
. [173](#), [17904](#), [17932](#), [17945](#)
- \clist_get:NN [178](#), [17955](#)
- \clist_get:NNTF [178](#), [17992](#)
- \clist_gpop:NN [179](#), [17966](#)
- \clist_gpop:NNTF [179](#), [17992](#)
- \clist_gpush:Nn [179](#), [18017](#)
- \clist_gput_left:Nn
. [173](#), [17929](#), [18025](#), [18026](#), [18027](#),
[18028](#), [18029](#), [18030](#), [18031](#), [18032](#)
- \clist_gput_right:Nn [173](#), [17942](#)
- \clist_gremove_all:Nn . . . [174](#), [18057](#)
- \clist_gremove_duplicates:N
. [174](#), [18035](#)
- \clist_greverse:N [174](#), [18096](#)
- .clist_gset:N [223](#), [21122](#)
- \clist_gset:Nn [173](#), [14114](#), [17923](#)
- \clist_gset_eq:NN . . [172](#), [17869](#), [18038](#)
- \clist_gset_from_seq:NN
. [172](#), [3450](#), [17877](#), [18060](#)
- \clist_gsort:Nn [175](#), [3435](#), [18114](#)
- \clist_if_empty:NTF
. [175](#), [17913](#), [18048](#), [18081](#), [18114](#),
[18171](#), [18210](#), [18242](#), [18448](#), [20903](#)
- \clist_if_empty:nTF [175](#), [18118](#)
- \clist_if_empty_p:N [175](#), [18114](#)
- \clist_if_empty_p:n [175](#), [18118](#)

- \clist_if_exist:NNTF
 [173](#), [11299](#), [11399](#), [17919](#), [18316](#)
- \clist_if_exist_p:N [173](#), [17919](#)
- \clist_if_in:NnTF
 [172](#), [175](#), [942](#), [18044](#), [18132](#)
- \clist_if_in:nnTF .. [175](#), [18132](#), [22729](#)
- \clist_item:Nn [179](#), [849](#), [18380](#), [18449](#)
- \clist_item:nn [179](#), [849](#), [18411](#), [18444](#)
- \clist_log:N [180](#), [18452](#)
- \clist_log:n [180](#), [18474](#)
- \clist_map_break: [176](#), [18176](#),
 [18188](#), [18197](#), [18198](#), [18220](#), [18247](#),
 [18260](#), [18268](#), [18269](#), [18281](#), [21543](#)
- \clist_map_break:n [177](#), [3443](#), [3449](#),
 [18152](#), [18281](#), [21497](#), [21572](#), [34996](#)
- \clist_map_function:NN
 [176](#), [844](#), [16252](#),
 [16262](#), [18155](#), [18169](#), [18290](#), [18465](#)
- \clist_map_function:nN . [176](#), [305](#),
 [845](#), [14117](#), [16257](#), [16267](#), [16278](#),
 [18193](#), [18479](#), [21674](#), [35157](#), [35233](#)
- \clist_map_inline:Nn
 [176](#), [3443](#), [3449](#), [9138](#),
 [18042](#), [18208](#), [21492](#), [21534](#), [21563](#)
- \clist_map_inline:nn
 [176](#), [3239](#), [10060](#), [11591](#),
 [11614](#), [11626](#), [18208](#), [20862](#), [20977](#),
 [21989](#), [22175](#), [34574](#), [34776](#), [34778](#),
 [34814](#), [34983](#), [35137](#), [35181](#), [35186](#)
- \clist_map_tokens:Nn
 [176](#), [844](#), [18231](#), [18240](#)
- \clist_map_tokens:nn [176](#), [18265](#)
- \clist_map_variable:NNn .. [176](#), [18230](#)
- \clist_map_variable:nNn .. [176](#), [18230](#)
- \clist_new:N [172](#), [831](#), [17856](#), [18033](#),
 [18481](#), [18482](#), [18483](#), [18484](#), [20685](#)
- \clist_pop:NN [178](#), [17966](#)
- \clist_pop:NNTF [179](#), [17992](#)
- \clist_push:Nn [179](#), [18017](#)
- \clist_put_left:Nn
 . [173](#), [17929](#), [18017](#), [18018](#), [18019](#),
 [18020](#), [18021](#), [18022](#), [18023](#), [18024](#)
- \clist_put_right:Nn
 [173](#), [17942](#), [21071](#), [21605](#), [21615](#), [21643](#)
- \clist_rand_item:N [180](#), [18439](#)
- \clist_rand_item:n ... [74](#), [180](#), [18439](#)
- \clist_remove_all:Nn
 [174](#), [9153](#), [18057](#), [21072](#)
- \clist_remove_duplicates:N
 [172](#), [174](#), [18035](#)
- \clist_reverse:N [174](#), [18096](#)
- \clist_reverse:n
 [174](#), [840](#), [18097](#), [18099](#), [18102](#)
- .clist_set:N [223](#), [21122](#)
- \clist_set:Nn [173](#),
 [178](#), [17923](#), [17930](#), [17932](#), [17943](#),
 [17945](#), [18138](#), [18226](#), [18237](#), [20902](#)
- \clist_set_eq:NN
 [172](#), [17869](#), [18036](#), [21477](#)
- \clist_set_from_seq:NN
 [172](#), [3444](#), [17877](#), [18058](#)
- \clist_show:N [180](#), [18452](#)
- \clist_show:n [180](#), [18474](#)
- \clist_sort:Nn [175](#), [3435](#), [18114](#)
- \clist_use:Nn [178](#), [18314](#)
- \clist_use:nn [178](#), [18347](#)
- \clist_use:Nnnn . [177](#), [178](#), [800](#), [18314](#)
- \clist_use:nnnn [178](#), [18347](#)
- \c_empty_clist
 [180](#), [17803](#), [17957](#), [17972](#), [17994](#), [18008](#)
- \g_tmpa_clist [180](#), [18481](#)
- \l_tmpa_clist [180](#), [18481](#)
- \g_tmpb_clist [180](#), [18481](#)
- \l_tmpb_clist [180](#), [18481](#)
- clist internal commands:
- __clist_concat:NNNN [17904](#)
- __clist_count:n [18285](#)
- __clist_count:w [18285](#)
- __clist_get:wN [17955](#), [17997](#)
- __clist_if_empty_n:w [18118](#)
- __clist_if_empty_n:wNw [18118](#)
- __clist_if_in_return:nnN [18132](#)
- __clist_if_wrap:nTF . [832](#), [17830](#),
 [17855](#), [17896](#), [18049](#), [18063](#), [18144](#)
- __clist_if_wrap:w [832](#), [17830](#)
- \l__clist_internal_clist
 [835](#), [17804](#), [17935](#),
 [17936](#), [17948](#), [17949](#), [18138](#), [18139](#),
 [18140](#), [18226](#), [18227](#), [18237](#), [18238](#)
- \l__clist_internal_remove_clist .
 [18033](#),
 [18041](#), [18044](#), [18046](#), [18048](#), [18053](#)
- \l__clist_internal_remove_seq ...
 [18033](#), [18065](#), [18066](#), [18067](#)
- __clist_item:nnnN [18380](#), [18413](#)
- __clist_item_n:nw [18411](#)
- __clist_item_n_end:n [18411](#)
- __clist_item_N_loop:nw [18380](#)
- __clist_item_n_loop:nw [18411](#)
- __clist_item_n_strip:n [18411](#)
- __clist_item_n_strip:w [18411](#)
- __clist_map_function:Nw
 [842](#), [18169](#), [18215](#)
- __clist_map_function_end:w
 [842](#), [18169](#)
- __clist_map_function_n:Nn [843](#), [18193](#)
- __clist_map_tokens:nw [18240](#)
- __clist_map_tokens_end:w [18240](#)

- `__clist_map_tokens_n:nw` [18265](#)
- `__clist_map_unbrace:wn`
..... [843](#), [18193](#), [18277](#), [18364](#)
- `__clist_map_variable:Nnn` [844](#), [18230](#)
- `__clist_pop:NNN` [17966](#)
- `__clist_pop:wN` [17966](#)
- `__clist_pop:wwNNN` . [836](#), [17966](#), [18011](#)
- `__clist_pop_TF:NNN` [17992](#)
- `__clist_put_left:NNNn` [17929](#)
- `__clist_put_right:NNNn` [17942](#)
- `__clist_rand_item:nn` [18439](#)
- `__clist_remove_all:` [18057](#)
- `__clist_remove_all:NNNn` [18057](#)
- `__clist_remove_all:w` ... [839](#), [18057](#)
- `__clist_remove_duplicates:NN` . [18035](#)
- `__clist_reverse:wwNww` ... [840](#), [18102](#)
- `__clist_reverse_end:ww` .. [840](#), [18102](#)
- `__clist_sanitize:n`
..... [17817](#), [17859](#), [17924](#), [17926](#)
- `__clist_sanitize:Nn` [832](#), [17817](#)
- `__clist_set_from_seq:n` [17877](#)
- `__clist_set_from_seq:NNNn` ... [17877](#)
- `__clist_show:NN` [18452](#)
- `__clist_show:Nn` [18474](#)
- `__clist_tmp:w`
.. [839](#), [17810](#), [18070](#), [18092](#), [18146](#),
[18155](#), [18159](#), [18161](#), [18295](#), [18313](#)
- `__clist_trim_next:w` [832](#),
[843](#), [17811](#), [17820](#), [17828](#), [18196](#), [18205](#)
- `__clist_use:Nw` [847](#), [18347](#)
- `__clist_use:nwn` [18314](#)
- `__clist_use:nwwwnwn` ... [846](#), [18314](#)
- `__clist_use:wn` [18314](#)
- `__clist_use_end:w` [847](#), [18347](#)
- `__clist_use_i_delimit_by_s_-`
stop:nw [17807](#), [18407](#)
- `__clist_use_more:w` [847](#), [18347](#)
- `__clist_use_none_delimit_by_s_-`
mark:w [17807](#), [18362](#)
- `__clist_use_none_delimit_by_s_-`
stop:w [839](#), [17807](#), [17825](#), [18073](#),
[18181](#), [18188](#), [18202](#), [18252](#), [18260](#),
[18275](#), [18308](#), [18349](#), [18393](#), [18398](#)
- `__clist_use_one:w` [18347](#)
- `__clist_wrap_item:w` [832](#), [17826](#), [17854](#)
- `\closein` [216](#)
- `\closeout` [217](#)
- `\clubpenalties` [515](#)
- `\clubpenalty` [218](#)
- `cm` [257](#)
- code commands:
 `.code:n` [224](#), [21120](#)
- coffin commands:
 `\coffin_attach:NnnNnnnn`
 [284](#), [1308](#), [33496](#)
- `\coffin_clear:N` [282](#), [32705](#)
- `\coffin_display_handles:Nn` [285](#), [33742](#)
- `\coffin_dp:N`
 [285](#), [32910](#), [33360](#), [33399](#), [33856](#)
- `\coffin_gattach:NnnNnnnn` . [284](#), [33496](#)
- `\coffin_gclear:N` [282](#), [32705](#)
- `\coffin_gjoin:NnnNnnnn` ... [285](#), [33445](#)
- `\coffin_gresize:Nnn` [284](#), [33339](#)
- `\coffin_grotate:Nn` [284](#), [33180](#)
- `\coffin_gscale:Nnn` [284](#), [33387](#)
- `\coffin_gset_eq:NN`
 [282](#), [32879](#), [33454](#), [33505](#)
- `\coffin_gset_horizontal_pole:Nnn`
 [283](#), [32940](#)
- `\coffin_gset_vertical_pole:Nnn` ..
 [284](#), [32940](#)
- `\coffin_ht:N`
 [285](#), [32910](#), [33360](#), [33399](#), [33855](#)
- `\coffin_if_exist:NTF` [282](#), [32684](#), [32698](#)
- `\coffin_if_exist_p:N` [282](#), [32684](#)
- `\coffin_join:NnnNnnnn` ... [285](#), [33445](#)
- `\coffin_log:N` [286](#), [33867](#)
- `\coffin_log:Nnn` [286](#), [33867](#)
- `\coffin_log_structure:N` .. [286](#), [33842](#)
- `\coffin_mark_handle:Nnnn` . [286](#), [33697](#)
- `\coffin_new:N`
 [282](#), [1284](#), [32723](#), [32903](#),
[32904](#), [32905](#), [32906](#), [32907](#), [32908](#),
[32909](#), [33629](#), [33639](#), [33640](#), [33641](#)
- `\coffin_resize:Nnn` [284](#), [33339](#)
- `\coffin_rotate:Nn` [284](#), [33180](#)
- `\coffin_scale:Nnn` [284](#), [33387](#)
- `\coffin_scale:NnnNN` [33387](#)
- `\coffin_set_eq:NN`
 [282](#), [32879](#), [33448](#), [33499](#), [33555](#), [33758](#)
- `\coffin_set_horizontal_pole:Nnn` .
 [283](#), [32940](#)
- `\coffin_set_vertical_pole:Nnn` ...
 [284](#), [32940](#)
- `\coffin_show:N` [286](#), [33867](#)
- `\coffin_show:Nnn` [286](#), [33867](#)
- `\coffin_show_structure:N`
 [286](#), [1309](#), [33842](#)
- `\coffin_typeset:Nnnnn` ... [285](#), [33631](#)
- `\coffin_wd:N`
 [285](#), [32910](#), [33356](#), [33403](#), [33857](#)
- `\c_empty_coffin` [286](#), [32903](#)
- `\g_tmpa_coffin` [287](#), [32906](#)
- `\l_tmpa_coffin` [286](#), [32906](#)
- `\g_tmpb_coffin` [287](#), [32906](#)
- `\l_tmpb_coffin` [286](#), [32906](#)

coffin internal commands:

- __coffin_align:NnnNnnnnN
.. 33459, 33510, 33531, 33538, 33634
- \l__coffin_aligned_coffin
..... 32903, 33460,
33461, 33465, 33471, 33474, 33477,
33493, 33494, 33511, 33512, 33513,
33514, 33515, 33518, 33522, 33526,
33527, 33532, 33533, 33534, 33535,
33536, 33569, 33585, 33635, 33636,
33827, 33834, 33836, 33838, 33840
- \l__coffin_aligned_internal_-
coffin 32903, 33548, 33555
- __coffin_attach:NnnNnnnnN ... 33496
- __coffin_attach_mark:NnnNnnnn ...
..... 33496, 33704, 33720, 33736
- \l__coffin_bottom_corner_dim ...
..... 33176, 33208, 33212,
33291, 33302, 33303, 33323, 33331
- \l__coffin_bounding_prop
..... 33172, 33199, 33228,
33230, 33236, 33238, 33247, 33310
- \l__coffin_bounding_shift_dim ...
.. 33175, 33207, 33309, 33315, 33316
- __coffin_calculate_intersection:Nnn
..... 33069, 33540, 33543, 33820
- __coffin_calculate_intersection:nnnnnn
..... 33069
- __coffin_calculate_intersection:nnnnnnnn
..... 33069, 33771
- \c__coffin_corners_prop
..... 32653, 32730, 32929, 32936
- \l__coffin_corners_prop
.... 33173, 33190, 33194, 33217,
33222, 33253, 33293, 33320, 33367,
33371, 33377, 33383, 33418, 33432
- \l__coffin_cos_fp
1292, 1294, 33170, 33189, 33274, 33283
- __coffin_display_attach:Nnnnn 33742
- \l__coffin_display_coffin
.... 33639, 33758, 33764, 33829,
33830, 33835, 33837, 33839, 33840
- \l__coffin_display_coord_coffin .
..... 33639, 33706,
33721, 33737, 33779, 33794, 33813
- \l__coffin_display_font_tl
..... 33684, 33709, 33782
- __coffin_display_handles_-
aux:nnnn 33742
- __coffin_display_handles_-
aux:nnnnnn 33742
- \l__coffin_display_handles_prop .
.. 33642, 33712, 33716, 33785, 33789
- \l__coffin_display_offset_dim ...
.. 33679, 33738, 33739, 33814, 33815
- \l__coffin_display_pole_coffin ...
.. 33639, 33699, 33705, 33744, 33777
- \l__coffin_display_poles_prop ...
..... 33683, 33749,
33754, 33757, 33759, 33761, 33768
- \l__coffin_display_x_dim
..... 33681, 33774, 33824
- \l__coffin_display_y_dim
..... 33681, 33775, 33826
- \c__coffin_empty_coffin 33629, 33634
- \l__coffin_error_bool
..... 32674, 33073, 33077,
33091, 33113, 33144, 33770, 33772
- __coffin_find_bounding_shift: ..
..... 33202, 33307
- __coffin_find_bounding_shift_-
aux:nn 33307
- __coffin_find_corner_maxima:N ..
..... 33201, 33287
- __coffin_find_corner_maxima_-
aux:nn 33287
- __coffin_get_pole:Nnn
32916, 33071, 33072, 33596, 33597,
33600, 33601, 33751, 33752, 33755
- __coffin_greset_structure:N ...
..... 32719, 32926, 32990
- __coffin_gupdate:N
.. 32758, 32771, 32830, 32848, 32982
- __coffin_gupdate_corners:N ...
..... 32991, 32994
- __coffin_gupdate_poles:N
..... 32992, 33025
- __coffin_if_exist:NTF
32696, 32707, 32716, 32738, 32751,
32776, 32811, 32824, 32853, 32881,
32893, 32948, 32966, 33850, 33881
- \l__coffin_internal_box
..... 32650, 32785,
32791, 32796, 32862, 32868, 32873,
33204, 33211, 33213, 33214, 33216
- \l__coffin_internal_dim
.... 32650, 33235, 33237, 33241,
33398, 33401, 33466, 33468, 33469
- \l__coffin_internal_tl ... 32650,
33567, 33568, 33570, 33713, 33714,
33717, 33718, 33726, 33731, 33786,
33787, 33790, 33791, 33800, 33805
- __coffin_join:NnnNnnnnN 33445
- \l__coffin_left_corner_dim
..... 33176, 33207, 33215,
33292, 33298, 33299, 33322, 33330

__coffin_mark_handle_aux:nnnnNnn
 [33697](#)
 __coffin_offset_corner:Nnnnn . [33576](#)
 __coffin_offset_corners:Nnn ...
 .. [33482](#), [33483](#), [33489](#), [33490](#), [33576](#)
 __coffin_offset_pole:Nnnnnnn . [33557](#)
 __coffin_offset_poles:Nnn
 [33480](#), [33481](#),
 [33486](#), [33487](#), [33523](#), [33524](#), [33557](#)
 \l__coffin_offset_x_dim
 [32675](#), [33463](#), [33464](#), [33467](#),
 [33478](#), [33480](#), [33482](#), [33488](#), [33491](#),
 [33525](#), [33544](#), [33552](#), [33823](#), [33831](#)
 \l__coffin_offset_y_dim
 [32675](#), [33481](#), [33483](#), [33488](#), [33491](#),
 [33525](#), [33546](#), [33553](#), [33825](#), [33832](#)
 \l__coffin_pole_a_tl
 [32677](#), [33071](#), [33076](#), [33596](#), [33599](#),
 [33600](#), [33603](#), [33751](#), [33753](#), [33756](#)
 \l__coffin_pole_b_tl [32677](#),
 [33072](#), [33076](#), [33597](#), [33599](#), [33601](#),
 [33603](#), [33752](#), [33753](#), [33755](#), [33756](#)
 \c__coffin_poles_prop
 [32660](#), [32732](#), [32931](#), [32938](#)
 \l__coffin_poles_prop
 [33173](#), [33192](#), [33196](#),
 [33219](#), [33224](#), [33261](#), [33328](#), [33369](#),
 [33373](#), [33379](#), [33385](#), [33424](#), [33439](#)
 __coffin_reset_structure:N
 .. [32710](#), [32926](#), [32984](#), [33471](#), [33515](#)
 __coffin_resize:NnnNN [33339](#)
 __coffin_resize_common:NnnN ...
 [33363](#), [33365](#), [33404](#)
 \l__coffin_right_corner_dim
 .. [33176](#), [33215](#), [33290](#), [33300](#), [33301](#)
 __coffin_rotate:NnnNN [33180](#)
 __coffin_rotate_bounding:nnn ...
 [33200](#), [33244](#)
 __coffin_rotate_corner:Nnnn ...
 [33195](#), [33244](#)
 __coffin_rotate_pole:Nnnnnn ...
 [33197](#), [33256](#)
 __coffin_rotate_vector:nnNN ...
 .. [33246](#), [33252](#), [33258](#), [33259](#), [33268](#)
 __coffin_rule:nn [33692](#), [33702](#), [33747](#)
 __coffin_scale:NnnNN
 [33388](#), [33391](#), [33393](#)
 __coffin_scale_corner:Nnnn
 [33372](#), [33415](#)
 __coffin_scale_pole:Nnnnnn
 [33374](#), [33415](#)
 __coffin_scale_vector:nnNN
 [33408](#), [33417](#), [33423](#)
 \l__coffin_scale_x_fp [33335](#), [33355](#),
 [33375](#), [33395](#), [33397](#), [33403](#), [33411](#)
 \l__coffin_scale_y_fp ... [33335](#),
 [33357](#), [33396](#), [33397](#), [33401](#), [33413](#)
 \l__coffin_scaled_total_height_-
 dim [33337](#), [33400](#), [33405](#)
 \l__coffin_scaled_width_dim
 [33337](#), [33402](#), [33405](#)
 __coffin_set_bounding:N [33198](#), [33226](#)
 __coffin_set_horizontal_-
 pole:NnnN [32940](#)
 __coffin_set_pole:Nnn
 [32786](#), [32863](#), [32940](#),
 [33569](#), [33609](#), [33613](#), [33621](#), [33625](#)
 __coffin_set_vertical:NnnNN . [32762](#)
 __coffin_set_vertical:NnnNNw [32837](#)
 __coffin_set_vertical_aux:
 [32762](#), [32857](#)
 __coffin_set_vertical_pole:NnnN
 [32940](#)
 __coffin_shift_corner:Nnnn
 [33218](#), [33318](#)
 __coffin_shift_pole:Nnnnnn
 [33220](#), [33318](#)
 __coffin_show:Nnnnn [33867](#)
 __coffin_show_structure:NN
 [33842](#), [33883](#)
 \l__coffin_sin_fp
 [1292](#), [1294](#), [33170](#), [33188](#), [33275](#), [33282](#)
 \l__coffin_slope_A_fp [32672](#)
 \l__coffin_slope_B_fp [32672](#)
 __coffin_to_value:N
 [32683](#), [32688](#), [32727](#), [32728](#), [32729](#),
 [32731](#), [32884](#), [32885](#), [32886](#), [32887](#),
 [32896](#), [32897](#), [32898](#), [32899](#), [32919](#),
 [32928](#), [32930](#), [32935](#), [32937](#), [32950](#),
 [32968](#), [32978](#), [33001](#), [33032](#), [33191](#),
 [33193](#), [33221](#), [33223](#), [33368](#), [33370](#),
 [33382](#), [33384](#), [33474](#), [33518](#), [33521](#),
 [33559](#), [33578](#), [33585](#), [33750](#), [33861](#)
 \l__coffin_top_corner_dim
 .. [33176](#), [33212](#), [33289](#), [33304](#), [33305](#)
 __coffin_update:N
 .. [32745](#), [32765](#), [32817](#), [32841](#), [32982](#)
 __coffin_update_B:nnnnnnnnN . [33594](#)
 __coffin_update_corners:N
 [32985](#), [32994](#)
 __coffin_update_corners:NN .. [32994](#)
 __coffin_update_corners:NNN . [32994](#)
 __coffin_update_poles:N
 [32986](#), [33025](#), [33477](#), [33522](#)
 __coffin_update_poles:NN [33025](#)
 __coffin_update_poles:NNN ... [33025](#)
 __coffin_update_T:nnnnnnnnN . [33594](#)

- _coffin_update_vertical_-
 - poles:NNN 33493, 33526, [33594](#)
- \l_coffin_x_dim . . . [32679](#), 33080,
 - 33089, 33115, 33146, 33164, 33246,
 - 33248, 33252, 33254, 33258, 33263,
 - 33417, 33419, 33423, 33426, 33541,
 - 33545, 33564, 33572, 33774, 33821
- \l_coffin_x_prime_dim
 - [32679](#), 33260,
 - 33264, 33541, 33545, 33821, 33824
- _coffin_x_shift_corner:Nnnn . . .
 - [33378](#), [33430](#)
- _coffin_x_shift_pole:Nnnnnn . . .
 - [33380](#), [33430](#)
- \l_coffin_y_dim [32679](#),
 - 33081, 33093, 33111, 33160, 33246,
 - 33248, 33252, 33254, 33258, 33263,
 - 33417, 33419, 33423, 33426, 33542,
 - 33547, 33565, 33572, 33775, 33822
- \l_coffin_y_prime_dim
 - [32679](#), 33260,
 - 33265, 33542, 33547, 33822, 33826
- color commands:
 - color.sc [292](#)
 - \color_ensure_current: . [288](#), [1281](#),
 - [32742](#), [32755](#), [32813](#), [32826](#), [33913](#)
 - \color_export:nnN [293](#), [34666](#)
 - \color_export:nnnN [293](#), [34666](#)
 - \color_fill:n [291](#), [34522](#)
 - \color_fill:nn [292](#), [34522](#)
 - \l_color_fixed_model_tl [291](#), [34044](#),
 - [34046](#), [34352](#), [34355](#), [34358](#), [34360](#),
 - [34364](#), [34399](#), [34400](#), [34406](#), [34425](#),
 - [34427](#), [34550](#), [34554](#), [34556](#), [34670](#)
 - \color_group_begin: . . [288](#), [32117](#),
 - [32121](#), [32126](#), [32133](#), [32138](#), [32146](#),
 - [32152](#), [32166](#), [32172](#), [32179](#), [32184](#),
 - [32197](#), [32199](#), [32203](#), [32208](#), [32213](#),
 - [32218](#), [32225](#), [32230](#), [32237](#), [32242](#),
 - [32250](#), [32256](#), [32271](#), [32277](#), [33911](#)
 - \color_group_end: [288](#),
 - [32117](#), [32121](#), [32126](#), [32133](#), [32138](#),
 - [32158](#), [32179](#), [32184](#), [32197](#), [32199](#),
 - [32203](#), [32208](#), [32213](#), [32218](#), [32225](#),
 - [32230](#), [32237](#), [32242](#), [32263](#), [33911](#)
 - \color_log:n [291](#), [35419](#)
 - \color_math:nn [292](#), [34432](#)
 - \color_math:nn(n) [1325](#)
 - \color_math:nnn [292](#), [34432](#)
 - \l_color_math_active_tl
 - [292](#), [34429](#), [34483](#)
 - \color_model_new:nnn [294](#), [34788](#)
 - \color_profile_apply:nn . . [294](#), [35390](#)
- \color_select:n
 - [291](#), [33701](#), [33708](#), [33746](#), [33781](#), [34381](#)
- \color_select:n(n) [292](#)
- \color_select:nn [291](#), [34381](#)
- \color_select:nn(n) [292](#)
- \color_set:nn [290](#), [34547](#)
- \color_set:nnn
 - . [291](#), [34547](#), [34655](#), [34656](#), [34657](#),
 - [34658](#), [34659](#), [34660](#), [34661](#), [34662](#)
- \color_set_eq:nn [291](#), [34547](#)
- \color_show:n [291](#), [35419](#)
- \color_stroke:n [291](#), [34522](#)
- \color_stroke:nn [292](#), [34522](#)
- color internal commands:
 - \g_color_alternative_model_prop
 - [34775](#), [34887](#), [34985](#)
 - \g_color_alternative_values_-
 - prop [34780](#),
 - [34902](#), [34916](#), [34926](#), [35140](#), [35242](#)
 - _color_backend_devicen_-
 - init:nnn [35155](#)
 - _color_backend_iccbased_-
 - device:nnn . . . [35407](#), [35412](#), [35417](#)
 - _color_backend_iccbased_-
 - init:nnn [35387](#)
 - _color_backend_pickup:N
 - [33915](#), [34026](#)
 - _color_backend_reset: [33922](#), [34503](#)
 - _color_backend_separation_-
 - init:nnnnn . . . [34903](#), [34917](#), [34927](#)
 - _color_backend_separation_-
 - init_CIELAB:nnn [34955](#)
 - _color_check_model:N . [34040](#), [34353](#)
 - _color_check_model:nn [34353](#)
 - \g_color_colorants_prop
 - [34758](#), [34888](#), [35203](#)
 - _color_convert:nnN
 - . . [33951](#), [34059](#), [34143](#), [34154](#), [34360](#)
 - _color_convert:nnnN
 - [33951](#), [34425](#), [34700](#)
 - _color_convert_cmyk_cmyk:w . [34018](#)
 - _color_convert_cmyk_gray:w . [34010](#)
 - _color_convert_cmyk_rgb:w . [34012](#)
 - _color_convert_devicen_-
 - cmyk:nnnnnnnnn [34967](#)
 - _color_convert_devicen_-
 - cmyk:nnnnw [34967](#)
 - _color_convert_devicen_cmyk_-
 - aux:nnnnw [34967](#)
 - _color_convert_devicen_-
 - gray:nnn [34967](#)
 - _color_convert_devicen_gray:nw
 - [34967](#)

- _color_convert_devicen_gray_-aux:nw [34967](#)
- _color_convert_devicen_-rgb:nnnnnn [34967](#)
- _color_convert_devicen_-rgb:nnnw [34967](#)
- _color_convert_devicen_rgb_-aux:nnnw [34967](#)
- _color_convert_gray_cmyk:w . [33992](#)
- _color_convert_gray_gray:w . [33988](#)
- _color_convert_gray_gray:w:uuuuu_color_convert_gray_rgb:w:uuuuu_color_convert_gray_cmyk:w:uuuuu_color_convert_cmyk_gray:w:uuuuu_color_convert_cmyk_rgb:w:uuuuu_color_convert_cmyk_cmyk:w:uuuuu_color_convert_rgb_gray:w:uuuuu_color_convert_rgb_rgb:w:uuuuu_color_convert_rgb_cmyk:w . [33951](#)
- _color_convert_gray_rgb:w . [33990](#)
- _color_convert_rgb_cmyk:nnnn [33951](#)
- _color_convert_rgb_gray:w . [33994](#)
- _color_convert_rgb_rgb:w . [33996](#)
- \l_color_current_tl [1311](#), [33911](#), [33915](#), [33916](#), [33928](#), [34026](#), [34028](#), [34031](#), [34078](#), [34375](#), [34379](#), [34383](#), [34385](#), [34389](#), [34392](#), [34435](#), [34441](#), [34447](#), [34449](#), [34504](#), [34511](#), [34512](#), [34524](#), [34525](#), [34529](#), [34530](#), [34534](#), [34536](#), [34540](#), [34542](#), [34642](#), [34644](#), [34665](#)
- _color_draw:nnn [34522](#)
- _color_export:nN [34666](#)
- _color_export:nnnN [34666](#)
- _color_export:nnnNN . [34695](#), [34715](#)
- _color_export_comma-sep-cmyk:Nw [34726](#)
- \c_color_export_comma-sep-cmyk_-tl [34705](#)
- _color_export_comma-sep-rgb:Nw [34731](#)
- \c_color_export_comma-sep-rgb_-tl [34705](#)
- _color_export_format_backend:nnN [34693](#)
- _color_export_format_comma-sep-cmyk:nnN [34710](#)
- _color_export_format_comma-sep-rgb:nnN [34710](#)
- _color_export_format_space-sep-cmyk:nnN [34710](#)
- _color_export_format_space-sep-rgb:nnN [34710](#)
- _color_export_HTML:n [34731](#)
- _color_export_HTML:Nw [34731](#)
- \c_color_export_HTML_tl [34705](#)
- _color_export_space-sep-cmyk:Nw [34726](#)
- \c_color_export_space-sep-cmyk_-tl [34705](#)
- _color_export_space-sep-rgb:Nw [34731](#)
- \c_color_export_space-sep-rgb_-tl [34705](#)
- _color_extract:nnN [33945](#), [34084](#), [34100](#), [34101](#), [34120](#), [34135](#)
- \c_color_fallback_cmyk_tl . . . [34755](#)
- \c_color_fallback_gray_tl . . . [34755](#)
- \c_color_fallback_rgb_tl . . . [34755](#)
- _color_finalise_current: [34372](#), [34384](#), [34391](#)
- \c_color_icc_colorspace_-signatures_prop [35343](#), [35369](#)
- _color_if_defined:nTF . . [33933](#), [34037](#), [34070](#), [34118](#), [34635](#), [35428](#)
- \l_color_internal_int [33930](#), [35136](#), [35139](#), [35195](#)
- \l_color_internal_prop [34753](#), [34803](#), [34834](#), [34847](#), [34862](#), [34941](#), [34969](#), [35356](#)
- \l_color_internal_tl [33930](#), [34086](#), [34089](#), [34576](#), [34582](#), [34584](#), [34586](#), [34587](#), [34625](#), [34627](#), [34628](#), [34835](#), [34838](#), [34848](#), [34851](#), [34863](#), [34865](#), [34942](#), [34946](#), [34949](#), [34970](#), [34973](#), [34986](#), [34989](#), [34991](#), [35134](#), [35145](#), [35168](#), [35191](#), [35357](#), [35360](#), [35370](#), [35373](#)
- _color_math:nn [34432](#)
- _color_math_scan:w [1327](#), [34451](#), [34453](#), [34514](#)
- _color_math_scan_auxi: [34453](#)
- _color_math_scan_auxii: [34453](#)
- _color_math_scan_auxiii:N [34490](#), [34496](#)
- _color_math_scan_end: [34453](#)
- _color_math_script_aux:N . . . [34506](#)
- _color_math_scripts:Nw [34469](#), [34506](#)
- \g_color_math_seq [34431](#), [34447](#), [34504](#), [34511](#)
- _color_model:N [33943](#), [34028](#), [34375](#), [34565](#), [34586](#), [34625](#), [34642](#), [34665](#)
- _color_model_convert:nnn . . . [34832](#)
- _color_model_devicen:n [34967](#)
- _color_model_devicen:nn [34967](#)
- _color_model_devicen:nnn . . . [34967](#)
- _color_model_devicen:nnnn . . [34967](#)

- _color_model_devicen_colorant:n [34967](#)
- _color_model_devicen_convert:n [34967](#)
- _color_model_devicen_convert:nnn [34967](#)
- _color_model_devicen_convert:nnnn [34967](#), [35031](#)
- _color_model_devicen_convert:nnnnn [35213](#), [35218](#), [35223](#), [35225](#)
- _color_model_devicen_convert:w [34967](#)
- _color_model_devicen_convert_-aux:n [34967](#)
- _color_model_devicen_convert_-aux:w [35246](#), [35247](#)
- _color_model_devicen_convert_-cmyk:n [34967](#)
- _color_model_devicen_convert_-cmyk:nnn [35210](#)
- _color_model_devicen_convert_-gray:n [34967](#)
- _color_model_devicen_convert_-gray:nnn [35215](#)
- _color_model_devicen_convert_-rgb:n [34967](#)
- _color_model_devicen_convert_-rgb:nnn [35220](#)
- _color_model_devicen_init:nnn [34967](#)
- _color_model_devicen_init:nnnn [34967](#)
- _color_model_devicen_mix:nw [34967](#)
- _color_model_devicen_parse:nw [34967](#)
- _color_model_devicen_parse_-1:nn [34967](#)
- _color_model_devicen_parse_-2:nn [34967](#)
- _color_model_devicen_parse_-3:nn [34967](#)
- _color_model_devicen_parse_-4:nn [34967](#)
- _color_model_devicen_parse_-generic:nn [34967](#)
- _color_model_devicen_tranform:nnn [34967](#)
- _color_model_devicen_tranform:w [34967](#)
- _color_model_devicen_tranform_-1:nnnnn [34967](#)
- _color_model_devicen_tranform_-3:nnnnn [34967](#)
- _color_model_devicen_tranform_-4:nnnnn [34967](#)
- _color_model_devicen_tranform:nnn [35178](#), [35182](#), [35187](#), [35189](#)
- _color_model_devicen_tranform:w [35142](#), [35171](#)
- _color_model_iccbased:n [35354](#)
- _color_model_iccbased:nn ... [35354](#)
- _color_model_iccbased:nnn .. [35354](#)
- _color_model_iccbased_aux:nnn [35354](#)
- _color_model_iccbased_aux:nnnnnn [35372](#), [35380](#)
- _color_model_init:nnnn [34811](#), [34881](#), [35023](#), [35382](#)
- \g_color_model_int [34754](#), [34813](#), [34819](#), [35406](#), [35411](#), [35416](#)
- _color_model_new:nnn [34788](#)
- \c_color_model_range_CIELAB_tl [34774](#)
- _color_model_separation:n .. [34832](#)
- _color_model_separation:nn .. [34832](#)
- _color_model_separation:nnn .. [34832](#)
- _color_model_separation:w .. [34832](#)
- _color_model_separation_-CIELAB:nnnnnn [34832](#)
- _color_model_separation_-CIELAB:nnnnnnnn [34832](#)
- _color_model_separation_-cmyk:nnnnnn [34832](#)
- _color_model_separation_-gray:nnnnnn [34832](#)
- _color_model_separation_-rgb:nnnnnn [34832](#)
- \l_color_model_tl [34020](#), [34056](#), [34057](#), [34060](#), [34084](#), [34087](#), [34102](#), [34121](#), [34123](#), [34130](#), [34135](#), [34141](#), [34143](#), [34145](#), [34151](#), [34156](#), [34358](#), [34360](#), [34369](#), [34982](#), [34988](#), [34989](#), [34991](#), [35009](#), [35014](#)
- \c_color_model_whitepoint_-CIELAB_a_tl [34767](#)
- \c_color_model_whitepoint_-CIELAB_b_tl [34767](#)
- \c_color_model_whitepoint_-CIELAB_d50_tl [34767](#)
- \c_color_model_whitepoint_-CIELAB_d55_tl [34767](#)
- \c_color_model_whitepoint_-CIELAB_d65_tl [34767](#)
- \c_color_model_whitepoint_-CIELAB_d75_tl [34767](#)
- \c_color_model_whitepoint_-CIELAB_e_tl [34767](#)

\l_color_named._prop	34663	_color_parse_model_hsb_-	
\l_color_named._tl	34663	aux:nnnn	34235 , 34239
\l_color_named_tl .	34546 , 34562 , 34565 , 34568 , 34624 , 34625 , 34629	_color_parse_model_hsb_-	
\l_color_named_white_prop ...	34824	aux:nnnnn	34243 , 34251
\l_color_next_model_tl ..	34020 , 34120 , 34121 , 34141 , 34142 , 34155	_color_parse_model_HTML:w ..	34216
\l_color_next_value_tl ..	34020 , 34120 , 34130 , 34146 , 34152 , 34157	_color_parse_model_HTML_aux:w .	34268 , 34269
_color_parse:nN	34024 , 34383 , 34435 , 34524 , 34529 , 34562 , 34582 , 34671	_color_parse_model_RGB:w ...	34216
_color_parse:Nw	34024	_color_parse_model_rgb:w ...	34191
_color_parse_aux:nN	34024	_color_parse_model_wave:w ..	34216
_color_parse_break:w	34024	_color_parse_model_wave_-	
_color_parse_end:	34024	auxi:nn	34216
_color_parse_eq:Nn	34024	_color_parse_model_wave_-	
_color_parse_eq:nNn	34024	auxii:nn	34216
_color_parse_gray:n	34024	_color_parse_model_wave_rho:n .	34216
_color_parse_loop:nn	34024	34216
_color_parse_loop:w	34024	_color_parse_number:n	
_color_parse_loop_init:Nnn .	34024	34191 , 34244 , 34884 , 35039 , 35045 , 35059 , 35060 , 35061 , 35073 , 35074 , 35075 , 35076 , 35103
_color_parse_mix:Nnnn	34024	_color_parse_number:w	34191
_color_parse_mix:nNnn	34024	_color_parse_set_eq:Nn	34038 , 34042 , 34073
_color_parse_mix_cmyk:nw	34024 , 35079	_color_parse_set_eq:nNn	34045 , 34046 , 34049
_color_parse_mix_gray:nw	34024 , 34882 , 35040	_color_parse_std:n	34024
_color_parse_mix_rgb:nw	34024 , 35064	_color_profile_apply:nn	35390
_color_parse_model_cmyk:w ..	34191	_color_profile_apply_cmyk:n .	35390
_color_parse_model_Gray:w ..	34216	_color_profile_apply_gray:n .	35390
_color_parse_model_gray:w ..	34191	_color_profile_apply_rgb:n .	35390
_color_parse_model_hsb:nnn .	34216	_color_select:N	33916 , 33919 , 34385 , 34392 , 34512
_color_parse_model_hsb:nnnn .	34216	_color_select:nn	33919
_color_parse_model_hsb:nnnnn .	34216	_color_select:nnN ...	34381 , 34624
_color_parse_model_HSB:w ...	34216	_color_select_loop:Nw	34381
_color_parse_model_Hsb:w ...	34216	_color_select_main:Nw	34381 , 34441 , 34534 , 34540 , 34678
_color_parse_model_hsb:w ...	34216	_color_select_math:N .	33919 , 34449
_color_parse_model_hsb_0:nnnn .	34216	_color_select_swap:Nnn	34381
_color_parse_model_hsb_1:nnnn .	34216	_color_set:nn	34547
_color_parse_model_hsb_2:nnnn .	34216	_color_set:nnn	34547
_color_parse_model_hsb_3:nnnn .	34216	_color_set:nnw	34547
_color_parse_model_hsb_4:nnnn .	34216	_color_set_aux:nnn	34547
_color_parse_model_hsb_5:nnnn .	34216	_color_set_colon:nnw	34547
_color_parse_model_hsb_aux:nnn	34216	_color_set_loop:nw	34547
		_color_show:n	35419
		_color_show:Nn	35419
		_color_tmp:w	34711 , 34719 , 34720 , 34721 , 34722 , 34723
		\l_color_value_tl	34020 , 34053 , 34054 , 34058 , 34060 , 34064 , 34084 , 34087 , 34102 , 34127 , 34130 , 34135 , 34144 , 34361 , 34364 , 34370 , 34425 , 34427 , 35141 , 35143

- `__color_values:N` .. [33943](#), [34031](#),
[34379](#), [34568](#), [34587](#), [34629](#), [34644](#)
- `\columnwidth` [32806](#)
- `\copy` [219](#)
- `\copyfont` [909](#)
- `cos` [254](#)
- `cosd` [254](#)
- `cot` [254](#)
- `cotd` [254](#)
- `\count` [220](#), [19153](#)
- `\countdef` [221](#)
- `\cr` [222](#)
- `\crampeddisplaystyle` [800](#)
- `\crampedscriptscriptstyle` [801](#)
- `\crampedscriptstyle` [803](#)
- `\crampedtextstyle` [804](#)
- `\crr` [223](#)
- `\creationdate` [773](#)
- `\cs` [24821](#)
- cs commands:
 - `\cs:w` [21](#), [530](#), [553](#), [554](#), [634](#),
[1431](#), [1453](#), [1455](#), [1508](#), [1818](#), [1846](#),
[2039](#), [2104](#), [2274](#), [2323](#), [2332](#), [2334](#),
[2338](#), [2339](#), [2340](#), [2402](#), [2408](#), [2414](#),
[2420](#), [2447](#), [2449](#), [2454](#), [2461](#), [2462](#),
[2527](#), [2531](#), [2570](#), [3188](#), [4367](#), [7205](#),
[7208](#), [8630](#), [8632](#), [10809](#), [13841](#),
[13847](#), [17021](#), [17108](#), [17745](#), [17797](#),
[18819](#), [19944](#), [20240](#), [20287](#), [20378](#),
[20948](#), [20949](#), [21597](#), [22503](#), [22522](#),
[22589](#), [23400](#), [23589](#), [23621](#), [24035](#),
[24061](#), [24074](#), [24108](#), [24150](#), [24666](#),
[26371](#), [27462](#), [31600](#), [31975](#), [35809](#)
 - `\cs_argument_spec:N`
..... [22](#), [2211](#), [13023](#), [13056](#), [36574](#)
 - `\cs_end:` [21](#), [406](#), [553](#),
[554](#), [635](#), [1431](#), [1453](#), [1455](#), [1459](#),
[1508](#), [1812](#), [1818](#), [1840](#), [1846](#), [1966](#),
[2039](#), [2104](#), [2274](#), [2323](#), [2332](#), [2334](#),
[2338](#), [2339](#), [2340](#), [2402](#), [2408](#), [2414](#),
[2420](#), [2447](#), [2449](#), [2454](#), [2461](#), [2462](#),
[2527](#), [2531](#), [2570](#), [3188](#), [4367](#), [7014](#),
[7221](#), [8627](#), [8633](#), [8635](#), [8637](#), [8639](#),
[8641](#), [8643](#), [8645](#), [8647](#), [8649](#), [8651](#),
[8653](#), [10809](#), [10824](#), [10827](#), [10828](#),
[13847](#), [13850](#), [17021](#), [17108](#), [17745](#),
[17750](#), [17778](#), [17787](#), [17797](#), [18819](#),
[19944](#), [20240](#), [20287](#), [20378](#), [20948](#),
[20949](#), [21597](#), [22506](#), [22522](#), [22597](#),
[23403](#), [23593](#), [23625](#), [24041](#), [24067](#),
[24080](#), [24111](#), [24153](#), [24672](#), [26371](#),
[27465](#), [31600](#), [31975](#), [35807](#), [35809](#)
 - `\cs_generate_from_arg_count:NNnn`
..... [19](#), [2019](#), [2062](#)
- `\cs_generate_variant:Nn`
..... [15](#), [31–33](#), [64](#),
[301](#), [400](#), [2887](#), [3230](#), [3232](#), [3234](#),
[3236](#), [3413](#), [3415](#), [3437](#), [3440](#), [3446](#),
[3452](#), [4173](#), [5139](#), [6274](#), [7029](#), [7070](#),
[8380](#), [8386](#), [8395](#), [8396](#), [8397](#), [8398](#),
[8401](#), [8402](#), [8413](#), [8414](#), [8430](#), [8437](#),
[8439](#), [8587](#), [8588](#), [8593](#), [8594](#), [8927](#),
[8959](#), [9672](#), [10054](#), [10055](#), [10056](#),
[10057](#), [10095](#), [10100](#), [10134](#), [10155](#),
[10157](#), [10159](#), [10330](#), [10349](#), [10357](#),
[10369](#), [10371](#), [10373](#), [10400](#), [10403](#),
[10421](#), [10529](#), [11017](#), [11026](#), [11027](#),
[11028](#), [11309](#), [11393](#), [11916](#), [11927](#),
[11928](#), [11933](#), [11934](#), [11939](#), [11940](#),
[11945](#), [11946](#), [11963](#), [11964](#), [11985](#),
[11986](#), [11987](#), [11988](#), [11989](#), [11990](#),
[11991](#), [11992](#), [12033](#), [12034](#), [12035](#),
[12036](#), [12037](#), [12038](#), [12039](#), [12040](#),
[12075](#), [12076](#), [12077](#), [12078](#), [12079](#),
[12080](#), [12081](#), [12082](#), [12130](#), [12131](#),
[12132](#), [12133](#), [12197](#), [12198](#), [12199](#),
[12200](#), [12261](#), [12262](#), [12267](#), [12268](#),
[12438](#), [12452](#), [12482](#), [12492](#), [12513](#),
[12518](#), [12520](#), [12529](#), [12541](#), [12542](#),
[12579](#), [12582](#), [12587](#), [12588](#), [12664](#),
[12675](#), [12873](#), [12884](#), [12885](#), [12908](#),
[12915](#), [12917](#), [12994](#), [13015](#), [13017](#),
[13098](#), [13113](#), [13114](#), [13117](#), [13118](#),
[13129](#), [13151](#), [13152](#), [13153](#), [13154](#),
[13187](#), [13188](#), [13193](#), [13194](#), [13271](#),
[13329](#), [13347](#), [13373](#), [13387](#), [13434](#),
[13495](#), [13573](#), [13592](#), [13630](#), [13645](#),
[13662](#), [13663](#), [13664](#), [13677](#), [13724](#),
[13731](#), [15953](#), [15954](#), [16024](#), [16031](#),
[16055](#), [16228](#), [16231](#), [16234](#), [16237](#),
[16240](#), [16269](#), [16270](#), [16271](#), [16272](#),
[16273](#), [16274](#), [16280](#), [16320](#), [16321](#),
[16322](#), [16323](#), [16328](#), [16329](#), [16351](#),
[16352](#), [16353](#), [16354](#), [16359](#), [16360](#),
[16361](#), [16362](#), [16379](#), [16380](#), [16405](#),
[16406](#), [16423](#), [16424](#), [16480](#), [16481](#),
[16531](#), [16544](#), [16545](#), [16563](#), [16589](#),
[16590](#), [16642](#), [16648](#), [16676](#), [16705](#),
[16715](#), [16738](#), [16739](#), [16816](#), [16839](#),
[16853](#), [16887](#), [16889](#), [17023](#), [17046](#),
[17061](#), [17062](#), [17067](#), [17068](#), [17070](#),
[17072](#), [17085](#), [17086](#), [17087](#), [17088](#),
[17097](#), [17098](#), [17099](#), [17100](#), [17105](#),
[17106](#), [17714](#), [17718](#), [17860](#), [17900](#),
[17901](#), [17902](#), [17903](#), [17917](#), [17918](#),
[17927](#), [17928](#), [17938](#), [17939](#), [17940](#),
[17941](#), [17951](#), [17952](#), [17953](#), [17954](#),
[17965](#), [17990](#), [17991](#), [18055](#), [18056](#),

18094, 18095, 18100, 18101, 18192,
 18229, 18232, 18264, 18293, 18337,
 18346, 18403, 18410, 18451, 18453,
 18455, 18608, 18609, 18610, 18611,
 18811, 18867, 19544, 19550, 19553,
 19556, 19559, 19562, 19578, 19581,
 19593, 19599, 19606, 19614, 19622,
 19654, 19655, 19656, 19657, 19664,
 19665, 19684, 19685, 19686, 19687,
 19698, 19708, 19780, 19782, 19784,
 19786, 19803, 19804, 19864, 19879,
 19902, 19908, 19910, 19946, 19952,
 19956, 19957, 19962, 19963, 19972,
 19973, 19976, 19979, 19987, 19988,
 19996, 19997, 20269, 20289, 20295,
 20298, 20299, 20304, 20305, 20314,
 20315, 20317, 20319, 20324, 20325,
 20330, 20331, 20359, 20360, 20362,
 20380, 20386, 20391, 20392, 20397,
 20398, 20407, 20408, 20410, 20412,
 20417, 20418, 20423, 20424, 20430,
 20720, 20803, 20806, 20822, 20877,
 20884, 20974, 21087, 21093, 21296,
 21310, 21316, 21326, 21352, 21358,
 21368, 21408, 21446, 21847, 21899,
 21932, 21943, 21981, 21984, 21994,
 21996, 22078, 22080, 22110, 22152,
 22161, 22170, 22179, 22187, 22240,
 22242, 22256, 22277, 22320, 22846,
 22849, 24541, 24548, 24549, 24550,
 24553, 24554, 24557, 24558, 24563,
 24564, 24571, 24572, 24573, 24574,
 24576, 24578, 24816, 24878, 27960,
 28014, 28092, 28137, 28152, 28206,
 28545, 28565, 28594, 28648, 28656,
 28664, 28750, 28789, 28802, 28861,
 28934, 28985, 29438, 29774, 29816,
 31542, 31977, 31982, 31983, 31988,
 31989, 31994, 31995, 32000, 32001,
 32009, 32010, 32011, 32014, 32020,
 32023, 32029, 32032, 32038, 32041,
 32044, 32045, 32073, 32074, 32082,
 32085, 32088, 32097, 32115, 32128,
 32129, 32140, 32141, 32154, 32155,
 32174, 32175, 32194, 32195, 32220,
 32221, 32232, 32233, 32244, 32245,
 32258, 32259, 32279, 32280, 32283,
 32284, 32287, 32293, 32308, 32311,
 32429, 32435, 32475, 32478, 32495,
 32498, 32515, 32518, 32532, 32535,
 32553, 32556, 32582, 32585, 32591,
 32597, 32713, 32722, 32735, 32748,
 32761, 32767, 32773, 32821, 32834,
 32843, 32850, 32890, 32902, 32942,
 32945, 32960, 32963, 32981, 33182,
 33185, 33345, 33352, 33389, 33392,
 33450, 33456, 33501, 33507, 33638,
 33741, 33817, 33844, 33847, 33869,
 33872, 33875, 33878, 33950, 33953,
 33954, 33987, 34168, 34831, 35209,
 35621, 35628, 35697, 35700, 35703,
 35706, 35754, 35757, 35839, 35842,
 35891, 35921, 35951, 35952, 36025,
 36026, 36189, 36192, 36237, 36241
 \cs_gset:Nn 18, 2034, 2099
 .cs_gset:Np 224, 21130
 \cs_gset:Npn 14, 16,
 1492, 1914, 1928, 1930, 7046, 9201,
 9203, 9241, 10999, 16105, 16106,
 16143, 16185, 16680, 18665, 21139,
 21141, 23280, 30152, 36385, 36593,
 36595, 36597, 36599, 36601, 36603,
 36610, 36620, 36623, 36626, 36629,
 36632, 36635, 36638, 36640, 36642,
 36644, 36646, 36648, 36650, 36652
 \cs_gset:Npx
 . . . 16, 1492, 1915, 1928, 1931, 16685
 \cs_gset_eq:NN 19,
 1946, 1963, 1971, 8392, 8394, 9090,
 10152, 10366, 11893, 11895, 11914,
 13998, 14002, 16226, 16451, 16690,
 16695, 18606, 19548, 19867, 19875
 \cs_gset_nopar:Nn 18, 2034, 2099
 \cs_gset_nopar:Npn
 . . . 16, 1492, 1912, 1920, 1924, 15913
 \cs_gset_nopar:Npx
 . 16, 665, 1368, 1492, 1913, 1920,
 1925, 11910, 11920, 11925, 16199,
 21835, 36133, 36159, 36164, 36191
 \cs_gset_protected:Nn 18, 2034, 2099
 .cs_gset_protected:Np . . . 224, 21130
 \cs_gset_protected:Npn . . 16, 1492,
 1918, 1940, 1942, 3364, 3372, 3385,
 3796, 4064, 4069, 10266, 12470,
 13333, 16748, 17332, 18213, 19870,
 20198, 21143, 21145, 24883, 28966,
 36330, 36347, 36356, 36380, 36605,
 36608, 36612, 36663, 36669, 36675
 \cs_gset_protected:Npx
 16, 1492, 1919,
 1940, 1943, 17343, 20205, 24890, 36336
 \cs_gset_protected_nopar:Nn
 19, 2034, 2099
 \cs_gset_protected_nopar:Npn
 17, 1492, 1916, 1934, 1936
 \cs_gset_protected_nopar:Npx
 17, 1492, 1917, 1934, 1937

- \cs_if_eq:NNTF . . . 27, [1375](#), [2131](#),
2138, 2139, 2142, 2143, 2146, 2147,
5767, 9136, 9251, 20997, 23045,
23055, 23081, 23083, 23085, 23285,
29633, 29653, 31476, 36330, 36356
- \cs_if_eq_p:NN 27,
[2131](#), [5784](#), [29707](#), [29708](#), [31523](#), [31524](#)
- \cs_if_exist 60
- \cs_if_exist:N
. 8459, 8461, 11965, 11966,
16330, 16332, 17073, 17075, 17919,
17921, 19805, 19807, 19964, 19966,
20306, 20308, 20399, 20401, 24617,
24618, 28935, 28937, 32002, 32004
- \cs_if_exist:NTF 21,
27, [343](#), [397](#), [598](#), [727](#), [868](#), [1798](#),
1855, 1857, 1859, 1861, 1863, 1865,
1867, 1869, 2151, 2277, 2347, 2383,
2480, 2504, 2572, 2603, 2854, 3120,
3362, 3380, 3383, 5143, 5495, 7191,
8708, 8709, 8710, 8712, 8716, 8745,
8978, 9048, 9134, 9172, 9970, 10084,
10088, 10114, 10315, 10319, 10777,
10938, 10997, 11029, 11038, 11181,
11257, 11411, 11464, 11589, 12629,
13969, 13978, 13982, 13996, 16435,
17048, 17049, 17050, 17051, 17726,
17727, 17773, 18662, 18815, 19117,
19136, 19709, 20454, 20612, 20734,
20829, 20920, 20925, 20946, 21475,
21517, 21525, 21537, 21549, 21555,
21566, 21582, 21669, 21730, 21739,
21819, 22120, 23278, 23452, 24531,
28811, 28911, 28964, 28999, 29686,
30002, 30010, 30188, 31320, 31327,
31680, 32686, 32688, 33962, 34416,
34796, 34801, 34860, 36066, 36085
- \cs_if_exist_p:N . . . 27, [343](#), [1798](#),
8776, 9067, 18950, 18951, 28887,
29669, 29716, 31488, 32802, 33686
- \cs_if_exist_use:N
. 21, [367](#), [1854](#), 6138, 9552, 9570,
10498, 21551, 21570, 29809, 29885
- \cs_if_exist_use:NTF [21](#), [1854](#), 1856,
1858, 1864, 1866, 3150, 3219, 4708,
4715, 5102, 5107, 5151, 5563, 5649,
7127, 9105, 16058, 22727, 23410,
23412, 30022, 30028, 30129, 30131,
33959, 33973, 34686, 35027, 35397
- \cs_if_free:NTF 27,
[62](#), [598](#), [1826](#), 1895, 3062, 3089, 9542
- \cs_if_free_p:N 26, 27, [62](#), [1826](#)
- \cs_log:N 20, [376](#), [2176](#)
- \cs_meaning:N 20, [353](#),
[1440](#), [1456](#), 1464, 2188, 8975, 28976
- \cs_new:Nn 17, [63](#), [2034](#), [2099](#)
- \cs_new:Npn 14, 15, 19, [62](#), [63](#),
[404](#), [419](#), [1375](#), 1582, 1599, [1904](#),
[1928](#), 1932, 2005, 2007, 2009, 2017,
2070, 2136, 2137, 2138, 2139, 2140,
2141, 2142, 2143, 2144, 2145, 2146,
2147, 2213, 2217, 2226, 2235, 2244,
2247, 2256, 2257, 2267, 2268, 2269,
2270, 2271, 2272, 2273, 2275, 2279,
2283, 2286, 2299, 2305, 2311, 2322,
2324, 2331, 2333, 2335, 2342, 2343,
2345, 2349, 2353, 2359, 2361, 2366,
2371, 2377, 2385, 2392, 2393, 2399,
2405, 2411, 2417, 2423, 2430, 2437,
2444, 2451, 2458, 2467, 2468, 2470,
2475, 2482, 2486, 2489, 2499, 2500,
2502, 2506, 2509, 2510, 2512, 2514,
2520, 2526, 2528, 2534, 2536, 2543,
2550, 2551, 2552, 2553, 2554, 2556,
2565, 2567, 2570, 2571, 2574, 2578,
2581, 2583, 2588, 2598, 2601, 2605,
2623, 2624, 2626, 2632, 2637, 2639,
2645, 2665, 2667, 2669, 2682, 2689,
2704, 2710, 2716, 2721, 2722, 2745,
2775, 2782, 2788, 2816, 2822, 2827,
2833, 2882, 2883, 2884, 2885, 2954,
2975, 2997, 3000, 3008, 3021, 3036,
3047, 3079, 3184, 3186, 3427, 3582,
3595, 3600, 3606, 3607, 3614, 3621,
3628, 3635, 3642, 3643, 3645, 3652,
3658, 3754, 3759, 3760, 3768, 3774,
3951, 3956, 3963, 4000, 4005, 4020,
4043, 4048, 4103, 4109, 4127, 4132,
4147, 4149, 4151, 4158, 4174, 4176,
4179, 4185, 4399, 4428, 4433, 4435,
4444, 4446, 4447, 4452, 4458, 4480,
4482, 4484, 4706, 4712, 4723, 4728,
4741, 4746, 4758, 4773, 4783, 4795,
4818, 4929, 4947, 4955, 4967, 4979,
5483, 5765, 5780, 5814, 5830, 5877,
5915, 5946, 5952, 5958, 5966, 5971,
5977, 5982, 5996, 6011, 6020, 6028,
6030, 6064, 6073, 6144, 6390, 6805,
6909, 6912, 6933, 6935, 6941, 6943,
6950, 6960, 7160, 7542, 7645, 7651,
7690, 7697, 7705, 7711, 7713, 8338,
8417, 8429, 8431, 8471, 8472, 8481,
8482, 8492, 8497, 8502, 8504, 8512,
8513, 8514, 8515, 8516, 8517, 8518,
8519, 8520, 8521, 8531, 8547, 8557,
8573, 8583, 8585, 8589, 8591, 8595,
8603, 8608, 8616, 8622, 8629, 8631,
8633, 8634, 8636, 8638, 8640, 8642,

8644, 8646, 8648, 8650, 8652, 8654,
8659, 8660, 8661, 8662, 8663, 8664,
8665, 8666, 8667, 8668, 8680, 8683,
9005, 9007, 9044, 9050, 9057, 9167,
9240, 9275, 9337, 9342, 9347, 9349,
9351, 9353, 9359, 9368, 9374, 9380,
9521, 9523, 9540, 9673, 9690, 9692,
9694, 10023, 10024, 10033, 10046,
10048, 10050, 10052, 10250, 10252,
10327, 10426, 10434, 10472, 10489,
10544, 10595, 10604, 10623, 10624,
10632, 10638, 10646, 10656, 10661,
10667, 10673, 10746, 10748, 10750,
10798, 10806, 10812, 10819, 10820,
10826, 10828, 10835, 10840, 10848,
10858, 10860, 10869, 10871, 10872,
10874, 10922, 10927, 10932, 10947,
10949, 10951, 10962, 10963, 10964,
10966, 10984, 10986, 11074, 11076,
11078, 11083, 11088, 11090, 11092,
11099, 11113, 11123, 11133, 11140,
11144, 11151, 11224, 11337, 11342,
11347, 11352, 11358, 11372, 11410,
11478, 11605, 11606, 11610, 11611,
12125, 12186, 12254, 12289, 12378,
12381, 12382, 12383, 12384, 12395,
12410, 12415, 12420, 12425, 12430,
12432, 12441, 12443, 12450, 12453,
12460, 12466, 12483, 12490, 12493,
12501, 12514, 12516, 12519, 12521,
12530, 12535, 12540, 12543, 12554,
12555, 12556, 12557, 12564, 12571,
12573, 12580, 12591, 12603, 12612,
12618, 12624, 12626, 12631, 12636,
12643, 12648, 12654, 12665, 12666,
12667, 12668, 12676, 12716, 12725,
12744, 12746, 12759, 12766, 12782,
12793, 12801, 12807, 12810, 12815,
12827, 12833, 12834, 12836, 12844,
12850, 12857, 12859, 12861, 12874,
12876, 12878, 12886, 12894, 12900,
12907, 12909, 12914, 12916, 12918,
12919, 12927, 12939, 12948, 12957,
12962, 12968, 12991, 12992, 12993,
12995, 13048, 13086, 13087, 13249,
13254, 13259, 13264, 13269, 13274,
13280, 13285, 13290, 13295, 13300,
13302, 13308, 13310, 13318, 13320,
13322, 13348, 13374, 13376, 13378,
13386, 13388, 13399, 13408, 13411,
13422, 13431, 13433, 13435, 13443,
13445, 13452, 13473, 13483, 13488,
13493, 13494, 13496, 13504, 13506,
13514, 13520, 13526, 13545, 13547,
13556, 13562, 13569, 13571, 13574,
13584, 13591, 13593, 13601, 13606,
13611, 13622, 13629, 13631, 13637,
13639, 13644, 13646, 13652, 13653,
13658, 13659, 13660, 13661, 13665,
13670, 13675, 13678, 13680, 13688,
13693, 13779, 13787, 13794, 13835,
13837, 13843, 13849, 13851, 13856,
13861, 13877, 13893, 13907, 14004,
14010, 14036, 14042, 14074, 14084,
14095, 14121, 14128, 14188, 14195,
14217, 14227, 14296, 14306, 14343,
14405, 14446, 14448, 14465, 14471,
14494, 14524, 14545, 14554, 14633,
14653, 14673, 14696, 14703, 14730,
14833, 14847, 14874, 14883, 14885,
14906, 14911, 14917, 14922, 14990,
15013, 15025, 15034, 15040, 15045,
15923, 15929, 15937, 15944, 15951,
15955, 15961, 15994, 16004, 16007,
16111, 16120, 16153, 16158, 16160,
16169, 16175, 16180, 16207, 16214,
16312, 16318, 16350, 16363, 16418,
16499, 16529, 16557, 16562, 16584,
16619, 16621, 16629, 16635, 16643,
16649, 16651, 16653, 16664, 16706,
16716, 16740, 16755, 16765, 16793,
16811, 16815, 16817, 16840, 16841,
16842, 16849, 16851, 16905, 16925,
16930, 16932, 16933, 16939, 16947,
16953, 16971, 16979, 16987, 17000,
17002, 17009, 17011, 17108, 17115,
17129, 17134, 17140, 17151, 17156,
17163, 17165, 17167, 17169, 17171,
17173, 17175, 17185, 17190, 17195,
17200, 17205, 17207, 17213, 17231,
17239, 17247, 17253, 17259, 17267,
17275, 17281, 17287, 17294, 17310,
17320, 17322, 17358, 17372, 17378,
17410, 17442, 17444, 17446, 17452,
17458, 17470, 17478, 17490, 17498,
17531, 17564, 17566, 17568, 17570,
17572, 17577, 17582, 17587, 17592,
17593, 17594, 17595, 17596, 17597,
17598, 17599, 17600, 17601, 17602,
17603, 17604, 17605, 17606, 17607,
17608, 17617, 17618, 17627, 17633,
17635, 17644, 17651, 17657, 17659,
17661, 17677, 17688, 17711, 17744,
17784, 17785, 17794, 17795, 17807,
17808, 17809, 17811, 17817, 17823,
17853, 17854, 17893, 17989, 18091,
18093, 18102, 18109, 18112, 18125,
18131, 18169, 18179, 18186, 18193,

18200, 18207, 18240, 18250, 18258,
18265, 18271, 18281, 18283, 18285,
18294, 18297, 18306, 18314, 18338,
18339, 18342, 18344, 18347, 18357,
18368, 18370, 18373, 18379, 18380,
18388, 18404, 18411, 18419, 18421,
18435, 18437, 18438, 18439, 18441,
18446, 18494, 18564, 18570, 18576,
18582, 18613, 18619, 18656, 18720,
18735, 18740, 18784, 18786, 18788,
18791, 18794, 18800, 18806, 18812,
18813, 18825, 18835, 18837, 18839,
18848, 18850, 18859, 18865, 18868,
18878, 18884, 18891, 18893, 18925,
18927, 18929, 18935, 18937, 18943,
19065, 19095, 19242, 19254, 19255,
19263, 19272, 19281, 19290, 19292,
19294, 19296, 19298, 19300, 19302,
19304, 19306, 19308, 19310, 19312,
19314, 19321, 19327, 19334, 19335,
19336, 19337, 19340, 19434, 19442,
19444, 19446, 19456, 19466, 19535,
19641, 19688, 19693, 19699, 19707,
19711, 19727, 19731, 19742, 19822,
19840, 19852, 19880, 19890, 19903,
19905, 19926, 19940, 19998, 20003,
20005, 20013, 20021, 20029, 20031,
20043, 20049, 20062, 20064, 20066,
20068, 20070, 20078, 20083, 20088,
20093, 20098, 20100, 20106, 20108,
20116, 20124, 20130, 20136, 20144,
20152, 20158, 20164, 20171, 20185,
20219, 20221, 20227, 20240, 20241,
20248, 20256, 20258, 20260, 20346,
20349, 20354, 20357, 20425, 20456,
20468, 20477, 20483, 20485, 20487,
20497, 20503, 20508, 20515, 20523,
20527, 20534, 20536, 20544, 20548,
20555, 20565, 20573, 20581, 20593,
20602, 20616, 20617, 20618, 20619,
20621, 20637, 20648, 20656, 20661,
20667, 20799, 21076, 21591, 21593,
21658, 21667, 21673, 21675, 21680,
21684, 21688, 21692, 21698, 21710,
21714, 21718, 21970, 21972, 21982,
21995, 22007, 22079, 22081, 22083,
22094, 22153, 22155, 22162, 22168,
22186, 22188, 22196, 22205, 22215,
22328, 22329, 22330, 22331, 22332,
22333, 22334, 22335, 22336, 22337,
22347, 22371, 22373, 22375, 22384,
22386, 22393, 22405, 22406, 22408,
22418, 22428, 22438, 22448, 22456,
22458, 22465, 22467, 22468, 22473,
22480, 22494, 22496, 22512, 22513,
22521, 22523, 22532, 22534, 22546,
22551, 22555, 22560, 22562, 22564,
22566, 22568, 22575, 22577, 22585,
22587, 22599, 22601, 22603, 22605,
22629, 22631, 22633, 22634, 22635,
22637, 22639, 22641, 22643, 22661,
22676, 22677, 22683, 22699, 22705,
22833, 22834, 22835, 22836, 22837,
22838, 22839, 22844, 22847, 22893,
22895, 22897, 22899, 22905, 22909,
22911, 22920, 22921, 22930, 22943,
22956, 22963, 22977, 22993, 23005,
23016, 23026, 23032, 23043, 23053,
23079, 23090, 23107, 23118, 23123,
23143, 23145, 23156, 23161, 23174,
23197, 23198, 23202, 23219, 23220,
23244, 23252, 23270, 23299, 23325,
23329, 23332, 23334, 23340, 23352,
23364, 23371, 23377, 23385, 23408,
23423, 23442, 23450, 23465, 23480,
23491, 23501, 23511, 23516, 23525,
23542, 23555, 23560, 23566, 23568,
23575, 23605, 23633, 23649, 23660,
23665, 23683, 23701, 23712, 23727,
23732, 23743, 23753, 23763, 23779,
23823, 23828, 23835, 23843, 23849,
23854, 23858, 23875, 23883, 23915,
23932, 23946, 23965, 23973, 23982,
23991, 24002, 24004, 24018, 24028,
24029, 24046, 24053, 24058, 24071,
24084, 24089, 24117, 24131, 24159,
24160, 24164, 24181, 24203, 24205,
24216, 24248, 24252, 24267, 24284,
24308, 24310, 24312, 24314, 24324,
24329, 24340, 24352, 24363, 24376,
24396, 24414, 24416, 24428, 24434,
24442, 24456, 24463, 24474, 24481,
24495, 24591, 24613, 24615, 24632,
24654, 24659, 24676, 24703, 24704,
24705, 24706, 24722, 24733, 24741,
24753, 24759, 24765, 24773, 24781,
24787, 24793, 24801, 24809, 24827,
24840, 24862, 24910, 24916, 24927,
24951, 24953, 24955, 24957, 24965,
24969, 24976, 24983, 24984, 24985,
24986, 24987, 24988, 24991, 24993,
25022, 25030, 25041, 25043, 25045,
25047, 25054, 25078, 25080, 25090,
25105, 25114, 25128, 25136, 25144,
25151, 25158, 25166, 25176, 25190,
25201, 25202, 25208, 25225, 25232,
25234, 25241, 25246, 25263, 25264,
25265, 25284, 25290, 25300, 25312,

25319, 25333, 25341, 25379, 25388,
25409, 25411, 25413, 25422, 25433,
25445, 25460, 25473, 25486, 25494,
25512, 25530, 25537, 25545, 25555,
25556, 25565, 25566, 25575, 25585,
25599, 25609, 25620, 25628, 25630,
25641, 25647, 25682, 25703, 25705,
25707, 25709, 25716, 25725, 25730,
25737, 25744, 25764, 25769, 25786,
25797, 25802, 25812, 25814, 25824,
25831, 25833, 25839, 25841, 25843,
25847, 25866, 25867, 25872, 25880,
25881, 25904, 25917, 25924, 25932,
25933, 25934, 25935, 25936, 25937,
25945, 25951, 25953, 25955, 25977,
25982, 25992, 26002, 26013, 26026,
26037, 26042, 26049, 26058, 26060,
26069, 26078, 26092, 26094, 26096,
26109, 26119, 26124, 26133, 26141,
26148, 26154, 26163, 26165, 26177,
26182, 26190, 26195, 26205, 26211,
26217, 26224, 26231, 26233, 26238,
26240, 26245, 26247, 26261, 26271,
26283, 26288, 26295, 26305, 26307,
26309, 26320, 26334, 26348, 26368,
26381, 26383, 26388, 26401, 26406,
26414, 26419, 26429, 26441, 26471,
26472, 26473, 26475, 26477, 26479,
26493, 26499, 26508, 26527, 26533,
26543, 26562, 26570, 26603, 26609,
26618, 26620, 26634, 26693, 26701,
26719, 26736, 26737, 26742, 26767,
26790, 26818, 26834, 26844, 26855,
26876, 26891, 26896, 26901, 26903,
26917, 26923, 26938, 26946, 26956,
26966, 26979, 26997, 27003, 27017,
27032, 27070, 27072, 27074, 27076,
27078, 27093, 27108, 27123, 27138,
27153, 27168, 27176, 27190, 27192,
27198, 27210, 27218, 27225, 27451,
27458, 27495, 27503, 27504, 27515,
27522, 27524, 27530, 27541, 27551,
27558, 27565, 27580, 27619, 27632,
27663, 27669, 27676, 27696, 27698,
27715, 27730, 27743, 27750, 27755,
27757, 27766, 27779, 27782, 27803,
27816, 27831, 27849, 27864, 27874,
27883, 27896, 27912, 27929, 27942,
27948, 27950, 27955, 27956, 27957,
27958, 27961, 27966, 27972, 27977,
27979, 28002, 28010, 28012, 28015,
28020, 28026, 28031, 28033, 28056,
28081, 28090, 28091, 28093, 28098,
28100, 28105, 28107, 28117, 28125,
28133, 28135, 28138, 28143, 28148,
28150, 28151, 28153, 28158, 28163,
28165, 28170, 28177, 28191, 28196,
28198, 28208, 28210, 28212, 28214,
28216, 28227, 28237, 28239, 28245,
28252, 28258, 28268, 28273, 28275,
28283, 28284, 28298, 28305, 28311,
28312, 28325, 28340, 28346, 28368,
28383, 28393, 28414, 28423, 28446,
28464, 28475, 28480, 28491, 28508,
28513, 28560, 28566, 28580, 28649,
28657, 28665, 28671, 28678, 28683,
28689, 28701, 28793, 28898, 28900,
28907, 28919, 28923, 28927, 29252,
29257, 29272, 29282, 29293, 29307,
29324, 29337, 29339, 29340, 29421,
29429, 29436, 29439, 29441, 29447,
29458, 29470, 29494, 29514, 29530,
29537, 29551, 29562, 29571, 29576,
29582, 29591, 29600, 29605, 29616,
29618, 29623, 29629, 29643, 29649,
29675, 29680, 29682, 29693, 29699,
29704, 29712, 29713, 29728, 29729,
29742, 29747, 29757, 29764, 29780,
29782, 29784, 29786, 29788, 29790,
29792, 29794, 29796, 29806, 29814,
29817, 29819, 29825, 29836, 29841,
29855, 29867, 29881, 29888, 29894,
29899, 29905, 29919, 29930, 29939,
29946, 29953, 29960, 29969, 29974,
29985, 29987, 29992, 29996, 29998,
30000, 30020, 30026, 30036, 30042,
30053, 30065, 30072, 30078, 30088,
30123, 30125, 30127, 30136, 30165,
30174, 30180, 30182, 30184, 30186,
30197, 30205, 30211, 30233, 30247,
30256, 30258, 30268, 30295, 30302,
30313, 30322, 30335, 30343, 30453,
30461, 30474, 30486, 30501, 30508,
30521, 30546, 30556, 30563, 30586,
30599, 30609, 30616, 30625, 30637,
30644, 30664, 30678, 30689, 30707,
30720, 31348, 31357, 31364, 31366,
31368, 31374, 31385, 31386, 31391,
31396, 31403, 31414, 31419, 31421,
31423, 31428, 31433, 31444, 31455,
31460, 31467, 31472, 31483, 31485,
31508, 31509, 31515, 31520, 31532,
31600, 31678, 31968, 33943, 33944,
33988, 33990, 33992, 33994, 33996,
33998, 34003, 34010, 34012, 34018,
34162, 34169, 34174, 34176, 34183,
34191, 34193, 34202, 34212, 34214,
34216, 34218, 34220, 34225, 34233,

- 34239, 34251, 34253, 34254, 34255,
 34256, 34257, 34258, 34259, 34260,
 34267, 34269, 34278, 34287, 34301,
 34343, 34350, 34684, 34742, 34883,
 34894, 34901, 34909, 34915, 34923,
 34925, 34957, 34959, 35038, 35044,
 35046, 35055, 35068, 35083, 35096,
 35110, 35201, 35227, 35238, 35245,
 35247, 35260, 35266, 35277, 35292,
 35298, 35303, 35313, 35319, 35329,
 35384, 35385, 35439, 35622, 35629,
 35630, 35634, 35642, 35654, 35681,
 35683, 35684, 35805, 35845, 35847,
 35849, 35851, 35853, 35855, 35857,
 35859, 35861, 35866, 35872, 35878,
 35881, 35882, 35892, 35900, 35902,
 35908, 35914, 36007, 36014, 36048,
 36208, 36216, 36236, 36238, 36239,
 36242, 36244, 36246, 36248, 36256
 \cs_new:Npx
 15, 39, 398, 399, 1904, 1928, 1933,
 2913, 3976, 4729, 4731, 4733, 4735,
 4737, 4739, 10471, 10483, 10975,
 15047, 22487, 23311, 23895, 25049,
 27057, 27063, 27675, 29475, 29663,
 30102, 30224, 30734, 34931, 35088,
 35228, 35818, 35820, 35822, 35829
 \cs_new_eq:NN
 19, 64, 368, 370, 863, 1718, 1946,
 2246, 2255, 2292, 2569, 2863, 2864,
 2865, 2866, 2867, 2868, 2869, 2870,
 2871, 2872, 2873, 2874, 2875, 2876,
 2877, 2880, 2881, 3122, 3730, 3731,
 3732, 4396, 4402, 4550, 4551, 4552,
 4651, 4659, 4680, 4719, 4720, 4721,
 4722, 4908, 6676, 6942, 7340, 7862,
 8377, 8379, 8399, 8400, 8694, 8695,
 8696, 8697, 8700, 8701, 8702, 8703,
 8969, 10094, 10116, 10184, 10329,
 10427, 10921, 11215, 11256, 11324,
 11330, 11600, 11601, 11602, 11909,
 11910, 12255, 12256, 13097, 13111,
 13112, 13115, 13116, 13182, 13203,
 13718, 13725, 14053, 14069, 14071,
 15061, 15921, 16195, 16198, 16221,
 16241, 16242, 16243, 16244, 16245,
 16246, 16247, 16248, 16478, 16854,
 16855, 16856, 16857, 16858, 16859,
 16860, 16861, 16862, 16863, 16864,
 16865, 16866, 16867, 16868, 16869,
 16870, 16871, 16872, 16873, 16874,
 16875, 16876, 16877, 16878, 16879,
 16918, 16919, 16920, 16921, 16922,
 17052, 17053, 17056, 17107, 17357,
 17713, 17717, 17803, 17856, 17857,
 17861, 17862, 17863, 17864, 17865,
 17866, 17867, 17868, 17869, 17870,
 17871, 17872, 17873, 17874, 17875,
 17876, 18017, 18018, 18019, 18020,
 18021, 18022, 18023, 18024, 18025,
 18026, 18027, 18028, 18029, 18030,
 18031, 18032, 18612, 18664, 18966,
 18968, 18969, 18970, 18972, 18975,
 18976, 19330, 19331, 19332, 19563,
 19564, 19565, 19566, 19567, 19568,
 19569, 19570, 19741, 19935, 19936,
 19937, 20239, 20268, 20272, 20273,
 20296, 20297, 20351, 20352, 20353,
 20356, 20361, 20365, 20366, 20427,
 20428, 20429, 20433, 20434, 20476,
 22053, 22054, 22325, 22326, 22327,
 22531, 22698, 22976, 23004, 23012,
 23013, 23014, 23023, 23025, 23822,
 23941, 23942, 23943, 24540, 24551,
 24552, 28205, 28207, 28251, 29853,
 29994, 30121, 30150, 30199, 30201,
 30203, 30756, 30758, 31967, 32006,
 32007, 32008, 32042, 32043, 32054,
 32055, 32056, 32161, 32192, 32193,
 32266, 32281, 32282, 32683, 32910,
 32911, 32912, 32913, 32914, 32915,
 33911, 33912, 34882, 35040, 35064,
 35079, 35946, 36145, 36146, 36580
 \cs_new_nopar:Nn 17, 2034, 2099
 \cs_new_nopar:Npn
 15, 368, 369, 1904, 1920, 1926
 \cs_new_nopar:Npx 15, 1904, 1920, 1927
 \cs_new_protected:Nn . 17, 2034, 2099
 \cs_new_protected:Npn
 15, 404, 1375, 1586,
 1603, 1904, 1940, 1944, 1946, 1947,
 1948, 1949, 1950, 1951, 1952, 1953,
 1954, 1959, 1960, 1961, 1962, 1964,
 2019, 2029, 2031, 2042, 2051, 2149,
 2158, 2160, 2162, 2164, 2166, 2174,
 2176, 2177, 2179, 2180, 2182, 2190,
 2192, 2194, 2258, 2293, 2465, 2494,
 2564, 2595, 2887, 2900, 2918, 2922,
 2925, 2934, 3058, 3075, 3085, 3094,
 3118, 3126, 3138, 3146, 3157, 3159,
 3161, 3163, 3165, 3176, 3178, 3191,
 3199, 3210, 3229, 3231, 3233, 3235,
 3237, 3322, 3341, 3348, 3360, 3393,
 3412, 3414, 3416, 3435, 3438, 3441,
 3447, 3453, 3468, 3477, 3497, 3507,
 3518, 3528, 3538, 3539, 3546, 3552,
 3562, 3572, 3668, 3674, 3676, 3691,
 3776, 3787, 3805, 3815, 3817, 3841,

3848, 3860, 3863, 3866, 3876, 3884,
3891, 3900, 3915, 3932, 3943, 4053,
4060, 4062, 4080, 4082, 4084, 4094,
4096, 4098, 4186, 4200, 4211, 4225,
4244, 4246, 4251, 4257, 4271, 4279,
4286, 4293, 4295, 4305, 4315, 4346,
4352, 4354, 4359, 4364, 4373, 4397,
4400, 4403, 4405, 4414, 4420, 4426,
4486, 4488, 4489, 4494, 4500, 4508,
4518, 4532, 4553, 4563, 4571, 4573,
4581, 4593, 4612, 4614, 4619, 4627,
4629, 4636, 4641, 4643, 4645, 4652,
4660, 4662, 4664, 4666, 4673, 4678,
4681, 4687, 4923, 4987, 5000, 5011,
5024, 5057, 5086, 5095, 5100, 5105,
5110, 5131, 5140, 5147, 5156, 5161,
5168, 5181, 5183, 5185, 5187, 5193,
5215, 5228, 5255, 5260, 5294, 5329,
5350, 5352, 5360, 5362, 5364, 5366,
5368, 5370, 5374, 5385, 5401, 5414,
5420, 5431, 5444, 5450, 5469, 5489,
5520, 5531, 5546, 5559, 5577, 5585,
5590, 5592, 5594, 5611, 5630, 5632,
5655, 5667, 5687, 5708, 5715, 5722,
5733, 5740, 5746, 5804, 5856, 5865,
5878, 5893, 5902, 5921, 5940, 6079,
6150, 6160, 6162, 6164, 6171, 6216,
6229, 6245, 6247, 6249, 6254, 6269,
6275, 6298, 6321, 6336, 6343, 6350,
6352, 6354, 6361, 6375, 6391, 6400,
6414, 6426, 6443, 6452, 6454, 6466,
6475, 6487, 6500, 6507, 6527, 6558,
6592, 6610, 6619, 6625, 6631, 6637,
6680, 6689, 6703, 6722, 6749, 6754,
6764, 6776, 6814, 6823, 6835, 6842,
6844, 6846, 6866, 6871, 6877, 6888,
6893, 6898, 6914, 6966, 6985, 6987,
7030, 7054, 7071, 7077, 7079, 7099,
7125, 7136, 7145, 7154, 7187, 7201,
7212, 7218, 7227, 7235, 7270, 7276,
7279, 7287, 7293, 7296, 7305, 7308,
7311, 7314, 7319, 7328, 7331, 7334,
7339, 7345, 7350, 7355, 7360, 7361,
7362, 7370, 7371, 7372, 7393, 7395,
7397, 7405, 7407, 7409, 7413, 7414,
7429, 7446, 7448, 7450, 7452, 7469,
7471, 7473, 7488, 7496, 7506, 7517,
7526, 7543, 7555, 7565, 7574, 7614,
7669, 7674, 7718, 7740, 7753, 7755,
7786, 7788, 7817, 7833, 7844, 7849,
7863, 7869, 7871, 7872, 7878, 7880,
7881, 7887, 7889, 7891, 7897, 7899,
7901, 7909, 7929, 7935, 7941, 7947,
7955, 7957, 7959, 7961, 7963, 7965,
7967, 7969, 7971, 7976, 8005, 8015,
8020, 8029, 8031, 8041, 8354, 8356,
8358, 8365, 8379, 8381, 8387, 8389,
8391, 8393, 8403, 8408, 8432, 8434,
8436, 8438, 8440, 8690, 8777, 8794,
8848, 8856, 8867, 8890, 8920, 8924,
8952, 8956, 8960, 8965, 9020, 9024,
9091, 9097, 9175, 9183, 9188, 9190,
9197, 9199, 9206, 9247, 9276, 9296,
9301, 9312, 9389, 9391, 9432, 9455,
9511, 9525, 9550, 9572, 9573, 9586,
9591, 9617, 9626, 9628, 9630, 9647,
9674, 9676, 9678, 9680, 9682, 9684,
9686, 9688, 10094, 10098, 10112,
10123, 10144, 10156, 10158, 10160,
10172, 10173, 10174, 10201, 10203,
10214, 10216, 10235, 10237, 10239,
10254, 10256, 10258, 10264, 10271,
10280, 10282, 10284, 10289, 10329,
10333, 10336, 10350, 10358, 10370,
10372, 10374, 10386, 10387, 10388,
10398, 10401, 10404, 10410, 10416,
10423, 10425, 10435, 10466, 10477,
10495, 10530, 10553, 10565, 10584,
10588, 10677, 10693, 10702, 10710,
10719, 10726, 10732, 10881, 10915,
11012, 11066, 11157, 11159, 11161,
11163, 11173, 11201, 11274, 11279,
11285, 11286, 11291, 11310, 11325,
11331, 11382, 11387, 11394, 11395,
11396, 11423, 11436, 11448, 11458,
11460, 11462, 11471, 11479, 11607,
11608, 11888, 11890, 11892, 11894,
11896, 11901, 11911, 11917, 11922,
11929, 11931, 11935, 11937, 11941,
11943, 11947, 11955, 11973, 11975,
11977, 11979, 11981, 11983, 11993,
11998, 12003, 12011, 12013, 12018,
12023, 12031, 12041, 12043, 12048,
12056, 12058, 12060, 12065, 12073,
12091, 12097, 12099, 12101, 12103,
12115, 12134, 12149, 12167, 12189,
12191, 12193, 12195, 12201, 12223,
12229, 12257, 12259, 12263, 12265,
12351, 12352, 12353, 12467, 12480,
12507, 12509, 12511, 12583, 12585,
12880, 12882, 13014, 13016, 13018,
13034, 13036, 13049, 13051, 13143,
13145, 13147, 13149, 13155, 13170,
13183, 13185, 13189, 13191, 13330,
13345, 13354, 13364, 13366, 13719,
13726, 13735, 13867, 13883, 13899,
13905, 13909, 13911, 13931, 13949,
13957, 13967, 13976, 14060, 14068,

14070, 14072, 14082, 14088, 14112,
14123, 14129, 14132, 14134, 14139,
14165, 14171, 14177, 14205, 14279,
14327, 14380, 14463, 14469, 14492,
14522, 14543, 14618, 14714, 14719,
14721, 14723, 14799, 14801, 14803,
14808, 14818, 14897, 14902, 14904,
14957, 14959, 14961, 14966, 14976,
15910, 16012, 16014, 16025, 16032,
16038, 16056, 16065, 16070, 16075,
16080, 16085, 16091, 16096, 16103,
16109, 16118, 16128, 16130, 16132,
16134, 16136, 16141, 16186, 16223,
16229, 16232, 16235, 16238, 16249,
16254, 16259, 16264, 16275, 16281,
16283, 16285, 16287, 16289, 16324,
16326, 16334, 16342, 16355, 16357,
16365, 16367, 16369, 16381, 16383,
16385, 16407, 16409, 16411, 16438,
16439, 16440, 16462, 16473, 16503,
16511, 16521, 16532, 16534, 16536,
16538, 16546, 16564, 16566, 16568,
16677, 16682, 16687, 16693, 16699,
16728, 16745, 16773, 16775, 16777,
16783, 16785, 16787, 16886, 16888,
16890, 17018, 17024, 17026, 17059,
17060, 17063, 17065, 17069, 17071,
17077, 17079, 17081, 17083, 17089,
17091, 17093, 17095, 17101, 17103,
17109, 17324, 17326, 17328, 17335,
17337, 17339, 17351, 17715, 17719,
17742, 17747, 17748, 17759, 17761,
17762, 17763, 17810, 17858, 17877,
17879, 17881, 17904, 17906, 17908,
17923, 17925, 17929, 17931, 17933,
17942, 17944, 17946, 17955, 17963,
17966, 17968, 17970, 17978, 18006,
18035, 18037, 18039, 18057, 18059,
18061, 18096, 18098, 18142, 18208,
18224, 18230, 18233, 18235, 18452,
18454, 18456, 18474, 18475, 18476,
18492, 18496, 18498, 18500, 18502,
18504, 18506, 18508, 18510, 18512,
18514, 18516, 18518, 18520, 18522,
18524, 18526, 18528, 18530, 18532,
18534, 18536, 18538, 18540, 18542,
18544, 18546, 18548, 18550, 18552,
18554, 18556, 18558, 18560, 18562,
18566, 18568, 18572, 18574, 18578,
18580, 18584, 18596, 19341, 19343,
19345, 19350, 19357, 19366, 19377,
19382, 19396, 19404, 19422, 19424,
19426, 19428, 19430, 19432, 19492,
19509, 19514, 19521, 19526, 19528,
19545, 19551, 19554, 19557, 19560,
19576, 19579, 19582, 19588, 19594,
19600, 19607, 19615, 19623, 19625,
19631, 19633, 19642, 19648, 19658,
19666, 19675, 19767, 19768, 19769,
19788, 19790, 19792, 19865, 19907,
19909, 19911, 19941, 19947, 19953,
19954, 19958, 19960, 19968, 19970,
19974, 19977, 19980, 19982, 19989,
19991, 20072, 20194, 20201, 20213,
20270, 20274, 20284, 20290, 20300,
20302, 20310, 20312, 20316, 20318,
20320, 20322, 20326, 20328, 20363,
20367, 20375, 20381, 20387, 20389,
20393, 20395, 20403, 20405, 20409,
20411, 20413, 20415, 20419, 20421,
20431, 20435, 20712, 20714, 20721,
20726, 20731, 20744, 20751, 20755,
20764, 20769, 20778, 20784, 20801,
20804, 20807, 20823, 20825, 20827,
20843, 20854, 20856, 20858, 20875,
20878, 20885, 20900, 20913, 20918,
20929, 20931, 20933, 20952, 20959,
20961, 20975, 20985, 21013, 21022,
21031, 21064, 21077, 21088, 21094,
21096, 21098, 21100, 21102, 21104,
21106, 21108, 21110, 21112, 21114,
21116, 21118, 21120, 21122, 21124,
21126, 21128, 21130, 21132, 21134,
21136, 21138, 21140, 21142, 21144,
21146, 21148, 21150, 21152, 21154,
21156, 21158, 21160, 21162, 21164,
21166, 21168, 21170, 21172, 21174,
21176, 21178, 21180, 21182, 21184,
21186, 21188, 21190, 21192, 21194,
21196, 21198, 21200, 21202, 21204,
21206, 21208, 21210, 21212, 21214,
21216, 21218, 21220, 21222, 21224,
21226, 21228, 21230, 21232, 21234,
21236, 21238, 21240, 21242, 21244,
21246, 21248, 21250, 21252, 21254,
21256, 21258, 21260, 21262, 21264,
21266, 21268, 21270, 21272, 21274,
21276, 21297, 21299, 21305, 21311,
21317, 21324, 21327, 21346, 21353,
21359, 21366, 21369, 21388, 21409,
21411, 21417, 21422, 21427, 21447,
21452, 21456, 21462, 21467, 21473,
21487, 21513, 21532, 21547, 21561,
21577, 21601, 21625, 21657, 21744,
21746, 21748, 21831, 21837, 21922,
21924, 21985, 22022, 22048, 22057,
22066, 22101, 22103, 22111, 22122,
22130, 22137, 22143, 22171, 22180,

22224, 22229, 22239, 22241, 22243,
 22254, 22259, 22264, 22278, 22283,
 22288, 22303, 22314, 22338, 22341,
 22452, 22725, 22742, 22744, 22746,
 22748, 22776, 22778, 22780, 22782,
 22802, 22804, 22806, 22808, 22810,
 22812, 22814, 22816, 22818, 24539,
 24542, 24544, 24546, 24555, 24556,
 24559, 24561, 24565, 24566, 24567,
 24568, 24569, 24575, 24577, 24579,
 24598, 24600, 24879, 24886, 24898,
 27708, 28533, 28546, 28585, 28595,
 28607, 28612, 28622, 28630, 28636,
 28715, 28720, 28727, 28729, 28731,
 28753, 28755, 28767, 28778, 28800,
 28805, 28818, 28831, 28839, 28848,
 28862, 28873, 28879, 28960, 28973,
 28980, 29769, 31537, 31970, 31972,
 31978, 31980, 31984, 31986, 31990,
 31992, 31996, 31998, 32012, 32015,
 32021, 32024, 32030, 32033, 32039,
 32046, 32048, 32050, 32052, 32069,
 32071, 32080, 32083, 32086, 32089,
 32091, 32098, 32116, 32118, 32123,
 32130, 32135, 32142, 32148, 32156,
 32162, 32168, 32176, 32181, 32186,
 32188, 32190, 32196, 32198, 32200,
 32205, 32210, 32215, 32222, 32227,
 32234, 32239, 32246, 32252, 32260,
 32267, 32273, 32285, 32288, 32306,
 32309, 32312, 32322, 32356, 32367,
 32378, 32389, 32400, 32411, 32424,
 32430, 32436, 32451, 32458, 32468,
 32473, 32476, 32479, 32493, 32496,
 32499, 32513, 32516, 32519, 32530,
 32533, 32536, 32551, 32554, 32557,
 32566, 32580, 32583, 32586, 32592,
 32598, 32610, 32696, 32705, 32714,
 32723, 32736, 32749, 32762, 32768,
 32774, 32809, 32822, 32835, 32836,
 32837, 32844, 32851, 32877, 32878,
 32879, 32891, 32916, 32926, 32933,
 32940, 32943, 32946, 32958, 32961,
 32964, 32976, 32982, 32988, 32994,
 32996, 32998, 33004, 33025, 33027,
 33029, 33035, 33069, 33084, 33180,
 33183, 33186, 33226, 33244, 33250,
 33256, 33268, 33287, 33296, 33307,
 33313, 33318, 33326, 33339, 33346,
 33353, 33365, 33387, 33390, 33393,
 33408, 33415, 33421, 33430, 33437,
 33445, 33451, 33457, 33496, 33502,
 33508, 33529, 33538, 33557, 33562,
 33576, 33581, 33594, 33605, 33617,
 33631, 33692, 33697, 33734, 33742,
 33766, 33810, 33818, 33842, 33845,
 33848, 33867, 33870, 33873, 33876,
 33879, 33913, 33919, 33924, 33926,
 33945, 33951, 33955, 34035, 34042,
 34049, 34068, 34081, 34091, 34116,
 34139, 34148, 34160, 34161, 34353,
 34367, 34381, 34387, 34394, 34403,
 34414, 34423, 34432, 34437, 34445,
 34453, 34462, 34481, 34496, 34501,
 34506, 34521, 34522, 34527, 34532,
 34538, 34544, 34547, 34552, 34558,
 34572, 34592, 34609, 34619, 34633,
 34666, 34676, 34682, 34693, 34695,
 34726, 34729, 34731, 34733, 34751,
 34788, 34794, 34811, 34832, 34845,
 34858, 34878, 34891, 34906, 34920,
 34929, 34939, 34951, 34967, 34980,
 35016, 35021, 35036, 35042, 35053,
 35066, 35081, 35119, 35132, 35171,
 35177, 35179, 35184, 35189, 35205,
 35210, 35215, 35220, 35225, 35354,
 35367, 35380, 35390, 35395, 35404,
 35409, 35414, 35419, 35421, 35423,
 35600, 35608, 35616, 35623, 35666,
 35668, 35673, 35685, 35687, 35695,
 35698, 35701, 35704, 35707, 35752,
 35755, 35758, 35813, 35815, 35817,
 35837, 35840, 35922, 35924, 35926,
 35932, 35934, 35936, 35942, 35944,
 35947, 35949, 35959, 35965, 35978,
 35996, 36021, 36023, 36033, 36041,
 36108, 36114, 36130, 36132, 36134,
 36136, 36147, 36152, 36157, 36162,
 36167, 36184, 36185, 36187, 36190,
 36193, 36204, 36206, 36218, 36223,
 36228, 36271, 36273, 36275, 36277,
 36294, 36307, 36312, 36320, 36322,
 36334, 36353, 36360, 36377, 36382
 \cs_new_protected:Npx
 15, 398, 399, 404, 1904,
 1940, 1945, 2036, 2101, 2902, 2906,
 2911, 3070, 3074, 5323, 5337, 5339,
 9400, 9402, 9404, 9406, 9408, 9417,
 9419, 9421, 10135, 10897, 11297,
 13124, 18602, 19481, 32799, 34024,
 34372, 34602, 34713, 34816, 34822
 \cs_new_protected_nopar:Nn
 17, 2034, 2099
 \cs_new_protected_nopar:Npn
 15, 1904, 1921, 1934, 1938
 \cs_new_protected_nopar:Npx
 15, 1904, 1934, 1939
 \cs_prefix_spec:N

- [22](#), [2211](#), [13023](#), [13056](#), [36576](#)
- \cs_replacement_spec:N
 [23](#), [2211](#), [21757](#), [36578](#)
- \cs_set:Nn [17](#), [373](#), [2034](#), [2099](#)
- .cs_set:Np [224](#), [21130](#)
- \cs_set:Npn ... [14](#), [16](#), [62](#), [63](#), [360](#),
[369](#), [373](#), [885](#), [1478](#), [1508](#), [1515](#),
[1517](#), [1520](#), [1521](#), [1522](#), [1523](#), [1524](#),
[1525](#), [1526](#), [1527](#), [1528](#), [1529](#), [1530](#),
[1531](#), [1532](#), [1533](#), [1534](#), [1535](#), [1536](#),
[1537](#), [1538](#), [1539](#), [1540](#), [1542](#), [1543](#),
[1544](#), [1545](#), [1546](#), [1547](#), [1548](#), [1549](#),
[1550](#), [1573](#), [1575](#), [1577](#), [1580](#), [1597](#),
[1646](#), [1649](#), [1711](#), [1712](#), [1713](#), [1714](#),
[1762](#), [1764](#), [1766](#), [1768](#), [1773](#), [1779](#),
[1780](#), [1784](#), [1791](#), [1794](#), [1854](#), [1856](#),
[1858](#), [1860](#), [1862](#), [1864](#), [1866](#), [1868](#),
[1887](#), [1904](#), [1920](#), [1928](#), [1928](#), [2034](#),
[2099](#), [3408](#), [4691](#), [4692](#), [4693](#), [5019](#),
[5020](#), [5598](#), [5600](#), [5617](#), [5619](#), [5859](#),
[5860](#), [6106](#), [6107](#), [6108](#), [6109](#), [6135](#),
[6180](#), [6725](#), [7032](#), [8973](#), [9192](#), [9194](#),
[10442](#), [10764](#), [11620](#), [12175](#), [12232](#),
[12360](#), [13172](#), [14824](#), [14981](#), [16151](#),
[16173](#), [16955](#), [16963](#), [18070](#), [18159](#),
[18651](#), [18953](#), [19354](#), [19368](#), [19635](#),
[21131](#), [21133](#), [21696](#), [22751](#), [22759](#),
[22768](#), [22785](#), [22793](#), [22821](#), [29099](#),
[31642](#), [31643](#), [31705](#), [31712](#), [31717](#),
[31718](#), [36283](#), [36285](#), [36291](#), [36358](#)
- \cs_set:Npx [16](#), [380](#), [1478](#), [1928](#), [1929](#),
[6377](#), [10500](#), [10501](#), [10502](#), [10503](#),
[10504](#), [12234](#), [14179](#), [14207](#), [18146](#),
[19352](#), [19369](#), [19409](#), [19415](#), [29101](#)
- \cs_set_eq:NN [19](#), [64](#),
[370](#), [563](#), [1716](#), [1946](#), [2906](#), [2924](#),
[3074](#), [3670](#), [3671](#), [3675](#), [3844](#), [3887](#),
[3912](#), [4267](#), [4548](#), [6098](#), [6132](#), [6731](#),
[6781](#), [7618](#), [7641](#), [7944](#), [7982](#), [7983](#),
[7985](#), [7986](#), [7987](#), [8009](#), [8388](#), [8390](#),
[8724](#), [10506](#), [10507](#), [10508](#), [10509](#),
[10511](#), [10513](#), [10514](#), [11889](#), [11891](#),
[13918](#), [13927](#), [14726](#), [16413](#), [16414](#),
[16416](#), [16570](#), [16571](#), [16582](#), [17751](#),
[18605](#), [18824](#), [19003](#), [19348](#), [19407](#),
[19414](#), [20889](#), [20905](#), [20909](#), [20980](#),
[20991](#), [21001](#), [28877](#), [36280](#), [36290](#)
- \cs_set_nopar:Nn [18](#), [2034](#), [2099](#)
- \cs_set_nopar:Npn [14](#), [16](#), [187](#), [369](#),
[1478](#), [1507](#), [1565](#), [1566](#), [1920](#), [1922](#),
[20845](#), [20916](#), [21471](#), [29134](#), [29136](#)
- \cs_set_nopar:Npx
 ... [16](#), [923](#), [927](#), [941](#), [1368](#), [1478](#),
[1511](#), [1920](#), [1923](#), [2055](#), [2295](#), [2496](#),
[4416](#), [4422](#), [11909](#), [20746](#), [20759](#),
[20766](#), [20773](#), [20774](#), [20894](#), [21459](#),
[21464](#), [36131](#), [36149](#), [36154](#), [36188](#)
- \cs_set_protected:Nn . [18](#), [2034](#), [2099](#)
- .cs_set_protected:Np [224](#), [21130](#)
- \cs_set_protected:Npn [15](#), [16](#), [370](#),
[161](#), [1478](#), [1494](#), [1496](#), [1498](#), [1500](#),
[1502](#), [1504](#), [1509](#), [1552](#), [1553](#), [1558](#),
[1563](#), [1564](#), [1567](#), [1579](#), [1581](#), [1583](#),
[1584](#), [1585](#), [1587](#), [1596](#), [1598](#), [1600](#),
[1601](#), [1602](#), [1604](#), [1613](#), [1625](#), [1651](#),
[1668](#), [1687](#), [1695](#), [1703](#), [1715](#), [1717](#),
[1719](#), [1731](#), [1745](#), [1782](#), [1870](#), [1883](#),
[1885](#), [1889](#), [1891](#), [1893](#), [1901](#), [1906](#),
[1940](#), [1940](#), [1974](#), [1995](#), [3092](#), [3253](#),
[3672](#), [4189](#), [4253](#), [5335](#), [5372](#), [6083](#),
[6092](#), [6094](#), [6096](#), [6099](#), [6101](#), [6110](#),
[6112](#), [6117](#), [6119](#), [6124](#), [6126](#), [6128](#),
[6130](#), [6133](#), [6890](#), [6891](#), [7411](#), [9320](#),
[9386](#), [10021](#), [10117](#), [10422](#), [10424](#),
[10563](#), [10644](#), [10744](#), [10760](#), [11183](#),
[11629](#), [12367](#), [12589](#), [12799](#), [13093](#),
[13120](#), [14325](#), [14378](#), [16487](#), [18295](#),
[18594](#), [18703](#), [19091](#), [19110](#), [19490](#),
[20338](#), [20452](#), [20591](#), [20635](#), [20876](#),
[21135](#), [21137](#), [21631](#), [22704](#), [23200](#),
[23276](#), [23856](#), [23930](#), [23944](#), [24162](#),
[24179](#), [24214](#), [24250](#), [24265](#), [24282](#),
[25845](#), [29111](#), [29121](#), [29131](#), [29156](#),
[29171](#), [29189](#), [29197](#), [29229](#), [30765](#),
[30768](#), [30801](#), [30812](#), [30828](#), [31053](#),
[31056](#), [31090](#), [31093](#), [31111](#), [31130](#),
[31250](#), [31253](#), [31282](#), [31315](#), [31620](#),
[31632](#), [31694](#), [31745](#), [32814](#), [32827](#),
[32858](#), [33141](#), [34711](#), [36351](#), [36357](#)
- \cs_set_protected:Npx
[16](#), [147](#), [1478](#), [1940](#), [1941](#), [9558](#), [20880](#)
- \cs_set_protected_nopar:Nn
 [18](#), [2034](#), [2099](#)
- \cs_set_protected_nopar:Npn
 [16](#), [369](#), [1478](#), [1934](#), [1934](#)
- \cs_set_protected_nopar:Npx
 [16](#), [1478](#), [1934](#), [1935](#)
- \cs_show:N [20](#), [27](#), [376](#), [2176](#)
- \cs_split_function:N
 . [22](#), [1592](#), [1609](#), [1724](#), [1725](#), [1782](#),
[2006](#), [2047](#), [2893](#), [3196](#), [16028](#), [16049](#)
- \cs_to_sr:N [1370](#)
- \cs_to_str:N . [6](#), [21](#), [107](#), [120](#), [364](#),
[365](#), [397](#), [717](#), [1773](#), [1788](#), [2691](#),
[2857](#), [4430](#), [5931](#), [10427](#), [13701](#),
[13702](#), [13703](#), [13704](#), [13705](#), [13706](#),
[13707](#), [13708](#), [13709](#), [13710](#), [13711](#),
[13712](#), [16156](#), [16178](#), [22702](#), [29678](#),

- 31604, 36135, 36221, 36226, 36234
- _cs_undefine:N 20, 828, 929, 1962, 9445, 9446, 9447, 10119, 11603, 11604, 12826, 13080, 28551, 28835, 28895
- cs internal commands:
- _cs_count_signature:N ... 363, 2005
- _cs_count_signature:n 2005
- _cs_count_signature:nnN 2005
- _cs_generate_from_signature:n . 2056, 2070
- _cs_generate_from_signature:NNn 2038, 2042
- _cs_generate_from_signature:nnNNnn 2046, 2051
- _cs_generate_internal_c:NN . 3159
- _cs_generate_internal_end:w ... 3142, 3176
- _cs_generate_internal_long:nnnNNn 3180, 3184
- _cs_generate_internal_long:w .. 3143, 3178
- _cs_generate_internal_loop:nwnnw 3140, 3146, 3158, 3160, 3162, 3164, 3167
- _cs_generate_internal_N:NN . 3157
- _cs_generate_internal_n:NN . 3161
- _cs_generate_internal_one_-go:NNn 404, 3115, 3138
- _cs_generate_internal_other:NN 3151, 3165
- _cs_generate_internal_test:Nw . 3100, 3122, 3126
- _cs_generate_internal_test_-aux:w .. 3102, 3118, 3123, 3129, 3132
- _cs_generate_internal_variant:n 408, 3065, 3070, 3250, 3256
- _cs_generate_internal_variant:NNn 404, 3090, 3094
- _cs_generate_internal_variant:wnNwn 3072, 3085
- _cs_generate_internal_variant_-loop:n 3070
- _cs_generate_internal_x:NN . 3163
- _cs_generate_variant:N . 2889, 2902
- _cs_generate_variant:n 3191
- _cs_generate_variant:nnNN 2892, 2925
- _cs_generate_variant:nnNnn . 3191
- _cs_generate_variant:Nnnw 2932, 2934
- _cs_generate_variant:w 3191
- _cs_generate_variant:ww 2902
- _cs_generate_variant:wwNN 400-402, 2941, 3058
- _cs_generate_variant:wwNw .. 2902
- _cs_generate_variant_F_-form:nnn 3191
- _cs_generate_variant_loop:nNwN 400, 401, 2942, 2954
- _cs_generate_variant_loop_-base:N 2954
- _cs_generate_variant_loop_-end:nwwwNNnn .. 400-402, 2944, 2954
- _cs_generate_variant_loop_-invalid:NNwNNnn 401, 2954
- _cs_generate_variant_loop_-long:wNNnn 401, 2947, 2954
- _cs_generate_variant_loop_-same:w 401, 2954
- _cs_generate_variant_loop_-special:NNwNNnn 2954, 3053
- _cs_generate_variant_p_-form:nnn 3191
- _cs_generate_variant_same:N ... 401, 2999, 3047
- _cs_generate_variant_T_-form:nnn 3191
- _cs_generate_variant_TF_-form:nnn 3191
- _cs_get_function_name:N 363
- _cs_get_function_signature:N . 363
- _cs_parm_from_arg_count_-test:nnTF 1974
- _cs_split_function_auxi:w .. 1782
- _cs_split_function_auxii:w . 1782
- _cs_tmp:w ... 363, 398, 404, 408, 1782, 1797, 1904, 1920, 1922, 1923, 1924, 1925, 1926, 1927, 1928, 1929, 1930, 1931, 1932, 1933, 1934, 1935, 1936, 1937, 1938, 1939, 1940, 1941, 1942, 1943, 1944, 1945, 2034, 2055, 2057, 2060, 2075, 2076, 2077, 2078, 2079, 2080, 2081, 2082, 2083, 2084, 2085, 2086, 2087, 2088, 2089, 2090, 2091, 2092, 2093, 2094, 2095, 2096, 2097, 2098, 2099, 2107, 2108, 2109, 2110, 2111, 2112, 2113, 2114, 2115, 2116, 2117, 2118, 2119, 2120, 2121, 2122, 2123, 2124, 2125, 2126, 2127, 2128, 2129, 2130, 2906, 2924, 3066, 3074, 3092, 3137, 3253, 3260, 3261, 3262, 3263, 3264, 3265, 3266, 3267, 3268, 3269, 3270, 3271, 3272, 3273, 3274, 3275, 3276, 3277, 3278, 3279, 3280, 3281, 3282, 3283, 3284, 3285, 3286, 3287, 3288, 3289, 3290, 3291,

- 3292, 3293, 3294, 3295, 3296, 3297,
 3298, 3299, 3300, 3301, 3302, 3303
 __cs_to_str:N 364, 1773
 __cs_to_str:w 364, 1773
 __cs_use_i_delimit_by_s_stop:nw
 2883, 3204
 __cs_use_none_delimit_by_q_
 recursion_stop:w
 2883, 2930, 2937, 3217
 __cs_use_none_delimit_by_s_
 stop:w 2883, 3208
 csc 254
 cscd 254
 \csname 635, 4, 21, 25, 30,
 34, 47, 70, 71, 73, 82, 107, 111, 130, 224
 \csstring 805
 \currentcjktoken 1109, 1166
 \currentgrouplevel 516
 \currentgrouptype 517
 \currentifbranch 518
 \currentiflevel 519
 \currentiftypetype 520
 \currentspacingmode 1110
 \currentxspacingmode 1111
- ### D
- \d 29389, 31738
 \date 344
 \day 225, 1314, 8997
 dd 257
 \deadcycles 226
 debug commands:
 \debug_off: 344
 \debug_off:n 29, 1374, 1375, 1553
 \debug_on: 344
 \debug_on:n 29, 1374, 1553
 \debug_resume: . 29, 1281, 1563, 32733
 \debug_suspend: 29, 1281, 1563, 32726
 debug internal commands:
 \g__debug_deprecation_off_tl . 1565
 \g__debug_deprecation_on_tl .. 1565
 \def 53, 54, 55, 81, 83, 88,
 89, 91, 104, 105, 108, 119, 143, 182, 227
 default commands:
 .default:n 224, 21146
 \defaultthyphenchar 228
 \defaultskewchar 229
 deg 256
 \delcode 230
 \delimiter 231
 \delimiterfactor 232
 \delimitershortfall 233
 deprecation internal commands:
 __deprecation_just_error:nnNN 36320
 __deprecation_old:Nnn 36377
 __deprecation_old_protected:Nnn
 36377
 __deprecation_patch_aux:Nn .. 36320
 __deprecation_patch_aux:nnNNnn .
 36320
 __deprecation_warn_once:nnNNnn 36320
 \detokenize 47, 130, 521
 \DH 29395, 31294, 31654
 \dh 29395, 31294, 31664
 dim commands:
 \dim_abs:n 207, 903, 19998
 \dim_add:Nn 207, 19980
 \dim_case:nn 210, 20078
 \dim_case:nnn 36447
 \dim_case:nnTF
 210, 20078, 20083, 20088, 36448
 \dim_compare:nNnTF . 208–211, 243,
 20033, 20102, 20138, 20146, 20155,
 20161, 20173, 20176, 20187, 33087,
 33090, 33095, 33109, 33112, 33117,
 33463, 33468, 33478, 33607, 33619,
 35715, 35732, 35766, 35780, 35790
 \dim_compare:nTF 208, 209,
 211, 20038, 20110, 20118, 20127, 20133
 \dim_compare_p:n 209, 20038
 \dim_compare_p:nNn 208, 20033
 \dim_const:Nn 206,
 896, 906, 19947, 20276, 20277, 22055
 \dim_do_until:nn 211, 20108
 \dim_do_until:nNnn 210, 20136
 \dim_do_while:nn 211, 20108
 \dim_do_while:nNnn 210, 20136
 \dim_eval:n 208, 209, 212, 896, 1258,
 1259, 19950, 20081, 20086, 20091,
 20096, 20191, 20219, 20271, 20275,
 32790, 32867, 32953, 32971, 33008,
 33012, 33013, 33017, 33021, 33022,
 33039, 33044, 33050, 33057, 33064,
 33229, 33232, 33233, 33240, 33322,
 33323, 33330, 33331, 33434, 33441,
 33590, 33591, 33855, 33856, 33857
 \dim_gadd:Nn 207, 19980
 \dim_gset:N 224, 21154
 \dim_gset:Nn 207, 896, 19968
 \dim_gset_eq:NN 207, 19974
 \dim_gsub:Nn 207, 19980
 \dim_gzero:N . 206, 19953, 19961, 20297
 \dim_gzero_new:N 206, 19958
 \dim_if_exist:NTF
 207, 19959, 19961, 19964
 \dim_if_exist_p:N 207, 19964
 \dim_log:N 214, 20272
 \dim_log:n 214, 20272

`\dim_max:nn` .. [207](#), [19998](#), [33301](#), [33305](#)
`\dim_min:nn`
 [207](#), [19998](#), [33299](#), [33303](#), [33316](#)
`\dim_new:N` [206](#), [19941](#), [19949](#), [19959](#),
 [19961](#), [20278](#), [20279](#), [20280](#), [20281](#),
 [32297](#), [32298](#), [32299](#), [32300](#), [32301](#),
 [32302](#), [32303](#), [32304](#), [32651](#), [32675](#),
 [32676](#), [32679](#), [32680](#), [32681](#), [32682](#),
 [33175](#), [33176](#), [33177](#), [33178](#), [33179](#),
 [33337](#), [33338](#), [33679](#), [33681](#), [33682](#)
`\dim_ratio:nn` . [208](#), [904](#), [20029](#), [20265](#)
`.dim_set:N` [224](#), [21154](#)
`\dim_set:Nn` [207](#), [19968](#),
 [32324](#), [32325](#), [32326](#), [32358](#), [32369](#),
 [32453](#), [32454](#), [32455](#), [32470](#), [32568](#),
 [32569](#), [32570](#), [32572](#), [32574](#), [32576](#),
 [32780](#), [32856](#), [33089](#), [33093](#), [33111](#),
 [33115](#), [33146](#), [33160](#), [33235](#), [33270](#),
 [33278](#), [33289](#), [33290](#), [33291](#), [33292](#),
 [33298](#), [33300](#), [33302](#), [33304](#), [33309](#),
 [33315](#), [33398](#), [33400](#), [33402](#), [33410](#),
 [33412](#), [33466](#), [33541](#), [33542](#), [33544](#),
 [33546](#), [33564](#), [33565](#), [33680](#), [33774](#),
 [33775](#), [33821](#), [33822](#), [33823](#), [33825](#)
`\dim_set_eq:NN` [207](#), [19974](#), [32805](#), [32806](#)
`\dim_show:N` [213](#), [20268](#)
`\dim_show:n` [214](#), [905](#), [20270](#)
`\dim_sign:n` [212](#), [20221](#)
`\dim_step_function:nnnN`
 [211](#), [902](#), [20164](#), [20216](#)
`\dim_step_inline:nnnn` ... [211](#), [20194](#)
`\dim_step_variable:nnnNn` . [212](#), [20194](#)
`\dim_sub:Nn` [207](#), [19980](#)
`\dim_to_decimal:n`
 [212](#), [20241](#), [20257](#), [20262](#)
`\dim_to_decimal_in_bp:n` .. [213](#), [20256](#)
`\dim_to_decimal_in_sp:n`
 [213](#), [1010](#), [20258](#), [23338](#), [23375](#), [23969](#)
`\dim_to_decimal_in_unit:nn` [213](#), [20260](#)
`\dim_to_fp:n` [213](#), [1010](#), [1029](#), [20268](#),
 [28170](#), [32362](#), [32363](#), [32373](#), [32374](#),
 [32442](#), [32445](#), [32446](#), [32471](#), [32486](#),
 [32487](#), [32506](#), [32507](#), [32525](#), [32542](#),
 [32545](#), [32546](#), [33100](#), [33101](#), [33102](#),
 [33122](#), [33123](#), [33124](#), [33134](#), [33135](#),
 [33151](#), [33152](#), [33153](#), [33154](#), [33164](#),
 [33165](#), [33274](#), [33275](#), [33282](#), [33283](#),
 [33356](#), [33359](#), [33360](#), [33411](#), [33413](#)
`\dim_until_do:nn` [211](#), [20108](#)
`\dim_until_do:nNnn` [210](#), [20136](#)
`\dim_use:N` [212](#), [1258](#), [20001](#), [20007](#),
 [20008](#), [20009](#), [20015](#), [20016](#), [20017](#),
 [20041](#), [20060](#), [20220](#), [20224](#), [20239](#),
 [20244](#), [20351](#), [20352](#), [20427](#), [20428](#),
 [33237](#), [33241](#), [33248](#), [33254](#), [33263](#),
 [33264](#), [33265](#), [33419](#), [33426](#), [33572](#)
`\dim_while_do:nn` [211](#), [20108](#)
`\dim_while_do:nNnn` [211](#), [20136](#)
`\dim_zero:N` [206](#), [19953](#), [19959](#), [20296](#),
 [32327](#), [32456](#), [32571](#), [33080](#), [33081](#)
`\dim_zero_new:N` [206](#), [19958](#)
`\c_max_dim` [214](#), [217](#), [958](#),
 [20276](#), [20370](#), [22082](#), [22124](#), [22132](#),
 [33289](#), [33290](#), [33291](#), [33292](#), [33309](#)
`\g_tmpa_dim` [214](#), [20278](#)
`\l_tmpa_dim` [214](#), [20278](#)
`\g_tmpb_dim` [214](#), [20278](#)
`\l_tmpb_dim` [214](#), [20278](#)
`\c_zero_dim` [214](#), [20173](#), [20176](#), [20229](#),
 [20276](#), [20369](#), [22149](#), [32183](#), [32207](#),
 [33087](#), [33090](#), [33095](#), [33109](#), [33112](#),
 [33117](#), [33463](#), [33468](#), [33478](#), [35719](#),
 [35730](#), [35736](#), [35748](#), [35766](#), [35770](#),
 [35778](#), [35780](#), [35784](#), [35790](#), [35800](#)
dim internal commands:
 `__dim_abs:N` [19998](#)
 `__dim_case:nnTF` [20078](#)
 `__dim_case:nw` [20078](#)
 `__dim_case_end:nw` [20078](#)
 `__dim_compare:w` [20038](#)
 `__dim_compare:wNn` [898](#), [20038](#)
 `__dim_compare_!:w` [20038](#)
 `__dim_compare_<:w` [20038](#)
 `__dim_compare_=:w` [20038](#)
 `__dim_compare_>:w` [20038](#)
 `__dim_compare_end:w` .. [20046](#), [20070](#)
 `__dim_compare_error:` ... [898](#), [20038](#)
 `__dim_eval:w` [904](#), [19935](#),
 [19969](#), [19971](#), [19981](#), [19985](#), [19990](#),
 [19994](#), [20001](#), [20007](#), [20008](#), [20009](#),
 [20015](#), [20016](#), [20017](#), [20032](#), [20035](#),
 [20041](#), [20060](#), [20065](#), [20167](#), [20168](#),
 [20169](#), [20220](#), [20224](#), [20244](#), [20259](#)
 `__dim_eval_end:` [19935](#),
 [19969](#), [19971](#), [19981](#), [19985](#), [19990](#),
 [19994](#), [20001](#), [20011](#), [20019](#), [20032](#),
 [20035](#), [20220](#), [20224](#), [20244](#), [20259](#)
 `__dim_maxmin:wwN` [19998](#)
 `__dim_ratio:n` [20029](#)
 `__dim_sign:Nw` [20221](#)
 `__dim_step:NnnnN` [20164](#)
 `__dim_step:Nnnnnn` [20194](#)
 `__dim_step:wwwN` [20164](#)
 `__dim_to_decimal:w` [20241](#)
 `__dim_use_none_delimit_by_s_`
 `stop:w` [19940](#), [20056](#)
`\dimen` [234](#), [19152](#)
`\dimendef` [235](#)

<code>\dimexpr</code>	522	4877, 4878, 4880, 4916, 4919, 4940,
<code>\directlua</code>	6, 38, 40, 806	4943, 4951, 4959, 4962, 4971, 4974,
<code>\disablecjktoken</code>	1167	4983, 4991, 4994, 5004, 5124, 5238,
<code>\discretionary</code>	236	5282, 5286, 5289, 5300, 5305, 5395,
<code>\disinhibitglue</code>	1112	5541, 5554, 5643, 5672, 5711, 5729,
<code>\displayindent</code>	237	5842, 5898, 5932, 5962, 6369, 6387,
<code>\displaylimits</code>	238	6406, 6440, 6493, 6540, 6544, 6551,
<code>\displaystyle</code>	239	6572, 6583, 6730, 6843, 6953, 6997,
<code>\displaywidowpenalties</code>	523	7000, 7120, 7131, 7140, 7168, 7180,
<code>\displaywidowpenalty</code>	240	7206, 7223, 7231, 7492, 7746, 8025,
<code>\displaywidth</code>	241	8036, 8424, 8467, 8487, 8509, 8527,
<code>\divide</code>	242	8543, 8553, 8569, 8579, 8671, 8673,
<code>\DJ</code>	29396, 31295, 31655	8675, 8677, 10191, 10194, 10197,
<code>\dj</code>	29396, 31295, 31665	11230, 11239, 11250, 11884, 12273,
<code>\do</code>	1220	12283, 12298, 12307, 12317, 12333,
<code>\doublehyphendemerits</code>	243	12347, 12374, 12391, 12406, 12685,
<code>\dp</code>	244	12703, 12721, 12729, 12739, 12755,
<code>\draftmode</code>	910	12778, 12789, 12795, 12941, 12953,
draw commands:		13002, 13005, 13008, 13209, 13214,
<code>\draw_begin:</code>	291	13219, 13226, 13231, 13477, 13533,
<code>\draw_end:</code>	291	13536, 13539, 13551, 13566, 13800,
<code>\dtou</code>	1113	13808, 13816, 13963, 14014, 14015,
<code>\dump</code>	245	14019, 14024, 14047, 14100, 14200,
<code>\dviextension</code>	807	14451, 14481, 14484, 14514, 14517,
<code>\dvifedback</code>	808	14534, 14537, 14640, 14645, 14663,
<code>\dvivvariable</code>	809	14682, 14685, 14734, 14739, 14742,
		14857, 14869, 14878, 15000, 15005,
		15933, 15971, 15979, 15990, 16000,
		16019, 16043, 16047, 16089, 16148,
		16165, 16429, 16507, 16516, 16943,
		16954, 16975, 16991, 16994, 17015,
		17055, 17154, 17181, 17219, 17227,
		17528, 17561, 17612, 17729, 17754,
		17780, 17789, 17959, 17974, 17996,
		18010, 18624, 18627, 18635, 18641,
		18670, 18676, 18748, 18759, 18779,
		18897, 18900, 18903, 18906, 18909,
		18912, 18915, 18985, 18990, 18995,
		19000, 19007, 19014, 19019, 19024,
		19029, 19034, 19039, 19044, 19049,
		19054, 19076, 19082, 19085, 19121,
		19124, 19259, 19268, 19276, 19285,
		19361, 19386, 19390, 19400, 19438,
		19452, 19461, 19471, 19505, 20004,
		20025, 20036, 20046, 20071, 20231,
		20234, 22087, 22379, 22396, 22397,
		22412, 22422, 22517, 22593, 22655,
		22658, 22672, 22690, 22694, 22934,
		22947, 22967, 22995, 22996, 23018,
		23039, 23062, 23063, 23096, 23113,
		23131, 23166, 23170, 23206, 23223,
		23229, 23233, 23237, 23396, 23429,
		23437, 23470, 23474, 23486, 23496,
		23506, 23537, 23550, 23585, 23595,
E		
<code>\edef</code>	94, 117, 246	
<code>\efcode</code>	685	
<code>\c_eight</code>	36415	
<code>\elapsedtime</code>	774	
<code>\c_eleven</code>	36421	
<code>\else</code>	5, 26, 28, 35, 71, 72, 75, 76, 247	
else commands:		
<code>\else:</code> ...	27, 64, 70, 93, 167, 168,	
	220, 281, 357, 359, 365, 399, 569,	
	690, 1055, 1416, 1461, 1639, 1647,	
	1673, 1802, 1805, 1814, 1820, 1830,	
	1833, 1842, 1848, 1968, 1990, 1999,	
	2013, 2072, 2073, 2134, 2317, 2597,	
	2749, 2777, 2792, 2800, 2837, 2907,	
	2958, 2959, 2961, 2965, 2977, 2978,	
	2979, 2980, 2981, 2982, 2983, 2984,	
	2985, 3049, 3050, 3052, 3101, 3131,	
	3218, 3353, 3354, 3822, 3825, 3828,	
	3838, 3853, 3880, 3895, 3922, 3938,	
	3971, 3979, 3981, 3983, 3985, 3987,	
	3989, 3991, 3993, 4011, 4032, 4036,	
	4115, 4119, 4221, 4231, 4240, 4275,	
	4321, 4332, 4439, 4536, 4537, 4542,	
	4543, 4560, 4567, 4767, 4777, 4827,	
	4836, 4848, 4849, 4851, 4853, 4856,	
	4857, 4860, 4861, 4870, 4872, 4874,	

- 23614, 23627, 23640, 23644, 23655,
 23678, 23695, 23707, 23721, 23734,
 23738, 23746, 23748, 23758, 23769,
 23785, 23799, 23805, 23808, 23815,
 23837, 23867, 23890, 23918, 23921,
 24095, 24099, 24106, 24125, 24137,
 24141, 24148, 24170, 24187, 24193,
 24225, 24257, 24273, 24293, 24334,
 24349, 24382, 24384, 24390, 24405,
 24458, 24623, 24639, 24650, 24688,
 24691, 24694, 24697, 24728, 24737,
 24746, 24749, 24920, 24933, 24936,
 24943, 24961, 24985, 24986, 25001,
 25011, 25060, 25063, 25072, 25084,
 25095, 25109, 25122, 25162, 25196,
 25216, 25253, 25271, 25274, 25280,
 25294, 25329, 25347, 25350, 25353,
 25356, 25417, 25490, 25560, 25561,
 25570, 25605, 25688, 25692, 25696,
 25758, 25793, 25808, 26073, 26102,
 26106, 26266, 26275, 26329, 26340,
 26356, 26364, 26423, 26503, 26514,
 26519, 26553, 26566, 26578, 26584,
 26705, 26713, 26752, 26759, 26781,
 26808, 26823, 26827, 26849, 26880,
 26883, 26908, 26911, 26952, 26960,
 26971, 26974, 27089, 27104, 27119,
 27134, 27149, 27164, 27185, 27230,
 27536, 27574, 27575, 27584, 27628,
 27683, 27684, 27685, 27789, 27811,
 27826, 27844, 27892, 27908, 28114,
 28181, 28186, 28350, 28386, 28399,
 28429, 28433, 28441, 28468, 28494,
 28502, 28519, 28522, 28571, 28575,
 28627, 28686, 28698, 29165, 29262,
 29266, 29277, 29298, 29302, 29311,
 29312, 29313, 29314, 29315, 29316,
 29317, 29318, 29319, 29330, 29344,
 29347, 29350, 29353, 29356, 29359,
 29362, 29760, 30360, 30364, 30367,
 30371, 30380, 30383, 30386, 30389,
 30392, 30395, 30409, 30412, 30415,
 30418, 30421, 30433, 30436, 30439,
 30442, 32058, 32060, 32066, 35808
 \em 31583
 em 257
 \emergencystretch 248
 \emph 31556
 \enablecjktoken 1168
 \end .. 340, 249, 24817, 29406, 29413, 31599
 \endcsname 635, 4, 21, 25, 30,
 34, 47, 70, 71, 73, 82, 107, 111, 130, 250
 \endgroup 19,
 20, 24, 29, 33, 53, 88, 92, 98, 115, 251
 \endinput 99, 252
 \endL 524
 \endlinechar 129, 141, 253
 \endR 525
 \enquote 24819
 \ensuremath 1206, 29408
 \epTeXinputencoding 1114
 \epTeXversion 1115
 \eqno 254
 \errhelp 89, 255
 \errmessage 91, 256
 \ERROR 18716
 \errorcontextlines 89, 257
 \errorstopmode 258
 \escapechar 259
 escapehex 11708
 \ETC 4385
 \TeXglueshrinkorder 810
 \TeXgluestretchorder 811
 \TeXrevision 526
 \TeXversion 527
 \etoksapp 812
 \etokspre 813
 \euc 1116
 \everycr 260
 \everydisplay 261
 \everyeof 528
 \everyhbox 262
 \everyjob 45, 46, 263
 \everymath 264
 \everypar 265
 \everyvbox 266
 ex 257
 \exceptionpenalty 814
 \exhyphenpenalty 267
 exp 252
 exp commands:
 \exp:w .. 41, 42, 357, 364, 382, 383,
 390, 391, 551, 552, 554, 571, 628,
 693, 711, 832, 1000, 1002, 1003,
 1006, 1007, 1025, 1030, 1368, 1438,
 1574, 1576, 2289, 2302, 2308, 2356,
 2360, 2364, 2369, 2375, 2381, 2397,
 2409, 2415, 2421, 2426, 2428, 2435,
 2442, 2473, 2478, 2487, 2492, 2501,
 2503, 2511, 2518, 2524, 2532, 2541,
 2548, 2562, 2582, 2586, 2591, 2593,
 2630, 2812, 3171, 7945, 7953, 7991,
 8030, 8038, 8046, 8478, 8624, 8681,
 10609, 12172, 12403, 12412, 12417,
 12422, 12427, 12865, 12946, 13251,
 13256, 13261, 13266, 13282, 13287,
 13292, 13297, 13460, 13469, 13524,
 17187, 17192, 17197, 17202, 17820,

17828, 17888, 18196, 18205, 18615,
 19291, 19293, 19295, 19297, 19299,
 19301, 19303, 19305, 19307, 19309,
 19311, 19313, 19380, 20045, 20080,
 20085, 20090, 20095, 22425, 22540,
 22544, 22910, 23036, 23037, 23038,
 23039, 23158, 23176, 23205, 23249,
 23261, 23266, 23274, 23282, 23303,
 23309, 23381, 23394, 23395, 23404,
 23417, 23435, 23436, 23456, 23469,
 23473, 23495, 23523, 23536, 23549,
 23573, 23584, 23594, 23613, 23626,
 23639, 23642, 23654, 23677, 23706,
 23720, 23737, 23757, 23768, 23774,
 23784, 23826, 23833, 23864, 23879,
 23887, 23904, 23920, 23924, 23933,
 23970, 23979, 23988, 23993, 23995,
 24006, 24008, 24023, 24026, 24033,
 24044, 24128, 24174, 24192, 24195,
 24209, 24222, 24272, 24290, 24361,
 24373, 24402, 24404, 24408, 24410,
 24468, 24478, 24488, 24500, 24630,
 24647, 24657, 24812, 24813, 24814,
 25005, 25008, 25016, 25026, 25034,
 26046, 26577, 26599, 26754, 26930,
 27206, 27949, 27964, 27981, 28018,
 28035, 28077, 28096, 28109, 28141,
 28156, 28167, 28289, 28336, 28376,
 28412, 28592, 28692, 29425, 29464,
 29745, 29800, 29847, 29861, 29873,
 31352, 35846, 35848, 35850, 35852,
 35854, 35856, 35858, 35860, 36150,
 36155, 36160, 36165, 36181, 36199
 \exp_after:wN 38, 40, 41, 194,
 357, 360, 380, 383, 427, 532, 551–
 554, 597, 686, 699, 820, 845, 888,
 975, 999, 1000, 1002, 1003, 1069,
 1070, 1134, 1368, 1435, 1453, 1455,
 1460, 1462, 1574, 1576, 1630, 1654,
 1672, 1674, 1736, 1741, 1748, 1777,
 1781, 1786, 1797, 1813, 1815, 1818,
 1841, 1843, 1846, 1967, 1969, 1978,
 1998, 2000, 2039, 2104, 2201, 2221,
 2230, 2239, 2251, 2261, 2267, 2274,
 2276, 2288, 2289, 2301, 2302, 2307,
 2308, 2313, 2318, 2320, 2323, 2332,
 2334, 2337, 2338, 2339, 2342, 2344,
 2346, 2350, 2355, 2360, 2363, 2368,
 2373, 2374, 2375, 2379, 2380, 2381,
 2387, 2388, 2395, 2396, 2397, 2401,
 2402, 2403, 2407, 2408, 2409, 2413,
 2414, 2415, 2419, 2420, 2421, 2425,
 2426, 2427, 2428, 2432, 2433, 2434,
 2435, 2439, 2440, 2441, 2442, 2446,
 2447, 2448, 2453, 2454, 2455, 2456,
 2460, 2461, 2462, 2463, 2469, 2472,
 2473, 2477, 2478, 2491, 2492, 2499,
 2501, 2503, 2507, 2511, 2513, 2516,
 2517, 2522, 2523, 2527, 2530, 2531,
 2535, 2538, 2539, 2540, 2545, 2546,
 2547, 2555, 2558, 2559, 2560, 2561,
 2566, 2568, 2570, 2571, 2582, 2585,
 2590, 2615, 2616, 2617, 2628, 2629,
 2641, 2642, 2643, 2648, 2656, 2657,
 2658, 2659, 2660, 2661, 2684, 2685,
 2686, 2687, 2747, 2748, 2750, 2772,
 2777, 2778, 2790, 2791, 2793, 2797,
 2798, 2799, 2802, 2804, 2807, 2811,
 2812, 2813, 2818, 2819, 2830, 2836,
 2838, 2842, 2843, 2846, 2847, 2857,
 2904, 2908, 2930, 2937, 2957, 3100,
 3102, 3129, 3130, 3132, 3169, 3171,
 3188, 3206, 3217, 3345, 3431, 3432,
 3474, 3493, 3494, 3509, 3510, 3511,
 3756, 3762, 3764, 3775, 3835, 3836,
 3837, 3838, 3844, 3845, 3861, 3879,
 3881, 3887, 3888, 3919, 3921, 3923,
 3953, 3968, 3970, 3972, 3997, 4007,
 4015, 4025, 4035, 4037, 4039, 4074,
 4105, 4114, 4117, 4118, 4120, 4121,
 4129, 4130, 4144, 4182, 4218, 4219,
 4220, 4222, 4227, 4236, 4238, 4239,
 4241, 4255, 4274, 4276, 4288, 4289,
 4290, 4324, 4367, 4404, 4417, 4418,
 4423, 4424, 4427, 4430, 4438, 4440,
 4497, 4504, 4511, 4515, 4522, 4528,
 4566, 4568, 4577, 4699, 4702, 4743,
 4761, 4766, 4768, 4769, 4776, 4779,
 4780, 4786, 4798, 4810, 4829, 4838,
 4950, 4952, 4958, 4961, 4963, 4970,
 4973, 4975, 4982, 4984, 4990, 4993,
 4996, 5090, 5332, 5396, 5408, 5540,
 5543, 5553, 5555, 5737, 5744, 5752,
 5837, 5931, 5963, 6332, 6622, 6628,
 6634, 6745, 6769, 6800, 6831, 6857,
 6895, 6945, 6946, 6954, 6957, 7075,
 7104, 7166, 7167, 7170, 7171, 7179,
 7181, 7182, 7205, 7208, 7285, 7648,
 7671, 7672, 7683, 7707, 7708, 7715,
 7735, 7846, 7847, 7945, 7949, 7950,
 7951, 7991, 8030, 8043, 8044, 8045,
 8476, 8494, 8499, 8503, 8625, 8881,
 8882, 8883, 8984, 9329, 9534, 9535,
 10026, 10027, 10113, 10275, 10293,
 10334, 10539, 10542, 10592, 10601,
 10604, 10607, 10608, 10610, 10650,
 10716, 10747, 10768, 10780, 10808,
 10816, 10905, 10906, 10907, 11328,

11432, 11951, 11952, 11959, 11960,
11976, 11982, 11996, 12001, 12007,
12008, 12012, 12016, 12021, 12027,
12028, 12032, 12042, 12046, 12052,
12053, 12057, 12059, 12063, 12069,
12070, 12074, 12094, 12117, 12118,
12119, 12120, 12121, 12180, 12181,
12182, 12233, 12243, 12248, 12291,
12329, 12343, 12387, 12389, 12519,
12572, 12577, 12639, 12650, 12657,
12670, 12673, 12718, 12728, 12752,
12762, 12763, 12764, 12767, 12771,
12772, 12796, 12863, 12942, 12944,
12945, 12946, 12951, 12952, 12954,
13026, 13043, 13044, 13312, 13313,
13325, 13390, 13413, 13447, 13448,
13459, 13460, 13468, 13476, 13478,
13485, 13490, 13508, 13509, 13510,
13522, 13523, 13550, 13552, 13558,
13564, 13578, 13598, 13609, 13625,
13633, 13641, 13648, 13655, 13667,
13774, 13790, 13809, 13818, 13839,
13840, 13845, 13846, 13871, 13872,
13887, 13888, 13936, 13941, 14006,
14335, 14371, 14373, 14388, 14394,
14558, 14560, 14627, 14646, 14647,
14661, 14662, 14689, 14690, 14805,
14827, 14840, 14841, 14868, 14963,
14984, 14993, 15926, 15932, 15934,
15958, 16009, 16010, 16089, 16123,
16164, 16172, 16183, 16325, 16327,
16339, 16347, 16491, 16525, 16537,
16550, 16551, 16552, 16574, 16575,
16620, 16655, 16656, 16657, 16757,
16758, 16760, 16761, 16769, 16770,
16797, 16798, 16824, 16825, 16828,
16895, 16935, 16949, 16954, 16957,
16958, 16965, 16966, 16982, 16983,
17004, 17005, 17014, 17126, 17131,
17136, 17159, 17161, 17289, 17290,
17291, 17316, 17317, 17500, 17528,
17533, 17561, 17574, 17584, 17611,
17613, 17614, 17622, 17639, 17683,
17745, 17752, 17788, 17790, 17797,
17819, 17827, 17888, 17960, 17975,
17997, 18011, 18078, 18086, 18092,
18173, 18195, 18204, 18321, 18322,
18325, 18326, 18615, 18616, 18659,
18724, 18725, 18726, 18817, 18818,
19061, 19080, 19128, 19237, 19266,
19267, 19269, 19275, 19278, 19360,
19363, 19379, 19385, 19388, 19391,
19398, 19399, 19401, 19437, 19439,
19449, 19450, 19451, 19453, 19459,
19460, 19462, 19469, 19470, 19472,
19499, 19504, 19506, 19512, 19638,
19691, 19717, 19741, 19819, 19842,
19843, 19844, 19916, 20000, 20004,
20007, 20008, 20015, 20016, 20040,
20045, 20056, 20059, 20166, 20167,
20168, 20223, 20243, 20342, 20747,
20775, 20788, 20948, 20956, 20964,
21069, 21437, 21597, 21598, 21677,
21700, 22105, 22106, 22107, 22125,
22133, 22157, 22158, 22202, 22209,
22210, 22221, 22378, 22380, 22381,
22399, 22400, 22401, 22411, 22413,
22421, 22423, 22430, 22431, 22432,
22433, 22434, 22435, 22440, 22441,
22442, 22443, 22444, 22445, 22446,
22489, 22502, 22505, 22516, 22518,
22533, 22537, 22538, 22539, 22542,
22543, 22607, 22609, 22636, 22640,
22665, 22669, 22686, 22693, 22695,
22765, 22773, 22790, 22799, 22845,
22910, 22979, 22980, 22981, 23041,
23051, 23070, 23076, 23095, 23097,
23099, 23110, 23111, 23114, 23125,
23129, 23136, 23137, 23148, 23149,
23158, 23165, 23167, 23168, 23176,
23205, 23223, 23224, 23227, 23228,
23230, 23231, 23235, 23236, 23238,
23239, 23248, 23249, 23254, 23260,
23266, 23274, 23282, 23301, 23302,
23305, 23306, 23308, 23315, 23316,
23318, 23336, 23337, 23365, 23368,
23373, 23374, 23379, 23380, 23382,
23391, 23392, 23393, 23394, 23397,
23398, 23399, 23402, 23417, 23434,
23435, 23445, 23446, 23456, 23468,
23472, 23485, 23487, 23495, 23505,
23507, 23513, 23518, 23520, 23522,
23528, 23529, 23533, 23535, 23547,
23548, 23570, 23572, 23578, 23581,
23583, 23587, 23592, 23597, 23598,
23608, 23609, 23611, 23612, 23615,
23619, 23624, 23638, 23641, 23653,
23662, 23669, 23670, 23671, 23672,
23674, 23676, 23687, 23688, 23689,
23690, 23692, 23694, 23696, 23697,
23698, 23704, 23705, 23715, 23719,
23720, 23722, 23723, 23724, 23729,
23735, 23736, 23747, 23749, 23756,
23757, 23759, 23760, 23767, 23773,
23783, 23847, 23860, 23861, 23862,
23863, 23877, 23878, 23880, 23885,
23886, 23901, 23903, 23920, 23924,
23933, 23967, 23968, 23969, 23975,

23976, 23977, 23978, 23984, 23985,
23986, 23987, 23994, 24007, 24015,
24021, 24022, 24024, 24025, 24031,
24032, 24034, 24060, 24073, 24093,
24094, 24096, 24097, 24104, 24105,
24107, 24110, 24122, 24123, 24124,
24126, 24127, 24128, 24135, 24136,
24138, 24139, 24146, 24147, 24149,
24152, 24167, 24168, 24169, 24172,
24173, 24174, 24184, 24185, 24186,
24189, 24190, 24191, 24194, 24198,
24207, 24208, 24219, 24220, 24221,
24224, 24226, 24227, 24228, 24255,
24256, 24258, 24259, 24260, 24270,
24271, 24272, 24274, 24275, 24276,
24288, 24289, 24292, 24294, 24295,
24296, 24316, 24317, 24318, 24319,
24320, 24321, 24322, 24332, 24333,
24335, 24336, 24337, 24343, 24354,
24355, 24356, 24357, 24358, 24359,
24360, 24361, 24366, 24367, 24368,
24369, 24370, 24371, 24372, 24388,
24389, 24391, 24392, 24399, 24400,
24401, 24406, 24407, 24409, 24425,
24440, 24449, 24459, 24465, 24466,
24467, 24472, 24485, 24486, 24487,
24493, 24585, 24629, 24646, 24656,
24681, 24682, 24729, 24811, 24919,
24921, 24960, 24962, 24965, 25000,
25002, 25004, 25007, 25014, 25015,
25018, 25019, 25024, 25025, 25032,
25033, 25068, 25069, 25070, 25072,
25083, 25108, 25110, 25116, 25117,
25121, 25124, 25146, 25148, 25161,
25163, 25169, 25171, 25174, 25180,
25182, 25184, 25185, 25186, 25188,
25193, 25195, 25197, 25201, 25204,
25210, 25211, 25215, 25217, 25218,
25219, 25227, 25229, 25230, 25237,
25243, 25250, 25251, 25256, 25257,
25258, 25259, 25278, 25279, 25280,
25286, 25287, 25288, 25293, 25295,
25303, 25305, 25307, 25308, 25310,
25321, 25323, 25325, 25326, 25331,
25382, 25383, 25390, 25391, 25393,
25395, 25397, 25400, 25403, 25405,
25407, 25416, 25418, 25424, 25426,
25428, 25429, 25430, 25436, 25438,
25440, 25441, 25442, 25463, 25464,
25467, 25475, 25477, 25481, 25482,
25483, 25484, 25489, 25491, 25498,
25501, 25504, 25507, 25516, 25519,
25522, 25525, 25532, 25534, 25540,
25548, 25550, 25552, 25569, 25571,
25578, 25580, 25583, 25589, 25591,
25593, 25594, 25595, 25597, 25611,
25612, 25615, 25633, 25635, 25637,
25649, 25652, 25655, 25658, 25661,
25664, 25667, 25670, 25674, 25686,
25690, 25694, 25697, 25712, 25718,
25720, 25722, 25732, 25756, 25759,
25771, 25773, 25777, 25778, 25779,
25781, 25782, 25784, 25791, 25799,
25800, 25806, 25807, 25813, 25816,
25817, 25818, 25819, 25827, 25869,
25874, 25876, 25883, 25886, 25889,
25892, 25895, 25898, 25906, 25907,
25919, 25927, 25929, 25939, 25941,
25948, 25957, 25959, 25962, 25965,
25968, 25971, 25984, 25986, 25994,
25996, 26004, 26006, 26016, 26019,
26022, 26029, 26044, 26045, 26062,
26064, 26065, 26122, 26135, 26137,
26143, 26156, 26158, 26160, 26184,
26198, 26200, 26207, 26209, 26250,
26251, 26252, 26254, 26255, 26256,
26258, 26259, 26265, 26267, 26268,
26274, 26276, 26277, 26278, 26279,
26291, 26297, 26299, 26336, 26343,
26350, 26370, 26371, 26373, 26375,
26377, 26390, 26395, 26396, 26397,
26398, 26399, 26403, 26408, 26410,
26416, 26422, 26424, 26425, 26431,
26432, 26433, 26434, 26435, 26436,
26437, 26438, 26443, 26445, 26447,
26449, 26451, 26456, 26458, 26460,
26462, 26464, 26466, 26484, 26488,
26496, 26497, 26502, 26504, 26513,
26516, 26517, 26518, 26520, 26521,
26522, 26530, 26536, 26548, 26551,
26552, 26554, 26555, 26579, 26580,
26583, 26585, 26601, 26605, 26606,
26607, 26623, 26629, 26695, 26696,
26697, 26704, 26706, 26707, 26712,
26714, 26715, 26724, 26725, 26727,
26730, 26733, 26749, 26753, 26754,
26758, 26760, 26795, 26801, 26802,
26804, 26806, 26807, 26809, 26810,
26820, 26821, 26824, 26825, 26826,
26828, 26829, 26830, 26847, 26848,
26850, 26851, 26857, 26859, 26862,
26865, 26868, 26871, 26879, 26882,
26884, 26887, 26894, 26898, 26906,
26907, 26910, 26912, 26914, 26919,
26920, 26926, 26931, 26932, 26940,
26941, 26942, 26943, 26986, 27008,
27009, 27012, 27013, 27022, 27023,
27024, 27028, 27035, 27036, 27037,

- 27178, 27179, 27180, 27182, 27200,
 27201, 27202, 27203, 27204, 27205,
 27212, 27221, 27228, 27229, 27453,
 27454, 27460, 27461, 27464, 27469,
 27472, 27475, 27478, 27481, 27484,
 27487, 27490, 27506, 27507, 27517,
 27526, 27534, 27535, 27537, 27538,
 27543, 27544, 27553, 27560, 27569,
 27570, 27583, 27585, 27613, 27614,
 27623, 27626, 27651, 27657, 27658,
 27700, 27701, 27703, 27717, 27718,
 27726, 27737, 27771, 27774, 27784,
 27785, 27788, 27790, 27796, 27810,
 27812, 27853, 27856, 27876, 27949,
 27959, 27963, 27981, 27984, 28005,
 28006, 28013, 28017, 28035, 28038,
 28067, 28068, 28074, 28075, 28076,
 28083, 28091, 28095, 28109, 28112,
 28128, 28129, 28136, 28140, 28151,
 28155, 28167, 28172, 28173, 28174,
 28180, 28182, 28185, 28187, 28249,
 28278, 28288, 28295, 28300, 28301,
 28311, 28338, 28349, 28351, 28353,
 28355, 28360, 28361, 28363, 28374,
 28375, 28395, 28401, 28402, 28404,
 28407, 28412, 28418, 28419, 28449,
 28451, 28454, 28457, 28459, 28467,
 28469, 28473, 28477, 28482, 28487,
 28498, 28562, 28563, 28587, 28588,
 28589, 28590, 28591, 28599, 28610,
 28616, 28642, 28643, 28644, 28651,
 28652, 28659, 28660, 28668, 28669,
 28673, 28674, 28675, 28680, 28685,
 28691, 28694, 28695, 28696, 28697,
 28698, 28902, 28903, 29103, 29149,
 29151, 29265, 29267, 29274, 29275,
 29278, 29301, 29303, 29309, 29321,
 29329, 29331, 29423, 29462, 29532,
 29533, 29576, 29577, 29625, 29645,
 29709, 29735, 29744, 29759, 29761,
 29798, 29845, 29859, 29871, 30772,
 30774, 30778, 30822, 30824, 30838,
 30840, 31066, 31068, 31122, 31124,
 31126, 31141, 31143, 31145, 31264,
 31266, 31337, 31350, 31398, 31399,
 31460, 31461, 31462, 31469, 31528,
 31645, 31647, 31720, 31722, 33921,
 33925, 33943, 33944, 34357, 34410,
 34498, 34525, 34530, 34536, 34542,
 34683, 34701, 34865, 35142, 35886,
 35896, 35901, 35904, 36001, 36011,
 36150, 36155, 36160, 36165, 36179,
 36182, 36197, 36199, 36200, 36207,
 36212, 36214, 36217, 36231, 36251,
 36252, 36259, 36260, 36301, 36315
 \exp_args:cc 34, 1452, 2331
 \exp_args:Nc 32,
 34, 374, 1452, 1456, 1464, 1678,
 1691, 1699, 1707, 1902, 1921, 1947,
 1952, 1959, 2018, 2030, 2103, 2136,
 2137, 2138, 2139, 2161, 2165, 2331,
 2901, 4057, 5693, 10031, 10037,
 10261, 11882, 12212, 12472, 13098,
 17765, 21757, 23059, 23298, 24178,
 24202, 24232, 24234, 24236, 24238,
 24240, 24242, 24244, 24246, 24264,
 24280, 24281, 24300, 24302, 28812,
 28813, 28814, 28842, 31596, 33000,
 33031, 36135, 36221, 36226, 36234
 \exp_args:Ncc 36,
 1949, 1953, 1961, 2144, 2145, 2146,
 2147, 2331, 13954, 14136, 16098, 16138
 \exp_args:Nccc 37, 2331
 \exp_args:Ncco 37, 2430
 \exp_args:Nccx 37, 3279
 \exp_args:Ncf 36, 2371
 \exp_args:NcNc 37, 2430
 \exp_args:NcNo 37, 2430
 \exp_args:Ncno 37, 3279
 \exp_args:NcnV 37, 3279
 \exp_args:Ncnx 37, 3279
 \exp_args:Nco 36, 385, 2371
 \exp_args:Ncoo 37, 3279
 \exp_args:NcV 36, 2371
 \exp_args:Ncv 36, 2371
 \exp_args:NcVV 37, 3279
 \exp_args:Ncx 36, 3253
 \exp_args:Ne 35,
 381, 1517, 2284, 2347, 2579, 10800,
 10802, 10924, 10948, 11080, 11089,
 11108, 11118, 11128, 11141, 11344,
 13025, 15027, 15042, 18827, 18831,
 29593, 29677, 29687, 29801, 29962,
 30251, 30510, 30525, 30588, 31353,
 31502, 31637, 31714, 34032, 34222,
 34229, 34598, 35121, 35241, 35392
 \exp_args:Nee 36, 3253,
 10035, 11219, 11374, 34235, 34790
 \exp_args:Neee
 37, 3279, 11094, 34262, 34345
 \exp_args:Nf 35,
 2006, 2359, 2625, 2691, 2726, 2740,
 4434, 6471, 8471, 9355, 9356, 10642,
 10700, 12161, 12888, 12889, 12905,
 12923, 12931, 12935, 12959, 12965,
 12975, 13022, 13055, 13437, 13439,
 13498, 13500, 13516, 13853, 13858,
 14191, 14219, 14743, 14744, 14908,

- 14919, 16623, 16624, 16640, 17188,
 17193, 17198, 17203, 17374, 17443,
 17445, 17463, 17472, 17483, 17492,
 17630, 17647, 18394, 18408, 18429,
 18440, 18497, 18567, 18573, 18579,
 18585, 18737, 18857, 18941, 20081,
 20086, 20091, 20096, 22176, 24872,
 27945, 28511, 28754, 28920, 28975,
 34000, 34164, 35105, 35814, 35816
 \exp_args:Nff
 . 36, 3253, 12929, 22706, 35962, 36037
 \exp_args:Nffo 37, 3279
 \exp_args:Nfo 36, 3253
 \exp_args:NNc
 . 36, 353, 1948, 1951, 1960, 2032,
 2140, 2141, 2142, 2143, 2178, 2181,
 2331, 3171, 10113, 10244, 10245,
 10334, 16750, 17331, 17342, 20197,
 20204, 24882, 24889, 28539, 28844
 \exp_args:Nnc 36, 3253
 \exp_args:NNcf 37, 3279
 \exp_args:NNe 36, 2371, 34933
 \exp_args:Nne
 36, 3253, 10930, 10950, 10985
 \exp_args:NNf 36,
 2371, 6743, 10112, 10333, 10526,
 20190, 22285, 22290, 27172, 27173
 \exp_args:Nnf
 36, 3253, 12139, 18880, 21755
 \exp_args:Nnff 37, 3279, 18886
 \exp_args:Nnnc 37, 3279
 \exp_args:Nnnf 37, 3279
 \exp_args:NNNo
 37, 2342, 5045, 6140, 6203, 7551
 \exp_args:NNno 37, 3279
 \exp_args:Nnno 37, 3279
 \exp_args:NNNV
 37, 2430, 34088, 34583, 34673
 \exp_args:NNNv 37, 2430, 10247
 \exp_args:NNnV 37, 3279
 \exp_args:NNNx 37, 410, 3279, 5053, 5536
 \exp_args:NNnx 37, 3279
 \exp_args:Nnnx 37, 3279
 \exp_args:NNo 30,
 36, 2342, 2634, 7025, 7938, 12097,
 17362, 19632, 19690, 19818, 21595
 \exp_args:Nno
 36, 3253, 7045, 8889, 10218,
 10914, 12124, 12185, 12293, 12302,
 12653, 12668, 12759, 12793, 13380,
 16144, 20048, 22750, 22758, 22767,
 22784, 22792, 22820, 23324, 23328
 \exp_args:NNoo 37, 3279
 \exp_args:NNox 37, 3279
 \exp_args:Nnox 37, 3279
 \exp_args:NNV 36, 2371
 \exp_args:Nnv 36, 2371
 \exp_args:NnV 36, 3253, 34143
 \exp_args:Nnv 36, 3253
 \exp_args:NNVV 37, 3279
 \exp_args:NNx 36, 2186,
 3253, 5861, 11401, 11415, 11481, 35369
 \exp_args:Nnx 36, 3253, 12815
 \exp_args:No 32, 35, 1368, 2170, 2175,
 2342, 2634, 2638, 2668, 2706, 2769,
 2786, 2824, 3137, 3160, 3167, 3239,
 3256, 4061, 4088, 4148, 4150, 4254,
 5135, 5199, 5219, 5906, 5955, 6445,
 6873, 7067, 7082, 7177, 7375, 7760,
 7764, 7793, 7794, 7988, 8864, 8879,
 9528, 10277, 10408, 10438, 10534,
 10902, 10970, 12112, 12351, 12352,
 12353, 12380, 12381, 12382, 12383,
 12384, 12451, 12481, 12491, 12512,
 12581, 12584, 12586, 12667, 12676,
 12881, 12883, 12907, 12914, 12916,
 12973, 12982, 13319, 13346, 13351,
 13365, 13386, 13433, 13444, 13494,
 13505, 13572, 13591, 13629, 13644,
 14064, 14117, 14403, 16315, 17362,
 17449, 17455, 18085, 18097, 18099,
 18134, 18139, 18313, 18423, 18427,
 18461, 20348, 21117, 21151, 21179,
 21209, 21298, 21307, 21313, 21348,
 21355, 21410, 21595, 21628, 29296,
 32090, 36111, 36236, 36240, 36284
 \exp_args:Noc 36, 3253
 \exp_args:Nof 36, 3253, 12154
 \exp_args:Noo ... 36, 3253, 5241, 6458
 \exp_args:Noof 37, 3279
 \exp_args:Nooo 37, 3279
 \exp_args:Noooo 10037
 \exp_args:Noox 37, 3279
 \exp_args:Nox 36, 3253
 \exp_args:NV 35,
 2359, 10676, 10752, 10890, 11461,
 11466, 21115, 21149, 21177, 21207,
 28857, 29896, 34046, 34549, 34837,
 34850, 34948, 34972, 35014, 35359
 \exp_args:Nv
 . 35, 2359, 29723, 31495, 34045, 35430
 \exp_args:NVo 36, 3253
 \exp_args:NVV 36, 2371, 10587
 \exp_args:Nx . 36, 1976, 2465, 3180,
 4603, 5383, 5384, 5755, 6868, 8446,
 8447, 9302, 10166, 10169, 10380,
 10383, 10444, 13971, 17025, 17767,

- 18707, 21119, 21153, 21181, 21211,
24589, 28854, 28867, 35018, 35387
- \exp_args:Nxo 36, [3253](#)
- \exp_args:Nxx 36, [3253](#)
- \exp_args_generate:n 301, [3237](#), 10032
- \exp_args:Nn 35300
- \exp_end: 41, 357,
360, 364, 382, 383, 390, 391, 551,
552, 554, 573, 685, 693, 711, 1000,
1030, 1367, 1368, 1439, 1679, 1692,
1700, 1708, 2320, 2329, [2593](#), 2623,
2813, 3171, 7945, 7950, 8000, 8030,
8038, 8044, 8656, 8659, 8660, 8661,
8662, 8663, 8664, 8665, 8666, 8667,
8669, 12188, 12442, 12835, 12954,
13309, 13493, 17214, 17815, 18646,
18656, 18659, 18665, 18673, 18720,
18725, 19328, 20107, 23041, 24000,
26601, 28338, 28340, 28412, 29444,
29822, 31371, 35879, 36141, 36170
- \exp_end_continue_f:nw 42, [2593](#)
- \exp_end_continue_f:w
... 41, 42, 382, 1002, 1003, 2289,
2360, 2397, 2421, 2492, 2511, 2524,
2548, 2562, 2582, [2593](#), 8478, 9955,
10609, 12403, 17888, 19380, 20045,
22425, 22540, 22544, 23158, 23176,
23197, 23261, 23266, 23274, 23282,
23303, 23381, 23417, 23425, 23456,
23826, 23833, 23879, 23926, 23933,
23970, 24014, 24020, 24023, 24033,
24044, 24209, 24402, 24404, 24408,
24410, 24468, 24478, 24488, 24500,
24630, 24647, 24657, 24812, 24813,
24814, 25005, 25016, 25026, 25034,
26046, 26754, 26930, 27206, 27949,
27964, 27981, 28018, 28035, 28077,
28096, 28109, 28141, 28156, 28167,
28289, 28376, 28592, 28692, 29745
- \exp_last_two_unbraced:Nnn
..... 38, [2565](#), 33074, 33598, 33602
- \exp_last_unbraced:cf
..... 33964, 33970, 33976
- \exp_last_unbraced:Nco 38, [2499](#), 18215
- \exp_last_unbraced:NcV 38, [2499](#)
- \exp_last_unbraced:Ne 38, [2499](#), 28220
- \exp_last_unbraced:Nf
.... 38, [2499](#), 4754, 6066, 14287,
14572, 14761, 14933, 16027, 16048,
16155, 16177, 17461, 17481, 22192,
22207, 22701, 24621, 25098, 33954
- \exp_last_unbraced:Nfo 38, [2499](#), 28542
- \exp_last_unbraced:NNf 38, [2499](#)
- \exp_last_unbraced:NNNf 38, [2499](#), 8405
- \exp_last_unbraced:NNNf
..... 38, [2499](#), 8410
- \exp_last_unbraced:NNNNo 38, [2499](#),
2917, 2921, 3084, 10818, 13687,
14404, 20798, 22493, 22511, 29337
- \exp_last_unbraced:NNNo 38, [2499](#)
- \exp_last_unbraced:NnNo 38, [2499](#)
- \exp_last_unbraced:NNNV 38, [2499](#)
- \exp_last_unbraced:NNo .. 38, [2499](#),
10441, 12843, 29470, 31386, 33569
- \exp_last_unbraced:Nno
..... 38, [2499](#), 16708, 18244, 19882
- \exp_last_unbraced:NNV 38, [2499](#)
- \exp_last_unbraced:No
.. 38, [2499](#), 33725, 33730, 33798, 33804
- \exp_last_unbraced:Noo 38, [2499](#)
- \exp_last_unbraced:NV
..... 38, [2499](#), 8043, 35372
- \exp_last_unbraced:Nv 38, [2499](#), 18728
- \exp_last_unbraced:Nx 38, [2499](#)
- \exp_not:N .. 39, 92, 156, 257, 383,
389, 394, 427, 436, 443, 523, 528,
551, 689–691, 867, 878, 1008, [1435](#),
1634, 1723, 1726, 2038, 2039, 2103,
2104, 2201, [2267](#), 2313, 2466, [2570](#),
2570, 2648, 2652, 2694, 2747, 2767,
2777, 2790, 2894, 2896, 2897, 2904,
2905, 2906, 2907, 2908, 2909, 2915,
2941, 2950, 3005, 3006, 3066, 3072,
3074, 3080, 3141, 3158, 3160, 3188,
3823, 3826, 3978, 3979, 3980, 3981,
3982, 3983, 3984, 3985, 3986, 3987,
3988, 3989, 3990, 3991, 3992, 3993,
3996, 3997, 3998, 4205, 4214, 4215,
4220, 4228, 4230, 4324, 4326, 4328,
4334, 4336, 4378, 4730, 4732, 4734,
4736, 4738, 4740, 5123, 5125, 5325,
5327, 5338, 5342, 5500, 6155, 6862,
7277, 7290, 8024, 9401, 9403, 9405,
9407, 9412, 9413, 9418, 9420, 9422,
10137, 10138, 10142, 10899, 10902,
10903, 10905, 10906, 10907, 10908,
10911, 10912, 10977, 10979, 11299,
11300, 11301, 11302, 11303, 11306,
11307, 12236, 12683, 12701, 12727,
12734, 12745, 12746, 12818, 12821,
12822, 13126, 13127, 13337, 13338,
14181, 14182, 14209, 14210, 14211,
15049, 15050, 15051, 15053, 15054,
15056, 16314, 16316, 16732, 16779,
17347, 17610, 18148, 18603, 18677,
18681, 18710, 18895, 18898, 18901,
18904, 18907, 18910, 18913, 18980,
18984, 18989, 18994, 18999, 19006,

- 19013, 19018, 19023, 19028, 19033,
 19038, 19048, 19053, 19058, 19061,
 19062, 19065, 19075, 19080, 19095,
 19114, 19119, 19120, 19121, 19122,
 19123, 19124, 19126, 19128, 19129,
 19130, 19134, 19135, 19138, 19139,
 19231, 19234, 19235, 19237, 19238,
 19242, 19245, 19246, 19248, 19251,
 19371, 19384, 19398, 19411, 19417,
 19448, 19451, 19458, 19459, 19468,
 19469, 19483, 19484, 19495, 19496,
 19773, 19796, 19929, 20209, 20248,
 20812, 20814, 20868, 20869, 20881,
 20937, 20939, 20968, 20969, 21082,
 21083, 21280, 21281, 21282, 21283,
 21284, 21287, 21289, 21291, 21292,
 21331, 21332, 21333, 21334, 21337,
 21339, 21341, 21342, 21373, 21374,
 21375, 21376, 21379, 21381, 21383,
 21384, 21392, 21393, 21394, 21395,
 21396, 21399, 21401, 21403, 21404,
 22489, 22490, 23222, 23223, 23318,
 23319, 23320, 23321, 23427, 23467,
 23471, 23493, 23586, 23618, 23703,
 23717, 23734, 23745, 23755, 23792,
 23794, 23897, 23898, 23900, 23901,
 23902, 23903, 23904, 23905, 23908,
 23910, 23912, 24091, 24092, 24133,
 24134, 24254, 24269, 24894, 25051,
 27059, 27060, 27061, 27065, 27066,
 27067, 28907, 29103, 29105, 29259,
 29260, 29263, 29274, 29342, 29345,
 29348, 29351, 29354, 29357, 29360,
 29477, 29478, 29479, 29481, 29483,
 29487, 29491, 29492, 29665, 29666,
 29671, 29672, 29687, 29759, 29881,
 29946, 30104, 30109, 30111, 30112,
 30113, 30115, 30116, 30118, 30226,
 30228, 30231, 30736, 30742, 30744,
 30745, 30749, 30750, 30752, 30754,
 30773, 30775, 30779, 30823, 30825,
 30839, 30841, 31067, 31069, 31123,
 31125, 31127, 31142, 31144, 31146,
 31265, 31267, 31646, 31648, 31721,
 31723, 32805, 32806, 34026, 34027,
 34028, 34029, 34030, 34031, 34032,
 34033, 34243, 34374, 34375, 34376,
 34377, 34378, 34379, 34604, 34605,
 34609, 34611, 34715, 34716, 34818,
 34824, 34826, 34827, 34933, 34934,
 34935, 34936, 35091, 35092, 35093,
 35230, 35232, 35235, 35270, 35323,
 35819, 35821, 35825, 35827, 35832,
 35834, 35938, 36044, 36297, 36298,
 36340, 36348, 36364, 36367, 36663,
 36666, 36669, 36672, 36675, 36678
 \exp_not:n 20, 34, 35,
 39, 40, 51, 92, 112–116, 145, 150,
 151, 156, 177–180, 194, 201, 234,
 235, 304, 307, 344, 421, 426, 427,
 433, 436, 451, 523, 530, 533, 544,
 667, 675, 676, 696, 790, 793, 833,
 834, 836, 840, 850, 918, 1209, 1210,
 1212, 1368, 1370, 1375, 1435, 1635,
 1641, 1643, 1649, 1650, 1728, 1978,
 2213, 2214, 2267, 2280, 2296, 2483,
 2496, 2570, 2651, 2693, 3010, 3025,
 3040, 3115, 3162, 3185, 3584, 3851,
 3965, 3995, 4232, 4378, 4455, 5123,
 5125, 5758, 6212, 6379, 6597, 6786,
 6851, 6863, 6941, 6942, 7205, 7208,
 7277, 7285, 7302, 7317, 7358, 7542,
 7643, 7657, 7665, 7693, 7698, 7700,
 7991, 8921, 8953, 9395, 9412, 9560,
 10402, 10419, 11606, 11609, 11611,
 12001, 12021, 12046, 12063, 12237,
 12238, 12239, 12817, 12820, 12824,
 12904, 13164, 13207, 13218, 13358,
 16201, 16286, 16288, 16338, 16339,
 16346, 16347, 16363, 16395, 16415,
 16418, 16421, 16530, 16562, 16586,
 16639, 16733, 16789, 16840, 16850,
 17348, 17855, 17897, 17898, 17912,
 17914, 17984, 18049, 18085, 18093,
 18113, 18149, 18338, 18343, 18369,
 18372, 18375, 18407, 18438, 18458,
 18711, 18822, 19133, 19352, 19372,
 19412, 19418, 19603, 19696, 19774,
 19777, 19778, 19797, 19801, 19929,
 20210, 20598, 20607, 20871, 20881,
 20896, 20937, 20939, 20971, 21084,
 21285, 21293, 21321, 21334, 21335,
 21343, 21363, 21376, 21377, 21385,
 21397, 21405, 21609, 21619, 21645,
 21647, 23313, 24543, 24545, 24547,
 24583, 24895, 25052, 28231, 29278,
 29301, 29595, 29596, 29712, 29733,
 29964, 29965, 31683, 31687, 31688,
 32922, 34825, 35168, 35895, 36010,
 36014, 36017, 36018, 36051, 36141,
 36142, 36150, 36155, 36160, 36165,
 36179, 36197, 36210, 36217, 36347
 \exp_stop_f:
 .. 40, 41, 167, 382, 421, 427, 628,
 695, 789, 805, 966, 979, 1049, 1141,
 1170, 2286, 2754, 2757, 2772, 2780,
 2835, 3344, 3603, 3789, 3798, 3799,
 3833, 3903, 3936, 3937, 3941, 4012,

- 4028, 4034, 4116, 4535, 4536, 4537,
 4543, 4565, 4825, 4845, 4846, 4850,
 4854, 4855, 4858, 4859, 4867, 4868,
 4871, 4875, 4876, 4879, 4938, 5298,
 5303, 5317, 5318, 5331, 5393, 5394,
 5433, 5710, 6384, 6437, 6952, 6956,
 7105, 7129, 7164, 7169, 7175, 7512,
 8679, 8681, 9021, 10113, 10334,
 10597, 10611, 10623, 10833, 12941,
 13000, 13006, 13475, 13491, 13531,
 13537, 13549, 13566, 13798, 13806,
 14013, 14014, 14015, 14020, 14021,
 14045, 14416, 14478, 14482, 14512,
 14515, 14531, 14535, 14556, 14636,
 14638, 14658, 14659, 14676, 14678,
 14732, 14735, 14736, 14855, 14860,
 15001, 16350, 16937, 16951, 16961,
 16969, 17153, 17158, 17312, 18493,
 18495, 18563, 18565, 18569, 18571,
 18575, 18577, 18581, 18583, 18621,
 18629, 18630, 18631, 18632, 18638,
 18655, 18660, 18668, 18728, 18742,
 18743, 18749, 20054, 20225, 22085,
 22088, 22217, 22294, 22298, 22328,
 22533, 22648, 22663, 22910, 22936,
 22985, 22997, 23063, 23204, 23234,
 23390, 23433, 23484, 23504, 23531,
 23545, 23580, 23607, 23616, 23635,
 23651, 23667, 23685, 23746, 23765,
 23781, 24015, 24103, 24145, 24383,
 24387, 24701, 24703, 24718, 24735,
 24743, 24744, 24941, 25061, 25067,
 25082, 25119, 25192, 25214, 25268,
 25269, 25277, 25614, 25632, 25776,
 25788, 25804, 25821, 26099, 26100,
 26197, 26290, 26325, 26343, 26352,
 26354, 26510, 26545, 26698, 26748,
 26794, 26799, 26881, 26948, 26954,
 26969, 26981, 27019, 27040, 27080,
 27095, 27110, 27125, 27140, 27155,
 27183, 27227, 27493, 27503, 27533,
 27685, 27687, 27736, 27809, 27818,
 27833, 27885, 27898, 27982, 28036,
 28087, 28110, 28386, 28387, 28388,
 28397, 28407, 28425, 28494, 28497,
 28500, 28568, 28572, 28614, 28686,
 28693, 29328, 30358, 30361, 30362,
 30365, 30378, 30381, 30384, 30387,
 30390, 30393, 30407, 30410, 30413,
 30416, 30419, 30431, 30434, 30437,
 30440, 36297, 36298, 36299, 36300
- exp internal commands:
- __exp_arg_last_unbraced:nn .. [2467](#)
 - __exp_arg_next:Nnn [2267](#), [2274](#)
 - __exp_arg_next:nnn
[382](#), [2267](#), [2276](#), [2284](#), [2288](#), [2301](#), [2307](#)
 - __exp_e:N [2609](#), [2639](#)
 - __exp_e:nn [384](#), [391](#), [2356](#),
[2487](#), [2605](#), [2625](#), [2630](#), [2638](#), [2666](#),
[2668](#), [2713](#), [2714](#), [2719](#), [2786](#), [2804](#)
 - __exp_e:Nnn [392](#), [2639](#)
 - __exp_e_end:nn [391](#), [2605](#), [2738](#)
 - __exp_e_expandable:Nnn ... [392](#), [2639](#)
 - __exp_e_group:n [2612](#), [2626](#)
 - __exp_e_if_toks_register:N .. [2850](#)
 - __exp_e_if_toks_register:NTF ...
..... [2801](#), [2850](#)
 - __exp_e_noexpand:Nnn [2659](#), [2694](#), [2716](#)
 - __exp_e_primitive:Nnn ... [2661](#), [2669](#)
 - __exp_e_primitive_aux:NNnn .. [2669](#)
 - __exp_e_primitive_aux:NNw ... [2669](#)
 - __exp_e_primitive_other:NNnn . [2669](#)
 - __exp_e_primitive_other_-
aux:nNNnn [2669](#)
 - __exp_e_protected:Nnn [392](#), [2639](#)
 - __exp_e_put:nn
[392](#), [394](#), [395](#), [2626](#), [2719](#), [2731](#), [2818](#)
 - __exp_e_put:nnn [396](#), [2626](#), [2824](#)
 - __exp_e_space:nn [2616](#), [2624](#)
 - __exp_e_the:N [2782](#)
 - __exp_e_the:Nnn [2660](#), [2695](#), [2782](#)
 - __exp_e_the_errhelp: [2850](#)
 - __exp_e_the_everycr: [2850](#)
 - __exp_e_the_everydisplay: ... [2850](#)
 - __exp_e_the_everyeof: [2850](#)
 - __exp_e_the_everyhbox: [2850](#)
 - __exp_e_the_everyjob: [2850](#)
 - __exp_e_the_everymath: [2850](#)
 - __exp_e_the_everypar: [2850](#)
 - __exp_e_the_everyvbox: [2850](#)
 - __exp_e_the_output: [2850](#)
 - __exp_e_the_pdfpageattr: [2850](#)
 - __exp_e_the_pdfpageresources: [2850](#)
 - __exp_e_the_pdfpagesattr: ... [2850](#)
 - __exp_e_the_pdfpkmode: [2850](#)
 - __exp_e_the_toks:N [396](#), [2822](#)
 - __exp_e_the_toks:n . [396](#), [2798](#), [2822](#)
 - __exp_e_the_toks:wnn [396](#), [2797](#), [2822](#)
 - __exp_e_the_toks_reg:N [2782](#)
 - __exp_e_the_XeTeXinterchartoks:
..... [2850](#)
 - __exp_e_unexpanded:N [2721](#)
 - __exp_e_unexpanded:nN [394](#), [2721](#)
 - __exp_e_unexpanded:nn [2721](#)
 - __exp_e_unexpanded:Nnn
..... [2658](#), [2693](#), [2721](#)
 - __exp_eval_error_msg:w [2311](#)

- _exp_eval_register:N 3433, 3475, 3486, 3495, 3504, 3559,
 2302, 2308, 2311, 2364,
 2369, 2375, 2381, 2409, 2415, 2427,
 2428, 2435, 2442, 2473, 2478, 2501,
 2503, 2518, 2532, 2541, 2586, 2591
 - _l_exp_internal_tl 354, 1507, 1511, 1512,
 2267, 2267, 2295, 2297, 2496, 2497
 - _exp_last_two_unbraced:nnN 2565
 - \expandafter 4, 20, 21, 24, 25, 29, 30, 33,
 34, 45, 46, 70, 71, 73, 82, 107, 115, 268
 - \expanded 355, 816
 - \expandglyphsinfont 911
 - \ExplFileDate 10, 11426, 11441, 11455, 11459
 - \ExplFileDescription 10, 11425, 11438
 - \ExplFileExtension 11428, 11443, 11452
 - \ExplFileName 10, 11427, 11442, 11451
 - \ExplFileVersion 10, 11429, 11444, 11453
 - \explicitdiscretionary 817
 - \explicithyphenpenalty 815
 - \ExplSyntaxOff 5,
 9, 171, 314, 315, 344, 117, 147, 161
 - \ExplSyntaxOn 5,
 9, 171, 314, 315, 344, 671, 854, 143
- F**
- fact 252
 - false 257
 - \fam 269
 - \fi 18, 23, 32, 37, 49, 50,
 51, 71, 76, 77, 78, 93, 101, 113, 114, 270
 - fi commands:
 - \fi: 27, 64, 70, 93, 167,
 168, 194, 220, 281, 357, 359, 360,
 364, 365, 391, 427, 446, 447, 544,
 548, 573, 636, 671, 676, 711, 713,
 716, 811, 820, 858, 892, 893, 979,
 1007, 1022, 1055, 1416, 1463, 1631,
 1639, 1647, 1655, 1675, 1680, 1693,
 1701, 1709, 1711, 1712, 1713, 1714,
 1737, 1742, 1749, 1776, 1781, 1807,
 1808, 1816, 1822, 1835, 1836, 1844,
 1850, 1970, 1991, 2001, 2015, 2073,
 2134, 2252, 2262, 2316, 2319, 2326,
 2327, 2600, 2607, 2617, 2630, 2643,
 2648, 2651, 2652, 2653, 2654, 2662,
 2671, 2687, 2751, 2779, 2785, 2794,
 2799, 2805, 2808, 2812, 2820, 2830,
 2839, 2843, 2847, 2915, 2931, 2938,
 2947, 2961, 2962, 2967, 2968, 2969,
 2987, 2988, 2989, 2990, 2991, 2992,
 2993, 2994, 2995, 3003, 3022, 3024,
 3054, 3055, 3056, 3103, 3133, 3205,
 3216, 3226, 3346, 3357, 3358, 3402,
 3433, 3475, 3486, 3495, 3504, 3559,
 3569, 3579, 3691, 3693, 3766, 3770,
 3774, 3801, 3812, 3830, 3831, 3832,
 3839, 3855, 3861, 3864, 3872, 3882,
 3897, 3905, 3913, 3924, 3940, 3960,
 3973, 3995, 4013, 4021, 4023, 4026,
 4033, 4038, 4122, 4123, 4183, 4214,
 4215, 4216, 4223, 4233, 4242, 4264,
 4277, 4327, 4328, 4335, 4336, 4341,
 4342, 4369, 4370, 4441, 4498, 4505,
 4506, 4512, 4516, 4523, 4524, 4529,
 4530, 4540, 4541, 4546, 4547, 4561,
 4569, 4578, 4579, 4606, 4762, 4770,
 4781, 4787, 4799, 4839, 4841, 4848,
 4851, 4852, 4856, 4860, 4861, 4862,
 4863, 4872, 4873, 4877, 4880, 4881,
 4882, 4918, 4921, 4942, 4945, 4953,
 4964, 4965, 4976, 4977, 4985, 4997,
 4998, 5008, 5009, 5022, 5041, 5042,
 5050, 5051, 5116, 5126, 5159, 5173,
 5177, 5240, 5288, 5291, 5292, 5307,
 5310, 5333, 5391, 5392, 5397, 5424,
 5425, 5436, 5440, 5474, 5479, 5487,
 5522, 5529, 5534, 5544, 5556, 5582,
 5645, 5674, 5713, 5720, 5731, 5819,
 5838, 5844, 5849, 5872, 5884, 5885,
 5888, 5900, 5934, 5964, 6333, 6372,
 6388, 6412, 6432, 6441, 6498, 6505,
 6525, 6543, 6554, 6556, 6586, 6589,
 6623, 6629, 6635, 6732, 6770, 6801,
 6802, 6832, 6858, 6903, 6955, 7002,
 7003, 7015, 7066, 7068, 7121, 7133,
 7143, 7152, 7172, 7183, 7209, 7225,
 7233, 7283, 7285, 7300, 7302, 7323,
 7494, 7515, 7593, 7610, 7611, 7631,
 7657, 7659, 7665, 7667, 7672, 7702,
 7738, 7749, 7838, 7840, 7841, 7847,
 8027, 8039, 8426, 8469, 8487, 8509,
 8529, 8545, 8555, 8571, 8581, 8671,
 8673, 8675, 8677, 8681, 8684, 8877,
 8885, 10193, 10196, 10199, 10276,
 10294, 10549, 10590, 10599, 10620,
 10630, 10634, 10641, 10649, 10844,
 10848, 10851, 10899, 10912, 11232,
 11241, 11252, 11886, 12106, 12113,
 12247, 12248, 12275, 12285, 12300,
 12309, 12319, 12335, 12349, 12359,
 12363, 12376, 12393, 12408, 12634,
 12639, 12646, 12651, 12658, 12661,
 12687, 12705, 12723, 12731, 12741,
 12751, 12757, 12764, 12775, 12780,
 12782, 12786, 12791, 12795, 12796,
 12943, 12955, 13004, 13010, 13011,
 13209, 13214, 13219, 13226, 13231,

13326, 13404, 13408, 13409, 13427,
13480, 13493, 13535, 13541, 13542,
13553, 13566, 13567, 13588, 13626,
13652, 13775, 13783, 13791, 13802,
13819, 13821, 13965, 14017, 14018,
14023, 14026, 14027, 14049, 14102,
14202, 14418, 14451, 14487, 14488,
14519, 14520, 14540, 14541, 14559,
14644, 14648, 14658, 14668, 14684,
14688, 14691, 14696, 14698, 14741,
14745, 14746, 14837, 14842, 14853,
14859, 14871, 14874, 14876, 14880,
14994, 15008, 15009, 15927, 15935,
15959, 15973, 15981, 15992, 16002,
16022, 16052, 16053, 16125, 16148,
16167, 16170, 16391, 16394, 16431,
16492, 16509, 16519, 16573, 16578,
16944, 16945, 16954, 16977, 16994,
16995, 16997, 17014, 17015, 17058,
17111, 17119, 17146, 17154, 17160,
17183, 17221, 17229, 17314, 17529,
17562, 17610, 17615, 17731, 17756,
17782, 17791, 17961, 17976, 17999,
18013, 18626, 18629, 18630, 18631,
18632, 18637, 18638, 18643, 18644,
18645, 18672, 18681, 18723, 18731,
18733, 18777, 18778, 18781, 18917,
18918, 18919, 18920, 18921, 18922,
18923, 18985, 18990, 18995, 19000,
19007, 19014, 19019, 19024, 19029,
19034, 19039, 19044, 19049, 19054,
19076, 19087, 19088, 19138, 19139,
19261, 19270, 19279, 19287, 19364,
19393, 19394, 19402, 19440, 19454,
19463, 19473, 19495, 19496, 19497,
19507, 19514, 19516, 19846, 19847,
19848, 19849, 19858, 19859, 19860,
19861, 19884, 19885, 19886, 19887,
19896, 19897, 19898, 19899, 20004,
20027, 20036, 20053, 20057, 20071,
20074, 20236, 20237, 22090, 22091,
22126, 22134, 22200, 22219, 22233,
22382, 22398, 22402, 22414, 22424,
22519, 22572, 22575, 22576, 22581,
22595, 22633, 22634, 22635, 22636,
22637, 22638, 22639, 22640, 22641,
22642, 22643, 22644, 22657, 22659,
22670, 22673, 22687, 22692, 22696,
22825, 22916, 22917, 22926, 22927,
22938, 22939, 22940, 22951, 22952,
22953, 22960, 22971, 22972, 22973,
22983, 22984, 22988, 22989, 22997,
23000, 23001, 23009, 23020, 23040,
23063, 23098, 23115, 23134, 23135,
23144, 23150, 23170, 23171, 23199,
23208, 23225, 23232, 23240, 23241,
23342, 23343, 23344, 23347, 23350,
23389, 23405, 23431, 23432, 23439,
23447, 23476, 23477, 23480, 23482,
23483, 23488, 23498, 23501, 23503,
23508, 23539, 23552, 23557, 23563,
23566, 23567, 23601, 23602, 23629,
23630, 23643, 23646, 23657, 23680,
23699, 23709, 23725, 23734, 23740,
23746, 23750, 23755, 23761, 23776,
23787, 23804, 23812, 23814, 23820,
23841, 23869, 23892, 23925, 23927,
24050, 24098, 24102, 24112, 24113,
24129, 24140, 24144, 24154, 24155,
24175, 24196, 24199, 24229, 24261,
24277, 24297, 24338, 24350, 24363,
24365, 24385, 24386, 24393, 24411,
24450, 24460, 24625, 24641, 24652,
24681, 24682, 24683, 24690, 24692,
24693, 24699, 24700, 24703, 24730,
24738, 24739, 24747, 24748, 24750,
24751, 24922, 24935, 24945, 24946,
24951, 24952, 24953, 24954, 24955,
24956, 24963, 24973, 24980, 24991,
24992, 25003, 25020, 25065, 25066,
25073, 25086, 25101, 25111, 25125,
25155, 25164, 25198, 25220, 25238,
25255, 25272, 25273, 25275, 25276,
25281, 25296, 25329, 25358, 25359,
25360, 25361, 25362, 25375, 25419,
25492, 25559, 25561, 25562, 25572,
25601, 25604, 25605, 25616, 25636,
25699, 25700, 25701, 25713, 25752,
25753, 25754, 25755, 25761, 25764,
25766, 25776, 25794, 25809, 25821,
25828, 26075, 26079, 26081, 26085,
26092, 26093, 26103, 26104, 26107,
26199, 26269, 26280, 26292, 26324,
26331, 26342, 26358, 26365, 26426,
26483, 26493, 26495, 26505, 26523,
26524, 26556, 26559, 26568, 26570,
26572, 26586, 26600, 26624, 26708,
26716, 26747, 26755, 26761, 26772,
26775, 26778, 26787, 26796, 26798,
26804, 26811, 26814, 26823, 26831,
26852, 26885, 26886, 26913, 26915,
26933, 26934, 26953, 26964, 26973,
26976, 26987, 26990, 26993, 27011,
27021, 27032, 27034, 27043, 27090,
27105, 27120, 27135, 27150, 27165,
27168, 27170, 27187, 27232, 27539,
27575, 27576, 27586, 27627, 27628,
27652, 27679, 27680, 27683, 27685,

- 27686, 27691, 27703, 27722, 27727,
27735, 27738, 27770, 27780, 27781,
27791, 27813, 27828, 27846, 27854,
27857, 27885, 27893, 27909, 27981,
27999, 28035, 28053, 28086, 28109,
28115, 28188, 28189, 28320, 28321,
28330, 28337, 28342, 28352, 28362,
28387, 28390, 28403, 28435, 28443,
28444, 28472, 28494, 28495, 28496,
28499, 28504, 28524, 28525, 28577,
28578, 28619, 28627, 28680, 28686,
28699, 29141, 29142, 29150, 29164,
29168, 29169, 29185, 29264, 29268,
29279, 29300, 29304, 29320, 29332,
29364, 29365, 29366, 29367, 29368,
29369, 29370, 29518, 29524, 29762,
30369, 30370, 30373, 30374, 30397,
30398, 30399, 30400, 30401, 30402,
30423, 30424, 30425, 30426, 30427,
30444, 30445, 30446, 30447, 32058,
32060, 32066, 35810, 36009, 36045,
36050, 36051, 36169, 36176, 36181,
36207, 36213, 36217, 36232, 36253,
36261, 36297, 36298, 36299, 36305
 \c_fifteen 36429
 file commands:
 \file_add_path:nN 36449
 \file_compare_timestamp:nNn 97
 \file_compare_timestamp:nNnTF ...
 97, 11216
 \file_compare_timestamp_p:nNn ...
 97, 11216
 \g_file_curr_dir_str
 93, 10755, 11314, 11320, 11333
 \g_file_curr_ext_str
 93, 10755, 11316, 11322, 11335
 \g_file_curr_name_str
 93, 9089, 9216,
 10755, 11315, 11321, 11334, 36462
 \g_file_current_name_tl 36461
 \file_full_name:n 94,
 10922, 11021, 11081, 11089, 11095,
 11141, 11220, 11221, 35388, 35393
 \file_get:nnN
 94, 10881, 36562, 36564, 36566, 36568
 \file_get:nnNTF 94, 10881, 10883
 \file_get_full_name:nN
 94, 345, 11012, 36450
 \file_get_full_name:nNTF 94,
 10104, 10888, 11012, 11014, 11026,
 11027, 11270, 11276, 11281, 11293
 \file_get_hex_dump:nN 95, 11157
 \file_get_hex_dump:nnnN ... 95, 11201
 \file_get_hex_dump:nnnNTF
 95, 11201, 11203
 \file_get_hex_dump:nNTF
 95, 11157, 11158
 \file_get_md5five_hash:nN
 96, 11159, 11167
 \file_get_md5five_hash:nN\file_-
 get_size:nN 11157
 \file_get_md5five_hash:nN\file_-
 get_size:nNTF 11157
 \file_get_md5five_hash:nNTF 96, 11160
 \file_get_size:nN 96
 \file_get_size:nNTF 96, 11162
 \file_get_timestamp:nN 96, 11157
 \file_get_timestamp:nNTF
 96, 11157, 11164
 \file_hex_dump:n 95, 11092
 \file_hex_dump:nnn
 95, 11092, 11210, 35370
 \file_if_exist:nTF
 94, 97, 11268, 11579,
 11581, 11585, 13984, 36452, 36454
 \file_if_exist_input:n 97, 11274
 \file_if_exist_input:nTF
 97, 11274, 36451, 36453
 \file_input:n
 97, 11291, 13988, 36452, 36454
 \file_input_stop: 97, 11285
 \file_list: 36455
 \file_log_list: 97, 11394, 36456
 \file_md5five_hash:n 96, 11074
 \file_parse_full_name:n 95, 647, 11337
 \file_parse_full_name:nnNN
 95, 11051, 11318, 11382
 \file_parse_full_name_apply:nN ..
 95, 647, 11337, 11384
 \file_path_include:n 36457
 \file_path_remove:n 36459
 \l_file_search_path_seq 94-
 97, 10789, 10936, 11036, 36458, 36460
 \file_show_list: 97, 11394
 \file_size:n 96, 11074
 \file_timestamp:n 96, 11074
 file internal commands:
 \l__file_base_name_tl
 10784, 11033, 11069
 __file_compare_timestamp:nnN . 11216
 __file_const:nn 11593
 __file_details:nn 11074
 __file_details_aux:nn . 11074, 11109
 \l__file_dir_str
 10786, 11052, 11319, 11320
 __file_ext_check:nn
 10945, 10957, 10964

__file_ext_check:nnn . [10979](#), [10984](#)
 __file_ext_check:nnnn . [10985](#), [10986](#)
 __file_ext_check:nnnw . [10970](#), [10975](#)
 __file_ext_check:nnw [10965](#), [10966](#), [10973](#)
 \l_file_ext_str
 [10786](#), [11052](#), [11053](#), [11319](#), [11322](#)
 __file_full_name:n [10922](#)
 __file_full_name_assign:nnnNNN [11385](#), [11387](#)
 __file_full_name_aux:nN [10922](#)
 __file_full_name_aux:nn [10922](#)
 __file_full_name_aux:Nnn [10922](#)
 __file_full_name_aux:nnN [10922](#)
 \l_file_full_name_tl
 [10784](#), [10888](#), [10891](#),
 [11043](#), [11045](#), [11051](#), [11056](#), [11058](#),
 [11061](#), [11068](#), [11070](#), [11270](#), [11276](#),
 [11277](#), [11281](#), [11282](#), [11293](#), [11294](#)
 __file_get_aux:nnN [10881](#)
 __file_get_details:nnN [11157](#)
 __file_get_do:Nw [10881](#)
 __file_get_full_name_search:nN [11012](#)
 __file_hex_dump:n [11092](#)
 __file_hex_dump_auxi:nnn [11092](#)
 __file_hex_dump_auxii:nnnn [11092](#)
 __file_hex_dump_auxiii:nnnn [11092](#)
 __file_hex_dump_auxiiv:nnn [11092](#)
 __file_hex_dump_auxiv:nnn
 [11126](#), [11128](#), [11133](#)
 __file_id_info_auxi:w [11423](#)
 __file_id_info_auxii:w [650](#), [11423](#)
 __file_id_info_auxiii:w [11423](#)
 __file_if_recursion_tail_-
 break:NN [10796](#)
 __file_if_recursion_tail_stop:N [10796](#)
 __file_if_recursion_tail_stop_-
 do:Nn [10796](#)
 __file_if_recursion_tail_stop_-
 do:nn [10797](#)
 __file_input:n [11277](#), [11282](#), [11291](#)
 __file_input_pop: [11291](#)
 __file_input_pop:nnn [11291](#)
 __file_input_push:n [11291](#)
 \g_file_internal_ior
 [11047](#), [11055](#), [11057](#), [11060](#), [11070](#),
 [11071](#), [11073](#), [11481](#), [11492](#), [11494](#)
 \l_file_internal_tl
 [10754](#), [11327](#), [11328](#)
 __file_kernel_dependency_-
 compare:nnn [11460](#)
 __file_list:N [11394](#)
 __file_list_aux:n [11394](#)
 \c_file_marker_tl
 [636](#), [10880](#), [10903](#), [10916](#)
 __file_md5ive_hash:n [11074](#)
 __file_mismatched_dependency_-
 error:nn [11476](#), [11479](#)
 __file_name_cleanup:w [10922](#)
 __file_name_end: [10922](#)
 __file_name_expand:n [10798](#)
 __file_name_expand_cleanup:Nw [635](#), [10798](#)
 __file_name_expand_cleanup:w [635](#), [10798](#)
 __file_name_expand_end: [635](#), [10798](#)
 __file_name_expand_error:Nw [635](#), [10798](#)
 __file_name_expand_error_aux:Nw [635](#), [10798](#)
 __file_name_ext_check:nn [10922](#)
 __file_name_ext_check:nnn [10922](#)
 __file_name_ext_check:nnnn [10922](#)
 __file_name_ext_check:nnnw [10922](#)
 __file_name_ext_check:nnw [10922](#)
 __file_name_quote:nw [10872](#)
 \l_file_name_str
 [10786](#), [11052](#), [11319](#), [11321](#)
 __file_name_strip_quotes:n [10798](#)
 __file_name_strip_quotes:nnn [10798](#)
 __file_name_strip_quotes:nnnw [10798](#)
 __file_name_strip_quotes:nw
 [10837](#), [10840](#), [10846](#), [10849](#)
 __file_name_strip_quotes_-
 end:wnnw [10843](#), [10848](#)
 __file_name_trim_spaces:n [10798](#)
 __file_name_trim_spaces:nw [10798](#)
 __file_name_trim_spaces_aux:n [10798](#)
 __file_name_trim_spaces_aux:w [10798](#)
 __file_parse_full_name_area:nw
 [647](#), [11347](#)
 __file_parse_full_name_auxi:nN [11344](#), [11347](#)
 __file_parse_full_name_base:nw
 [648](#), [11355](#), [11358](#)
 __file_parse_full_name_tidy:nnnN [648](#), [11365](#), [11366](#), [11368](#), [11372](#)
 __file_parse_version:w [11460](#)
 __file_quark_if_nil:nTF
 [10793](#), [10862](#), [10876](#), [10968](#), [10977](#)
 __file_quark_if_nil_p:n [10793](#)
 \g_file_record_seq [646](#),
 [649](#), [10783](#), [11301](#), [11404](#), [11418](#), [11419](#)
 __file_size:n
 [10921](#), [10930](#), [10950](#), [10985](#)

- \g__file_stack_seq [646](#), [10758](#), [11312](#), [11327](#)
- __file_str_cmp:nn [11215](#), [11245](#)
- __file_timestamp:n [11216](#)
- __file_tmp:w [10760](#), [10764](#), [10768](#), [10774](#), [10780](#)
- \l__file_tmp_seq ... [10790](#), [11398](#),
[11401](#), [11404](#), [11405](#), [11407](#), [11415](#),
[11420](#), [11491](#), [11493](#), [11516](#), [11520](#)
- \filedump [775](#)
- \filemoddate [776](#)
- \filesize [777](#)
- \finalhyphendemerits [271](#)
- \firstmark [272](#)
- \firstmarks [529](#)
- \firstvalidlanguage [818](#)
- \c_five [36409](#)
- flag commands:
 - \flag_clear:n [169](#), [170](#), [5749](#), [7616](#),
[7617](#), [14062](#), [14090](#), [14184](#), [14213](#),
[14282](#), [14330](#), [14331](#), [14383](#), [14384](#),
[14620](#), [14621](#), [14622](#), [14623](#), [14624](#),
[14725](#), [14820](#), [14821](#), [14822](#), [14823](#),
[14978](#), [14979](#), [14980](#), [17747](#), [17760](#)
 - \flag_clear_new:n [170](#), [743](#),
[14565](#), [14566](#), [14567](#), [14568](#), [14748](#),
[14749](#), [14750](#), [14928](#), [14929](#), [17759](#)
 - \flag_height:n [170](#),
[7626](#), [7628](#), [13908](#), [17768](#), [17784](#), [17798](#)
 - \flag_if_exist:n [170](#)
 - \flag_if_exist:nTF . [170](#), [17760](#), [17771](#)
 - \flag_if_exist_p:n [170](#), [17771](#)
 - \flag_if_raised:n [170](#)
 - \flag_if_raised:nTF
... [170](#), [5757](#), [13901](#), [13906](#), [13908](#),
[14592](#), [14598](#), [14603](#), [14610](#), [14782](#),
[14787](#), [14792](#), [14943](#), [14950](#), [17776](#)
 - \flag_if_raised_p:n [170](#), [17776](#)
 - \flag_log:n [170](#), [17761](#)
 - \flag_new:n [169](#), [170](#), [743](#), [828](#), [5739](#),
[7476](#), [7477](#), [13770](#), [13771](#), [17742](#),
[17760](#), [22721](#), [22722](#), [22723](#), [22724](#)
 - \flag_raise:n [170](#), [7656](#),
[7664](#), [14048](#), [14098](#), [14198](#), [14231](#),
[14302](#), [14315](#), [14353](#), [14358](#), [14439](#),
[14641](#), [14642](#), [14665](#), [14666](#), [14679](#),
[14680](#), [14699](#), [14700](#), [14706](#), [14707](#),
[14737](#), [14887](#), [14888](#), [14997](#), [14998](#),
[15002](#), [15003](#), [15017](#), [15018](#), [17795](#)
 - \flag_raise_if_clear:n
... [302](#), [5776](#), [5798](#), [22755](#), [22764](#),
[22772](#), [22789](#), [22798](#), [22829](#), [35805](#)
 - \flag_show:n [170](#), [17761](#)
- flag internal commands:
 - __flag_clear:wn [17747](#)
 - __flag_height_end:wn [17784](#)
 - __flag_height_loop:wn [17784](#)
 - __flag_show:Nn [17761](#)
 - \floatingpenalty [273](#)
 - floor [253](#)
 - \fmtname [106](#), [8745](#), [8748](#),
[8749](#), [8761](#), [8771](#), [29669](#), [29670](#),
[32802](#), [32803](#), [33686](#), [33687](#), [35591](#)
 - \font [274](#)
 - \fontchardp [530](#)
 - \fontcharht [531](#)
 - \fontcharic [532](#)
 - \fontcharwd [533](#)
 - \fontdimen [950](#), [275](#)
 - \fontencoding [31545](#)
 - \fontfamily [31546](#)
 - \fontid [819](#)
 - \fontname [276](#)
 - \fontseries [31547](#)
 - \fontshape [31548](#)
 - \fontsize [31551](#)
 - \footnotesize [31587](#)
 - \forcecjktoken [1169](#)
 - \formatname [820](#)
 - \c_four [36407](#)
 - \c_fourteen [36427](#)
- fp commands:
 - \c_e_fp [247](#), [249](#), [24602](#)
 - flag_fp_division_by_zero . [248](#), [22721](#)
 - flag_fp_invalid_operation [248](#), [22721](#)
 - flag_fp_overflow [248](#), [22721](#)
 - flag_fp_underflow [248](#), [22721](#)
 - \fp_abs:n
... [252](#), [257](#), [1164](#), [28210](#), [32471](#),
[32573](#), [32575](#), [32577](#), [33401](#), [33403](#)
 - \fp_add:Nn [240](#), [1164](#), [24565](#)
 - \fp_compare:nNnTF [243–245](#),
[24643](#), [24784](#), [24790](#), [24795](#), [24803](#),
[24864](#), [24870](#), [32328](#), [32330](#), [32335](#),
[32604](#), [32619](#), [32628](#), [33143](#), [33375](#),
[34291](#), [34295](#), [34303](#), [34310](#), [34317](#),
[34324](#), [34331](#), [34744](#), [34747](#), [35193](#)
 - \fp_compare:nTF [243–245](#),
[251](#), [24627](#), [24756](#), [24762](#), [24767](#), [24775](#)
 - \fp_compare_p:n [244](#), [24627](#)
 - \fp_compare_p:nNn [243](#), [24643](#)
 - \fp_const:Nn
[240](#), [24542](#), [24602](#), [24603](#), [24604](#), [24605](#)
 - \fp_do_until:nn [245](#), [24753](#)
 - \fp_do_until:nNnn [244](#), [24781](#)
 - \fp_do_while:nn [245](#), [24753](#)
 - \fp_do_while:nNnn [244](#), [24781](#)

- \fp_eval:n [241](#),
[244](#), [251–257](#), [1044](#), [28205](#), [33993](#),
[33995](#), [34001](#), [34005](#), [34006](#), [34007](#),
[34011](#), [34014](#), [34015](#), [34016](#), [34165](#),
[34175](#), [34179](#), [34180](#), [34181](#), [34186](#),
[34187](#), [34188](#), [34189](#), [34217](#), [34222](#),
[34230](#), [34236](#), [34245](#), [34246](#), [34247](#),
[34263](#), [34264](#), [34265](#), [34273](#), [34274](#),
[34275](#), [34282](#), [34283](#), [34284](#), [34346](#),
[34351](#), [34896](#), [34897](#), [34898](#), [34899](#),
[34911](#), [34912](#), [34913](#), [34924](#), [35049](#),
[35050](#), [35112](#), [35271](#), [35272](#), [35273](#),
[35274](#), [35282](#), [35283](#), [35284](#), [35285](#),
[35301](#), [35307](#), [35324](#), [35325](#), [35326](#),
[35334](#), [35335](#), [35336](#), [36060](#), [36079](#)
- \fp_format:nn [258](#)
- \fp_gadd:Nn [240](#), [24565](#)
- .fp_gset:N [224](#), [21162](#)
- \fp_gset:Nn .. [240](#), [24542](#), [24566](#), [24568](#)
- \fp_gset_eq:NN [240](#), [24551](#), [24556](#)
- \fp_gsub:Nn [241](#), [24565](#)
- \fp_gzero:N [240](#), [24555](#), [24562](#)
- \fp_gzero_new:N [240](#), [24559](#)
- \fp_if_exist:NTF
..... [242](#), [24560](#), [24562](#), [24617](#)
- \fp_if_exist_p:N [242](#), [24617](#)
- \fp_if_nan:n [301](#)
- \fp_if_nan:nTF [258](#), [301](#), [24619](#)
- \fp_if_nan_p:n [301](#), [24619](#)
- \fp_log:N [248](#), [24575](#)
- \fp_log:n [248](#), [24598](#)
- \fp_max:nn [257](#), [28212](#)
- \fp_min:nn [257](#), [28212](#)
- \fp_new:N [240](#), [24539](#), [24560](#), [24562](#),
[24606](#), [24607](#), [24608](#), [24609](#), [32294](#),
[32295](#), [32296](#), [32422](#), [32423](#), [32672](#),
[32673](#), [33170](#), [33171](#), [33335](#), [33336](#)
- .fp_set:N [224](#), [21162](#)
- \fp_set:Nn [240](#), [24542](#), [24565](#), [24567](#),
[32316](#), [32317](#), [32318](#), [32441](#), [32443](#),
[32484](#), [32504](#), [32524](#), [32541](#), [32543](#),
[32561](#), [32562](#), [32602](#), [32603](#), [33188](#),
[33189](#), [33355](#), [33357](#), [33395](#), [33396](#)
- \fp_set_eq:NN .. [240](#), [24551](#), [24555](#),
[32489](#), [32509](#), [32526](#), [32605](#), [32606](#)
- \fp_show:N [248](#), [24575](#)
- \fp_show:n [248](#), [24598](#)
- \fp_sign:n [241](#), [28208](#)
- \fp_step_function:nnnN
..... [246](#), [24809](#), [24901](#)
- \fp_step_inline:nnnn [246](#), [24879](#)
- \fp_step_variable:nnnNn .. [246](#), [24879](#)
- \fp_sub:Nn [241](#), [24565](#)
- \fp_to_decimal:N
..... [241](#), [242](#), [22712](#), [28012](#), [28043](#), [28205](#)
- \fp_to_decimal:n . [241](#), [242](#), [28012](#),
[28207](#), [28209](#), [28211](#), [28213](#), [28215](#)
- \fp_to_dim:N [241](#), [1162](#), [28135](#)
- \fp_to_dim:n [241](#), [247](#), [28135](#), [32360](#),
[32371](#), [32471](#), [33098](#), [33120](#), [33148](#),
[33162](#), [33272](#), [33280](#), [33411](#), [33413](#)
- \fp_to_int:N [241](#), [28151](#)
- \fp_to_int:n [241](#), [28151](#), [34748](#)
- \fp_to_scientific:N
..... [242](#), [27958](#), [27989](#), [27996](#)
- \fp_to_scientific:n [242](#), [27958](#)
- \fp_to_tl:N
..... [242](#), [260](#), [22713](#), [24589](#), [28091](#)
- \fp_to_tl:n [242](#),
[22339](#), [22754](#), [22763](#), [22788](#), [22797](#),
[22826](#), [24423](#), [24438](#), [24599](#), [24601](#),
[24836](#), [24837](#), [24856](#), [24867](#), [28091](#)
- \fp_trap:nn [248](#),
[984](#), [22725](#), [22840](#), [22841](#), [22842](#), [22843](#)
- \fp_until_do:nn [245](#), [24753](#)
- \fp_until_do:nNnn [245](#), [24781](#)
- \fp_use:N [242](#), [260](#), [28205](#)
- \fp_while_do:nn [245](#), [24753](#)
- \fp_while_do:nNnn [245](#), [24781](#)
- \fp_zero:N [240](#), [24555](#), [24560](#)
- \fp_zero_new:N [240](#), [24559](#)
- \c_inf_fp [247](#),
[256](#), [22353](#), [23935](#), [25372](#), [25454](#),
[25792](#), [26552](#), [26575](#), [26777](#), [26780](#),
[26784](#), [26807](#), [27009](#), [27172](#), [28697](#)
- \c_minus_inf_fp
..... [247](#), [256](#), [22353](#), [25373](#),
[25457](#), [25790](#), [26327](#), [27173](#), [28698](#)
- \c_minus_zero_fp
..... [246](#), [22353](#), [25369](#), [27892](#), [28696](#)
- \c_nan_fp ... [256](#), [987](#), [1012](#), [22353](#),
[22765](#), [22773](#), [22845](#), [23051](#), [23070](#),
[23076](#), [23099](#), [23266](#), [23274](#), [23282](#),
[23360](#), [23417](#), [23456](#), [23847](#), [23924](#),
[23936](#), [24425](#), [24440](#), [24860](#), [26751](#),
[28249](#), [28295](#), [28610](#), [28669](#), [28695](#)
- \c_one_degree_fp [247](#), [256](#), [23938](#), [24604](#)
- \c_one_fp [246](#), [1040](#),
[1148](#), [23939](#), [24368](#), [24389](#), [24602](#),
[24960](#), [25813](#), [26546](#), [26746](#), [26797](#),
[26982](#), [27096](#), [27126](#), [27675](#), [28311](#)
- \c_pi_fp . [247](#), [256](#), [1022](#), [23937](#), [24604](#)
- \g_tmpa_fp [247](#), [24606](#)
- \l_tmpa_fp [247](#), [24606](#)
- \g_tmpb_fp [247](#), [24606](#)
- \l_tmpb_fp [247](#), [24606](#)

- \c_zero_fp [246](#),
[1043](#), [1060](#), [1176](#), [22353](#), [22407](#),
[23940](#), [24380](#), [24392](#), [24540](#), [24555](#),
[24556](#), [24962](#), [24965](#), [25201](#), [25368](#),
[26555](#), [26576](#), [26774](#), [26810](#), [27890](#),
[27996](#), [28180](#), [28694](#), [32328](#), [32330](#),
[32335](#), [32619](#), [32628](#), [33375](#), [35193](#)
- fp internal commands:
 - __fp_ [24969](#), [24976](#), [24985](#), [24986](#)
 - __fp_&o:ww [1046](#), [1055](#), [24966](#)
 - __fp_&tuple_o:ww [24966](#)
 - __fp_*_o:ww [25333](#)
 - __fp_*_tuple_o:ww [25839](#)
 - __fp+_o:ww . [1058](#), [1059](#), [1088](#), [25054](#)
 - __fp-_o:ww [1058](#), [1059](#), [25049](#)
 - __fp/_o:ww . [1067](#), [1068](#), [1111](#), [25445](#)
 - __fp^_o:ww [26742](#)
 - __fp_acos_o:w [1152](#), [1155](#), [27831](#)
 - __fp_acot_o:Nw . [27071](#), [27073](#), [27663](#)
 - __fp_acotii_o:Nww [27673](#), [27676](#)
 - __fp_acotii_o:ww [1148](#)
 - __fp_acsc_normal_o:NnwNnw
[1154](#), [27889](#), [27904](#), [27912](#)
 - __fp_acsc_o:w [27883](#)
 - __fp_add:NNNn [24565](#)
 - __fp_add_big_i:wNww [1061](#)
 - __fp_add_big_i_o:wNww
[1058](#), [1061](#), [25121](#), [25128](#)
 - __fp_add_big_ii:wNww [1061](#)
 - __fp_add_big_ii_o:wNww [25124](#), [25128](#)
 - __fp_add_inf_o:Nww ... [25070](#), [25090](#)
 - __fp_add_normal_o:Nww
[1061](#), [25069](#), [25105](#)
 - __fp_add_npos_o:NnwNnw
[1061](#), [25108](#), [25114](#)
 - __fp_add_return_ii_o:Nww
[25072](#), [25078](#), [25083](#)
 - __fp_add_significand_carry_-
o:wwwNN [1063](#), [25161](#), [25176](#)
 - __fp_add_significand_no_carry_-
o:wwwNN [1062](#), [25163](#), [25166](#)
 - __fp_add_significand_o:NnnwnnnnN
.... [1061](#), [1062](#), [25131](#), [25139](#), [25144](#)
 - __fp_add_significand_pack:NNNNNNN
..... [25144](#)
 - __fp_add_significand_test_o:N [25144](#)
 - __fp_add_zeros_o:Nww . [25068](#), [25080](#)
 - __fp_and_return:wNw [24966](#)
 - __fp_array_bounds:NNnTF
[28566](#), [28597](#), [28667](#)
 - __fp_array_bounds_error:NNn . [28566](#)
 - __fp_array_count:n [22456](#),
[23035](#), [24710](#), [24711](#), [25852](#), [27931](#)
 - __fp_array_gset:NNNww [28585](#)
 - __fp_array_gset:w [28585](#)
 - __fp_array_gset_normal:w [28585](#)
 - __fp_array_gset_recover:Nw .. [28585](#)
 - __fp_array_gset_special:nnNNN ..
..... [28585](#), [28642](#)
 - __fp_array_gzero:N [1176](#)
 - __fp_array_if_all_fp:nTF
..... [22468](#), [24418](#)
 - __fp_array_if_all_fp_loop:w . [22468](#)
 - \g__fp_array_int
..... [28531](#), [28538](#), [28540](#), [28552](#)
 - __fp_array_item:N [28649](#)
 - __fp_array_item:NNNnN [28649](#)
 - __fp_array_item:NwN [28649](#)
 - __fp_array_item:w [28649](#)
 - __fp_array_item_normal:w [28649](#)
 - __fp_array_item_special:w ... [28649](#)
 - \l__fp_array_loop_int
..... [28532](#), [28638](#), [28641](#), [28644](#)
 - __fp_array_new:nnnn [28533](#)
 - __fp_array_new:nnnnn . [28542](#), [28546](#)
 - __fp_array_to_clist:n
..... [23103](#), [28216](#), [28335](#)
 - __fp_array_to_clist_loop:Nw . [28216](#)
 - __fp_asec_o:w [27896](#)
 - __fp_asin_auxi_o:NnNww
.... [1153](#), [1155](#), [27861](#), [27864](#), [27923](#)
 - __fp_asin_isqrt:wn [27864](#)
 - __fp_asin_normal_o:NnwNnnnw ...
..... [27822](#), [27838](#), [27849](#)
 - __fp_asin_o:w [27816](#)
 - __fp_atan_auxi:ww [1150](#), [27741](#), [27755](#)
 - __fp_atan_auxii:w [27755](#)
 - __fp_atan_combine_aux:ww [27782](#)
 - __fp_atan_combine_o:NwwwwwN ...
..... [1149](#), [27700](#), [27717](#), [27782](#)
 - __fp_atan_default:w [1040](#), [1148](#), [27663](#)
 - __fp_atan_div:wnwnw
..... [1149](#), [27728](#), [27730](#)
 - __fp_atan_inf_o:NNw [1148](#), [27688](#),
[27689](#), [27690](#), [27698](#), [27834](#), [27907](#)
 - __fp_atan_near:wwwn [27730](#)
 - __fp_atan_near_aux:wnw [27730](#)
 - __fp_atan_normal_o:NNnwNnw
..... [1148](#), [27692](#), [27708](#)
 - __fp_atan_o:Nw . [27075](#), [27077](#), [27663](#)
 - __fp_atan_Taylor_break:w [27766](#)
 - __fp_atan_Taylor_loop:www
..... [1150](#), [27761](#), [27766](#)
 - __fp_atan_test_o:NwNwNwN
..... [1154](#), [27711](#), [27715](#), [27871](#)
 - __fp_atanii_o:Nww [27667](#), [27676](#)

```

\__fp_basics_pack_high:NNNNw ...
    . 1062, 1079, 22566, 25169, 25321,
    25424, 25436, 25578, 25771, 26297
\__fp_basics_pack_high_carry:w ..
    ..... 976, 22566
\__fp_basics_pack_low:NNNNw ...
    ..... 1069, 1079,
    22566, 25171, 25323, 25426, 25438,
    25580, 25720, 25722, 25773, 26299
\__fp_basics_pack_weird_high:NNNNNNNw
    ..... 22577, 25180, 25589
\__fp_basics_pack_weird_low:NNNNw
    ..... 22577, 25182, 25591
\c__fp_big_leading_shift_int ...
    .. 22552, 25650, 25985, 25995, 26005
\c__fp_big_middle_shift_int ....
    .... 22552, 25653, 25656, 25659,
    25662, 25665, 25668, 25672, 25987,
    25997, 26007, 26017, 26020, 26023
\c__fp_big_trailing_shift_int ...
    ..... 22552, 25676, 26030
\c__fp_Bigg_leading_shift_int ...
    ..... 22557, 25499, 25517
\c__fp_Bigg_middle_shift_int ...
    .. 22557, 25502, 25505, 25520, 25523
\c__fp_Bigg_trailing_shift_int ..
    ..... 22557, 25508, 25526
\__fp_binary_rev_type_o:Nww ....
    ..... 24058, 25842, 25844
\__fp_binary_type_o:Nww .....
    ..... 24058, 25840, 25853
\c__fp_block_int ..... 22358, 26249
\__fp_case_return:nw .....
    . 979, 22634, 22664, 22667, 22672,
    23164, 26511, 27006, 27688, 27689,
    27690, 27983, 28037, 28111, 28113,
    28114, 28180, 28615, 28617, 28618
\__fp_case_return_i_o:ww . 22641,
    25071, 25085, 25094, 25366, 27679
\__fp_case_return_ii_o:ww .....
    .. 22641, 25367, 26795, 26813, 27680
\__fp_case_return_o:Nw . 979, 980,
    22635, 25792, 26546, 26551, 26554,
    26746, 26751, 26774, 26777, 26780,
    26982, 27096, 27126, 27890, 27892
\__fp_case_return_o:Nww .....
    ..... 22639, 25368, 25369,
    25372, 25373, 26797, 26806, 26809
\__fp_case_return_same_o:w . 979,
    980, 22637, 25601, 25605, 25793,
    25805, 25808, 26330, 26558, 26771,
    26986, 26989, 27081, 27089, 27104,
    27119, 27134, 27141, 27149, 27164,
    27819, 27827, 27845, 27891, 27908
\__fp_case_use:nw ..... 979,
    22633, 25096, 25364, 25365, 25370,
    25371, 25453, 25456, 25603, 25789,
    26323, 26326, 26782, 26992, 27082,
    27087, 27097, 27102, 27112, 27117,
    27127, 27132, 27142, 27147, 27157,
    27162, 27821, 27824, 27834, 27836,
    27842, 27886, 27888, 27899, 27902,
    27907, 27986, 27993, 28040, 28047
\__fp_change_func_type:NNN .....
    .... 22496, 23851, 25835, 27968,
    28022, 28099, 28145, 28160, 28599
\__fp_change_func_type_aux:w . 22496
\__fp_change_func_type_chk:NNN 22496
\__fp_chk:w .....
    . 965-967, 1022, 1059, 1061, 1063,
    1069, 1072, 22340, 22353, 22354,
    22355, 22356, 22357, 22367, 22372,
    22374, 22375, 22403, 22406, 22408,
    22418, 22431, 22450, 22645, 22661,
    22821, 22826, 23053, 23107, 23116,
    23118, 23949, 24586, 24592, 24595,
    24596, 24677, 24678, 24840, 24856,
    24860, 24924, 24925, 24928, 24939,
    24940, 24948, 24949, 24957, 24969,
    24972, 24976, 24979, 25055, 25075,
    25076, 25078, 25079, 25080, 25088,
    25091, 25102, 25103, 25105, 25114,
    25190, 25342, 25376, 25377, 25380,
    25461, 25599, 25607, 25609, 25786,
    25795, 25797, 25802, 25810, 25812,
    25814, 25818, 26320, 26332, 26334,
    26543, 26560, 26562, 26743, 26762,
    26764, 26765, 26768, 26785, 26788,
    26791, 26815, 26816, 26818, 26834,
    26923, 26936, 26938, 26942, 26946,
    26979, 26995, 27078, 27091, 27093,
    27106, 27108, 27121, 27123, 27136,
    27138, 27151, 27153, 27166, 27176,
    27677, 27693, 27694, 27698, 27709,
    27816, 27829, 27831, 27847, 27850,
    27860, 27883, 27894, 27896, 27910,
    27912, 27917, 27979, 28000, 28003,
    28033, 28054, 28057, 28107, 28123,
    28126, 28201, 28202, 28312, 28314,
    28346, 28612, 28620, 28623, 28702
\__fp_compare:wNNNNw ..... 24308
\__fp_compare_aux:wn ..... 24643
\__fp_compare_back:ww .....
    ..... 1170, 24659, 24938, 28330
\__fp_compare_back_any:ww .....
    1047-1049, 24383, 24656, 24659, 24727
\__fp_compare_back_tuple:ww .. 24704
\__fp_compare_nan:w .... 1048, 24659

```

- _fp_compare_npos:nwnw ... [1046](#),
[1048](#), [1049](#), [24687](#), [24733](#), [25192](#), [26099](#)
- _fp_compare_return:w [24627](#)
- _fp_compare_significand:nnnnnnnn
..... [24733](#)
- _fp_cos_o:w [27093](#)
- _fp_cot_o:w [1133](#), [27153](#)
- _fp_cot_zero_o:Nnw
..... [1132](#), [1133](#), [27111](#), [27153](#)
- _fp_csc_o:w [27108](#)
- _fp_decimate:nNnnnn
..... [977](#), [980](#), [1126](#), [22587](#),
[22652](#), [22679](#), [23120](#), [25130](#), [25138](#),
[25217](#), [26589](#), [26593](#), [26961](#), [28063](#)
- _fp_decimate:Nnnnn [22599](#)
- _fp_decimate_auxi:Nnnnn [978](#), [22603](#)
- _fp_decimate_auxii:Nnnnn ... [22603](#)
- _fp_decimate_auxiii:Nnnnn ... [22603](#)
- _fp_decimate_auxiv:Nnnnn ... [22603](#)
- _fp_decimate_auxix:Nnnnn ... [22603](#)
- _fp_decimate_auxv:Nnnnn ... [22603](#)
- _fp_decimate_auxvi:Nnnnn ... [22603](#)
- _fp_decimate_auxvii:Nnnnn ... [22603](#)
- _fp_decimate_auxviii:Nnnnn ... [22603](#)
- _fp_decimate_auxx:Nnnnn ... [22603](#)
- _fp_decimate_auxxi:Nnnnn ... [22603](#)
- _fp_decimate_auxxii:Nnnnn ... [22603](#)
- _fp_decimate_auxxiii:Nnnnn ... [22603](#)
- _fp_decimate_auxxiv:Nnnnn ... [22603](#)
- _fp_decimate_auxxv:Nnnnn ... [22603](#)
- _fp_decimate_auxxvi:Nnnnn ... [22603](#)
- _fp_decimate_pack:nnnnnnnnnw .
..... [978](#), [22610](#), [22629](#)
- _fp_decimate_pack:nnnnnnw ...
..... [22630](#), [22631](#)
- _fp_decimate_tiny:Nnnnn ... [22599](#)
- _fp_div_npos_o:Nww
..... [1071](#), [1072](#), [25450](#), [25460](#)
- _fp_div_significand_calc:wnnnnnnn
..... [1075](#),
[25477](#), [25486](#), [25534](#), [26403](#), [26410](#)
- _fp_div_significand_calc_-
i:wnnnnnnn [25486](#)
- _fp_div_significand_calc_-
ii:wnnnnnnn [25486](#)
- _fp_div_significand_i_o:wnw ..
..... [1072](#), [1075](#), [25467](#), [25473](#)
- _fp_div_significand_ii:wnw ...
... [1077](#), [25481](#), [25482](#), [25483](#), [25530](#)
- _fp_div_significand_iii:wnnnnn
..... [1077](#), [25484](#), [25537](#)
- _fp_div_significand_iv:wnnnnnnn
..... [1077](#), [25540](#), [25545](#)
- _fp_div_significand_large_-
o:wnnnnnwN [1079](#), [25571](#), [25585](#)
- _fp_div_significand_pack:NNN ..
..... [1079](#), [1112](#),
[25532](#), [25565](#), [26390](#), [26408](#), [26416](#)
- _fp_div_significand_small_-
o:wnnnnnwN [1079](#), [25569](#), [25575](#)
- _fp_div_significand_test_o:w ..
..... [1079](#), [25475](#), [25566](#)
- _fp_div_significand_v:NN
..... [25550](#), [25552](#), [25555](#)
- _fp_div_significand_v:NNw .. [25545](#)
- _fp_div_significand_vi:Nw ...
..... [1078](#), [25545](#)
- _fp_division_by_zero_o:Nnw ...
..... [984](#), [22785](#),
[22833](#), [25790](#), [26327](#), [27172](#), [27173](#)
- _fp_division_by_zero_o:NNww ...
[984](#), [22793](#), [22833](#), [25454](#), [25457](#), [26784](#)
- \c_fp_empty_tuple_fp
..... [22451](#), [23260](#), [23910](#), [23920](#)
- _fp_ep_compare:www . [26094](#), [27724](#)
- _fp_ep_compare_aux:www [26094](#)
- _fp_ep_div:wwwwn
..... [1145](#), [26124](#), [26235](#),
[27653](#), [27740](#), [27744](#), [27753](#), [27920](#)
- _fp_ep_div_eps_pack:NNNNwN . [26154](#)
- _fp_ep_div_epsi:wnNNNNn [1102](#)
- _fp_ep_div_epsi:wnNNNNNN
..... [26151](#), [26154](#)
- _fp_ep_div_epsii:wnNNNNNN . [26154](#)
- _fp_ep_div_esti:wwwwn
..... [1102](#), [26130](#), [26133](#)
- _fp_ep_div_estii:wnnnwnw ... [26133](#)
- _fp_ep_div_estiii:NNNNwwww . [26133](#)
- _fp_ep_inv_to_float_o:wN ... [1134](#)
- _fp_ep_inv_to_float_o:wwN
... [1144](#), [26231](#), [26239](#), [27115](#), [27130](#)
- _fp_ep_isqrt:wnw [26177](#), [27881](#)
- _fp_ep_isqrt_aux:wnw [26177](#)
- _fp_ep_isqrt_auxi:wnw [26180](#), [26182](#)
- _fp_ep_isqrt_auxii:wnnnwn . [26177](#)
- _fp_ep_isqrt_epsi:wN
..... [1105](#), [26214](#), [26217](#)
- _fp_ep_isqrt_epsii:wwN [26217](#)
- _fp_ep_isqrt_esti:wnnnwn ...
..... [26192](#), [26195](#)
- _fp_ep_isqrt_estii:wnnnwn . [26195](#)
- _fp_ep_isqrt_estiii:NNNNwwww .
..... [26195](#)
- _fp_ep_mul:wwwwn
..... [1128](#), [26109](#), [27022](#),
[27035](#), [27610](#), [27640](#), [27868](#), [27879](#)

- __fp_ep_mul_raw:wwwN
..... [26109](#), [27194](#), [27560](#)
- __fp_ep_to_ep:wwN . [26060](#), [26111](#),
[26114](#), [26126](#), [26129](#), [26179](#), [27869](#)
- __fp_ep_to_ep_end:www [26060](#)
- __fp_ep_to_ep_loop:N
..... [1143](#), [26060](#), [27561](#)
- __fp_ep_to_ep_zero:ww [26060](#)
- __fp_ep_to_fixed:wn [26042](#),
[27191](#), [27747](#), [27756](#), [27866](#), [28355](#)
- __fp_ep_to_fixed_auxi:ww [26042](#)
- __fp_ep_to_fixed_auxii:nnnnnnwn
..... [26042](#)
- __fp_ep_to_float_o:wN [1134](#)
- __fp_ep_to_float_o:wwN
..... [1131](#), [1144](#), [26231](#),
[26243](#), [27046](#), [27085](#), [27100](#), [27659](#)
- __fp_error:nnnn [22754](#),
[22762](#), [22771](#), [22788](#), [22796](#), [22824](#),
[22847](#), [23046](#), [23048](#), [23069](#), [23074](#),
[23846](#), [24421](#), [24436](#), [24836](#), [24855](#),
[24866](#), [27974](#), [28028](#), [28102](#), [28609](#)
- __fp_exp_after_?_f:nw
..... [973](#), [1008](#), [23244](#)
- __fp_exp_after_any_f:Nnw [22521](#)
- __fp_exp_after_any_f:nw
..... [974](#), [22521](#), [22547](#), [23246](#), [24015](#)
- __fp_exp_after_array_f:w
..... [974](#), [22532](#),
[23900](#), [25006](#), [25017](#), [25027](#), [25035](#)
- __fp_exp_after_expr_mark_f:nw ..
..... [1008](#), [23244](#)
- __fp_exp_after_expr_stop_f:nw [22521](#)
- __fp_exp_after_f:nw
..... [970](#), [1008](#), [22408](#), [22526](#), [23948](#), [24086](#)
- __fp_exp_after_normal:nnw
..... [22411](#), [22421](#), [22438](#)
- __fp_exp_after_normal:Nwwww
..... [22440](#), [22448](#)
- __fp_exp_after_o:w .. [970](#), [22408](#),
[22638](#), [22642](#), [22644](#), [23114](#), [23158](#),
[23176](#), [24403](#), [24956](#), [24974](#), [24983](#),
[24992](#), [25079](#), [25816](#), [26935](#), [26940](#)
- __fp_exp_after_special:nnw
..... [971](#), [22413](#), [22423](#), [22428](#)
- __fp_exp_after_tuple_f:nw
..... [22532](#), [24210](#)
- __fp_exp_after_tuple_o:w
..... [22532](#), [24981](#), [24984](#), [24987](#), [24989](#)
- \c__fp_exp_intarray
..... [26636](#), [26722](#), [26729](#), [26732](#), [26734](#)
- __fp_exp_intarray:w [26693](#)
- __fp_exp_intarray_aux:w [26693](#)
- __fp_exp_large:NwN [1120](#), [26693](#), [26919](#)
- __fp_exp_large_after:wn [1120](#), [26693](#)
- __fp_exp_normal_o:w .. [26548](#), [26562](#)
- __fp_exp_o:w [26306](#), [26543](#)
- __fp_exp_overflow:NN [26562](#)
- __fp_exp_pos_large:NnnNwn
..... [26594](#), [26693](#)
- __fp_exp_pos_o:NNwnw
..... [26565](#), [26567](#), [26570](#)
- __fp_exp_pos_o:Nnnwn [26562](#)
- __fp_exp_Taylor:Nnnwn
..... [26590](#), [26609](#), [26739](#)
- __fp_exp_Taylor_break:Nw [26609](#)
- __fp_exp_Taylor_ii:ww . [26615](#), [26618](#)
- __fp_exp_Taylor_loop:ww [26609](#)
- __fp_expand:n [1164](#)
- __fp_exponent:w [22375](#)
- __fp_factorial_int_o:n [1128](#)
- __fp_fact_int_o:n [27000](#), [27003](#)
- __fp_fact_int_o:w [26997](#)
- __fp_fact_loop_o:w [27015](#), [27017](#)
- \c__fp_fact_max_arg_int [26978](#), [27005](#)
- __fp_fact_o:w [26310](#), [26979](#)
- __fp_fact_pos_o:w [26994](#), [26997](#)
- __fp_fact_small_o:w .. [27020](#), [27032](#)
- \c__fp_five_int [22908](#),
[22932](#), [22945](#), [22958](#), [22965](#), [23018](#)
- __fp_fixed<calculation>:wn . [1090](#)
- __fp_fixed_add:nnnnnnwn [25935](#)
- __fp_fixed_add:Nnnnnwn [25935](#)
- __fp_fixed_add:wn [1090](#),
[1093](#), [25935](#), [26175](#), [26485](#), [26493](#),
[26504](#), [26522](#), [27752](#), [27812](#), [28370](#)
- __fp_fixed_add_after:NNNNwn . [25935](#)
- __fp_fixed_add_one:wN [1091](#), [25867](#),
[26168](#), [26626](#), [26635](#), [27878](#), [28361](#)
- __fp_fixed_add_pack:NNNNwn . [25935](#)
- __fp_fixed_continue:wn
..... [25866](#), [26112](#),
[26117](#), [26127](#), [26704](#), [26894](#), [27229](#),
[27598](#), [27870](#), [27879](#), [28353](#), [28365](#)
- __fp_fixed_div_int:wn [25904](#)
- __fp_fixed_div_int:wwN
..... [1092](#), [25904](#), [26484](#), [26625](#), [27771](#)
- __fp_fixed_div_int_after:Nw
..... [1093](#), [25904](#)
- __fp_fixed_div_int_auxi:wn . [25904](#)
- __fp_fixed_div_int_auxii:wn
..... [1093](#), [25904](#)
- __fp_fixed_div_int_pack:Nw
..... [1093](#), [25904](#)
- __fp_fixed_div_myriad:wn
..... [25872](#), [26172](#)
- __fp_fixed_inv_to_float_o:wN
..... [26238](#), [26567](#), [26830](#)

- __fp_fixed_mul:nnnnnnnw [25955](#)
- __fp_fixed_mul:wnn [1090](#),
[1092](#), [1094](#), [1142](#), [1144](#), [25955](#),
[26121](#), [26152](#), [26167](#), [26169](#), [26173](#),
[26226](#), [26229](#), [26242](#), [26486](#), [26496](#),
[26536](#), [26627](#), [26725](#), [26740](#), [26840](#),
[27567](#), [27621](#), [27759](#), [27792](#), [27794](#)
- __fp_fixed_mul_add:nnnnwnnnn ...
..... [1097](#), [26024](#), [26026](#)
- __fp_fixed_mul_add:nnnnwnnW ...
..... [1098](#), [26031](#), [26037](#)
- __fp_fixed_mul_add:Nwnnnwnnn ...
... [1097](#), [25988](#), [25998](#), [26009](#), [26013](#)
- __fp_fixed_mul_add:wnwn
..... [1095](#), [25982](#), [28375](#)
- __fp_fixed_mul_after:wnn
..... [1095](#), [25874](#), [25880](#), [25883](#),
[25957](#), [25984](#), [25994](#), [26004](#), [26857](#)
- __fp_fixed_mul_one_minus_-
mul:wnn [25982](#)
- __fp_fixed_mul_short:wnn
..... [1092](#), [25881](#),
[26150](#), [26171](#), [26213](#), [26215](#), [27805](#)
- __fp_fixed_mul_sub_back:wnwn ...
..... [1095](#), [25982](#),
[26227](#), [27588](#), [27590](#), [27591](#), [27592](#),
[27593](#), [27594](#), [27595](#), [27596](#), [27597](#),
[27601](#), [27603](#), [27604](#), [27605](#), [27606](#),
[27607](#), [27608](#), [27609](#), [27634](#), [27636](#),
[27637](#), [27638](#), [27639](#), [27642](#), [27644](#),
[27645](#), [27646](#), [27647](#), [27772](#), [27780](#)
- __fp_fixed_one_minus_mul:wnn ...
..... [1095](#)–[1097](#), [26002](#)
- __fp_fixed_sub:wnn [25935](#), [26219](#),
[26502](#), [26518](#), [26530](#), [27233](#), [27753](#),
[27810](#), [27876](#), [28363](#), [28372](#), [28404](#)
- __fp_fixed_to_float_o:Nw
..... [26245](#), [26511](#)
- __fp_fixed_to_float_o:wN
..... [1091](#), [1106](#),
[1151](#), [26232](#), [26245](#), [26531](#), [26541](#),
[26565](#), [26826](#), [27800](#), [28303](#), [28409](#)
- __fp_fixed_to_float_pack:ww ...
..... [26278](#), [26288](#)
- __fp_fixed_to_float_rad_o:wN ...
..... [26240](#), [27800](#)
- __fp_fixed_to_float_round_-
up:wnnnnw [26291](#), [26295](#)
- __fp_fixed_to_float_zero:w ...
..... [26274](#), [26283](#)
- __fp_fixed_to_loop:N
..... [26251](#), [26261](#), [26265](#)
- __fp_fixed_to_loop_end:w
..... [26267](#), [26271](#)
- __fp_from_dim:wNnnnnnnn [28170](#)
- __fp_from_dim:wnnnnwNn [28197](#), [28198](#)
- __fp_from_dim:wnnnnwNw [28170](#)
- __fp_from_dim:wNw [28170](#)
- __fp_from_dim_test:ww
... [1163](#), [23336](#), [23373](#), [23967](#), [28170](#)
- __fp_func_to_name:N
..... [22699](#), [23846](#), [23855](#)
- __fp_func_to_name_aux:w [22699](#)
- \c__fp_half_prec_int
..... [22358](#), [23577](#), [23609](#)
- __fp_if_type_fp:NTwFw . [972](#), [1040](#),
[22388](#), [22467](#), [22475](#), [22482](#), [22498](#),
[22525](#), [24430](#), [24444](#), [24635](#), [24661](#),
[24662](#), [24829](#), [24830](#), [24831](#), [24997](#)
- __fp_inf_fp:N [22371](#), [22809](#)
- __fp_int:wTF [22645](#), [28314](#)
- __fp_int_eval:w
... [975](#), [990](#), [992](#), [1007](#), [1022](#), [1061](#),
[1069](#), [1073](#), [1077](#), [1106](#), [22325](#),
[22385](#), [22460](#), [22591](#), [22594](#), [22982](#),
[22986](#), [22998](#), [22999](#), [23035](#), [23126](#),
[23130](#), [23169](#), [23383](#), [23388](#), [23430](#),
[23519](#), [23530](#), [23579](#), [23610](#), [23616](#),
[23617](#), [23663](#), [23673](#), [23675](#), [23691](#),
[23693](#), [23716](#), [23718](#), [23881](#), [24101](#),
[24143](#), [24343](#), [24648](#), [25118](#), [25126](#),
[25147](#), [25149](#), [25170](#), [25172](#), [25181](#),
[25183](#), [25212](#), [25218](#), [25228](#), [25230](#),
[25304](#), [25306](#), [25322](#), [25324](#), [25328](#),
[25344](#), [25384](#), [25392](#), [25394](#), [25396](#),
[25398](#), [25401](#), [25404](#), [25406](#), [25425](#),
[25427](#), [25437](#), [25439](#), [25465](#), [25468](#),
[25476](#), [25478](#), [25499](#), [25502](#), [25505](#),
[25508](#), [25517](#), [25520](#), [25523](#), [25526](#),
[25533](#), [25535](#), [25541](#), [25549](#), [25551](#),
[25553](#), [25559](#), [25579](#), [25581](#), [25590](#),
[25592](#), [25613](#), [25634](#), [25638](#), [25650](#),
[25653](#), [25656](#), [25659](#), [25662](#), [25665](#),
[25668](#), [25671](#), [25675](#), [25687](#), [25691](#),
[25695](#), [25698](#), [25719](#), [25721](#), [25723](#),
[25733](#), [25772](#), [25774](#), [25783](#), [25870](#),
[25875](#), [25877](#), [25884](#), [25887](#), [25890](#),
[25893](#), [25896](#), [25899](#), [25908](#), [25920](#),
[25928](#), [25930](#), [25940](#), [25942](#), [25949](#),
[25958](#), [25960](#), [25963](#), [25966](#), [25969](#),
[25972](#), [25985](#), [25987](#), [25995](#), [25997](#),
[26005](#), [26007](#), [26017](#), [26020](#), [26023](#),
[26030](#), [26045](#), [26063](#), [26066](#), [26122](#),
[26136](#), [26138](#), [26144](#), [26157](#), [26159](#),
[26161](#), [26185](#), [26201](#), [26208](#), [26209](#),
[26232](#), [26249](#), [26253](#), [26298](#), [26300](#),
[26344](#), [26355](#), [26374](#), [26376](#), [26378](#),
[26391](#), [26404](#), [26409](#), [26411](#), [26417](#),

- 26434, 26435, 26436, 26437, 26438,
 26439, 26444, 26446, 26448, 26450,
 26452, 26457, 26459, 26461, 26463,
 26465, 26467, 26489, 26497, 26581,
 26630, 26707, 26715, 26723, 26729,
 26732, 26837, 26858, 26860, 26863,
 26866, 26869, 26872, 26888, 26914,
 26928, 26944, 27014, 27024, 27029,
 27181, 27213, 27222, 27454, 27468,
 27471, 27474, 27477, 27480, 27483,
 27486, 27489, 27492, 27508, 27518,
 27527, 27545, 27554, 27561, 27572,
 27582, 27615, 27625, 27650, 27659,
 27702, 27719, 27721, 27733, 27734,
 27775, 27786, 27797, 27855, 28007,
 28130, 28183, 28279, 28302, 28356,
 28408, 28430, 28432, 28434, 28439,
 28458, 28470, 28478, 28483, 28488
 _fp_int_eval_end:
 22325, 22385, 22463, 22582, 23035,
 23140, 23144, 24344, 24648, 25328,
 25363, 25555, 25930, 26066, 26888,
 26944, 27214, 27223, 27572, 27582,
 27625, 27650, 27734, 28437, 28439
 _fp_int_p:w 22645
 _fp_int_to_roman:w 22325,
 22594, 23591, 23623, 26371, 28540
 _fp_invalid_operation:nnw 983,
 984, 22751, 22833, 22845, 27988,
 27995, 28042, 28049, 28149, 28164
 _fp_invalid_operation_o:nw ...
 . 984, 22844, 23855, 25603, 25829,
 26323, 26992, 27001, 27088, 27103,
 27118, 27133, 27148, 27163, 27825,
 27843, 27859, 27887, 27900, 27916
 _fp_invalid_operation_o:Nww ...
 984, 22759, 22833,
 24056, 25098, 25370, 25371, 26929
 _fp_invalid_operation_o:nnw . 25854
 _fp_invalid_operation_tl_o:nn .
 984, 22768, 22833, 23101, 28334
 _fp_kind:w 22386, 23094, 24621
 \c_fp_leading_shift_int
 22548, 25875,
 25884, 25958, 26858, 27508, 27545
 _fp_ln_c:NwNw
 1114, 1115, 26468, 26499
 _fp_ln_div_after:Nw
 1112, 26370, 26419
 _fp_ln_div_i:w 26392, 26401
 _fp_ln_div_ii:wnn
 .. 26395, 26396, 26397, 26398, 26406
 _fp_ln_div_vi:wnn ... 26399, 26414
 _fp_ln_exponent:wn
 1115, 26346, 26508
 _fp_ln_exponent_one:ww 26513, 26527
 _fp_ln_exponent_small:NNww ...
 26516, 26520, 26533
 \c_fp_ln_i_fixed_tl 26311
 \c_fp_ln_ii_fixed_tl 26311
 \c_fp_ln_iii_fixed_tl 26311
 \c_fp_ln_iv_fixed_tl 26311
 \c_fp_ln_ix_fixed_tl 26311
 _fp_ln_npos_o:w
 1108, 1109, 26332, 26334
 _fp_ln_o:w . 1108, 1123, 26308, 26320
 _fp_ln_significand:NNNNnnN ...
 1110, 26345, 26348, 26838
 _fp_ln_square_t_after:w
 26443, 26475
 _fp_ln_square_t_pack:NNNNnw ...
 .. 26445, 26447, 26449, 26451, 26473
 _fp_ln_t_large:NNw
 1113, 26424, 26431, 26441
 _fp_ln_t_small:Nw ... 26422, 26429
 _fp_ln_t_small:w 1113
 _fp_ln_Taylor:wwNw
 1114, 26476, 26477
 _fp_ln_Taylor_break:w 26482, 26493
 _fp_ln_Taylor_loop:www
 26478, 26479, 26488
 _fp_ln_twice_t_after:w 26456, 26472
 _fp_ln_twice_t_pack:Nw . 26458,
 26460, 26462, 26464, 26466, 26471
 \c_fp_ln_vi_fixed_tl 26311
 \c_fp_ln_vii_fixed_tl 26311
 \c_fp_ln_viii_fixed_tl 26311
 \c_fp_ln_x_fixed_tl
 26311, 26530, 26537
 _fp_ln_x_ii:wnnnn ... 26350, 26368
 _fp_ln_x_iii:NNNNNNw . 26377, 26381
 _fp_ln_x_iii_var:NNNNnw
 26375, 26383
 _fp_ln_x_iv:wnnnnnnnn
 1112, 26373, 26388
 _fp_logb_aux_o:w 25786
 _fp_logb_o:w 25044, 25786
 \c_fp_max_exp_exponent_int
 22364, 26573
 \c_fp_max_exponent_int .. 22362,
 22368, 22396, 26083, 26285, 26893
 \c_fp_middle_shift_int
 22548, 25887,
 25890, 25893, 25896, 25960, 25963,
 25966, 25969, 26860, 26863, 26866,
 26869, 27511, 27518, 27548, 27554
 _fp_minmax_aux_o:Nw 24910

- __fp_minmax_auxi:ww [24932](#), [24944](#), [24951](#)
- __fp_minmax_auxii:ww [24934](#), [24942](#), [24951](#)
- __fp_minmax_break_o:w . [24925](#), [24955](#)
- __fp_minmax_loop:Nww [1053](#), [24919](#), [24921](#), [24927](#)
- __fp_minmax_o:Nw [1046](#), [24614](#), [24616](#), [24910](#)
- \c__fp_minus_min_exponent_int ... [22362](#), [22397](#)
- __fp_misused:n . [22338](#), [22342](#), [22453](#)
- __fp_mul_cases_o:NnNnw [1071](#), [25335](#), [25341](#), [25447](#)
- __fp_mul_cases_o:nNnnw [25341](#)
- __fp_mul_npos_o:Nw . [1068](#), [1069](#), [1071](#), [1162](#), [1163](#), [25338](#), [25379](#), [28200](#)
- __fp_mul_significand_drop:NNNNw [1069](#), [25388](#)
- __fp_mul_significand_keep:NNNNw [25388](#)
- __fp_mul_significand_large_
f:NwNNNN [25418](#), [25422](#)
- __fp_mul_significand_o:nnnnNnnnn [1069](#), [25386](#), [25388](#)
- __fp_mul_significand_small_
f:NNwwwN [25416](#), [25433](#)
- __fp_mul_significand_test_f:NNN [1070](#), [25390](#), [25413](#)
- \c__fp_myriad_int [22361](#), [25870](#), [25901](#), [25902](#), [25979](#), [26040](#)
- __fp_neg_sign:N [1059](#), [22384](#), [25052](#), [25205](#)
- __fp_not_o:w [1046](#), [23874](#), [24957](#)
- \c__fp_one_fixed_tl [25864](#), [26484](#), [26697](#), [26894](#), [26921](#), [27704](#), [27771](#), [27876](#), [28353](#), [28363](#), [28404](#)
- __fp_overflow:w [970](#), [984](#), [986](#), [22399](#), [22833](#), [26575](#), [27008](#)
- \c__fp_overflowing_fp [22365](#), [27989](#), [28043](#)
- __fp_pack:NNNNw . [22548](#), [25876](#), [25886](#), [25889](#), [25892](#), [25895](#), [25898](#), [25959](#), [25962](#), [25965](#), [25968](#), [25971](#), [26859](#), [26862](#), [26865](#), [26868](#), [26871](#)
- __fp_pack_big:NNNNNw ... [22552](#), [25652](#), [25655](#), [25658](#), [25661](#), [25664](#), [25667](#), [25670](#), [25674](#), [25986](#), [25996](#), [26006](#), [26016](#), [26019](#), [26022](#), [26029](#)
- __fp_pack_Bigg:NNNNNw [22557](#), [25501](#), [25504](#), [25507](#), [25519](#), [25522](#), [25525](#)
- __fp_pack_eight:wNNNNNNNN [976](#), [1065](#), [22564](#), [25314](#), [25623](#), [26051](#), [27200](#), [27201](#)
- __fp_pack_twice_four:wNNNNNNNN [976](#), [22562](#), [23151](#), [23152](#), [25256](#), [25257](#), [26052](#), [26053](#), [26054](#), [26086](#), [26087](#), [26088](#), [26276](#), [26277](#), [26612](#), [26613](#), [26614](#), [27202](#), [27203](#), [27497](#), [27498](#), [27499](#), [27500](#), [28193](#)
- __fp_parse:n [998](#), [1010](#), [1022](#), [1030](#), [1043](#), [1044](#), [1051](#), [1164](#), [1176](#), [23182](#), [23333](#), [23991](#), [24543](#), [24545](#), [24547](#), [24570](#), [24621](#), [24630](#), [24647](#), [24657](#), [24814](#), [24874](#), [25799](#), [27964](#), [28018](#), [28096](#), [28141](#), [28156](#), [28209](#), [28211](#), [28213](#), [28215](#), [28592](#)
- __fp_parse_after:ww [23991](#)
- __fp_parse_apply_binary:NwNwN . . [1002](#), [1003](#), [1006](#), [1034](#), [24029](#), [24220](#)
- __fp_parse_apply_binary_chk:NN [24029](#), [24060](#), [24073](#)
- __fp_parse_apply_binary_
error:NNN [24029](#)
- __fp_parse_apply_comma:NwNwN ... [1034](#), [24179](#)
- __fp_parse_apply_compare:NwNNNNNwN [24367](#), [24376](#)
- __fp_parse_apply_compare_
aux:NNwN [24388](#), [24391](#), [24396](#)
- __fp_parse_apply_function:NNNwN [1025](#), [23823](#), [23984](#)
- __fp_parse_apply_unary:NNNwN ... [23828](#), [23860](#), [23975](#)
- __fp_parse_apply_unary_chk:nNNNNw [23839](#), [23840](#), [23843](#)
- __fp_parse_apply_unary_chk:nNNNw [23828](#)
- __fp_parse_apply_unary_chk:NwNw [23828](#)
- __fp_parse_apply_unary_error:NNw [23828](#), [25836](#)
- __fp_parse_apply_unary_type:NNN [23828](#)
- __fp_parse_caseless_inf:N ... [23941](#)
- __fp_parse_caseless_infinity:N [23941](#)
- __fp_parse_caseless_nan:N ... [23941](#)
- __fp_parse_compare:NNNNNNN . [24308](#)
- __fp_parse_compare_auxi:NNNNNNN [24308](#)
- __fp_parse_compare_auxii:NNNNN [24308](#)
- __fp_parse_compare_end:NNNNw . [24308](#)

- _fp_parse_continue:NwN
 . [1002](#), [1003](#), [1030](#), [24018](#), [24031](#),
 [24207](#), [24406](#), [25014](#), [25024](#), [25032](#)
- _fp_parse_continue_compare:NNwNN
 [24399](#), [24414](#)
- _fp_parse_digits_:N [23200](#)
- _fp_parse_digits_i:N [23200](#)
- _fp_parse_digits_ii:N [23200](#)
- _fp_parse_digits_iii:N [23200](#)
- _fp_parse_digits_iv:N [23200](#)
- _fp_parse_digits_v:N [23200](#)
- _fp_parse_digits_vi:N
 [23200](#), [23535](#), [23583](#)
- _fp_parse_digits_vii:N
 [1015](#), [23200](#), [23522](#), [23572](#)
- _fp_parse_excl_error: [24308](#)
- _fp_parse_expand:w
 [1006](#), [1007](#), [23197](#), [23199](#),
 [23209](#), [23249](#), [23309](#), [23353](#), [23362](#),
 [23365](#), [23369](#), [23406](#), [23440](#), [23478](#),
 [23480](#), [23499](#), [23501](#), [23523](#), [23540](#),
 [23553](#), [23573](#), [23603](#), [23631](#), [23647](#),
 [23658](#), [23681](#), [23710](#), [23720](#), [23727](#),
 [23741](#), [23757](#), [23777](#), [23788](#), [23870](#),
 [23893](#), [23905](#), [23980](#), [23989](#), [23997](#),
 [24010](#), [24128](#), [24174](#), [24198](#), [24224](#),
 [24272](#), [24292](#), [24361](#), [24374](#), [25010](#)
- _fp_parse_exponent:N [1020](#),
 [23308](#), [23514](#), [23663](#), [23730](#), [23732](#)
- _fp_parse_exponent:Nw
 [23538](#), [23551](#),
 [23600](#), [23628](#), [23679](#), [23708](#), [23727](#)
- _fp_parse_exponent_aux:NN .. [23732](#)
- _fp_parse_exponent_body:N
 [23759](#), [23763](#)
- _fp_parse_exponent_digits:N ...
 [23767](#), [23779](#)
- _fp_parse_exponent_keep:N .. [23790](#)
- _fp_parse_exponent_keep:NTF ...
 [23770](#), [23790](#)
- _fp_parse_exponent_sign:N
 [23749](#), [23753](#)
- _fp_parse_function:NNN
 [22894](#), [22896](#), [22898](#),
 [22901](#), [23973](#), [24614](#), [24616](#), [27071](#),
 [27073](#), [27075](#), [27077](#), [28238](#), [28240](#)
- _fp_parse_function_all_fp_-
 o:nnw [23028](#), [24416](#), [24912](#)
- _fp_parse_function_one_two:nnw
 ... [1148](#), [24428](#), [27665](#), [27671](#), [28307](#)
- _fp_parse_function_one_two_-
 aux:nnw [24428](#)
- _fp_parse_function_one_two_-
 auxii:nnw [24428](#)
- _fp_parse_function_one_two_-
 error_o:w [24428](#)
- _fp_parse_infix:NN
 [1008](#), [1012](#), [1028](#),
 [1033](#), [23248](#), [23418](#), [23457](#), [23933](#),
 [23948](#), [23970](#), [24086](#), [24089](#), [24172](#)
- _fp_parse_infix_!:N [24308](#)
- _fp_parse_infix_&:Nw [24265](#)
- _fp_parse_infix(:N [24248](#)
- _fp_parse_infix):N [24162](#)
- _fp_parse_infix*:N [24250](#)
- _fp_parse_infix+:N
 [1006](#), [23197](#), [24214](#)
- _fp_parse_infix_,:N [24179](#)
- _fp_parse_infix_=:N [24214](#)
- _fp_parse_infix_/:N [24214](#)
- _fp_parse_infix_:N .. [24282](#), [24995](#)
- _fp_parse_infix_<:N [24308](#)
- _fp_parse_infix_=:N [24308](#)
- _fp_parse_infix_>:N [24308](#)
- _fp_parse_infix_?:N [24282](#)
- _fp_parse_infix_<operation>:N [1006](#)
- _fp_parse_infix_~:N [24214](#)
- _fp_parse_infix_after_operand:NwN
 ... [1012](#), [23301](#), [23379](#), [23877](#), [24084](#)
- _fp_parse_infix_after_paren:NN
 [23902](#), [23928](#), [24131](#)
- _fp_parse_infix_and:N [24214](#), [24281](#)
- _fp_parse_infix_check:NNN
 [24107](#), [24117](#), [24149](#)
- _fp_parse_infix_comma:w [1034](#), [24179](#)
- _fp_parse_infix_end:N
 [1030](#), [1034](#), [23998](#), [24003](#), [24011](#), [24160](#)
- _fp_parse_infix_juxt:N
 [1033](#), [24097](#), [24105](#), [24214](#)
- _fp_parse_infix_mark:NNN
 [24094](#), [24136](#), [24159](#)
- _fp_parse_infix_mul:N
 [1033](#), [1036](#), [24122](#),
 [24139](#), [24147](#), [24214](#), [24249](#), [24258](#)
- _fp_parse_infix_or:N .. [24214](#), [24280](#)
- _fp_parse_infix_|:Nw [24265](#)
- _fp_parse_large:N [1014](#), [23485](#), [23568](#)
- _fp_parse_large_leading:wwNN ..
 [1018](#), [23570](#), [23575](#)
- _fp_parse_large_round:NN
 [1018](#), [23611](#), [23683](#)
- _fp_parse_large_round_aux:wNN ..
 [23683](#)
- _fp_parse_large_round_test:NN ..
 [23683](#)
- _fp_parse_large_trailing:wwNN ..
 [1018](#), [23581](#), [23605](#)

__fp_parse_letters:N
 [1012](#), [23394](#), [23408](#)
 __fp_parse_lparen_after:NwN . [23883](#)
 __fp_parse_o:n
 [998](#), [23991](#), [24812](#), [24813](#)
 __fp_parse_one:Nw
 ... [1001–1006](#), [1013](#), [1028](#), [1030](#),
 [23197](#), [23220](#), [23462](#), [23822](#), [24024](#)
 __fp_parse_one_digit:NN
 [1026](#), [23236](#), [23377](#)
 __fp_parse_one_fp:NN
 [1008](#), [23228](#), [23244](#)
 __fp_parse_one_other:NN [23239](#), [23385](#)
 __fp_parse_one_register:NN
 [23231](#), [23299](#)
 __fp_parse_one_register_aux:Nw .
 [23299](#)
 __fp_parse_one_register_-
 auxii:wwwNw [23299](#)
 __fp_parse_one_register_dim:ww .
 [23299](#)
 __fp_parse_one_register_int:www
 [23299](#)
 __fp_parse_one_register_-
 math:NNw [23340](#)
 __fp_parse_one_register_mu:www .
 [23299](#)
 __fp_parse_one_register_-
 special:N [23304](#), [23340](#)
 __fp_parse_one_register_wd:Nw [23340](#)
 __fp_parse_one_register_wd:w . [23340](#)
 __fp_parse_operand:Nw
 ... [1001–1004](#), [1006](#), [1030](#), [1034](#),
 [23197](#), [23866](#), [23868](#), [23889](#), [23891](#),
 [23980](#), [23989](#), [23996](#), [24009](#), [24018](#),
 [24197](#), [24223](#), [24291](#), [24374](#), [25009](#)
 __fp_parse_pack_carry:w [1017](#), [23555](#)
 __fp_parse_pack_leading:NNNNnw
 [23518](#), [23555](#), [23578](#)
 __fp_parse_pack_trailing:NNNNnw
 .. [23528](#), [23555](#), [23597](#), [23608](#), [23615](#)
 __fp_parse_prefix:NNN . [23397](#), [23442](#)
 __fp_parse_prefix!:Nw [23856](#)
 __fp_parse_prefix(:Nw [23883](#)
 __fp_parse_prefix):Nw [23915](#)
 __fp_parse_prefix+:Nw [23822](#)
 __fp_parse_prefix-:Nw [23856](#)
 __fp_parse_prefix.:Nw [23875](#)
 __fp_parse_prefix_unknown:NNN [23442](#)
 __fp_parse_return_semicolon:w ..
 [23198](#), [23207](#), [23438](#),
 [23645](#), [23656](#), [23739](#), [23771](#), [23786](#)
 __fp_parse_round:Nw [22899](#)
 __fp_parse_round_after:wN
 [1020](#), [23660](#), [23665](#), [23715](#)
 __fp_parse_round_loop:N
 [1020](#), [1021](#), [23633](#), [23676](#), [23694](#), [23719](#)
 __fp_parse_round_up:N [23633](#)
 __fp_parse_small:N [1015](#), [23505](#), [23516](#)
 __fp_parse_small_leading:wwNN ..
 [1016](#), [23520](#), [23525](#), [23587](#)
 __fp_parse_small_round:NN
 [23547](#), [23665](#), [23704](#)
 __fp_parse_small_trailing:wwNN .
 [1016](#), [23533](#), [23542](#), [23619](#)
 __fp_parse_strim_end:w [23491](#)
 __fp_parse_strim_zeros:N
 [1014](#), [1026](#), [23472](#), [23491](#), [23881](#)
 __fp_parse_trim_end:w [23465](#)
 __fp_parse_trim_zeros:N [23383](#), [23465](#)
 __fp_parse_unary_function:NNN ..
 [23973](#), [25042](#), [25044](#), [25046](#), [25048](#),
 [26306](#), [26308](#), [26310](#), [27059](#), [27065](#)
 __fp_parse_word:Nw [1012](#), [23391](#), [23408](#)
 __fp_parse_word_abs:N [25041](#)
 __fp_parse_word_acos:N [27051](#)
 __fp_parse_word_acosd:N [27051](#)
 __fp_parse_word_acot:N [27070](#)
 __fp_parse_word_acotd:N [27070](#)
 __fp_parse_word_acsc:N [27051](#)
 __fp_parse_word_acscd:N [27051](#)
 __fp_parse_word_asec:N [27051](#)
 __fp_parse_word_asecd:N [27051](#)
 __fp_parse_word_asin:N [27051](#)
 __fp_parse_word_asind:N [27051](#)
 __fp_parse_word_atan:N [27070](#)
 __fp_parse_word_atand:N [27070](#)
 __fp_parse_word_bp:N [23944](#)
 __fp_parse_word_cc:N [23944](#)
 __fp_parse_word_ceil:N [22893](#)
 __fp_parse_word_cm:N [23944](#)
 __fp_parse_word_cos:N [27051](#)
 __fp_parse_word_cosd:N [27051](#)
 __fp_parse_word_cot:N [27051](#)
 __fp_parse_word_cotd:N [27051](#)
 __fp_parse_word_csc:N [27051](#)
 __fp_parse_word_cscd:N [27051](#)
 __fp_parse_word_dd:N [23944](#)
 __fp_parse_word_deg:N [23930](#)
 __fp_parse_word_em:N [23963](#)
 __fp_parse_word_ex:N [23963](#)
 __fp_parse_word_exp:N [26305](#)
 __fp_parse_word_fact:N [26305](#)
 __fp_parse_word_false:N [23930](#)
 __fp_parse_word_floor:N [22893](#)
 __fp_parse_word_in:N [23944](#)

- __fp_parse_word_inf:N [23930](#), [23941](#), [23942](#)
- __fp_parse_word_ln:N [26305](#)
- __fp_parse_word_logb:N [25041](#)
- __fp_parse_word_max:N [24613](#)
- __fp_parse_word_min:N [24613](#)
- __fp_parse_word_mm:N [23944](#)
- __fp_parse_word_nan:N . [23930](#), [23943](#)
- __fp_parse_word_nc:N [23944](#)
- __fp_parse_word_nd:N [23944](#)
- __fp_parse_word_pc:N [23944](#)
- __fp_parse_word_pi:N [23930](#)
- __fp_parse_word_pt:N [23944](#)
- __fp_parse_word_rand:N [28237](#)
- __fp_parse_word_randint:N ... [28237](#)
- __fp_parse_word_round:N [22899](#)
- __fp_parse_word_sec:N [27051](#)
- __fp_parse_word_secd:N [27051](#)
- __fp_parse_word_sign:N [25041](#)
- __fp_parse_word_sin:N [27051](#)
- __fp_parse_word_sind:N [27051](#)
- __fp_parse_word_sp:N [23944](#)
- __fp_parse_word_sqrt:N [25041](#)
- __fp_parse_word_tan:N [27051](#)
- __fp_parse_word_tand:N [27051](#)
- __fp_parse_word_true:N [23930](#)
- __fp_parse_word_trunc:N [22893](#)
- __fp_parse_zero: [1014](#), [23487](#), [23507](#), [23511](#)
- __fp_pow_B:wwN [26841](#), [26876](#)
- __fp_pow_C_neg:w [26879](#), [26896](#)
- __fp_pow_C_overflow:w [26884](#), [26891](#), [26912](#)
- __fp_pow_C_pack:w [26898](#), [26906](#), [26917](#)
- __fp_pow_C_pos:w [26882](#), [26901](#)
- __fp_pow_C_pos_loop:wN [26902](#), [26903](#), [26910](#)
- __fp_pow_exponent:Nwnnnnw ... [26847](#), [26850](#), [26855](#)
- __fp_pow_exponent:wnN . [26839](#), [26844](#)
- __fp_pow_neg:www . [1126](#), [26753](#), [26923](#)
- __fp_pow_neg_aux:wNN .. [1125](#), [26923](#)
- __fp_pow_neg_case:w .. [26925](#), [26946](#)
- __fp_pow_neg_case_aux:nnnnn . [26946](#)
- __fp_pow_neg_case_aux:Nnnw [1126](#), [26946](#)
- __fp_pow_normal_o:ww [1121](#), [26758](#), [26790](#)
- __fp_pow_npos_aux:NNnw [26824](#), [26828](#), [26834](#)
- __fp_pow_npos_o:Nww [1123](#), [26801](#), [26818](#)
- __fp_pow_zero_or_inf:ww [1121](#), [26760](#), [26767](#)
- \c__fp_prec_and_int ... [23182](#), [24245](#)
- \c__fp_prec_colon_int [23182](#), [24303](#), [25009](#)
- \c__fp_prec_comma_int [1027](#), [23182](#), [23256](#), [23889](#), [23917](#), [24183](#), [24188](#), [24197](#)
- \c__fp_prec_comp_int [23182](#), [24331](#), [24374](#)
- \c__fp_prec_end_int .. [1030](#), [1034](#), [23182](#), [23258](#), [23996](#), [24009](#), [24166](#)
- \c__fp_prec_func_int [1027](#), [23182](#), [23888](#), [23980](#), [23989](#)
- \c__fp_prec_hat_int ... [23182](#), [24233](#)
- \c__fp_prec_hatii_int . [23182](#), [24233](#)
- \c__fp_prec_int [22358](#), [22591](#), [22652](#), [22679](#), [23120](#), [26593](#), [26958](#), [26961](#), [28061](#), [28063](#), [28069](#), [28120](#), [28318](#), [28357](#), [28408](#)
- \c__fp_prec_juxt_int .. [23182](#), [24235](#)
- \c__fp_prec_not_int [1026](#), [23182](#), [23873](#), [23874](#)
- \c__fp_prec_or_int [23182](#), [24247](#)
- \c__fp_prec_plus_int [1001](#), [23182](#), [24241](#), [24243](#)
- \c__fp_prec_quest_int [23182](#), [24286](#), [24301](#)
- \c__fp_prec_times_int [23182](#), [24237](#), [24239](#)
- \c__fp_prec_tuple_int [1027](#), [23182](#), [23257](#), [23891](#), [23919](#)
- __fp_rand_myriads:n [1169](#), [1170](#), [28273](#), [28290](#), [28376](#)
- __fp_rand_myriads_get:w [28273](#)
- __fp_rand_myriads_loop:w [28273](#)
- __fp_rand_o:Nw [28238](#), [28245](#), [28251](#), [28284](#)
- __fp_rand_o:w [28284](#)
- __fp_randinat_wide_aux:w [28446](#)
- __fp_randinat_wide_auxii:w .. [28446](#)
- __fp_randint:n [28508](#)
- __fp_randint:ww [28414](#), [28518](#)
- __fp_randint_auxi_o:ww [28305](#)
- __fp_randint_auxii:wn [28305](#)
- __fp_randint_auxiii_o:ww [28305](#)
- __fp_randint_auxiv_o:ww [28305](#)
- __fp_randint_auxv_o:w [28305](#)
- __fp_randint_badarg:w .. [1170](#), [28305](#)
- __fp_randint_default:w [28305](#)
- __fp_randint_o:Nw [28240](#), [28251](#), [28305](#)
- __fp_randint_o:w [28305](#)
- __fp_randint_split_aux:w [28446](#)
- __fp_randint_split_o:Nw [1173](#), [28446](#)
- __fp_randint_wide_aux:w [1173](#), [28449](#), [28480](#)

__fp_randint_wide_auxii:w
 28482, 28491
 __fp_reverse_args:Nww
 1154, 1155, 22334,
 27651, 27726, 27839, 27905, 28402
 __fp_round:NNN .. 990, 992, 1070,
 1086, 22909, 22979, 25173, 25184,
 25428, 25440, 25582, 25593, 25777
 __fp_round:Nwn . 23037, 23090, 28168
 __fp_round:Nww . 23038, 23059, 23090
 __fp_round:Nwww 23039, 23053
 __fp_round_aux_o:Nw 23026
 __fp_round_digit:Nw
 978, 992, 1069,
 1070, 1086, 22609, 22993, 25187,
 25330, 25431, 25443, 25596, 25782
 __fp_round_name_from_cs:N
 .. 23029, 23049, 23075, 23079, 23102
 __fp_round_neg:NNN 990,
 993, 1066, 23004, 25292, 25307, 25325
 __fp_round_no_arg_o:Nw 23036, 23043
 __fp_round_normal:NnnwNnn .. 23090
 __fp_round_normal:NNwNnn 23090
 __fp_round_normal:NwNNnw 23090
 __fp_round_normal_end:wwNnn . 23090
 __fp_round_o:Nw
 .. 22894, 22896, 22898, 22902, 23026
 __fp_round_pack:Nw 23090
 __fp_round_return_one:
 990, 22909, 22915,
 22925, 22933, 22937, 22946, 22950,
 22959, 22966, 22970, 23008, 23019
 __fp_round_s:NNNw
 . 990, 992, 1020, 22977, 23669, 23687
 __fp_round_special:NwwNnn ... 23090
 __fp_round_special_aux:Nw ... 23090
 __fp_round_to_nearest:NNN
 993, 994, 22902, 22905,
 22909, 23013, 23045, 23055, 28168
 __fp_round_to_nearest_neg:NNN 23004
 __fp_round_to_nearest_ninf:NNN .
 994, 22909, 23024
 __fp_round_to_nearest_ninf_-
 neg:NNN 23004
 __fp_round_to_nearest_pinf:NNN .
 994, 22909, 23015
 __fp_round_to_nearest_pinf_-
 neg:NNN 23004
 __fp_round_to_nearest_zero:NNN .
 994, 22909
 __fp_round_to_nearest_zero_-
 neg:NNN 23004
 __fp_round_to_ninf:NNN
 22896, 22909, 23012, 23083
 __fp_round_to_ninf_neg:NNN .. 23004
 __fp_round_to_pinf:NNN
 22898, 22909, 23004, 23085
 __fp_round_to_pinf_neg:NNN .. 23004
 __fp_round_to_zero:NNN
 22894, 22909, 23081
 __fp_round_to_zero_neg:NNN .. 23004
 __fp_rrrot:www 22335, 27772
 __fp_sanitize:Nw
 1061, 1064, 1069, 1072, 1080, 1128,
 1144, 1151, 1170, 22393, 23159,
 23177, 25116, 25210, 25382, 25463,
 25611, 26336, 26579, 26820, 27012,
 27613, 27657, 27784, 28300, 28395
 __fp_sanitize:wN
 1011, 1015, 22393, 23382, 23880
 __fp_sanitize_zero:w 22393
 __fp_sec_o:w 27123
 __fp_set_sign_o:w
 .. 23873, 25042, 25813, 25814, 25835
 __fp_show:NN 24575
 __fp_show_validate:w 24575
 __fp_sign_aux_o:w 25802
 __fp_sign_o:w 25046, 25802
 __fp_sin_o:w . 982, 1025, 1153, 27078
 __fp_sin_series_aux_o:NNwww . 27565
 __fp_sin_series_o:NNwww
 1131, 1145,
 27084, 27099, 27114, 27129, 27565
 __fp_small_int:wTF
 1127, 22661, 23092, 26999
 __fp_small_int_normal:NnwTF . 22661
 __fp_small_int_test:NnnwNwTF . 22661
 __fp_small_int_test:NnnwNw
 22680, 22683
 __fp_small_int_true:wTF 22661
 __fp_sqrt_auxi_o:NNNNwnnN
 25633, 25641
 __fp_sqrt_auxii_o:NnnnnnnnN ...
 1082, 1084, 25643, 25647, 25727, 25739
 __fp_sqrt_auxiii_o:wnnnnnnnn ...
 25644, 25682, 25728
 __fp_sqrt_auxiv_o:NNNNNw 25682
 __fp_sqrt_auxix_o:wnwnw 25716
 __fp_sqrt_auxv_o:NNNNNw 25682
 __fp_sqrt_auxvi_o:NNNNNw 25682
 __fp_sqrt_auxvii_o:NNNNNw ... 25682
 __fp_sqrt_auxviii_o:nnnnnnn ...
 .. 25704, 25706, 25708, 25714, 25716
 __fp_sqrt_auxx_o:Nnnnnnnn
 25712, 25730
 __fp_sqrt_auxxi_o:wnnnN 25730
 __fp_sqrt_auxxii_o:nnnnnnnnw ...
 25740, 25744

__fp_sqrt_auxxiii_o:w [25744](#)
 __fp_sqrt_auxxiv_o:wnnnnnnN ...
 [25756](#), [25759](#), [25767](#), [25769](#)
 __fp_sqrt_Newton_o:wwn
 [1081](#), [25618](#), [25629](#), [25630](#)
 __fp_sqrt_npos_auxi_o:wnnnN . [25609](#)
 __fp_sqrt_npos_auxii_o:wNNNNNNNN
 [25609](#)
 __fp_sqrt_npos_o:w ... [25606](#), [25609](#)
 __fp_sqrt_o:w [25048](#), [25599](#)
 __fp_step:NNnnnn [24879](#)
 __fp_step:NnnnnN [24809](#)
 __fp_step:wwwN [24809](#)
 __fp_step_fp:wwwN [24809](#)
 __fp_str_if_eq:nn [22698](#),
 [23794](#), [23806](#), [24092](#), [24134](#), [26793](#)
 __fp_sub_back_far_o:NnnwnnnN ..
 [1065](#), [25219](#), [25265](#)
 __fp_sub_back_near_after:wNNNNw
 [25225](#), [25303](#)
 __fp_sub_back_near_o:nnnnnnnnN .
 [1064](#), [25215](#), [25225](#)
 __fp_sub_back_near_pack:NNNNNNw
 [25225](#), [25305](#)
 __fp_sub_back_not_far_o:wwwNN .
 [25280](#), [25300](#)
 __fp_sub_back_quite_far_ii:NN [25284](#)
 __fp_sub_back_quite_far_o:wwNN .
 [25278](#), [25284](#)
 __fp_sub_back_shift:wnnnn
 [1065](#), [25237](#), [25241](#)
 __fp_sub_back_shift_ii:ww ... [25241](#)
 __fp_sub_back_shift_iii:NNNNNNNNw
 [25241](#)
 __fp_sub_back_shift_iv:nnnw . [25241](#)
 __fp_sub_back_very_far_ii_-
 o:nnNwNN [25312](#)
 __fp_sub_back_very_far_o:wwwNN
 [25279](#), [25312](#)
 __fp_sub_eq_o:Nnnnw [25190](#)
 __fp_sub_npos_i_o:Nnnnw
 [1063](#), [25195](#), [25204](#), [25208](#)
 __fp_sub_npos_ii_o:Nnnnw [25190](#)
 __fp_sub_npos_o:NnnNnw
 [1063](#), [25110](#), [25190](#)
 __fp_tan_o:w [27138](#)
 __fp_tan_series_aux_o:Nnnww . [27619](#)
 __fp_tan_series_o:NNwww
 [1133](#), [27145](#), [27160](#), [27619](#)
 __fp_ternary:NwN [1046](#), [24301](#), [24993](#)
 __fp_ternary_auxi:NwN
 [1046](#), [1056](#), [24993](#)
 __fp_ternary_auxii:NwN
 [1046](#), [1056](#), [24303](#), [24993](#)
 __fp_tmp:w [978](#), [1035](#),
 [22603](#), [22613](#), [22614](#), [22615](#), [22616](#),
 [22617](#), [22618](#), [22619](#), [22620](#), [22621](#),
 [22622](#), [22623](#), [22624](#), [22625](#), [22626](#),
 [22627](#), [22628](#), [22704](#), [22706](#), [23200](#),
 [23212](#), [23213](#), [23214](#), [23215](#), [23216](#),
 [23217](#), [23218](#), [23276](#), [23298](#), [23856](#),
 [23873](#), [23874](#), [23930](#), [23935](#), [23936](#),
 [23937](#), [23938](#), [23939](#), [23940](#), [23944](#),
 [23952](#), [23953](#), [23954](#), [23955](#), [23956](#),
 [23957](#), [23958](#), [23959](#), [23960](#), [23961](#),
 [23962](#), [24162](#), [24178](#), [24179](#), [24202](#),
 [24214](#), [24232](#), [24234](#), [24236](#), [24238](#),
 [24240](#), [24242](#), [24244](#), [24246](#), [24250](#),
 [24264](#), [24265](#), [24280](#), [24281](#), [24282](#),
 [24300](#), [24302](#), [25845](#), [25859](#), [25860](#)
 __fp_to_decimal:w
 .. [28023](#), [28033](#), [28150](#), [28167](#), [28654](#)
 __fp_to_decimal_dispatch:w [1158](#),
 [1161](#), [1162](#), [24872](#), [28013](#), [28017](#), [28020](#)
 __fp_to_decimal_huge:wnnnn .. [28033](#)
 __fp_to_decimal_large:Nnnw .. [28033](#)
 __fp_to_decimal_normal:wnnnn ..
 [28033](#), [28121](#)
 __fp_to_decimal_recover:w ... [28020](#)
 __fp_to_dim:w [28135](#)
 __fp_to_dim_dispatch:w . [1161](#), [28135](#)
 __fp_to_dim_recover:w [28135](#)
 __fp_to_int:w ... [1162](#), [28160](#), [28165](#)
 __fp_to_int_dispatch:w [28151](#)
 __fp_to_int_recover:w [28151](#)
 __fp_to_scientific:w
 [1159](#), [27969](#), [27979](#)
 __fp_to_scientific_dispatch:w ..
 [1157](#), [1161](#), [27959](#), [27963](#), [27966](#)
 __fp_to_scientific_normal:wnnnn
 [27979](#)
 __fp_to_scientific_normal:wNw [27979](#)
 __fp_to_scientific_recover:w . [27966](#)
 __fp_to_tl:w ... [28099](#), [28107](#), [28662](#)
 __fp_to_tl_dispatch:w
 [1156](#), [1160](#), [28091](#), [28095](#), [28098](#), [28231](#)
 __fp_to_tl_normal:nnnn [28107](#)
 __fp_to_tl_recover:w [28098](#)
 __fp_to_tl_scientific:wnnnn . [28107](#)
 __fp_to_tl_scientific:wNw ... [28107](#)
 \c__fp_trailing_shift_int
 [22548](#), [25877](#),
 [25899](#), [25972](#), [26872](#), [27511](#), [27548](#)
 __fp_trap_division_by_zero_-
 set:N [22776](#)
 __fp_trap_division_by_zero_set_-
 error: [22776](#)


```

\__fp_trap_division_by_zero_set_-
  flag: ..... 22776
\__fp_trap_division_by_zero_set_-
  none: ..... 22776
\__fp_trap_invalid_operation_-
  set:N ..... 22742
\__fp_trap_invalid_operation_-
  set_error: ..... 22742
\__fp_trap_invalid_operation_-
  set_flag: ..... 22742
\__fp_trap_invalid_operation_-
  set_none: ..... 22742
\__fp_trap_overflow_set:N .... 22802
\__fp_trap_overflow_set:NnNn . 22802
\__fp_trap_overflow_set_error: 22802
\__fp_trap_overflow_set_flag: . 22802
\__fp_trap_overflow_set_none: . 22802
\__fp_trap_underflow_set:N ... 22802
\__fp_trap_underflow_set_error: .
  ..... 22802
\__fp_trap_underflow_set_flag: 22802
\__fp_trap_underflow_set_none: 22802
\__fp_trig:NNNNwn . 27084, 27099,
  27114, 27129, 27144, 27159, 27176
\c__fp_trig_intarray ..... 1141,
  27237, 27467, 27470, 27473, 27476,
  27479, 27482, 27485, 27488, 27491
\__fp_trig_large:ww ... 27184, 27451
\__fp_trig_large_auxi:w ..... 27451
\__fp_trig_large_auxii:w 1141, 27451
\__fp_trig_large_auxiii:w 1141, 27451
\__fp_trig_large_auxix:Nw .... 27524
\__fp_trig_large_auxv:www .....
  ..... 27501, 27504
\__fp_trig_large_auxvi:wnnnnnnnn
  ..... 27504
\__fp_trig_large_auxvii:w .....
  ..... 27507, 27524
\__fp_trig_large_auxviii:w ... 27524
\__fp_trig_large_auxviii:ww ....
  ..... 27526, 27530
\__fp_trig_large_auxx:wnnnnn . 27524
\__fp_trig_large_auxxi:w ..... 27524
\__fp_trig_large_pack:NNNNw ...
  ..... 27504, 27553
\__fp_trig_small:ww .....
  1135, 1143, 27186, 27190, 27196, 27563
\__fp_trigd_large:ww .. 27184, 27198
\__fp_trigd_large_auxi:nnnnwnnnn
  ..... 27198
\__fp_trigd_large_auxii:wNw .. 27198
\__fp_trigd_large_auxiii:www . 27198
\__fp_trigd_small:ww .....
  ..... 1135, 27186, 27192, 27235

\__fp_trim_zeros:w .....
  ..... 27950, 28074, 28083, 28134
\__fp_trim_zeros_dot:w ..... 27950
\__fp_trim_zeros_end:w ..... 27950
\__fp_trim_zeros_loop:w ..... 27950
\__fp_tuple_ 24983, 24984, 24987, 24988
\__fp_tuple_&o:ww ..... 24966
\__fp_tuple_&tuple_o:ww ..... 24966
\__fp_tuple_*o:ww ..... 25839
\__fp_tuple_+tuple_o:ww ..... 25845
\__fp_tuple_-tuple_o:ww ..... 25845
\__fp_tuple/_o:ww ..... 25839
\__fp_tuple_chk:w ..... 971,
  22451, 22457, 22458, 22535, 22538,
  24211, 24423, 24438, 24463, 24466,
  24482, 24483, 24486, 24707, 24708,
  25848, 25849, 25855, 25856, 27929
\__fp_tuple_compare_back:ww .. 24704
\__fp_tuple_compare_back_loop:w .
  ..... 24704
\__fp_tuple_compare_back_-
  tuple:ww ..... 24704
\__fp_tuple_convert:Nw .....
  ..... 27929, 27978, 28032, 28106
\__fp_tuple_convert_end:w .... 27929
\__fp_tuple_convert_loop:nNw . 27929
\__fp_tuple_count:w ..... 22456
\__fp_tuple_count_loop:Nw .... 22456
\__fp_tuple_map_loop_o:nw .... 24463
\__fp_tuple_map_o:nw .....
  .. 24463, 25832, 25840, 25842, 25844
\__fp_tuple_mapthread_loop_o:nw .
  ..... 24481
\__fp_tuple_mapthread_o:nww ....
  ..... 24481, 25853
\__fp_tuple_not_o:w ..... 24957
\__fp_tuple_set_sign_aux_o:Nnw 25824
\__fp_tuple_set_sign_aux_o:w . 25824
\__fp_tuple_set_sign_o:w ..... 25824
\__fp_tuple_to_decimal:w ..... 28020
\__fp_tuple_to_scientific:w .. 27966
\__fp_tuple_to_tl:w ..... 28098
\__fp_tuple_|o:ww ..... 24966
\__fp_tuple_|tuple_o:ww ..... 24966
\__fp_type_from_scan:N .....
  . 972, 22480, 24037, 24039, 24063,
  24065, 24076, 24078, 24668, 24670
\__fp_type_from_scan:w ..... 22480
\__fp_type_from_scan_other:N ...
  ..... 22480, 22504, 22522
\__fp_underflow:w .....
  .. 970, 984, 986, 22400, 22833, 26576
\__fp_use_i:ww .....
  .... 1099, 1153, 22336, 26089, 27858

```

- `__fp_use_i:www` [22336](#)
- `__fp_use_i_delimit_by_s_stop:nw`
..... [22347](#), [24636](#), [24998](#)
- `__fp_use_i_until_s:nw` [1143](#), [22331](#),
[22380](#), [22390](#), [22653](#), [27228](#), [27506](#),
[27512](#), [27543](#), [28318](#), [28389](#), [28600](#)
- `__fp_use_ii_until_s:nnw`
..... [22331](#), [22378](#), [22389](#)
- `__fp_use_none_stop_f:n`
..... [22328](#), [26254](#), [26255](#), [26256](#)
- `__fp_use_none_until_s:w`
.. [22331](#), [25635](#), [26932](#), [27853](#), [27856](#)
- `__fp_use_s:n` [22329](#)
- `__fp_use_s:nn` [22329](#)
- `__fp_zero_fp:N` . [22371](#), [22817](#), [23165](#)
- `__fp_l_o:ww` [1046](#), [24966](#)
- `__fp_l_tuple_o:ww` [24966](#)
- fpararray commands:
- `\fpararray_count:N`
.... [260](#), [28560](#), [28572](#), [28583](#), [28639](#)
- `\fpararray_gset:Nnn` ... [260](#), [1177](#), [28585](#)
- `\fpararray_gzero:N` [260](#), [28636](#)
- `\fpararray_item:Nn` ... [260](#), [1177](#), [28649](#)
- `\fpararray_item_to_tl:Nn` ... [260](#), [28649](#)
- `\fpararray_new:Nn` [260](#), [28533](#)
- `\futurelet` [277](#)
- G**
- `\gdef` [278](#)
- get commands:
- `get_luaadata` [11863](#)
- `\GetIdInfo` [10](#), [11423](#)
- `\gleaders` [826](#)
- `\global` [179](#), [279](#)
- `\globaldefs` [280](#)
- `\glueexpr` [534](#)
- `\glueshrink` [535](#)
- `\glueshrinkorder` [536](#)
- `\gluestretch` [537](#)
- `\gluestretchorder` [538](#)
- `\gluetomu` [539](#)
- group commands:
- `\group_align_safe_begin/end:` [430](#), [573](#)
- `\group_align_safe_begin:` [71](#), [566](#),
[675](#), [680](#), [3779](#), [8475](#), [8678](#), [12231](#),
[12852](#), [19353](#), [19374](#), [19406](#), [20470](#),
[20614](#), [29431](#), [29808](#), [31359](#), [36279](#)
- `\group_align_safe_end:`
..... [71](#), [675](#), [680](#),
[3782](#), [8477](#), [8678](#), [12252](#), [12835](#),
[19362](#), [19371](#), [19411](#), [19417](#), [20473](#),
[20614](#), [29443](#), [29821](#), [31370](#), [36287](#)
- `\group_begin:` [13](#),
[671](#), [1311](#), [1327](#), [1445](#), [2178](#), [2181](#),
[2184](#), [2593](#), [3064](#), [3255](#), [3418](#), [3455](#),
[3734](#), [3778](#), [3785](#), [3858](#), [4018](#), [4202](#),
[4597](#), [4689](#), [5013](#), [5524](#), [5858](#), [6081](#),
[6182](#), [6612](#), [6989](#), [7269](#), [7519](#), [7545](#),
[7557](#), [7567](#), [7576](#), [7757](#), [7790](#), [7911](#),
[8678](#), [8723](#), [8876](#), [8972](#), [9279](#), [9303](#),
[9319](#), [9385](#), [10241](#), [10451](#), [10497](#),
[10759](#), [10900](#), [11430](#), [11968](#), [12105](#),
[12327](#), [12340](#), [13092](#), [13119](#), [13393](#),
[13416](#), [13823](#), [13933](#), [13986](#), [14281](#),
[14329](#), [14375](#), [14382](#), [14711](#), [14893](#),
[16448](#), [16485](#), [18592](#), [18598](#), [18649](#),
[18715](#), [18960](#), [18978](#), [19002](#), [19090](#),
[19109](#), [19489](#), [20451](#), [20590](#), [20634](#),
[21695](#), [24966](#), [28782](#), [29004](#), [29098](#),
[29144](#), [29228](#), [29255](#), [30760](#), [30767](#),
[30796](#), [31055](#), [31092](#), [31252](#), [31281](#),
[31314](#), [31606](#), [31693](#), [32102](#), [33911](#),
[34083](#), [34580](#), [34668](#), [34710](#), [36264](#)
- `\c_group_begin_token`
.... [106](#), [194](#), [309](#), [446](#), [690](#), [864](#),
[3823](#), [4322](#), [12698](#), [12736](#), [18960](#),
[18984](#), [29311](#), [29516](#), [32145](#), [32151](#),
[32165](#), [32171](#), [32249](#), [32255](#), [32270](#),
[32276](#), [34509](#), [34510](#), [34517](#), [36302](#)
- `\group_end:` . [13](#), [14](#), [443](#), [550](#), [791](#),
[1184](#), [1187](#), [1311](#), [1445](#), [2178](#), [2181](#),
[2187](#), [2602](#), [3067](#), [3258](#), [3422](#), [3463](#),
[3672](#), [3748](#), [3783](#), [3804](#), [3865](#), [4042](#),
[4193](#), [4609](#), [4703](#), [5046](#), [5054](#), [5537](#),
[5862](#), [6141](#), [6189](#), [6196](#), [6204](#), [6616](#),
[6617](#), [7026](#), [7333](#), [7524](#), [7552](#), [7640](#),
[7784](#), [7830](#), [7912](#), [7913](#), [8682](#), [8742](#),
[8893](#), [9000](#), [9298](#), [9311](#), [9330](#), [9539](#),
[10247](#), [10455](#), [10526](#), [10782](#), [10918](#),
[11433](#), [11971](#), [12127](#), [12177](#), [12330](#),
[12344](#), [13110](#), [13142](#), [13398](#), [13421](#),
[13833](#), [13946](#), [13989](#), [14294](#), [14341](#),
[14400](#), [14462](#), [14892](#), [15024](#), [16457](#),
[16495](#), [16500](#), [18600](#), [18607](#), [18718](#),
[18734](#), [18977](#), [18981](#), [19009](#), [19108](#),
[19157](#), [19513](#), [20589](#), [20615](#), [20674](#),
[21709](#), [24990](#), [28786](#), [29037](#), [29153](#),
[29225](#), [29242](#), [29271](#), [30785](#), [30795](#),
[31078](#), [31106](#), [31276](#), [31280](#), [31307](#),
[31340](#), [31610](#), [31963](#), [32108](#), [33912](#),
[34088](#), [34583](#), [34673](#), [34725](#), [36268](#)
- `\c_group_end_token` [309](#), [864](#),
[3826](#), [18960](#), [18989](#), [29312](#), [29522](#),
[32159](#), [32264](#), [34513](#), [34521](#), [36303](#)
- `\group_insert_after:N` [14](#),
[1451](#), [4605](#), [33922](#), [34513](#), [34514](#), [34827](#)
- `\group_log_list:` [14](#), [2190](#)
- `\group_show_list:` [14](#), [2190](#)

- groups commands:
 groups:n [224](#), [21170](#)
- H**
- \H [86](#), [29389](#),
 [31736](#), [31883](#), [31884](#), [31911](#), [31912](#)
- \halign [281](#)
- \hangingafter [282](#)
- \hangindent [283](#)
- \hbadness [284](#)
- \hbox [285](#)
- hbox commands:
 \hbox:n
 [270](#), [274](#), [32116](#), [32343](#), [32639](#), [33695](#)
- \hbox_gset:Nn
 [274](#), [32118](#), [32310](#), [32433](#),
 [32477](#), [32497](#), [32517](#), [32534](#), [32555](#),
 [32584](#), [32595](#), [32753](#), [33184](#), [35699](#)
- \hbox_gset:Nw [275](#), [32142](#), [32826](#)
- \hbox_gset_end: ... [275](#), [32142](#), [32829](#)
- \hbox_gset_to_wd:Nnn [275](#), [32130](#)
- \hbox_gset_to_wd:Nnw [275](#), [32162](#)
- \hbox_overlap_center:n ... [275](#), [32186](#)
- \hbox_overlap_left:n ... [275](#), [32186](#)
- \hbox_overlap_right:n ... [275](#), [32186](#)
- \hbox_set:Nn .. [270](#), [274](#), [275](#), [288](#),
 [32118](#), [32307](#), [32339](#), [32340](#), [32427](#),
 [32474](#), [32494](#), [32514](#), [32531](#), [32552](#),
 [32581](#), [32589](#), [32612](#), [32740](#), [33181](#),
 [33204](#), [33461](#), [33548](#), [33827](#), [35696](#),
 [35709](#), [35717](#), [35725](#), [35734](#), [35743](#),
 [35760](#), [35768](#), [35776](#), [35782](#), [35795](#)
- \hbox_set:Nw [275](#), [32142](#), [32813](#)
- \hbox_set_end: ... [275](#), [32142](#), [32816](#)
- \hbox_set_to_wd:Nnn [275](#), [32130](#)
- \hbox_set_to_wd:Nnw [275](#), [32162](#)
- \hbox_to_wd:nn ... [274](#), [32176](#), [32630](#)
- \hbox_to_zero:n
 [274](#), [32176](#), [32187](#), [32189](#), [32191](#), [35689](#)
- \hbox_unpack:N ... [275](#), [32192](#), [33465](#)
- \hbox_unpack_clear:N [36463](#)
- \hbox_unpack_drop:N [278](#), [32192](#), [36464](#)
- hcoffin commands:
 \hcoffin_gset:Nn [283](#), [32736](#)
- \hcoffin_gset:Nw [283](#), [32809](#)
- \hcoffin_gset_end: [283](#), [32809](#)
- \hcoffin_set:Nn
 [283](#), [32736](#), [33699](#), [33706](#), [33744](#), [33779](#)
- \hcoffin_set:Nw [283](#), [32809](#)
- \hcoffin_set_end: [283](#), [32809](#)
- \hfi [1117](#)
- \hfil [286](#)
- \hfill [287](#)
- \hfilneg [288](#)
- \hfuzz [289](#)
- \hjcode [821](#)
- \hoffset [290](#)
- \holdinginserts [291](#)
- hook commands:
 \hook_gput_code:nnn ... [28911](#), [28913](#)
- \hpack [822](#)
- \hrule [292](#)
- \hsize [293](#)
- \hskip [294](#)
- \hss [295](#)
- \ht [296](#)
- \Huge [31584](#)
- \huge [31588](#)
- \hyphenation [297](#)
- \hyphenationbounds [823](#)
- \hyphenationmin [824](#)
- \hyphenchar [298](#)
- \hyphenpenalty [299](#)
- \hyphenpenaltymode [825](#)
- I**
- \i [31305](#),
 [31666](#), [31792](#), [31794](#), [31796](#), [31798](#),
 [31849](#), [31852](#), [31855](#), [31858](#), [31929](#)
- \if [225](#), [300](#)
- if commands:
 \if:w [28](#), [181](#), [363](#), [364](#),
 [401](#), [467](#), [677](#), [678](#), [681](#), [691](#), [692](#),
 [705](#), [1416](#), [1776](#), [2072](#), [2073](#), [2956](#),
 [2959](#), [2960](#), [2961](#), [2962](#), [2977](#), [2978](#),
 [2979](#), [2980](#), [2981](#), [2982](#), [2983](#), [2984](#),
 [2985](#), [3049](#), [3050](#), [3052](#), [4914](#), [12281](#),
 [12291](#), [12387](#), [12750](#), [12770](#), [12785](#),
 [13218](#), [13225](#), [13230](#), [17610](#), [19274](#),
 [23094](#), [23467](#), [23471](#), [23493](#), [23586](#),
 [23618](#), [23637](#), [23703](#), [23717](#), [23734](#),
 [23755](#), [23794](#), [23806](#), [24092](#), [24134](#),
 [24254](#), [24269](#), [24621](#), [26793](#), [26823](#),
 [28330](#), [29158](#), [29166](#), [29183](#), [29295](#)
- \if_bool:N .. [70](#), [562](#), [1426](#), [8377](#), [8422](#)
- \if_box_empty:N ... [281](#), [32054](#), [32066](#)
- \if_case:w [167](#),
 [711](#), [713](#), [750](#), [820](#), [979](#), [1071](#), [1126](#),
 [1170](#), [1979](#), [2647](#), [3834](#), [4028](#), [4556](#),
 [4828](#), [5641](#), [5670](#), [5727](#), [6384](#), [6437](#),
 [7058](#), [7105](#), [7203](#), [7512](#), [8022](#), [8033](#),
 [10611](#), [13491](#), [13565](#), [13809](#), [14838](#),
 [16918](#), [17501](#), [17534](#), [18727](#), [22395](#),
 [22648](#), [22663](#), [23034](#), [23063](#), [24342](#),
 [24383](#), [25057](#), [25192](#), [25267](#), [25292](#),
 [25344](#), [25788](#), [25804](#), [25821](#), [26098](#),
 [26325](#), [26352](#), [26510](#), [26545](#), [26703](#),
 [26748](#), [26799](#), [26925](#), [26948](#), [26981](#),

27040, 27080, 27095, 27110, 27125,
 27140, 27155, 27681, 27734, 27818,
 27833, 27885, 27898, 27982, 28036,
 28110, 28327, 28614, 28693, 36296
 \if_catcode:w
 28, 690, 877, 1416, 3096,
 3823, 3826, 3980, 3982, 3984, 3986,
 3988, 3990, 3992, 4214, 4215, 4322,
 12693, 12734, 18895, 18898, 18901,
 18904, 18907, 18910, 18913, 18984,
 18989, 18994, 18999, 19006, 19013,
 19018, 19023, 19028, 19033, 19038,
 19048, 19075, 19384, 19443, 19448,
 19495, 19496, 22231, 23222, 23427,
 23745, 23792, 24091, 24133, 29259,
 29260, 29296, 29311, 29312, 29313,
 29314, 29315, 29316, 29317, 29318,
 29319, 29342, 29345, 29348, 29351,
 29354, 29357, 29360, 36297, 36298
 \if_charcode:w 28, 181,
 435, 689, 690, 716, 877, 1416, 3893,
 3917, 3966, 4234, 4765, 4775, 5284,
 5840, 6995, 6998, 11228, 11237,
 12679, 12727, 13649, 13789, 14451,
 19053, 19445, 22651, 24634, 24996
 \if_cs_exist:N 28,
 1431, 1803, 1831, 2596, 19083, 19283
 \if_cs_exist:w 28, 1431, 1459, 1812,
 1840, 1966, 17750, 17778, 17787, 35807
 \if_dim:w
 220, 19935, 20023, 20035, 20058, 20229
 \if_eof:w
 . 93, 617, 10184, 10189, 10274, 10292
 \if_false: 27, 64,
 194, 391, 427, 446, 532, 544, 548,
 573, 636, 671, 676, 680, 688, 794,
 811, 858, 892, 1416, 2607, 2617,
 2630, 2643, 2671, 2687, 2785, 2799,
 2805, 2812, 2820, 2830, 2843, 2847,
 3770, 3812, 3861, 3864, 4326, 4327,
 4334, 4335, 5022, 5041, 5042, 5051,
 5116, 5159, 5173, 5177, 5391, 5424,
 5436, 5440, 5474, 5479, 5487, 5522,
 5529, 5534, 5582, 5819, 5838, 5849,
 5872, 5884, 5885, 5888, 7285, 7302,
 7631, 7657, 7665, 7672, 7702, 7838,
 7840, 7841, 7847, 8681, 8877, 8885,
 10590, 10630, 10634, 10641, 10649,
 10899, 10912, 11891, 11895, 12106,
 12113, 12247, 12248, 12359, 12363,
 12402, 12634, 12639, 12646, 12651,
 12661, 12751, 12764, 12782, 12786,
 12796, 16391, 16394, 16573, 16578,
 17131, 18679, 18723, 19858, 19859,
 19860, 19861, 19896, 19897, 19898,
 19899, 20045, 29518, 29524, 36050,
 36051, 36169, 36181, 36207, 36217
 \if_hbox:N 281, 32054, 32058
 \if_int_compare:w
 27, 167, 705, 811, 812, 1449, 2835,
 3343, 3400, 3429, 3470, 3481, 3484,
 3502, 3557, 3567, 3577, 3798, 3870,
 3903, 3911, 3934, 3958, 4009, 4024,
 4113, 4116, 4262, 4437, 4496, 4502,
 4503, 4510, 4514, 4520, 4521, 4526,
 4527, 4535, 4536, 4537, 4543, 4575,
 4576, 4825, 4845, 4846, 4847, 4850,
 4854, 4855, 4858, 4859, 4867, 4868,
 4871, 4875, 4876, 4879, 4938, 4960,
 4972, 4981, 4989, 4992, 5002, 5005,
 5033, 5120, 5232, 5298, 5303, 5331,
 5389, 5422, 5533, 5550, 5896, 5929,
 5960, 6331, 6402, 6428, 6489, 6502,
 6513, 6529, 6580, 6621, 6627, 6633,
 6798, 6799, 6826, 6853, 6952, 7010,
 7129, 7138, 7149, 7164, 7220, 7229,
 7281, 7298, 7321, 7579, 7608, 7653,
 7661, 7743, 8684, 10187, 10188,
 10597, 11244, 12941, 12950, 12999,
 13000, 13006, 13206, 13213, 13475,
 13530, 13531, 13537, 13549, 13565,
 13798, 13806, 14013, 14014, 14015,
 14020, 14021, 14045, 14097, 14197,
 14416, 14478, 14482, 14512, 14515,
 14531, 14535, 14556, 14636, 14638,
 14657, 14658, 14676, 14678, 14732,
 14735, 14736, 14854, 14855, 14996,
 15001, 16918, 16973, 17014, 17015,
 17111, 17164, 17166, 17168, 17170,
 17172, 17174, 17176, 17179, 17312,
 18621, 18622, 18629, 18630, 18631,
 18632, 18637, 18638, 18668, 18742,
 18743, 18749, 19265, 20074, 22085,
 22088, 22132, 22198, 22217, 22396,
 22397, 22591, 22688, 22914, 22924,
 22932, 22945, 22958, 22965, 22986,
 22998, 23007, 23018, 23127, 23132,
 23204, 23234, 23387, 23389, 23426,
 23431, 23484, 23504, 23531, 23545,
 23580, 23607, 23635, 23651, 23667,
 23685, 23745, 23765, 23781, 23865,
 23888, 23917, 23919, 24100, 24102,
 24142, 24144, 24166, 24183, 24188,
 24218, 24286, 24331, 24645, 24692,
 24695, 24726, 24735, 24738, 24743,
 24744, 24747, 24750, 24937, 25061,
 25082, 25119, 25214, 25268, 25269,
 25272, 25275, 25345, 25354, 25559,

- 25632, 25685, 25689, 25693, 25711,
 25746, 25747, 25748, 25749, 25750,
 25776, 26100, 26103, 26197, 26290,
 26338, 26354, 26481, 26515, 26573,
 26582, 26622, 26794, 26805, 26823,
 26846, 26878, 26881, 26928, 26958,
 27005, 27019, 27183, 27227, 27685,
 27723, 27732, 27768, 27852, 27855,
 28084, 28317, 28385, 28386, 28387,
 28397, 28425, 28430, 28431, 28494,
 28495, 28496, 28500, 28515, 28520,
 28568, 28572, 29159, 29328, 30358,
 30361, 30362, 30365, 30378, 30381,
 30384, 30387, 30390, 30393, 30407,
 30410, 30413, 30416, 30419, 30431,
 30434, 30437, 30440, 36250, 36258
- `\if_int_odd:w` 168, 1146,
 4034, 4213, 4565, 4949, 4957, 4969,
 5395, 5710, 16918, 17047, 17217,
 17225, 17725, 18628, 18636, 18653,
 19494, 22936, 22983, 22995, 24379,
 25328, 25614, 26969, 27533, 27572,
 27582, 27625, 27649, 27809, 28493
- `\if_meaning:w`
 27, 690, 777, 1055, 1199,
 1364, 1416, 1627, 1653, 1671, 1733,
 1738, 1747, 1800, 1818, 1828, 1846,
 1997, 2011, 2133, 2250, 2313, 2314,
 2648, 2651, 2652, 2653, 2654, 2747,
 2777, 2790, 2796, 2904, 2927, 2936,
 3128, 3201, 3213, 3214, 3353, 3354,
 3820, 3850, 3878, 3978, 4181, 4216,
 4227, 4273, 4319, 4366, 4367, 4604,
 4760, 4785, 4797, 4913, 4937, 5280,
 5283, 5717, 6366, 6537, 6548, 6563,
 6724, 6768, 6903, 7176, 7490, 7607,
 7720, 8487, 8509, 10547, 10842,
 11882, 12271, 12315, 12331, 12345,
 12718, 13324, 13402, 13425, 13586,
 13624, 13959, 14686, 14835, 14850,
 14877, 14992, 15925, 15931, 15957,
 15969, 15977, 16009, 16016, 16040,
 16044, 16122, 16147, 16162, 16427,
 16490, 16505, 16513, 16941, 16944,
 16954, 16989, 16994, 16995, 17146,
 17957, 17972, 17994, 18008, 19043,
 19080, 19119, 19122, 19257, 19359,
 19387, 19398, 19436, 19497, 20004,
 20051, 20232, 22377, 22398, 22410,
 22420, 22515, 22570, 22579, 22670,
 22685, 22687, 22825, 22913, 22923,
 22935, 22948, 22949, 22968, 22969,
 22983, 22984, 22995, 22996, 23062,
 23109, 23144, 23147, 23163, 23170,
 23223, 23226, 23342, 23343, 23344,
 23345, 23348, 23444, 23557, 23563,
 23793, 23837, 24048, 24119, 24380,
 24398, 24448, 24458, 24681, 24682,
 24683, 24684, 24685, 24686, 24918,
 24930, 24931, 24959, 24971, 24978,
 24995, 25058, 25093, 25107, 25153,
 25160, 25236, 25248, 25348, 25351,
 25362, 25415, 25488, 25558, 25561,
 25568, 25601, 25602, 25605, 25826,
 26071, 26082, 26263, 26273, 26322,
 26421, 26501, 26550, 26564, 26711,
 26745, 26757, 26770, 26773, 26776,
 26779, 26804, 26905, 26909, 26968,
 26985, 26991, 27575, 27628, 27679,
 27680, 27682, 27683, 27703, 27720,
 27787, 27885, 27981, 28035, 28109,
 28179, 28184, 28316, 28348, 28359,
 28466, 28627, 28680, 28686, 29135,
 29137, 29148, 29274, 29759, 36009,
 36043, 36174, 36211, 36230, 36299
- `\if_mode_horizontal:` . 28, 1427, 8673
`\if_mode_inner:` 28, 1427, 8675
`\if_mode_math:` 28, 1427, 8677
`\if_mode_vertical:` 28, 1427, 2260, 8671
`\if_predicate:w` . 62, 64, 70, 8377,
 8465, 8525, 8540, 8551, 8566, 8577
`\if_true:` 27,
 64, 1416, 11889, 11893, 12776, 12782
`\if_vbox:N` 281, 32054, 32060
- `\ifabsdim` 912
`\ifabsnum` 913
`\ifcase` 301
`\ifcat` 302
`\ifcondition` 827
`\ifcsname` 344, 634, 540
`\ifdbbox` 1118
`\ifddir` 1119
`\ifdefined` 541
`\ifdim` 303
`\ifeof` 304
`\iffalse` 305
`\iffontchar` 542
`\ifhbox` 306
`\ifhmode` 307
`\ifincsname` 686
`\ifinner` 308
`\ifjfont` 1120
`\ifmbox` 1121
`\ifmdir` 1122
`\ifmmode` 309
`\ifnum` 27, 39, 69, 76, 310
`\ifodd` 311
`\ifpdfabsdim` 647

- \ifpdfabsnum 648
- \ifpdfprimitive 649
- \ifprimitive 779
- \iftbox 1123
- \iftdir 1125
- \iftfont 1124
- \iftrue 312, 11882
- \ifvbox 313
- \ifvmode 314
- \ifvoid 315
- \ifx 4, 21,
25, 30, 34, 70, 71, 73, 82, 106, 107, 316
- \ifybox 1126
- \ifydir 1127
- \ignoreligaturesinfont 914
- \ignorespaces 317
- \IJ 29397, 31296, 31656
- \ij 29397, 31296, 31668
- \immediate 89, 318
- \immediateassigned 828
- \immediateassignment 829
- in 257
- \indent 319
- inf 256
- \infty 23345, 23346
- inherit commands:
 - .inherit:n 225, 21172
- \inhibitglue 1128
- \inhibitxspcode 1129
- \initcatcodetable 830
- initial commands:
 - .initial:n 225, 21174
- \input 31, 320
- \inputlineno 321
- \insert 322
- \insertht 915
- \insertpenalties 323
- int commands:
 - \int_abs:n 156, 805, 16947, 22132
 - \int_add:Nn 158, 4544, 5712,
6519, 6520, 6778, 6850, 10706, 17077
 - \int_case:nn 160, 820, 17185, 17364,
17370, 29077, 30512, 30527, 30590
 - \int_case:nnn 36465
 - \int_case:nnTF 160,
4297, 8343, 16821, 17185, 17190,
17195, 18318, 23254, 27931, 36466
 - \int_compare:nNnTF
.. 158–161, 243, 3330, 4160, 4583,
4595, 4748, 5075, 5077, 5942, 6727,
7081, 7240, 7632, 7824, 8360, 8752,
8758, 9104, 10406, 10519, 10991,
11115, 11125, 11473, 11516, 12108,
12136, 12151, 12159, 12896, 12903,
12970, 13454, 13456, 13465, 14153,
14229, 15036, 16442, 16631, 16638,
17028, 17034, 17177, 17209, 17261,
17269, 17278, 17284, 17296, 17299,
17360, 17448, 17454, 17460, 17480,
17634, 17653, 17655, 17697, 18390,
18392, 18397, 18406, 18426, 18443,
18460, 18861, 18945, 20252, 21841,
22070, 22075, 22082, 22190, 22266,
22292, 24710, 25851, 27914, 28059,
28061, 28548, 28758, 28929, 28962,
30038, 30067, 30074, 30154, 30156,
30167, 30213, 30226, 30235, 30276,
30666, 30669, 30709, 30715, 30722,
30736, 34097, 34826, 35098, 35104,
35894, 35967, 35968, 35970, 35972
 - \int_compare:nTF
.. 159, 161, 244, 898, 6147, 6187,
8079, 8308, 8309, 8314, 8316, 10146,
10360, 17124, 17233, 17241, 17250,
17256, 28119, 28968, 35992, 35994
 - \int_compare_p:n 159, 6194, 17124
 - \int_compare_p:nNn 27,
159, 5787, 5788, 8766, 8983, 9083,
9085, 9087, 10305, 11104, 11105,
17177, 28884, 30095, 30307, 30308,
30467, 30468, 30570, 30571, 30572,
30620, 30628, 30629, 30650, 30651,
30693, 34095, 35637, 35638, 35645,
35648, 35649, 35657, 35660, 35661
 - \int_const:Nn
.... 157, 4474, 4475, 4476, 4477,
4886, 4887, 4888, 4889, 4890, 4891,
4895, 4896, 4897, 4898, 4899, 4900,
4901, 4902, 4903, 4904, 4905, 4906,
4907, 8897, 8993, 8995, 8997, 8998,
8999, 9070, 10077, 10234, 10300,
10301, 13738, 13739, 17024, 17663,
17664, 17665, 17666, 17667, 17668,
17669, 17670, 17671, 17672, 17673,
17674, 17675, 17676, 17721, 17722,
17723, 22358, 22359, 22360, 22361,
22362, 22363, 22364, 22548, 22549,
22550, 22552, 22553, 22554, 22557,
22558, 22559, 22908, 23182, 23183,
23184, 23185, 23186, 23187, 23188,
23189, 23190, 23191, 23192, 23193,
23194, 23195, 23196, 26978, 28267
 - \int_decr:N . 158, 3490, 3491, 3492,
3555, 3556, 3565, 3566, 3575, 3576,
3813, 7222, 7299, 7514, 7609, 17089
 - \int_div_round:nn 156, 16979
 - \int_div_truncate:nn
..... 156, 157, 8996, 13854,

- 13859, 14505, 14506, 14561, 14743,
 14909, 14920, 16979, 17375, 17473,
 17493, 18755, 18768, 18773, 18785
 \int_do_until:nn 161, 17231
 \int_do_until:nNnn 160, 17259
 \int_do_while:nn 161, 17231
 \int_do_while:nNnn 160, 17259
 \int_eval:n .. 19, 33, 156–160, 167,
303, 343, 371, 450, 685, 807, 823,
953, 954, 958, 959, 965, 999, 1048,
1073, 1075, 1359, 1979, 2008, 2024,
 3432, 3696, 3697, 3936, 4012, 4016,
 4039, 4434, 4565, 4829, 5710, 5956,
 6146, 6351, 6353, 6367, 6368, 6370,
 6371, 6513, 6603, 6646, 6821, 6869,
 6968, 7142, 7148, 7151, 8035, 8038,
 8083, 8128, 8129, 8349, 8626, 9021,
 9908, 10130, 10345, 10642, 10700,
 11096, 11097, 11119, 11129, 11136,
 11137, 12162, 12532, 12537, 12545,
 12889, 12897, 12905, 12932, 12936,
 12945, 12952, 12987, 12997, 13448,
 13461, 13486, 13510, 13511, 13523,
 13528, 13559, 13576, 13613, 13809,
 13829, 13847, 14122, 14647, 14662,
 14690, 14839, 14844, 14862, 15006,
 16624, 16632, 16640, 16795, 16930,
 17025, 17188, 17193, 17198, 17203,
 17357, 17443, 17445, 17575, 17585,
 17620, 17631, 17637, 17648, 17679,
 17716, 17720, 18287, 18299, 18384,
 18394, 18408, 18415, 18430, 18493,
 18495, 18563, 18565, 18569, 18571,
 18575, 18577, 18581, 18583, 18616,
 18617, 18738, 18790, 18793, 18798,
 18804, 19701, 21840, 21926, 21974,
 21992, 22010, 22069, 22107, 22108,
 22159, 22176, 22262, 22286, 22295,
 22299, 22368, 28256, 28416, 28419,
 28420, 28510, 28511, 28543, 28591,
 28653, 28661, 28730, 28754, 28920,
 32084, 32094, 35106, 35149, 35195,
 35897, 35963, 36038, 36252, 36260
 \int_eval:w 156,
345, 348, 3763, 3998, 4008, 10593,
 10602, 10627, 10639, 13479, 13841,
 16771, 16930, 17753, 17788, 22024,
 22203, 22210, 22211, 22222, 25800
 \int_from_alph:n 164, 17618
 \int_from_base:nn
 165, 17635, 17658, 17660, 17662
 \int_from_bin:n ... 165, 17657, 36468
 \int_from_binary:n 36467
 \int_from_hex:n
 165, 17657, 34273, 34274, 34275, 36470
 \int_from_hexadecimal:n 36469
 \int_from_oct:n ... 165, 17657, 36472
 \int_from_octal:n 36471
 \int_from_roman:n 165, 17677
 \int_gadd:Nn 158, 17077
 \int_gdecr:N ... 158, 4078, 10269,
 12478, 13343, 16697, 16753, 17089,
 17355, 18221, 19874, 20217, 24902
 \int_gincr:N 158,
 4056, 4188, 6292, 10260, 12469,
 13332, 16689, 16747, 17089, 17330,
 17341, 18212, 19869, 20196, 20203,
 21834, 22060, 24881, 24888, 28538,
 28841, 34813, 35406, 35411, 35416
 .int_gset:N 225, 21182
 \int_gset:Nn 158, 808, 6312, 9336, 17101
 \int_gset_eq:NN 157, 17069
 \int_gsub:Nn 158, 17077, 28552
 \int_gzero:N
 157, 6272, 6289, 17059, 17066
 \int_gzero_new:N 157, 17063
 \int_if_even:nTF 160, 17215
 \int_if_even_p:n 160, 17215
 \int_if_exist:NTF
 157, 5636, 5691, 17064,
 17066, 17073, 17691, 17695, 35612
 \int_if_exist_p:N 157, 17073
 \int_if_odd:nTF 160,
 7431, 7454, 7528, 10852, 17215, 26186
 \int_if_odd_p:n 160, 6220, 17215
 \int_incr:N
 158, 3404, 3500, 3501, 3854, 3896,
 3909, 3927, 4410, 4411, 5527, 6152,
 6319, 6359, 6448, 6779, 6875, 7210,
 7282, 7508, 7513, 7548, 7606, 7678,
 7679, 7728, 7750, 7853, 7854, 8017,
 16464, 17089, 20864, 21991, 22148,
 22182, 22235, 22308, 28641, 35139
 \int_log:N 166, 17717
 \int_log:n 166, 17719
 \int_max:nn 157,
1164, 6015, 6016, 6023, 6024, 6304,
 6472, 7820, 7822, 16947, 26047, 27207
 \int_min:nn 157, 1168, 16947
 \int_mod:nn 157, 8994,
 14158, 14222, 14506, 14507, 14744,
16979, 17365, 17464, 17484, 18787
 \int_new:N 157, 3309, 3310,
 3311, 3312, 3313, 3314, 3315, 3316,
 3317, 3318, 3319, 3749, 3750, 3751,
 3752, 4178, 4461, 4462, 4463, 4473,
 4884, 4885, 4892, 4893, 4910, 6237,
 6239, 6240, 6241, 6244, 6267, 6268,

- 6652, 6653, 6654, 6655, 6656, 6657,
- 6658, 6660, 6661, 6662, 6663, 6666,
- 6667, 6668, 6929, 7475, 7478, 7479,
- 7480, 7486, 7487, 8364, 8685, 10428,
- 10431, 10433, 10446, 14204, 17018,
- 17030, 17036, 17064, 17066, 17733,
- 17734, 17735, 17736, 17737, 17738,
- 20683, 21818, 21821, 21822, 22056,
- 28531, 28532, 28709, 33930, 34754
- \int_rand:n 165,
- 302, 21983, 22169, 28258, 28261, 28508
- \int_rand:nn
 .. 74, 165, 302, 1167, 1168, 1175,
- 1361, 12912, 16646, 17721, 18444,
- 18449, 28252, 28255, 28414, 35887
- \int_range:nn 1169
- .int_set:N 225, 21182
- \int_set:Nn
 158, 343, 2185, 2199, 2200,
- 2203, 2205, 2207, 3324, 3326, 3328,
- 3350, 3351, 3366, 3374, 3375, 3387,
- 3388, 3406, 3409, 3808, 3871, 4208,
- 4260, 4263, 4398, 5718, 6238, 6300,
- 6302, 6308, 6346, 6348, 6417, 6468,
- 6469, 6479, 6490, 6514, 6532, 6581,
- 6713, 6715, 6718, 6739, 6783, 6784,
- 6825, 6860, 7553, 7625, 7627, 7766,
- 7796, 7819, 7821, 10220, 10222,
- 10412, 10414, 10429, 10439, 10452,
- 10499, 10505, 10517, 10522, 12109,
- 12144, 14283, 14332, 14385, 16465,
- 17101, 20869, 22226, 28809, 28810,
- 28825, 28990, 29005, 29041, 32093,
- 32095, 32103, 32104, 32105, 32106
- \int_set_eq:NN 157,
- 3367, 3397, 4534, 5003, 5007, 5016,
- 5018, 5061, 5128, 5426, 5526, 5539,
- 5638, 6258, 6278, 6295, 6323, 6357,
- 6358, 6408, 6511, 6512, 6564, 6613,
- 6691, 6714, 6719, 6733, 6737, 6741,
- 6780, 6790, 6921, 6922, 7504, 7721,
- 8013, 8878, 10901, 12107, 12110, 17069
- \int_show:N 166, 17713
- \int_show:n 166, 826, 1359, 17715
- \int_sign:n ... 156, 903, 16933, 28891
- \int_step.... 60
- \int_step_function:nN
 162, 16455, 17287, 29009, 29010
- \int_step_function:nnN . 162, 3459,
- 6791, 7621, 17287, 18714, 18719,
- 29011, 29012, 29013, 29014, 29035
- \int_step_function:nnnN 162, 305,
- 816, 1051, 7799, 7807, 17287, 17354
- \int_step_inline:nn 162,
- 959, 17324, 22063, 28735, 28769, 28820
- \int_step_inline:nnn 162,
- 6706, 8367, 10081, 10312, 14142,
- 14151, 17324, 28742, 28745, 28991
- \int_step_inline:nnnn 162, 1053, 17324
- \int_step_variable:nNn ... 162, 17324
- \int_step_variable:nnNn .. 162, 17324
- \int_step_variable:nnnNn . 162, 17324
- \int_sub:Nn 158, 4538, 5239,
- 6567, 6575, 6584, 9280, 10714, 17077
- \int_to_Alph:n 163, 164, 17378
- \int_to_alph:n 163, 164, 17378
- \int_to_arabic:n 163, 17357
- \int_to_Base:n 164
- \int_to_base:n 164
- \int_to_Base:nn 164, 165, 17442, 17569
- \int_to_base:nn
 164, 165, 17442, 17565, 17567, 17571
- \int_to_bin:n . 164, 165, 17564, 36474
- \int_to_binary:n 36473
- \int_to_Hex:n
 164, 165, 4751, 17564, 34748
- \int_to_hex:n . 164, 165, 17564, 36476
- \int_to_hexadecimal:n 36475
- \int_to_oct:n . 164, 165, 17564, 36478
- \int_to_octal:n 36477
- \int_to_Roman:n 164, 165, 17572
- \int_to_roman:n 164, 165, 17572
- \int_to_symbols:nnn
 163, 17358, 17380, 17412
- \int_until_do:nn 161, 17231
- \int_until_do:nnnn 161, 17259
- \int_use:N 155, 158,
- 992, 998, 2204, 2206, 2208, 4058,
- 4190, 4206, 5035, 5122, 5200, 5211,
- 5220, 5224, 5235, 5236, 5242, 5243,
- 5249, 5250, 5409, 6220, 6313, 6318,
- 6339, 6341, 6446, 6459, 6460, 6861,
- 6913, 7012, 7023, 7178, 7553, 7637,
- 7638, 7827, 7828, 8361, 9240, 9910,
- 10223, 10262, 10408, 12471, 12473,
- 13334, 13338, 14643, 14667, 14681,
- 14701, 14708, 14890, 14999, 15004,
- 15020, 16690, 16696, 16749, 16751,
- 17107, 17333, 17344, 18214, 18216,
- 19868, 19876, 20199, 20206, 20870,
- 21835, 21929, 21977, 24884, 24891,
- 28843, 28845, 28876, 28924, 34819,
- 36060, 36069, 36071, 36074, 36079,
- 36088, 36090, 36094, 36097, 36102
- \int_value:w
 156, 167, 348, 396, 427, 567, 805,
- 811, 898, 953, 954, 958, 959, 969,

975, 979, 992, 1000, 1007, 1010,
 1015, 1022, 1049, 1059, 1067, 1075,
 1141, 1145, 1159, 1361, 1781, 2797,
 2799, 3432, 3763, 3775, 3954, 3996,
 3998, 4008, 4016, 4035, 4037, 4045,
 4254, 4310, 4330, 4339, 4744, 5268,
 5274, 5306, 5308, 5317, 5318, 5433,
 5918, 5933, 6946, 6947, 6958, 7649,
 8500, 8503, 8626, 9009, 9051, 9059,
 10593, 10602, 12945, 12952, 13448,
 13449, 13461, 13479, 13486, 13509,
 13510, 13511, 13523, 13559, 14075,
 14199, 14561, 14639, 14647, 14662,
 14690, 16759, 16771, 16918, 16935,
 16936, 16949, 16950, 16957, 16958,
 16959, 16965, 16966, 16967, 16981,
 16983, 16984, 17001, 17004, 17005,
 17006, 17013, 17127, 17131, 17161,
 17290, 17291, 17292, 17318, 17528,
 17561, 17753, 17788, 17798, 18616,
 18617, 20032, 20223, 20259, 22079,
 22082, 22107, 22108, 22154, 22159,
 22203, 22210, 22211, 22222, 22442,
 22443, 22444, 22445, 22446, 22460,
 22608, 22669, 22687, 22982, 23112,
 23126, 23128, 23130, 23133, 23169,
 23307, 23337, 23338, 23375, 23383,
 23514, 23519, 23521, 23530, 23534,
 23571, 23579, 23582, 23588, 23599,
 23610, 23616, 23617, 23620, 23663,
 23673, 23675, 23691, 23693, 23716,
 23730, 23806, 23807, 23881, 23969,
 24680, 24713, 25068, 25069, 25070,
 25072, 25118, 25121, 25124, 25147,
 25149, 25170, 25172, 25181, 25183,
 25187, 25205, 25212, 25218, 25228,
 25230, 25244, 25252, 25260, 25304,
 25306, 25322, 25324, 25327, 25330,
 25384, 25392, 25394, 25396, 25398,
 25401, 25404, 25406, 25425, 25427,
 25431, 25437, 25439, 25443, 25465,
 25468, 25476, 25478, 25481, 25482,
 25483, 25484, 25499, 25502, 25505,
 25508, 25517, 25520, 25523, 25526,
 25533, 25535, 25541, 25549, 25551,
 25553, 25579, 25581, 25590, 25592,
 25596, 25613, 25634, 25638, 25650,
 25653, 25656, 25659, 25662, 25665,
 25668, 25671, 25675, 25687, 25691,
 25695, 25698, 25719, 25721, 25723,
 25733, 25757, 25760, 25772, 25774,
 25780, 25783, 25800, 25820, 25870,
 25875, 25877, 25884, 25887, 25890,
 25893, 25896, 25899, 25908, 25920,
 25928, 25930, 25940, 25942, 25949,
 25958, 25960, 25963, 25966, 25969,
 25972, 25985, 25987, 25995, 25997,
 26005, 26007, 26017, 26020, 26023,
 26030, 26045, 26063, 26066, 26122,
 26136, 26138, 26144, 26157, 26159,
 26161, 26185, 26201, 26208, 26209,
 26253, 26255, 26256, 26257, 26298,
 26300, 26337, 26344, 26351, 26372,
 26374, 26376, 26378, 26391, 26395,
 26396, 26397, 26398, 26399, 26404,
 26409, 26411, 26417, 26434, 26435,
 26436, 26437, 26438, 26439, 26444,
 26446, 26448, 26450, 26452, 26457,
 26459, 26461, 26463, 26465, 26467,
 26489, 26497, 26513, 26518, 26522,
 26581, 26630, 26698, 26707, 26715,
 26726, 26728, 26731, 26734, 26822,
 26858, 26860, 26863, 26866, 26869,
 26872, 26879, 26882, 26884, 26888,
 26910, 26912, 26944, 27014, 27024,
 27029, 27039, 27181, 27213, 27222,
 27454, 27455, 27466, 27469, 27472,
 27475, 27478, 27481, 27484, 27487,
 27490, 27508, 27518, 27527, 27545,
 27554, 27561, 27571, 27615, 27624,
 27659, 27702, 27719, 27775, 27786,
 27797, 28007, 28083, 28130, 28175,
 28183, 28185, 28187, 28279, 28302,
 28356, 28396, 28408, 28419, 28420,
 28450, 28453, 28456, 28458, 28460,
 28467, 28470, 28478, 28483, 28488,
 28591, 28653, 28661, 28674, 28675,
 28676, 28686, 29178, 29310, 35887,
 35897, 36002, 36012, 36252, 36260
 \int_while_do:nn 161, 17231
 \int_while_do:nNnn 161, 17259
 \int_zero:N 157, 2193,
 3809, 3810, 3811, 3910, 5015, 5237,
 5675, 6184, 6257, 6288, 6712, 6991,
 7498, 7499, 7547, 7734, 8008, 10559,
 16449, 17059, 17064, 20861, 21988,
 22145, 22174, 22305, 28638, 35136
 \int_zero_new:N 157, 17063
 \c_max_char_int 166, 4748, 17723, 18638
 \c_max_int
 . 166, 236, 513, 962, 1168, 17722,
 28461, 32081, 32087, 33868, 33871
 \c_max_register_int
 166, 412, 1469, 3326,
 3351, 3388, 9908, 9910, 16442, 16918
 \c_one_int .. 166, 3911, 4217, 6502,
 6513, 17090, 17092, 17094, 17096,
17721, 19498, 22203, 22222, 25685,

- 25689, 25693, 25747, 26338, 26481,
 26622, 26928, 27768, 27852, 28319,
 28323, 28330, 28515, 36250, 36258
- \g_tmpa_int [166](#), [17733](#)
 \l_tmpa_int [4](#), [52](#), [166](#), [17733](#)
 \g_tmpb_int [166](#), [17733](#)
 \l_tmpb_int [4](#), [166](#), [17733](#)
 \c_zero_int [166](#),
[353](#), [364](#), [685](#), [1468](#), [1779](#), [1781](#),
[3934](#), [3958](#), [4009](#), [4113](#), [4214](#), [4215](#),
[4216](#), [4437](#), [5033](#), [5533](#), [5929](#), [6402](#),
[6428](#), [6489](#), [6529](#), [6580](#), [7010](#), [7149](#),
[7220](#), [7229](#), [7281](#), [7298](#), [7321](#), [7653](#),
[7661](#), [8684](#), [8878](#), [9012](#), [9062](#), [10901](#),
[11248](#), [12107](#), [12950](#), [12999](#), [13208](#),
[13213](#), [13530](#), [13565](#), [14197](#), [14996](#),
[17014](#), [17015](#), [17028](#), [17059](#), [17060](#),
[17111](#), [17119](#), [17296](#), [17299](#), [17721](#),
[18622](#), [18637](#), [19495](#), [19496](#), [19497](#),
[20074](#), [20252](#), [22064](#), [22190](#), [22688](#),
[22914](#), [22918](#), [22920](#), [22924](#), [22928](#),
[22941](#), [22954](#), [22961](#), [22974](#), [22986](#),
[22998](#), [23007](#), [23010](#), [23021](#), [23127](#),
[23132](#), [24695](#), [24727](#), [25711](#), [25746](#),
[25748](#), [25749](#), [25750](#), [26515](#), [26582](#),
[26805](#), [26823](#), [26846](#), [26878](#), [27724](#),
[28084](#), [28302](#), [28331](#), [28431](#), [28495](#)
- int internal commands:
- __int_abs:N [16947](#)
 __int_case:nnTF [17185](#)
 __int_case:nw [17185](#)
 __int_case_end:nw [17185](#)
 __int_compare:nnN [812](#), [17124](#)
 __int_compare:NNw .. [811](#), [812](#), [17124](#)
 __int_compare:Nw ... [811](#), [812](#), [17124](#)
 __int_compare:w [811](#), [17124](#)
 __int_compare_!=:NNw [17124](#)
 __int_compare_<:NNw [17124](#)
 __int_compare_<=:NNw [17124](#)
 __int_compare_=:NNw [17124](#)
 __int_compare_==:NNw [17124](#)
 __int_compare_>:NNw [17124](#)
 __int_compare_>=:NNw [17124](#)
 __int_compare_end=:NNw . [812](#), [17124](#)
 __int_compare_error:
 [810](#), [811](#), [17109](#), [17127](#), [17129](#)
 __int_compare_error:Nw
 [810-812](#), [17109](#), [17149](#)
 __int_const:nN [17024](#)
 __int_constdef:Nw [17024](#)
 __int_div_truncate:NwNw [16979](#)
 __int_eval:w
[343](#), [805](#), [806](#), [811](#), [16918](#), [16931](#),
[16932](#), [16936](#), [16950](#), [16958](#), [16959](#),
[16966](#), [16967](#), [16981](#), [16983](#), [16984](#),
[17001](#), [17004](#), [17005](#), [17006](#), [17013](#),
[17044](#), [17078](#), [17080](#), [17082](#), [17084](#),
[17102](#), [17104](#), [17127](#), [17161](#), [17179](#),
[17217](#), [17225](#), [17290](#), [17291](#), [17292](#),
[17318](#), [17501](#), [17528](#), [17534](#), [17561](#)
 __int_eval_end: ... [16918](#), [16931](#),
[16936](#), [16950](#), [16985](#), [17001](#), [17007](#),
[17016](#), [17044](#), [17078](#), [17080](#), [17082](#),
[17084](#), [17102](#), [17104](#), [17179](#), [17217](#),
[17225](#), [17501](#), [17528](#), [17534](#), [17561](#)
 __int_from_alph:N [823](#), [17618](#)
 __int_from_alph:nN [823](#), [17618](#)
 __int_from_base:N [823](#), [17635](#)
 __int_from_base:nnN [823](#), [17635](#)
 __int_from_roman:NN [17677](#)
 \c__int_from_roman_C_int [17663](#)
 \c__int_from_roman_c_int [17663](#)
 \c__int_from_roman_D_int [17663](#)
 \c__int_from_roman_d_int [17663](#)
 __int_from_roman_error:w [17677](#)
 \c__int_from_roman_I_int [17663](#)
 \c__int_from_roman_i_int [17663](#)
 \c__int_from_roman_L_int [17663](#)
 \c__int_from_roman_l_int [17663](#)
 \c__int_from_roman_M_int [17663](#)
 \c__int_from_roman_m_int [17663](#)
 \c__int_from_roman_V_int [17663](#)
 \c__int_from_roman_v_int [17663](#)
 \c__int_from_roman_X_int [17663](#)
 \c__int_from_roman_x_int [17663](#)
 __int_if_recursion_tail_stop:N .
 [16928](#), [17690](#)
 __int_if_recursion_tail_stop_-
 do:Nn ... [16928](#), [17629](#), [17646](#), [17693](#)
 \l__int_internal_a_int [17737](#)
 \l__int_internal_b_int [17737](#)
 \c__int_max_constdef_int [17024](#)
 __int_maxmin:wwN [16947](#)
 __int_mod:ww [16979](#)
 __int_pass_signs:wn
 [822](#), [17608](#), [17622](#), [17639](#)
 __int_pass_signs_end:wn [17608](#)
 __int_show:nN [17713](#)
 __int_sign:Nw [16933](#)
 __int_step:NNnnnn [17324](#)
 __int_step:NwnnN [17287](#)
 __int_step:wwN [17287](#)
 __int_to_Base:nn [17442](#)
 __int_to_base:nn [17442](#)
 __int_to_Base:nnN [17442](#)
 __int_to_base:nnN [17442](#)
 __int_to_Base:nnnN [17442](#)
 __int_to_base:nnnN [17442](#)

- __int_to_Letter:n [17442](#)
- __int_to_letter:n [17442](#)
- __int_to_roman:N [17572](#)
- __int_to_roman:w [811](#),
[822](#), [1449](#), [16918](#), [17137](#), [17575](#), [17585](#)
- __int_to_Roman_aux:N
..... [17584](#), [17587](#), [17590](#)
- __int_to_Roman_c:w [17572](#)
- __int_to_roman_c:w [17572](#)
- __int_to_Roman_d:w [17572](#)
- __int_to_roman_d:w [17572](#)
- __int_to_Roman_i:w [17572](#)
- __int_to_roman_i:w [17572](#)
- __int_to_Roman_l:w [17572](#)
- __int_to_roman_l:w [17572](#)
- __int_to_Roman_m:w [17572](#)
- __int_to_roman_m:w [17572](#)
- __int_to_Roman_Q:w [17572](#)
- __int_to_roman_Q:w [17572](#)
- __int_to_Roman_v:w [17572](#)
- __int_to_roman_v:w [17572](#)
- __int_to_Roman_x:w [17572](#)
- __int_to_roman_x:w [17572](#)
- __int_to_symbols:nnnn [17358](#)
- __int_use_none_delimit_by_s-
stop:w [16925](#), [17159](#)
- intarray commands:
 - \intarray_const_from_clist:Nn ...
..... [237](#), [21985](#), [22171](#), [26636](#), [27237](#)
 - \intarray_count:N
..... [236](#), [237](#), [345](#), [14159](#),
[14222](#), [21841](#), [21844](#), [21885](#), [21929](#),
[21977](#), [21983](#), [22070](#), [22073](#), [22075](#),
[22076](#), [22079](#), [22088](#), [22098](#), [22146](#),
[22169](#), [22190](#), [22249](#), [22306](#), [28563](#)
 - \intarray_gset:Nnn
..... [236](#), [345](#), [958](#), [960](#), [14143](#), [14155](#),
[14168](#), [14174](#), [21921](#), [22101](#), [28730](#)
 - \intarray_gset_rand:Nn ... [302](#), [22254](#)
 - \intarray_gset_rand:Nnn ... [302](#), [22254](#)
 - \intarray_gzero:N .. [237](#), [21934](#), [22143](#)
 - \intarray_item:Nn
..... [237](#), [345](#), [954](#), [958](#), [960](#),
[14153](#), [14158](#), [14192](#), [14221](#), [14229](#),
[21945](#), [21983](#), [22153](#), [22169](#), [28930](#)
 - \intarray_log:N [237](#), [22239](#)
 - \intarray_new:Nn ... [236](#), [951](#), [957](#),
[960](#), [6669](#), [6670](#), [7481](#), [7482](#), [7483](#),
[7484](#), [7485](#), [14141](#), [14150](#), [21830](#),
[22057](#), [28555](#), [28556](#), [28557](#), [28728](#)
 - \intarray_rand_item:N
..... [237](#), [21982](#), [22168](#)
 - \intarray_show:N .. [237](#), [955](#), [960](#), [22239](#)
 - \intarray_to_clist:N [302](#), [21995](#), [22186](#)
- intarray internal commands:
 - __intarray:w [21824](#), [21835](#)
 - \l__intarray_bad_index_int
..... [21821](#), [21929](#), [21977](#)
 - __intarray_bounds:NNnTF
..... [22083](#), [22113](#), [22164](#)
 - __intarray_bounds_error:NNnw .. [22083](#)
 - __intarray_const_from_clist:nN ..
..... [22171](#)
 - __intarray_count:w
.. [22053](#), [22069](#), [22079](#), [22177](#), [22198](#)
 - __intarray_entry:w
..... [22053](#), [22102](#), [22149](#), [22154](#)
 - \g__intarray_font_int
..... [22056](#), [22060](#), [22062](#)
 - __intarray_gset:Nnn [22101](#)
 - __intarray_gset:Nww .. [22105](#), [22111](#)
 - __intarray_gset:w [21901](#), [21923](#)
 - __intarray_gset:wTF .. [21901](#), [21926](#)
 - __intarray_gset_all_same:Nn [22254](#)
 - __intarray_gset_count:Nw
..... [950](#), [21819](#), [21840](#), [21885](#)
 - __intarray_gset_overflow:Nnn .. [22101](#)
 - __intarray_gset_overflow:NNnn ..
..... [22125](#), [22133](#), [22137](#)
 - __intarray_gset_overflow_-
test:nw
..... [956](#), [960](#), [962](#), [22047](#), [22115](#),
[22122](#), [22130](#), [22183](#), [22273](#), [22280](#)
 - __intarray_gset_rand:Nnn [22254](#)
 - __intarray_gset_rand_auxi:Nnnn ..
..... [22254](#)
 - __intarray_gset_rand_auxii:Nnnn [22254](#)
 - __intarray_gset_rand_auxiii:Nnnn [22254](#)
 - __intarray_gset_range:nNw ... [22021](#)
 - __intarray_gset_range:Nw [22224](#)
 - __intarray_gset_range:w [22024](#)
 - __intarray_item:Nn [22153](#)
 - __intarray_item:Nw ... [22157](#), [22162](#)
 - __intarray_item:w [21945](#)
 - __intarray_item:wTF [21945](#)
 - \l__intarray_loop_int [21818](#), [21988](#),
[21991](#), [21992](#), [22145](#), [22148](#), [22149](#),
[22174](#), [22177](#), [22182](#), [22184](#), [22226](#),
[22234](#), [22235](#), [22305](#), [22308](#), [22309](#)
 - __intarray_new:N
..... [21830](#), [21987](#), [22057](#), [22173](#)
 - __intarray_range_to_clist:w .. [22006](#)
 - __intarray_range_to_clist:ww .. [22205](#)
 - __intarray_show:NN
..... [22239](#), [22241](#), [22243](#)

- __intarray_signed_max_dim:n ... [22081](#), [22140](#), [22141](#)
- \c__intarray_sp_dim ... [22055](#), [22062](#), [22102](#)
- __intarray_table ... [21859](#)
- \g__intarray_table_int ... [21821](#), [21834](#), [21835](#)
- __intarray_to_clist:Nn ... [955](#), [21995](#), [22186](#), [22250](#)
- __intarray_to_clist:w . [21995](#), [22186](#)
- \interactionmode ... [543](#)
- \interlinepenalties ... [544](#)
- \interlinepenalty ... [324](#)
- ior commands:
 - \ior_close:N ... [86](#), [302](#), [10125](#), [10144](#), [11047](#), [11060](#), [11494](#), [29154](#), [29187](#), [29224](#), [29241](#)
 - \ior_get:NN . [87](#), [88](#), [302](#), [10201](#), [10281](#)
 - \ior_get:NNTF ... [87](#), [10201](#), [10202](#)
 - \ior_get_str:NN ... [36479](#)
 - \ior_get_term:nN ... [302](#), [10235](#)
 - \ior_if_eof:N ... [617](#)
 - \ior_if_eof:NNTF [90](#), [10185](#), [10207](#), [10227](#), [10267](#), [10286](#), [11057](#), [11071](#)
 - \ior_if_eof_p:N ... [90](#), [10185](#)
 - \ior_list_streams: ... [36481](#)
 - \ior_log:N ... [86](#), [10156](#)
 - \ior_log_list: ... [87](#), [10172](#), [36484](#)
 - \ior_log_streams: ... [36483](#)
 - \ior_map_break: [89](#), [10250](#), [10268](#), [10275](#), [10287](#), [10293](#), [29149](#), [29220](#)
 - \ior_map_break:n ... [89](#), [10250](#)
 - \ior_map_inline:Nn . [88](#), [10254](#), [11492](#)
 - \ior_map_variable:NNn [88](#), [10280](#), [29146](#)
 - \ior_new:N ... [86](#), [10094](#), [10096](#), [10097](#), [11073](#), [29095](#)
 - \ior_open:Nn ... [86](#), [646](#), [10098](#), [29110](#), [29155](#), [29188](#), [29240](#)
 - \ior_open:NnTF ... [86](#), [10099](#), [10102](#)
 - \ior_shell_open:Nn . [302](#), [11481](#), [36108](#)
 - \ior_show:N ... [86](#), [10156](#)
 - \ior_show_list: ... [87](#), [10172](#), [36482](#)
 - \ior_str_get:NN ... [87](#), [88](#), [302](#), [10214](#), [10283](#), [36480](#)
 - \ior_str_get:NNTF ... [88](#), [10214](#), [10215](#)
 - \ior_str_get_term:nN ... [302](#), [10235](#)
 - \ior_str_map_inline:Nn ... [88](#), [89](#), [10254](#), [29181](#), [29211](#)
 - \ior_str_map_variable:NNn . [89](#), [10280](#)
 - \c_term_ior ... [36445](#)
 - \g_tmpa_ior ... [93](#), [10096](#)
 - \g_tmppb_ior ... [93](#), [10096](#)
- ior internal commands:
 - \l__ior_file_name_tl ... [10101](#), [10104](#), [10106](#)
 - __ior_get:NN ... [10201](#), [10236](#), [10255](#)
 - __ior_get_term:NnN ... [10235](#)
 - \l__ior_internal_tl ... [10076](#), [10164](#), [10167](#), [10273](#), [10277](#)
 - __ior_list:N ... [10172](#)
 - __ior_map_inline:NNn ... [10254](#)
 - __ior_map_inline:NNNn ... [10254](#)
 - __ior_map_inline_loop:NNN ... [10254](#)
 - __ior_map_variable:NNNn ... [10280](#)
 - __ior_map_variable_loop:NNNn . [10280](#)
 - __ior_new:N ... [613](#), [10112](#), [10129](#)
 - __ior_new_aux:N ... [10116](#), [10120](#)
 - __ior_open_stream:Nn ... [10123](#)
 - __ior_shell_open:nN ... [36108](#)
 - __ior_show:NN ... [10156](#)
 - __ior_str_get:NN [10214](#), [10238](#), [10257](#)
 - \l__ior_stream_tl ... [10079](#), [10126](#), [10130](#), [10137](#)
 - \g__ior_streams_prop ... [614](#), [10080](#), [10138](#), [10149](#), [10164](#), [10179](#)
 - \g__ior_streams_seq ... [10078](#), [10126](#), [10150](#), [10151](#)
 - \c__ior_term_ior ... [10077](#), [10094](#), [10146](#), [10152](#), [10188](#), [10245](#)
 - \c__ior_term_noprompt_ior ... [10234](#), [10244](#)
- iow commands:
 - \iow_allow_break: ... [92](#), [302](#), [10466](#), [10508](#), [10513](#)
 - \iow_allow_break:n ... [624](#)
 - \iow_char:N ... [80](#), [91](#), [3684](#), [3687](#), [3688](#), [3712](#), [3713](#), [3720](#), [3721](#), [4722](#), [4723](#), [4730](#), [4732](#), [4734](#), [4736](#), [4738](#), [4740](#), [5383](#), [5384](#), [6100](#), [6107](#), [6108](#), [6109](#), [6233](#), [8051](#), [8054](#), [8055](#), [8060](#), [8094](#), [8103](#), [8107](#), [8112](#), [8132](#), [8134](#), [8135](#), [8137](#), [8140](#), [8142](#), [8147](#), [8149](#), [8151](#), [8156](#), [8160](#), [8163](#), [8164](#), [8167](#), [8169](#), [8173](#), [8175](#), [8181](#), [8183](#), [8187](#), [8189](#), [8193](#), [8198](#), [8200](#), [8242](#), [8244](#), [8249](#), [8251](#), [8257](#), [8262](#), [8267](#), [8271](#), [8281](#), [8284](#), [8288](#), [8289](#), [8293](#), [8301](#), [8372](#), [9765](#), [9768](#), [9769](#), [9801](#), [9829](#), [9963](#), [10427](#), [11555](#), [11557](#), [11558](#), [11559](#), [14261](#), [26742](#), [29056](#), [29058](#), [29059](#), [29062](#), [29064](#), [29065](#), [29068](#), [29070](#), [29071](#), [29072](#), [29076](#), [29083](#)
 - \iow_close:N ... [86](#), [10340](#), [10358](#)
 - \iow_indent:n ... [92](#), [625](#), [626](#), [8328](#), [9728](#), [9863](#), [9925](#), [9933](#), [9941](#), [10477](#), [10509](#), [10514](#), [14259](#), [14584](#)

- 14772, 22857, 22869, 35448, 35477,
35499, 35515, 35524, 35533, 35548
- \l_iow_line_count_int
..... 92, 441, 626, 4162,
4166, 9280, 10428, 10518, 10523, 10561
- \iow_list_streams: 36485
- \iow_log:N 86, 10370
- \iow_log:n 90,
1889, 9489, 9496, 10422, 13050, 36506
- \iow_log_list: 87, 10386, 36488
- \iow_log_streams: 36487
- \iow_new:N ... 86, 10329, 10331, 10332
- \iow_newline:
... 80, 90–92, 346, 592, 622, 699,
4112, 6155, 9302, 10426, 10506,
10515, 10521, 11410, 33855, 33856,
33857, 35819, 35821, 35824, 35831
- \iow_now:Nn 90, 91, 8925,
10416, 10422, 10423, 10424, 10425
- \iow_open:Nn 86, 10336
- \iow_shipout:Nn 90, 91, 622, 8957, 10401
- \iow_shipout_x:Nn . 90, 91, 622, 10398
- \iow_show:N 86, 10370
- \iow_show_list: 87, 10386, 36486
- \iow_term:n 90, 302, 1889, 8362, 9314,
9479, 9484, 9502, 9528, 10422, 36508
- \iow_wrap:nnnN .. 90, 92, 302, 626,
699, 1359, 9278, 9281, 9293, 9460,
9494, 9500, 9507, 10469, 10475,
10480, 10492, 10495, 13035, 13050
- \c_log_iow
. 93, 618, 10300, 10360, 10422, 10423
- \c_term_iow 93, 618, 10300,
10329, 10360, 10366, 10424, 10425
- \g_tmpa_iow 93, 10331
- \g_tmpb_iow 93, 10331
- iow internal commands:
- __iow_allow_break: 624, 10466, 10508
- __iow_allow_break_error:
..... 624, 10466, 10513
- \l_iow_file_name_tl
..... 10335, 10338, 10342, 10346
- __iow_indent:n ... 625, 10477, 10509
- __iow_indent_error:n
..... 625, 10477, 10514
- \l_iow_indent_int 10445,
10559, 10577, 10689, 10706, 10714
- \l_iow_indent_tl .. 10445, 10560,
10576, 10688, 10707, 10715, 10716
- \l_iow_internal_tl
..... 10299, 10378, 10381
- \l_iow_line_break_bool
10449, 10555, 10683, 10697, 10705,
10713, 10721, 10723, 10728, 10730
- \l__iow_line_part_tl
..... 628–630, 10447, 10557,
10569, 10590, 10648, 10651, 10682,
10696, 10698, 10704, 10712, 10735
- \l_iow_line_target_int
..... 631, 10431, 10517,
10519, 10522, 10684, 10689, 10724
- \l__iow_line_tl 10447, 10556, 10573,
10663, 10679, 10695, 10696, 10704,
10712, 10734, 10735, 10740, 10742
- __iow_list:N 10386
- __iow_new:N 10333, 10344
- \l_iow_newline_tl 10430,
10515, 10516, 10518, 10521, 10739
- \l__iow_one_indent_int
..... 10432, 10706, 10714
- \l_iow_one_indent_tl
..... 623, 10432, 10707
- __iow_open_stream:Nn 10336
- __iow_set_indent:n 623, 10432
- __iow_show:NN 10370
- \l_iow_stream_tl
..... 10310, 10341, 10345, 10352
- \g__iow_streams_prop
620, 10311, 10353, 10363, 10378, 10393
- \g__iow_streams_seq
..... 10309, 10341, 10364, 10365
- __iow_tmp:w 629, 10563,
10587, 10644, 10676, 10744, 10752
- __iow_unindent:w .. 623, 10432, 10716
- __iow_use_i_delimit_by_s-
stop:nw 10327, 10548
- __iow_with:nNnn 10404
- __iow_wrap_allow_break:n 10693
- \c_iow_wrap_allow_break_marker_-
tl 10451, 10471
- __iow_wrap_break:w ... 10630, 10644
- __iow_wrap_break_end:w .. 629, 10644
- __iow_wrap_break_first:w 10644
- __iow_wrap_break_loop:w 10644
- __iow_wrap_break_none:w 10644
- __iow_wrap_chunk:nw 10561, 10563,
10699, 10700, 10708, 10717, 10724
- __iow_wrap_do: 10525, 10530
- __iow_wrap_end:n 10719
- __iow_wrap_end_chunk:w
..... 627, 10581, 10588, 10680
- \c__iow_wrap_end_marker_tl
..... 10451, 10535
- __iow_wrap_fix_newline:w 10530
- __iow_wrap_indent:n 10702
- \c__iow_wrap_indent_marker_tl ...
..... 10451, 10485

- __iow_wrap_line:nw
 627, 630, 10575, 10579, 10588, 10687
- __iow_wrap_line_aux:Nw 10588
- __iow_wrap_line_end:NnnnnnnN 10588
- __iow_wrap_line_end:nw
 629, 10588, 10664, 10665, 10674
- __iow_wrap_line_loop:w 10588
- __iow_wrap_line_seven:nnnnnn 10588
- \c__iow_wrap_marker_tl
 624, 627, 10451, 10587
- __iow_wrap_newline:n 10719
- \c__iow_wrap_newline_marker_tl ..
 626, 10451, 10550
- __iow_wrap_next:nw
 10563, 10642, 10684
- __iow_wrap_next_line:w 10636, 10677
- __iow_wrap_start:w 10530
- __iow_wrap_store_do:n
 10635, 10722, 10729, 10732
- \l__iow_wrap_tl . 626, 631, 10450,
 10512, 10527, 10532, 10534, 10537,
 10539, 10542, 10558, 10736, 10738
- __iow_wrap_trim:N
 631, 10665, 10696, 10722, 10729, 10744
- __iow_wrap_trim:w 10744
- __iow_wrap_trim_aux:w 10744
- __iow_wrap_unindent:n 10702
- \c__iow_wrap_unindent_marker_tl .
 10451, 10487
- \itshape 31579
- J**
- \j 31306, 31667, 31862, 31941
- \jcharwidowpenalty 1130
- \jfam 1131
- \jfont 1132
- \jis 1133
- \jobname 325
- K**
- \k 29389, 31740, 31815,
 31816, 31833, 31834, 31856, 31857,
 31858, 31913, 31914, 31939, 31940
- \kanjiskip 1134
- \kansuji 1135
- \kansujichar 1136
- \kcatcode 1137
- \kchar 1170
- \kchardef 1171
- \kern 326
- kernel internal commands:
 - __kernel_backend_align_begin: . 350
 - __kernel_backend_align_end: .. 350
 - \g__kernel_backend_header_bool . 350
 - __kernel_backend_literal:n ... 349
 - __kernel_backend_literal_pdf:n 349
 - __kernel_backend_literal_-
 postscript:n 349
 - __kernel_backend_literal_svg:n 349
 - __kernel_backend_matrix:n 350
 - __kernel_backend_postscript:n . 350
 - __kernel_backend_scope_begin: . 350
 - __kernel_backend_scope_end: .. 350
 - __kernel_chk_cs_exist:N 343
 - __kernel_chk_defined:NTF
 ... 343, 2149, 2168, 8442, 10162,
 10376, 13020, 13053, 17765, 22245
 - __kernel_chk_expr:nNnN 343
 - __kernel_chk_if_free_cs:N
 589, 863, 1893, 1908, 1956,
 8383, 11913, 11919, 11924, 15912,
 16225, 17020, 17040, 18961, 18963,
 18973, 19547, 19943, 20286, 20377,
 21833, 22059, 28717, 28733, 31974
 - __kernel_chk_tl_type:NnnTF 343,
 802, 850, 893, 7374, 13051, 13721,
 13728, 16892, 18458, 19913, 24581
 - \l__kernel_color_stack_int 350
 - __kernel_cs_parm_from_arg_-
 count:nnTF .. 344, 1615, 1974, 2021
 - __kernel_dependency_version_-
 check:Nn 344, 11460
 - __kernel_dependency_version_-
 check:nn 344, 11460
 - __kernel_deprecation_code:nn ...
 344,
 1374, 1565, 36324, 36350, 36357, 36358
 - __kernel_deprecation_error:Nnn .
 1374, 36327, 36360
 - __kernel_exp_not:w ... 344, 389,
 686, 888, 2569, 2571, 2575, 2579,
 2582, 2585, 2590, 11920, 11951,
 11952, 11959, 11960, 11974, 11976,
 11980, 11982, 11996, 12001, 12007,
 12008, 12012, 12016, 12021, 12027,
 12028, 12032, 12042, 12046, 12052,
 12053, 12057, 12059, 12063, 12069,
 12070, 12074, 12233, 12572, 12577,
 12633, 12638, 12645, 12670, 12863,
 13026, 17142, 19713, 19721, 19728,
 19733, 20458, 29423, 29798, 31350
 - \l__kernel_expl_bool
 142, 145, 160, 174, 1415
 - \c__kernel_expl_date_tl
 652, 1415, 11464, 11467, 11507, 11511
 - __kernel_file_input_pop: 344, 11291
 - __kernel_file_input_push:n
 344, 11291

- __kernel_file_missing:n [344](#), [10099](#), [11286](#), [11295](#)
- __kernel_file_name_quote:n [1367](#),
[10142](#), [10355](#), [10872](#), [10911](#), [11306](#)
- __kernel_file_name_sanitiz:n ..
..... [344](#), [647](#), [10339](#),
[10798](#), [10925](#), [11034](#), [11289](#), [11345](#)
- __kernel_group_show:NN [2190](#)
- __kernel_if_debug:TF .. [1552](#), [36338](#)
- __kernel_int_add:nnn
..... [345](#), [17011](#), [28461](#)
- __kernel_intarray_gset:Nnn
.. [345](#), [953](#), [955](#), [958](#), [6709](#), [6816](#),
[6819](#), [7510](#), [7582](#), [7584](#), [7590](#), [7598](#),
[7600](#), [7603](#), [7724](#), [7726](#), [7730](#), [7732](#),
[7744](#), [7747](#), [21921](#), [21992](#), [22064](#),
[22076](#), [22101](#), [22184](#), [22234](#), [22309](#),
[28625](#), [28626](#), [28628](#), [28632](#), [28633](#),
[28634](#), [28736](#), [28737](#), [28771](#), [28774](#)
- __kernel_intarray_gset_range_-
from_clist:Nnn
..... [345](#), [6879](#), [22021](#), [22224](#)
- __kernel_intarray_item:Nn
..... [345](#), [954](#), [959](#), [1141](#),
[4442](#), [6827](#), [6854](#), [6938](#), [6939](#), [6963](#),
[6964](#), [6972](#), [6979](#), [7036](#), [7040](#), [7059](#),
[7587](#), [7777](#), [7996](#), [21945](#), [22153](#),
[22201](#), [22220](#), [26722](#), [26728](#), [26731](#),
[26734](#), [27467](#), [27470](#), [27473](#), [27476](#),
[27479](#), [27482](#), [27485](#), [27488](#), [27491](#),
[28674](#), [28675](#), [28676](#), [28823](#), [28826](#)
- __kernel_intarray_range_to_-
clist:Nnn . [345](#), [6808](#), [22006](#), [22205](#)
- __kernel_ior_open:Nn . [345](#), [1367](#),
[10106](#), [10123](#), [11055](#), [11070](#), [36121](#)
- __kernel_iow_with:Nnn
.. [346](#), [592](#), [622](#), [699](#), [9315](#), [9317](#),
[9530](#), [9532](#), [10404](#), [10418](#), [13039](#), [13041](#)
- __kernel_kern:n [346](#), [1415](#),
[31970](#), [32342](#), [32632](#), [32641](#), [33206](#),
[33464](#), [33469](#), [33551](#), [33552](#), [33830](#),
[33831](#), [35711](#), [35713](#), [35762](#), [35764](#)
- \l__kernel_keyval_allow_blank_-
keys_bool [19609](#),
[19617](#), [19627](#), [19629](#), [20450](#), [20623](#)
- __kernel_msg_error:nnn [9678](#)
- __kernel_msg_error:nnnn [9678](#)
- __kernel_msg_error:nnnnn [9678](#)
- __kernel_msg_expandable_-
error:nnn [9690](#)
- __kernel_msg_expandable_-
error:nnnn [9690](#)
- __kernel_msg_info:nnnn [9678](#)
- __kernel_msg_new:nnn ... [9674](#), [9896](#)
- __kernel_msg_new:nnnn [9674](#)
- __kernel_msg_warning:nnn [9678](#)
- __kernel_msg_warning:nnnn ... [9678](#)
- __kernel_patch_deprecation:nnNNpn
..... [1374](#),
[36320](#), [36379](#), [36384](#), [36592](#), [36594](#),
[36596](#), [36598](#), [36600](#), [36602](#), [36604](#),
[36607](#), [36609](#), [36611](#), [36619](#), [36622](#),
[36625](#), [36628](#), [36631](#), [36634](#), [36637](#),
[36639](#), [36641](#), [36643](#), [36645](#), [36647](#),
[36649](#), [36651](#), [36662](#), [36668](#), [36674](#)
- __kernel_prefix_arg_replacement:wN
..... [2211](#)
- \g__kernel_prg_map_int
..... [346](#), [439](#), [683](#), [816](#),
[902](#), [1415](#), [4056](#), [4058](#), [4078](#), [4188](#),
[4190](#), [4206](#), [8685](#), [10260](#), [10262](#),
[10269](#), [12469](#), [12471](#), [12473](#), [12478](#),
[13332](#), [13334](#), [13338](#), [13343](#), [16689](#),
[16690](#), [16696](#), [16697](#), [16747](#), [16749](#),
[16751](#), [16753](#), [17330](#), [17333](#), [17341](#),
[17344](#), [17355](#), [18212](#), [18214](#), [18216](#),
[18221](#), [19868](#), [19869](#), [19874](#), [19876](#),
[20196](#), [20199](#), [20203](#), [20206](#), [20217](#),
[24881](#), [24884](#), [24888](#), [24891](#), [24902](#)
- __kernel_primitive:NN
..... [317](#), [182](#), [187](#),
[188](#), [189](#), [190](#), [191](#), [192](#), [193](#), [194](#),
[195](#), [196](#), [197](#), [198](#), [199](#), [200](#), [201](#),
[202](#), [203](#), [204](#), [205](#), [206](#), [207](#), [208](#),
[209](#), [210](#), [211](#), [212](#), [213](#), [214](#), [215](#),
[216](#), [217](#), [218](#), [219](#), [220](#), [221](#), [222](#),
[223](#), [224](#), [225](#), [226](#), [227](#), [228](#), [229](#),
[230](#), [231](#), [232](#), [233](#), [234](#), [235](#), [236](#),
[237](#), [238](#), [239](#), [240](#), [241](#), [242](#), [243](#),
[244](#), [245](#), [246](#), [247](#), [248](#), [249](#), [250](#),
[251](#), [252](#), [253](#), [254](#), [255](#), [256](#), [257](#),
[258](#), [259](#), [260](#), [261](#), [262](#), [263](#), [264](#),
[265](#), [266](#), [267](#), [268](#), [269](#), [270](#), [271](#),
[272](#), [273](#), [274](#), [275](#), [276](#), [277](#), [278](#),
[279](#), [280](#), [281](#), [282](#), [283](#), [284](#), [285](#),
[286](#), [287](#), [288](#), [289](#), [290](#), [291](#), [292](#),
[293](#), [294](#), [295](#), [296](#), [297](#), [298](#), [299](#),
[300](#), [301](#), [302](#), [303](#), [304](#), [305](#), [306](#),
[307](#), [308](#), [309](#), [310](#), [311](#), [312](#), [313](#),
[314](#), [315](#), [316](#), [317](#), [318](#), [319](#), [320](#),
[321](#), [322](#), [323](#), [324](#), [325](#), [326](#), [327](#),
[328](#), [329](#), [330](#), [331](#), [332](#), [333](#), [334](#),
[335](#), [336](#), [337](#), [338](#), [339](#), [340](#), [341](#),
[342](#), [343](#), [344](#), [345](#), [346](#), [347](#), [348](#),
[349](#), [350](#), [351](#), [352](#), [353](#), [354](#), [355](#),
[356](#), [357](#), [358](#), [359](#), [360](#), [361](#), [362](#),
[363](#), [364](#), [365](#), [366](#), [367](#), [368](#), [369](#),
[370](#), [371](#), [372](#), [373](#), [374](#), [375](#), [376](#),

377, 378, 379, 380, 381, 382, 383,
384, 385, 386, 387, 388, 389, 390,
391, 392, 393, 394, 395, 396, 397,
398, 399, 400, 401, 402, 403, 404,
405, 406, 407, 408, 409, 410, 411,
412, 413, 414, 415, 416, 417, 418,
419, 420, 421, 422, 423, 424, 425,
426, 427, 428, 429, 430, 431, 432,
433, 434, 435, 436, 437, 438, 439,
440, 441, 442, 443, 444, 445, 446,
447, 448, 449, 450, 451, 452, 453,
454, 455, 456, 457, 458, 459, 460,
461, 462, 463, 464, 465, 466, 467,
468, 469, 470, 471, 472, 473, 474,
475, 476, 477, 478, 479, 480, 481,
482, 483, 484, 485, 486, 487, 488,
489, 490, 491, 492, 493, 494, 495,
496, 497, 498, 499, 500, 501, 502,
503, 504, 505, 506, 507, 508, 509,
510, 511, 512, 513, 514, 515, 516,
517, 518, 519, 520, 521, 522, 523,
524, 525, 526, 527, 528, 529, 530,
531, 532, 533, 534, 535, 536, 537,
538, 539, 540, 541, 542, 543, 544,
545, 546, 547, 548, 549, 550, 551,
552, 553, 554, 555, 556, 557, 558,
559, 560, 561, 562, 563, 564, 565,
566, 567, 568, 569, 570, 571, 572,
573, 574, 575, 576, 577, 578, 579,
580, 581, 582, 583, 584, 585, 586,
587, 588, 589, 590, 591, 592, 593,
594, 595, 596, 597, 598, 599, 600,
601, 602, 604, 605, 606, 607, 608,
609, 610, 612, 613, 614, 615, 616,
617, 618, 619, 620, 621, 622, 623,
624, 625, 626, 627, 628, 629, 630,
631, 632, 633, 634, 635, 636, 637,
638, 639, 640, 641, 642, 643, 644,
645, 646, 647, 648, 649, 650, 651,
652, 653, 654, 655, 656, 657, 658,
659, 660, 661, 662, 663, 664, 665,
666, 667, 668, 669, 670, 671, 672,
673, 674, 675, 676, 677, 678, 679,
680, 681, 682, 683, 684, 685, 686,
687, 688, 689, 690, 691, 692, 693,
694, 699, 708, 709, 710, 711, 712,
713, 715, 716, 717, 718, 719, 720,
721, 722, 723, 724, 725, 727, 729,
731, 732, 733, 735, 736, 737, 738,
739, 740, 742, 744, 745, 747, 749,
750, 751, 752, 753, 754, 755, 756,
757, 758, 759, 760, 761, 762, 763,
764, 765, 766, 767, 768, 769, 770,
771, 772, 773, 774, 775, 776, 777,
778, 779, 780, 781, 782, 783, 784,
785, 786, 787, 789, 790, 792, 793,
794, 795, 796, 797, 798, 799, 800,
801, 803, 804, 805, 806, 807, 808,
809, 810, 811, 812, 813, 814, 815,
816, 817, 818, 819, 820, 821, 822,
823, 824, 825, 826, 827, 828, 829,
830, 831, 832, 833, 834, 835, 836,
837, 838, 839, 840, 841, 842, 843,
844, 845, 846, 847, 848, 849, 850,
851, 852, 853, 854, 855, 856, 857,
858, 859, 860, 861, 862, 863, 864,
865, 866, 867, 868, 869, 870, 871,
872, 873, 874, 875, 876, 877, 878,
879, 880, 881, 882, 883, 884, 885,
886, 887, 888, 889, 890, 891, 892,
893, 894, 895, 896, 897, 898, 899,
900, 902, 903, 904, 905, 906, 907,
908, 909, 910, 911, 912, 913, 914,
915, 916, 918, 920, 922, 923, 924,
925, 926, 927, 928, 929, 930, 931,
932, 933, 934, 935, 936, 937, 938,
939, 940, 941, 942, 943, 944, 945,
946, 947, 948, 949, 950, 951, 952,
953, 954, 955, 956, 957, 958, 959,
960, 961, 962, 963, 964, 965, 967,
969, 970, 971, 972, 974, 975, 976,
977, 979, 980, 982, 984, 985, 986,
987, 988, 990, 992, 993, 994, 995,
997, 998, 999, 1000, 1001, 1002,
1003, 1004, 1005, 1006, 1007, 1008,
1009, 1010, 1011, 1012, 1013, 1014,
1015, 1016, 1017, 1018, 1019, 1020,
1021, 1022, 1023, 1024, 1025, 1026,
1027, 1028, 1029, 1030, 1031, 1032,
1033, 1034, 1036, 1038, 1039, 1041,
1043, 1044, 1045, 1046, 1048, 1049,
1050, 1052, 1054, 1056, 1057, 1058,
1059, 1060, 1061, 1062, 1063, 1064,
1065, 1066, 1067, 1069, 1071, 1072,
1073, 1074, 1075, 1076, 1077, 1078,
1079, 1080, 1081, 1082, 1083, 1084,
1085, 1086, 1087, 1089, 1091, 1092,
1093, 1094, 1095, 1096, 1097, 1098,
1099, 1100, 1101, 1102, 1103, 1104,
1105, 1106, 1107, 1108, 1109, 1110,
1111, 1112, 1113, 1114, 1115, 1116,
1117, 1118, 1119, 1120, 1121, 1122,
1123, 1124, 1125, 1126, 1127, 1128,
1129, 1130, 1131, 1132, 1133, 1134,
1135, 1136, 1137, 1138, 1139, 1140,
1141, 1142, 1143, 1144, 1145, 1146,
1147, 1148, 1149, 1150, 1152, 1154,
1155, 1156, 1157, 1158, 1160, 1161,

- 1162, 1163, 1164, 1165, 1166, 1167,
- 1168, 1169, 1170, 1171, 1172, 1173,
- 1174, 1175, 1176, 1177, 1178, 1179,
- 1180, 1181, 1182, 1183, 1184, 1185
- _kernel_quark_new_conditional:Nn
 - 348, 4485, 10793,
 - 12089, 16012, 18491, 20711, 29249
- _kernel_quark_new_test:N . 347,
 - 776, 777, 779, 780, 8419, 10796,
 - 10797, 12088, 13090, 13091, 16012,
 - 16928, 16929, 19543, 29254, 31347
- _kernel_randint:n
 - 348, 962, 1169, 1173,
 - 22299, 28268, 28280, 28438, 28523
- _kernel_randint:nn
 - 348, 962, 22295, 28442, 28446, 28521
- \c_kernel_randint_max_int
 - 1172, 1415, 22292, 28267, 28436, 28520
- _kernel_register_log:N
 - 348, 2158, 17717, 20272,
 - 20273, 20365, 20366, 20433, 20434
- _kernel_register_show:N .. 348,
 - 698, 2158, 17713, 20268, 20361, 20429
- _kernel_register_show_aux:NN 2158
- _kernel_register_show_aux:nnN 2158
- _kernel_show:NN 2176
- _kernel_str_to_other:n
 - 349, 707, 710, 714, 13388, 13440, 13501
- _kernel_str_to_other_fast:n ...
 - 349, 4696, 5906, 10438,
 - 10534, 13339, 13359, 13411, 13935
- _kernel_str_to_other_fast_-
 - loop:w 13411
- _kernel_sys_configuration_-
 - load:n 8790, 8851, 36615
- _kernel_tl_gset:Nn
 - 349, 544, 665, 3420, 3945,
 - 4695, 5904, 7619, 7630, 7681, 7836,
 - 9117, 11909, 11957, 11980, 11982,
 - 11984, 12015, 12020, 12025, 12032,
 - 12059, 12062, 12067, 12074, 12192,
 - 12196, 12586, 12883, 13146, 13150,
 - 13869, 13885, 13935, 14063, 14115,
 - 14126, 14284, 14333, 14386, 14392,
 - 14625, 14825, 14982, 16261, 16266,
 - 16284, 16288, 16344, 16384, 16410,
 - 16567, 16610, 16776, 16786, 17880,
 - 17907, 17926, 17969, 18005, 18060,
 - 18099, 19768, 19791, 24545, 35925,
 - 35935, 35950, 35956, 36030, 36225
- _kernel_tl_set:Nn 349, 4585, 5474,
 - 5479, 5750, 5819, 7771, 7805, 10130,
 - 10338, 10345, 10437, 10512, 10515,
 - 10516, 10532, 10537, 10695, 10715,
 - 10734, 10736, 11020, 11033, 11068,
 - 11175, 11209, 11909, 11949, 11974,
 - 11976, 11978, 11995, 12000, 12005,
 - 12012, 12042, 12045, 12050, 12057,
 - 12190, 12194, 12584, 12881, 13144,
 - 13148, 13824, 16251, 16256, 16282,
 - 16286, 16308, 16336, 16382, 16408,
 - 16523, 16548, 16565, 16579, 16607,
 - 16774, 16784, 17878, 17905, 17924,
 - 17967, 18003, 18058, 18097, 18682,
 - 19767, 19789, 21321, 21363, 21429,
 - 21627, 24543, 35923, 35933, 35948,
 - 35954, 36028, 36036, 36205, 36220
- _kernel_tl_to_str:w
 - 349, 680, 681, 1442, 12291,
 - 12388, 12519, 13314, 13382, 16010
- keys commands:
 - \l_keys_choice_int . 223, 226, 228,
 - 229, 20683, 20861, 20864, 20869, 20870
 - \l_keys_choice_tl
 - 223, 226, 228, 229, 20683, 20868
 - \keys_define:nn 222, 9861, 20712
 - \keys_if_choice_exist:nnnTF
 - 233, 21736
 - \keys_if_choice_exist_p:nnn
 - 233, 21736
 - \keys_if_exist:nnTF
 - 233, 947, 21728, 21753
 - \keys_if_exist_p:nn 233, 21728
 - \l_keys_key_str 230, 20686, 20818,
 - 20943, 21438, 21439, 21538, 21542,
 - 21567, 21570, 21571, 21607, 21664
 - \l_keys_key_tl 20686, 21439
 - \keys_log:nn 233, 21744
 - \l_keys_path_str 230, 20691, 20740,
 - 20759, 20766, 20774, 20775, 20776,
 - 20793, 20811, 20813, 20815, 20830,
 - 20833, 20837, 20845, 20847, 20848,
 - 20851, 20866, 20880, 20890, 20895,
 - 20905, 20909, 20916, 20921, 20925,
 - 20926, 20936, 20938, 20940, 20954,
 - 20960, 20964, 20966, 20981, 20992,
 - 20998, 21002, 21018, 21027, 21069,
 - 21080, 21121, 21429, 21437, 21475,
 - 21478, 21517, 21521, 21526, 21535,
 - 21549, 21551, 21552, 21556, 21564,
 - 21587, 21617, 21640, 21652, 21661
 - \l_keys_path_tl . 20691, 20776, 20837
 - \keys_set:nn 222,
 - 225, 230, 231, 20956, 20960, 21276
 - \keys_set_filter:nnn 232, 21346
 - \keys_set_filter:nnnN ... 232, 21346
 - \keys_set_filter:nnnnN ... 232, 21346
 - \keys_set_groups:nnn 232, 21346

- \keys_set_known:nn [231](#), [21305](#)
- \keys_set_known:nnN . [231](#), [939](#), [21305](#)
- \keys_set_known:nnnN [231](#), [21305](#)
- \keys_show:nn [233](#), [21744](#)
- \l_keys_usage_load_prop
 - [230](#), [20705](#), [21037](#), [21044](#), [21051](#)
- \l_keys_usage_preamble_prop
 - [230](#), [20705](#), [21039](#), [21046](#), [21053](#)
- \l_keys_value_tl [230](#), [20701](#), [21018](#),
 - [21520](#), [21524](#), [21530](#), [21541](#), [21552](#),
 - [21571](#), [21584](#), [21609](#), [21619](#), [21647](#)
- keys internal commands:
 - __keys_bool_set:Nn
 - .. [20801](#), [21095](#), [21097](#), [21099](#), [21101](#)
 - __keys_bool_set:Nnn [20801](#)
 - __keys_bool_set_inverse:Nn
 - .. [20801](#), [21103](#), [21105](#), [21107](#), [21109](#)
 - __keys_check_forbidden: [20985](#)
 - __keys_check_groups: . [21479](#), [21487](#)
 - __keys_check_required: [20985](#)
 - \c_keys_check_root_str .. [20676](#),
 - [20992](#), [20998](#), [21002](#), [21551](#), [21570](#)
 - __keys_choice_find:n . [20824](#), [21658](#)
 - __keys_choice_find:nn [21658](#)
 - __keys_choice_make:
 - .. [20810](#), [20823](#), [20855](#), [20935](#), [21111](#)
 - __keys_choice_make:N [20823](#)
 - __keys_choice_make_aux:N [20823](#)
 - __keys_choices_make:nn
 - .. [20854](#), [21113](#), [21115](#), [21117](#), [21119](#)
 - __keys_choices_make:Nnn [20854](#)
 - __keys_cmd_set:nn
 - [20811](#), [20813](#), [20815](#), [20847](#), [20848](#),
 - [20865](#), [20875](#), [20936](#), [20938](#), [20940](#),
 - [20954](#), [20960](#), [20966](#), [21080](#), [21121](#)
 - \c_keys_code_root_str
 - [944](#), [20676](#), [20876](#),
 - [20880](#), [20925](#), [21549](#), [21567](#), [21583](#),
 - [21597](#), [21669](#), [21731](#), [21740](#), [21759](#)
 - __keys_cs_set:NNpn
 - [20878](#), [21131](#), [21133](#), [21135](#),
 - [21137](#), [21139](#), [21141](#), [21143](#), [21145](#)
 - __keys_default_inherit: [21513](#)
 - \c_keys_default_root_str
 - [20676](#), [20890](#),
 - [20895](#), [21517](#), [21521](#), [21538](#), [21542](#)
 - __keys_default_set:n [20820](#), [20885](#),
 - [20945](#), [21147](#), [21149](#), [21151](#), [21153](#)
 - __keys_define:n [20717](#), [20721](#)
 - __keys_define:nn [20717](#), [20721](#)
 - __keys_define:nnn [20712](#)
 - __keys_define_aux:nn [20721](#)
 - __keys_define_code:n . [20735](#), [20784](#)
 - __keys_define_code:w [20784](#)
 - __keys_execute:
 - .. [21443](#), [21483](#), [21505](#), [21509](#), [21547](#)
 - __keys_execute:nn
 - [20926](#), [21547](#), [21670](#), [21671](#)
 - __keys_execute_inherit: [20922](#), [21547](#)
 - __keys_execute_unknown: . [943](#), [21547](#)
 - \l_keys_filtered_bool ... [20697](#),
 - [21281](#), [21288](#), [21289](#), [21332](#), [21338](#),
 - [21339](#), [21374](#), [21380](#), [21381](#), [21393](#),
 - [21400](#), [21401](#), [21482](#), [21503](#), [21508](#)
 - __keys_find_key_module:wNN
 - [20964](#), [21417](#)
 - __keys_find_key_module_auxi:Nw .
 - [21417](#)
 - __keys_find_key_module_auxii:Nw
 - [21417](#)
 - __keys_find_key_module_auxiii:Nn
 - [21417](#)
 - __keys_find_key_module_auxiiii:Nw
 - [21460](#), [21462](#)
 - __keys_find_key_module_auxiv:Nw
 - [21417](#)
 - \l_keys_groups_clist ... [20685](#),
 - [20902](#), [20903](#), [20910](#), [21477](#), [21492](#)
 - \c_keys_groups_root_str
 - .. [20676](#), [20905](#), [20909](#), [21475](#), [21478](#)
 - __keys_groups_set:n .. [20900](#), [21171](#)
 - __keys_inherit:n [20913](#), [21173](#)
 - \c_keys_inherit_root_str
 - [20676](#), [20916](#),
 - [20921](#), [21526](#), [21535](#), [21556](#), [21564](#)
 - \l_keys_inherit_str [20693](#),
 - [20924](#), [21436](#), [21569](#), [21660](#), [21664](#)
 - __keys_initialise:n
 - .. [20918](#), [21175](#), [21177](#), [21179](#), [21181](#)
 - __keys_legacy_if_inverse:nn . [20929](#)
 - __keys_legacy_if_inverse:nnnn [20929](#)
 - __keys_legacy_if_set:nn
 - [20929](#), [21191](#), [21193](#)
 - __keys_legacy_if_set:nnnn
 - [20930](#), [20932](#), [20933](#)
 - __keys_legacy_if_set_inverse:nn
 - [20931](#), [21195](#), [21197](#)
 - __keys_meta_make:n ... [20952](#), [21199](#)
 - __keys_meta_make:nn .. [20952](#), [21201](#)
 - \l_keys_module_str
 - [20688](#), [20713](#), [20716](#), [20718](#),
 - [20760](#), [20956](#), [21066](#), [21073](#), [21298](#),
 - [21301](#), [21303](#), [21420](#), [21425](#), [21435](#),
 - [21438](#), [21444](#), [21583](#), [21584](#), [21587](#)
 - __keys_multichoice_find:n
 - [20826](#), [21658](#)
 - __keys_multichoice_make:
 - [20823](#), [20857](#), [21203](#)

```

__keys_multichoices_make:nn ...
  .. 20854, 21205, 21207, 21209, 21211
\l__keys_no_value_bool .....
  ..... 20689, 20723,
  20728, 20786, 21015, 21024, 21419,
  21424, 21515, 21608, 21618, 21646
\l__keys_only_known_bool .....
  .... 20690, 21280, 21286, 21287,
  21331, 21336, 21337, 21373, 21378,
  21379, 21392, 21398, 21399, 21579
__keys_parent:n .....
  .... 20830, 20833, 20837, 20921,
  21526, 21535, 21556, 21564, 21675
__keys_parent_auxi:w ..... 21675
__keys_parent_auxii:w ..... 21675
__keys_parent_auxiii:n ..... 21675
__keys_parent_auxiv:w ..... 21675
__keys_prop_put:Nn .....
  .. 20961, 21221, 21223, 21225, 21227
__keys_property_find:n 20733, 20744
__keys_property_find_auxi:w . 20744
__keys_property_find_auxii:w . 20744
__keys_property_find_auxiii:w 20744
__keys_property_find_auxiv:w . 20744
__keys_property_find_err:w ....
  ..... 20749, 20757, 20778, 20779
\l__keys_property_str 20696, 20734,
  20737, 20740, 20746, 20747, 20773,
  20781, 20789, 20790, 20793, 20796
\c__keys_props_root_str .....
  20682, 20734, 20790, 20796, 21094,
  21096, 21098, 21100, 21102, 21104,
  21106, 21108, 21110, 21112, 21114,
  21116, 21118, 21120, 21122, 21124,
  21126, 21128, 21130, 21132, 21134,
  21136, 21138, 21140, 21142, 21144,
  21146, 21148, 21150, 21152, 21154,
  21156, 21158, 21160, 21162, 21164,
  21166, 21168, 21170, 21172, 21174,
  21176, 21178, 21180, 21182, 21184,
  21186, 21188, 21190, 21192, 21194,
  21196, 21198, 21200, 21202, 21204,
  21206, 21208, 21210, 21212, 21214,
  21216, 21218, 21220, 21222, 21224,
  21226, 21228, 21230, 21232, 21234,
  21236, 21238, 21240, 21242, 21244,
  21246, 21248, 21250, 21252, 21254,
  21256, 21258, 21260, 21262, 21264,
  21266, 21268, 21270, 21272, 21274
__keys_quark_if_no_value:NTF ...
  ..... 20711, 21603
__keys_quark_if_no_value_p:N . 20711
\l__keys_relative_tl .... 20694,
  21283, 21292, 21293, 21334, 21342,
  21343, 21376, 21384, 21385, 21395,
  21404, 21405, 21603, 21613, 21627,
  21628, 21632, 21633, 21641, 21653
\l__keys_selective_bool .....
  .... 20697, 21282, 21290, 21291,
  21333, 21340, 21341, 21375, 21382,
  21383, 21394, 21402, 21403, 21441
\l__keys_selective_seq .....
  .. 20699, 21410, 21413, 21415, 21490
__keys_set:nn .. 21276, 21335, 21414
__keys_set:nnn ..... 21276
__keys_set_filter:nnnn ..... 21346
__keys_set_filter:nnnnN .... 21346
__keys_set_keyval:n .. 21302, 21417
__keys_set_keyval:nn . 21302, 21417
__keys_set_keyval:nnn ..... 21417
__keys_set_known:nnn ..... 21305
__keys_set_known:nnnnN ..... 21305
__keys_set_selective: ..... 21417
__keys_set_selective:nnn .... 21346
__keys_set_selective:nnnn .... 21346
__keys_show:Nnn ..... 21744
__keys_store_unused: .....
  ..... 21484, 21504, 21510, 21547
__keys_store_unused:w .....
  ..... 21631, 21652, 21657
__keys_store_unused_aux: .... 21547
__keys_tmp:w ..... 21696, 21708
\l__keys_tmp_bool .....
  ..... 20702, 21489, 21496, 21501
\l__keys_tmpa_tl ... 20702, 20965,
  21066, 21067, 21071, 21072, 21074
\l__keys_tmpp_tl ..... 20702,
  20965, 20970, 21068, 21071, 21072
__keys_trim_spaces:n .....
  . 923, 20716, 20746, 20775, 20866,
  21301, 21433, 21628, 21669, 21670,
  21695, 21731, 21740, 21751, 21760
__keys_trim_spaces_auxi:w ... 21695
__keys_trim_spaces_auxii:w .. 21695
__keys_trim_spaces_auxiii:w . 21695
\c__keys_type_root_str .....
  ..... 20676, 20830, 20833, 20845
__keys_undefine: 20915, 20975, 21269
\l__keys_unused_clist .....
  . 938, 20700, 21308, 21314, 21319,
  21321, 21322, 21349, 21356, 21361,
  21363, 21364, 21605, 21615, 21643
__keys_usage:n ..... 21031, 21271
__keys_usage:NN ..... 21031
__keys_usage:w ..... 21031
__keys_value_or_default:n .....
  ..... 21440, 21513

```

- _keys_value_requirement:nn ... 20897, 20985, 21091, 21273, 21275
- _keys_variable_set:NnnN 21077, 21123, 21125, 21127, 21129, 21237, 21239, 21241, 21243, 21245, 21247, 21249, 21251, 21253, 21255, 21257, 21259, 21261, 21263, 21265, 21267
- _keys_variable_set_required:NnnN 21077, 21155, 21157, 21159, 21161, 21163, 21165, 21167, 21169, 21183, 21185, 21187, 21189, 21213, 21215, 21217, 21219, 21229, 21231, 21233, 21235
- keyval commands:
 - \keyval_parse:NNn 235, 919, 20454, 20717, 21302
 - \keyval_parse:nnn 234, 235, 913, 918, 19628, 20454
- keyval internal commands:
 - _keyval_blank_key_error:w 20595, 20604, 20619
 - _keyval_blank_true:w 20551, 20619
 - _keyval_clean_up_active:w 916, 20493, 20506, 20527, 20559, 20579
 - _keyval_clean_up_other:w 917, 20532, 20537, 20548
 - _keyval_end_loop_active:w 20480, 20573
 - _keyval_end_loop_other:w 917, 20490, 20573
 - _keyval_if_blank:w 20551, 20595, 20604, 20616
 - _keyval_if_empty:w 20616
 - _keyval_if_recursion_tail:w 20479, 20489, 20616
 - _keyval_key:nn 917, 20553, 20590, 20619
 - _keyval_loop_active:nnw 20461, 20471, 20477, 20580
 - _keyval_loop_other:nnw 914, 20481, 20487, 20563, 20571, 20583, 20600, 20609, 20620, 20621, 20626
 - _keyval_misplaced_equal_after_active_error:w 915, 20500, 20504, 20555
 - _keyval_misplaced_equal_in_split_error:w 20511, 20516, 20520, 20524, 20540, 20545, 20555
 - _keyval_pair:nnnn 916, 20526, 20547, 20590
 - _keyval_split_active:w 914, 915, 20483, 20491, 20510, 20575
 - _keyval_split_active_auxi:w 20492, 20497, 20528, 20578
 - _keyval_split_active_auxii:w 915, 20497, 20557
 - _keyval_split_active_auxiii:w 915, 20497
 - _keyval_split_active_auxiv:w 915, 20497
 - _keyval_split_active_auxv:w 20497
 - _keyval_split_other:w 20483, 20499, 20519, 20530, 20539
 - _keyval_split_other_auxi:w 916, 20531, 20534, 20549
 - _keyval_split_other_auxii:w 20534
 - _keyval_split_other_auxiii:w 916, 20534
 - _keyval_tmp:w 918, 20452, 20588, 20591, 20613, 20614, 20635, 20673
 - _keyval_trim:nN 20507, 20526, 20535, 20547, 20553, 20619, 20634
 - _keyval_trim_auxi:w 20634
 - _keyval_trim_auxii:w 20634
 - _keyval_trim_auxiii:w 20634
 - _keyval_trim_auxiv:w 20634
 - \kuten 1138, 1172
- L
- \L 29398, 31297, 31657
- \l 29398, 31297, 31669
- \label 29406, 29413, 31605
- \language 327
- \LARGE 31585
- \Large 31586
- \large 31589
- \lastallocatedtoks 3381
- \lastbox 328
- \lastkern 329
- \lastlinefit 545
- \lastnamedcs 831
- \lastnodechar 1139
- \lastnodesubtype 1140
- \lastnodetype 546
- \lastpenalty 330
- \lastsavedboxresourceindex 916
- \lastsavedimageresourceindex 918
- \lastsavedimageresourcepages 920
- \lastskip 331
- \lastxpos 922
- \lastypos 923
- \latelua 832
- \lateluafunction 833
- \lccode 85, 86, 332
- \leaders 333
- \left 334
- \leftghost 834
- \lefthyphenmin 335

- \leftmarginkern 687
- \leftskip 336
- legacy commands:
 - \legacy_if:nTF 100, 11880
 - .legacy_if_gset:n 225, 21190
 - \legacy_if_gset:nn 100, 11896
 - \legacy_if_gset_false:n 100, 11888, 11903
 - .legacy_if_gset_inverse:n 225, 21190
 - \legacy_if_gset_true:n 100, 11888, 11903
 - \legacy_if_p:n 100, 11880
 - .legacy_if_set:n 225, 21190
 - \legacy_if_set:nn 100, 11896
 - \legacy_if_set_false:n 100, 11888, 11898
 - .legacy_if_set_inverse:n 225, 21190
 - \legacy_if_set_true:n 100, 11888, 11898
- \leqno 337
- \let 22, 179, 180, 338
- \lcharcode 835
- \letterspacefont 688
- \limits 1325, 339
- \LineBreak 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 81, 83
- \linedir 836
- \linedirection 837
- \linepenalty 340
- \lineskip 341
- \lineskiplimit 342
- \linewidth 32805
- \ln 26847, 26850
- ln 252
- \localbrokenpenalty 838
- \localinterlinepenalty 839
- \lcalleftbox 844
- \lcalrightbox 845
- \loccount 10088, 10319
- \loctoks 3353, 3354, 3380
- logb 253
- \long 182, 343, 19146, 19150
- \LongText 55, 97
- \looseness 344
- \lower 345
- \lowercase 88, 346
- \lpcode 689
- ltx.utils 99, 11644
- ltx.utils.filedump 99, 11712
- ltx.utils.filemd5sum 99, 11733
- ltx.utils.filemoddate 99, 11742
- ltx.utils.filesize 99, 11795
- lua commands:
 - \lua_escape:n 99, 11603, 11605, 36490
 - \lua_escape_x:n 36489
 - \lua_now:n 98, 8724, 8733, 11604, 11605, 28924, 36492
 - \lua_now_x:n 36491
 - \lua_shipout:n 98, 11605
 - \lua_shipout_e:n 98, 11605, 36494
 - \lua_shipout_x:n 36493
- lua internal commands:
 - _lua_escape:n 11600, 11610
 - _lua_now:n 11600, 11605
 - _lua_shipout:n 11600, 11607
 - \luabytecode 840
 - \luabytecodecall 841
 - \luacopyinputnodes 842
 - \luaedef 843
 - luaedef 11805
 - \luaescapestring 846
 - \luafunction 847
 - \luafunctioncall 848
- luatex commands:
 - \luatex_if_engine:TF 36497, 36499, 36501
 - \luatex_if_engine_p: 36495
 - \luatexalignmark 1242
 - \luatexaligntab 1243
 - \luatexattribute 1244
 - \luatexattributedef 1245
 - \luatexbanner 849
 - \luatexbodydir 1281
 - \luatexboxdir 1282
 - \luatexcatcodetable 1246
 - \luatexclearmarks 1247
 - \luatexcrampeddisplaystyle 1248
 - \luatexcrampedscriptscriptstyle 1250
 - \luatexcrampedscriptstyle 1251
 - \luatexcrampedtextstyle 1252
 - \luatexfontid 1253
 - \luatexformatname 1254
 - \luatexgladers 1255
 - \luatexinitcatcodetable 1256
 - \luatexlatalua 1257
 - \luatexleftghost 1283
 - \luatexlocalbrokenpenalty 1284
 - \luatexlocalinterlinepenalty 1286
 - \luatexlcalleftbox 1287
 - \luatexlcalrightbox 1288
 - \luatexluaescapestring 1258
 - \luatexluafunction 1259
 - \luatexmathdir 1289
 - \luatexmathstyle 1260
 - \luatexnokerns 1261
 - \luatexnoligs 1262
 - \luatexoutputbox 1263
 - \luatexpagebottomoffset 1290

<code>\luatexpagedir</code>	1291	<code>\mathscriptsmode</code>	861
<code>\luatexpageheight</code>	1292	<code>\mathstyle</code>	864
<code>\luatexpageleftoffset</code>	1264	<code>\mathsurround</code>	362
<code>\luatexpagerightoffset</code>	1293	<code>\mathsurroundmode</code>	865
<code>\luatexpagetopoffset</code>	1265	<code>\mathsurroundskip</code>	866
<code>\luatexpagewidth</code>	1294	<code>max</code>	253
<code>\luatexpardir</code>	1295	<code>\maxdeadcycles</code>	363
<code>\luatexpostexhyphenchar</code>	1266	<code>\maxdepth</code>	364
<code>\luatexposthyphenchar</code>	1267	<code>md5.HEX</code>	11725
<code>\luatexpreeexhyphenchar</code>	1268	<code>\mdfivesum</code>	778
<code>\luatexprehyphenchar</code>	1269	<code>\mdseries</code>	31578
<code>\luatexrevision</code>	850	<code>\meaning</code>	365
<code>\luatexrightghost</code>	1296	<code>\medmuskip</code>	366
<code>\luatexsavecatcodetable</code>	1270	<code>\message</code>	367
<code>\luatexscantextokens</code>	1271	<code>\MessageBreak</code>	81
<code>\luatexsuppressfontnotfounderror</code>	1241, 1280	meta commands:	
<code>\luatexsuppressifcsnameerror</code>	1273	<code>.meta:n</code>	225, 21198
<code>\luatexsuppresslongerror</code>	1274	<code>.meta:nn</code>	225, 21200
<code>\luatexsuppressmathparerror</code>	1276	<code>\middle</code>	548
<code>\luatexsuppressoutererror</code>	1277	<code>min</code>	253
<code>\luatextextdir</code>	1297	<code>\c_minus_one</code>	36397
<code>\luatexUchar</code>	1278	<code>\mkern</code>	368
<code>\luatexversion</code>	27, 76, 851	<code>mm</code>	257
M			
<code>\mag</code>	347	mode commands:	
<code>\mark</code>	348	<code>\mode_if_horizontal:TF</code>	70, 8672
<code>\marks</code>	547	<code>\mode_if_horizontal_p:</code>	70, 8672
<code>\mathaccent</code>	349	<code>\mode_if_inner:TF</code>	70, 8674
<code>\mathbin</code>	350	<code>\mode_if_inner_p:</code>	70, 8674
<code>\mathchar</code>	351, 19145	<code>\mode_if_math:TF</code>	70, 8676
<code>\mathchardef</code>	352	<code>\mode_if_math_p:</code>	70, 8676
<code>\mathchoice</code>	353	<code>\mode_if_vertical:TF</code>	70, 8670
<code>\mathclose</code>	354	<code>\mode_if_vertical_p:</code>	70, 8670
<code>\mathcode</code>	355	<code>\mode_leave_vertical:</code>	28, 2258, 33633, 33694
<code>\mathcolor</code>	1324	<code>\month</code>	369, 1320, 8998
<code>\mathdelimitersmode</code>	852	<code>\moveleft</code>	370
<code>\mathdir</code>	853	<code>\moveright</code>	371
<code>\mathdirection</code>	854	msg commands:	
<code>\mathdisplayskipmode</code>	855	<code>\msg_critical:nn</code>	81, 97, 9437
<code>\matheqnogapstep</code>	856	<code>\msg_critical:nnn</code>	81, 9437
<code>\mathinner</code>	356	<code>\msg_critical:nnnn</code>	81, 9437
<code>\mathnolimitsmode</code>	857	<code>\msg_critical:nnnnn</code>	81, 9437
<code>\mathop</code>	357	<code>\msg_critical_text:n</code>	79, 9337, 9440
<code>\mathopen</code>	358	<code>\msg_error:nn</code>	81, 1870, 4995, 5028, 5076, 5079, 5552, 5823, 7238, 7322, 8783, 9445, 28871, 28916, 33079, 36112
<code>\mathoption</code>	858	<code>\msg_error:nnn</code>	81, 1555, 1560, 1628, 1683, 1734, 1739, 1870, 2066, 2154, 2928, 3202, 3678, 5034, 5257, 5651, 5664, 5703, 5736, 5850, 7011, 7018, 7230, 7336, 8871, 9026, 9445, 9543, 9685, 11186, 11288, 11631, 12205, 13159, 13902,
<code>\mathord</code>	359		
<code>\mathpenaltiesmode</code>	859		
<code>\mathpunct</code>	360		
<code>\mathrel</code>	361		
<code>\mathrulesfam</code>	860		
<code>\mathscriptboxmode</code>	862		
<code>\mathscriptcharmode</code>	863		

- 13961, 16017, 16041, 16045, 16190,
16444, 16475, 19624, 20782, 20817,
20942, 21008, 21026, 21843, 22072,
22316, 22339, 22737, 28550, 28889,
28947, 28953, 32111, 32701, 34077,
34134, 34421, 34652, 34688, 34798,
34807, 34841, 34854, 34869, 34874,
34944, 34963, 34976, 34993, 35003,
35011, 35363, 35376, 35399, 36118
- \msg_error:nnnn
 . 81, 1619, 1659, 1753, 1870, 1897,
 2023, 3013, 3222, 3245, 3541, 3548,
 5234, 5299, 5514, 7242, 7258, 8805,
 8824, 8840, 9179, 9445, 9569, 9687,
 10468, 11495, 13993, 16060, 18469,
 20739, 20792, 20836, 20850, 21017,
 21058, 21586, 21639, 22733, 32921
- \msg_error:nnnnn
 . 81, 2059, 3694, 7635, 7826, 8450,
 9445, 9689, 10479, 13029, 13068,
 21928, 22113, 28597, 35982, 36367
- \msg_error:nnnnnn 81, 83, 3028, 3042,
 7433, 7456, 7530, 9445, 13062, 22139
- \msg_error_text:n 79, 9337, 9451, 10044
- \msg_expandable_error:nn 83, 2599,
 4725, 8657, 10032, 10822, 16216,
 18623, 18625, 18633, 18639, 18669,
 19536, 20561, 20569, 20625, 23265
- \msg_expandable_error:nnn
 83, 2328, 2675, 2735, 2759,
 4820, 5841, 9010, 9060, 9691, 9693,
 10032, 10854, 11001, 11262, 11622,
 12525, 16835, 17120, 17301, 18333,
 20178, 23272, 23287, 23292, 23358,
 23415, 23454, 23460, 23796, 23801,
 23810, 23817, 23908, 23922, 24120,
 24171, 24851, 28247, 28254, 28260
- \msg_expandable_error:nnnn
 83, 4750, 7141,
 9695, 10032, 10474, 10831, 22268,
 24305, 24326, 25012, 28426, 28516
- \msg_expandable_error:nnnnn
 83, 10032, 10491, 21976,
 22164, 22848, 28293, 28667, 36364
- \msg_expandable_error:nnnnnn ...
 83, 10032
- \msg_fatal:nn 80, 9424
- \msg_fatal:nnn 80, 9424
- \msg_fatal:nnnn 80, 9424
- \msg_fatal:nnnnn 80, 9424
- \msg_fatal_text:n 79, 9337, 9427
- \msg_gset:nnn 78, 9183
- \msg_gset:nnnn 78, 9183
- \msg_if_exist:nnTF 78, 9170, 9177, 9553
- \msg_if_exist_p:nn 78, 9170
- \msg_info:nn 82, 9455
- \msg_info:nnn 82, 9455
- \msg_info:nnnn 82, 9455, 9679
- \msg_info:nnnnn 82, 9455
- \msg_info:nnnnnn 82, 9455
- \msg_info_text:n . 79, 9337, 9484, 9489
- \msg_interrupt:nnn 36503
- \msg_line_context: 78,
 590, 1887, 9240, 20629, 20631, 29056
- \msg_line_number: 79, 9240
- \msg_log:n 36505
- \msg_log:nn 82, 9492
- \msg_log:nnn 82, 9492
- \msg_log:nnnn 82, 9492
- \msg_log:nnnnn 82, 9492
- \msg_log:nnnnnn .. 82, 4083, 4097,
 7361, 7371, 9492, 10173, 10387,
 11395, 16888, 18454, 18475, 19909,
 21747, 22241, 33846, 33877, 35422
- \msg_log_eval:Nn 303, 8435, 17720,
 20275, 20368, 20436, 24601, 35813
- \msg_module_name:n
 78, 80, 9250, 9356, 9374, 9458
- \g_msg_module_name_prop
 . 78, 3724, 8336, 9364, 9376, 9377,
 10058, 10066, 10069, 10071, 14616,
 20632, 21810, 22716, 29090, 35576
- \msg_module_type:n . 78, 79, 9355, 9368
- \g_msg_module_type_prop
 . 78, 3725, 8337, 9364, 9370, 9371,
 10059, 10067, 10070, 10072, 14617,
 20633, 21811, 22717, 29091, 35577
- \msg_new:nnn 78, 4386, 8050,
 8052, 8057, 8318, 8330, 9183, 9677,
 9800, 9833, 9844, 9846, 9848, 9850,
 9852, 9954, 9956, 9958, 9960, 9962,
 9964, 9966, 9968, 9972, 9975, 9982,
 9984, 9991, 9998, 11542, 14233,
 14235, 14244, 20628, 20630, 21803,
 21816, 22876, 22878, 22880, 22882,
 22884, 22886, 22888, 24503, 24505,
 24507, 24509, 24511, 24513, 24515,
 24517, 24519, 24521, 24523, 24525,
 24527, 24529, 24533, 24904, 24906,
 24908, 28243, 33903, 35578, 36372
- \msg_new:nnnn
 . 78, 589, 3683, 3700, 3707, 3716,
 8063, 8070, 8076, 8086, 8092, 8116,
 8123, 8131, 8139, 8146, 8153, 8159,
 8166, 8172, 8180, 8186, 8192, 8202,
 8209, 8218, 8221, 8229, 8235, 8241,
 8248, 8255, 8265, 8276, 8286, 8296,

- 8305, 8311, 8320, 8323, 9054, [9183](#),
 9675, 9696, 9704, 9712, 9719, 9730,
 9738, 9747, 9754, 9761, 9771, 9780,
 9787, 9793, 9802, 9809, 9817, 9825,
 9854, 9857, 9866, 9872, 9879, 9886,
 9898, 9905, 9913, 9920, 9936, 9945,
 10009, 10015, 11006, 11503, 11536,
 11548, 11554, 11561, 11566, 11636,
 14104, 14237, 14252, 14266, 14272,
 14319, 14364, 14454, 14569, 14751,
 14758, 14930, 21767, 21770, 21773,
 21779, 21785, 21791, 21797, 22708,
 22850, 22865, 29049, 29055, 29061,
 29067, 29074, 33887, 33894, 33897,
 35443, 35453, 35459, 35466, 35472,
 35481, 35487, 35495, 35504, 35510,
 35519, 35528, 35537, 35543, 35552,
 35558, 35564, 35570, 35987, 36123
 \msg_none:nn [82](#), [9504](#)
 \msg_none:nnn [82](#), [9504](#)
 \msg_none:nnnn [82](#), [9504](#)
 \msg_none:nnnnn [82](#), [9504](#)
 \msg_note:nn [82](#), [9455](#)
 \msg_note:nnn [82](#), [9455](#)
 \msg_note:nnnn [82](#), [9455](#)
 \msg_note:nnnnn [82](#), [9455](#)
 \msg_redirect_class:nn [84](#), [9626](#)
 \msg_redirect_module:nnn ... [84](#), [9626](#)
 \msg_redirect_name:nnn [84](#), [9617](#)
 \msg_see_documentation_text:n ...
 [80](#), [9374](#)
 \msg_set:nnn [78](#), [9183](#)
 \msg_set:nnnn [78](#), [9183](#)
 \msg_show:nn [83](#), [9505](#)
 \msg_show:nnn [83](#), [9505](#)
 \msg_show:nnnn [83](#), [9505](#)
 \msg_show:nnnnn [83](#), [9505](#)
 \msg_show:nnnnnn ... [83](#), [303](#), [1309](#),
 4081, 4095, 7360, 7370, [9505](#), 10172,
 10386, 11394, 16886, 18452, 18474,
 19907, 21745, 22239, 33843, 35420
 \msg_show_eval:Nn [303](#), 8433, 17716,
 20271, 20364, 20432, 24599, [35813](#)
 \msg_show_item:n
 . [83](#), [303](#), 16901, 18465, 18479, [35818](#)
 \msg_show_item:nn
 [303](#), [893](#), 19922, [35818](#)
 \msg_show_item_unbraced:n [303](#), [35818](#)
 \msg_show_item_unbraced:nn
 [303](#), [614](#), 10180, 10394,
 21755, 33862, 35433, 35441, [35818](#)
 \msg_term:n [36507](#)
 \msg_term:nn [82](#), [9492](#)
 \msg_term:nnn [82](#), [9492](#)
 \msg_term:nnnn [82](#), [9492](#)
 \msg_term:nnnnn [82](#), [9492](#)
 \msg_term:nnnnnn [82](#), [1309](#), [9492](#), 33874
 \msg_warning:nn [81](#), [5542](#), [9455](#)
 \msg_warning:nnn .. [81](#), 5458, 5462,
 5504, 5566, 5604, 5623, [9455](#), 9681
 \msg_warning:nnnn
 [81](#), 5164, 5313, [9455](#), 9683
 \msg_warning:nnnnn .. [81](#), [9455](#), 36340
 \msg_warning:nnnnnn .. [81](#), [9455](#), 9657
 \msg_warning_text:n .. [79](#), [9337](#), 9479
 msg internal commands:
 __msg_chk_free:nn [9175](#), [9185](#)
 __msg_chk_if_free:nn [9175](#)
 __msg_class_chk_exist:nTF
 [9540](#), 9555, 9622, 9632, 9637
 \l__msg_class_loop_seq . [601](#), [9549](#),
 9641, 9649, 9659, 9660, 9663, 9665
 __msg_class_new:nn
 598, [9385](#), 9424, 9437, 9448, 9477,
 9482, 9487, 9492, 9498, 9504, 9505
 \l__msg_class_tl [599](#),
 601, 9545, 9562, 9575, 9596, 9600,
 9603, 9611, 9650, 9652, 9654, 9668
 \c__msg_coding_error_text_tl ...
 ... [9208](#), 9699, 9707, 9733, 9741,
 9750, 9757, 9764, 9774, 9796, 9805,
 9812, 9820, 9828, 9860, 9869, 9875,
 9882, 9889, 9901, 9916, 9923, 9939
 \c__msg_continue_text_tl . [9208](#), 9257
 \c__msg_critical_text_tl . [9208](#), 9442
 \l__msg_current_class_tl
 [601](#), [9545](#), 9557,
 9595, 9600, 9603, 9611, 9640, 9654
 __msg_expandable_error:n [609](#)
 __msg_expandable_error:nn
 [10021](#), 10035
 __msg_fatal_exit: [9424](#)
 \c__msg_fatal_text_tl ... [9208](#), 9429
 \c__msg_help_text_tl [9208](#), 9267
 \l__msg_hierarchy_seq
 [599](#), [600](#), [9548](#), 9578, 9588, 9593
 __msg_info_aux:NNnnnnnn [9455](#)
 \l__msg_internal_tl
 .. [9162](#), 9293, 9299, 9435, 9529, 9535
 __msg_interrupt:n [9294](#), [9303](#)
 __msg_interrupt:Nnnn [9247](#)
 __msg_interrupt:NnnnN
 [9247](#), 9426, 9439, 9450
 __msg_interrupt_more_text:n ...
 [591](#), [9276](#)
 __msg_interrupt_text:n [9276](#)

- _msg_interrupt_wrap:nnn [9255](#), [9265](#), [9276](#)
- \c_msg_more_text_prefix_tl [9168](#), [9194](#), [9203](#), [9252](#), [9269](#)
- \l_msg_name_str [9163](#), [9250](#), [9283](#), [9287](#), [9458](#), [9466](#), [9470](#)
- \c_msg_no_info_text_tl .. [9208](#), [9259](#)
- _msg_no_more_text:nnnn [9247](#)
- \c_msg_on_line_text_tl .. [9208](#), [9243](#)
- _msg_redirect:nnn [9626](#)
- _msg_redirect_loop_chk:nnn ... [9626](#), [9668](#)
- _msg_redirect_loop_list:n .. [9626](#)
- \l_msg_redirect_prop [9547](#), [9575](#), [9620](#), [9623](#)
- \c_msg_return_text_tl [9208](#), [9702](#), [9710](#), [9717](#)
- _msg_show:n [597](#), [9505](#)
- _msg_show:nn [9505](#)
- _msg_show:w [9505](#)
- _msg_show_dot:w [9505](#)
- _msg_show_eval:nnN [35813](#)
- _msg_text:n [9337](#)
- _msg_text:nn [9337](#)
- \c_msg_text_prefix_tl [609](#), [9168](#), [9172](#), [9192](#), [9201](#), [9256](#), [9266](#), [9463](#), [9495](#), [9501](#), [9508](#), [10038](#)
- \l_msg_text_str [9163](#), [9249](#), [9281](#), [9286](#), [9457](#), [9462](#), [9469](#)
- _msg_tmp:w [10021](#), [10031](#)
- \c_msg_trouble_text_tl [9208](#)
- _msg_use:nnnnnn [9395](#), [9550](#)
- _msg_use_code: [599](#), [9550](#)
- _msg_use_hierarchy:nwN [9550](#)
- _msg_use_none_delimit_by_s_-stop:w [9167](#), [9581](#), [10027](#)
- _msg_use_redirect_module:n ... [600](#), [9550](#)
- _msg_use_redirect_name:n ... [9550](#)
- \mskip [372](#)
- \muexpr [549](#)
- multichoice commands:
 - .multichoice: [226](#), [21202](#)
- multichoices commands:
 - .multichoices:nn [226](#), [21202](#)
- \multiply [373](#)
- \muskip [374](#), [19154](#)
- muskip commands:
 - \c_max_muskip [220](#), [20437](#)
 - \muskip_add:Nn [218](#), [20413](#)
 - \muskip_const:Nn [218](#), [20381](#), [20437](#), [20438](#)
 - \muskip_eval:n [219](#), [20384](#), [20425](#), [20432](#), [20436](#)
 - \muskip_gadd:Nn [218](#), [20413](#)
 - .muskip_gset:N [226](#), [21212](#)
 - \muskip_gset:Nn [219](#), [20403](#)
 - \muskip_gset_eq:NN [219](#), [20409](#)
 - \muskip_gsub:Nn [219](#), [20413](#)
 - \muskip_gzero:N ... [218](#), [20387](#), [20396](#)
 - \muskip_gzero_new:N [218](#), [20393](#)
 - \muskip_if_exist:NTF [218](#), [20394](#), [20396](#), [20399](#)
 - \muskip_if_exist_p:N [218](#), [20399](#)
 - \muskip_log:N [220](#), [20433](#)
 - \muskip_log:n [220](#), [20433](#)
 - \muskip_new:N [218](#), [20375](#), [20383](#), [20394](#), [20396](#), [20439](#), [20440](#), [20441](#), [20442](#)
 - .muskip_set:N [226](#), [21212](#)
 - \muskip_set:Nn [219](#), [20403](#)
 - \muskip_set_eq:NN [219](#), [20409](#)
 - \muskip_show:N [219](#), [20429](#)
 - \muskip_show:n [220](#), [912](#), [20431](#)
 - \muskip_sub:Nn [219](#), [20413](#)
 - \muskip_use:N [219](#), [20426](#), [20427](#)
 - \muskip_zero:N [218](#), [20387](#), [20394](#)
 - \muskip_zero_new:N [218](#), [20393](#)
 - \g_tmpa_muskip [220](#), [20439](#)
 - \l_tmpa_muskip [220](#), [20439](#)
 - \g_tmpb_muskip [220](#), [20439](#)
 - \l_tmpb_muskip [220](#), [20439](#)
 - \c_zero_muskip [220](#), [20388](#), [20390](#), [20437](#)
 - \muskipdef [375](#)
 - \mutoglu [550](#)
 - my commands:
 - \l_my_int [156](#)

N

 - \n [8906](#), [8908](#), [8910](#)
 - nan [256](#)
 - nc [257](#)
 - nd [257](#)
 - \newbox [807](#)
 - \newcatcodetable [28722](#)
 - \newcount [807](#)
 - \newdimen [807](#)
 - \newlinechar [80](#), [376](#)
 - \newluabytcode [36](#)
 - \next [53](#), [94](#), [102](#), [105](#), [108](#), [116](#)
 - \NG [29399](#), [31298](#), [31658](#)
 - \ng [29399](#), [31298](#), [31670](#)
 - \c_nine [36417](#)
 - \noalign [377](#)
 - \noautospacing [1141](#)
 - \noautoxspacing [1142](#)
 - \noboundary [378](#)
 - \nobreakspace [31613](#)

<code>\noexpand</code>	81, 96, 99, 119, 379	<code>\or</code>	390
<code>\nohrule</code>	867	or commands:	
<code>\noindent</code>	380	<code>\or:</code>	167, 711, 713, 979,
<code>\nokerns</code>	868		1416, 1981, 1982, 1983, 1984, 1985,
<code>\noligs</code>	869		1986, 1987, 1988, 1989, 2657, 2658,
<code>\nolimits</code>	381		2659, 2660, 2661, 3836, 3837, 4030,
<code>\nonscript</code>	382		4031, 4557, 4558, 4559, 4560, 4831,
<code>\nonstopmode</code>	383		4832, 4833, 4834, 4835, 6386, 6439,
<code>\normaldeviate</code>	924		7073, 7075, 7107, 7108, 7109, 7110,
<code>\normalend</code> 1343, 1344, 10084, 10114, 10315			7111, 7112, 7113, 7114, 7115, 7116,
<code>\normaleveryjob</code>	1345		7117, 7118, 7119, 7513, 7514, 10613,
<code>\normalexpanded</code>	1354		10614, 10615, 10616, 10617, 10618,
<code>\normalfont</code>	31573		10619, 13491, 13567, 13811, 13812,
<code>\normalhoffset</code>	1357		13813, 13814, 13815, 14840, 14841,
<code>\normalinput</code>	1346		16918, 17503, 17504, 17505, 17506,
<code>\normalitaliccorrection</code>	1356, 1358		17507, 17508, 17509, 17510, 17511,
<code>\normallanguage</code>	1347		17512, 17513, 17514, 17515, 17516,
<code>\normalleft</code>	1364, 1365		17517, 17518, 17519, 17520, 17521,
<code>\normalmathop</code>	1348		17522, 17523, 17524, 17525, 17526,
<code>\normalmiddle</code>	1366		17527, 17536, 17537, 17538, 17539,
<code>\normalmonth</code>	1349		17540, 17541, 17542, 17543, 17544,
<code>\normalouter</code>	1350		17545, 17546, 17547, 17548, 17549,
<code>\normalover</code>	1351		17550, 17551, 17552, 17553, 17554,
<code>\normalright</code>	1367		17555, 17556, 17557, 17558, 17559,
<code>\normalshowtokens</code>	1360		17560, 18677, 18681, 18684, 18686,
<code>\normalsize</code>	31590		18687, 18689, 18691, 18693, 18694,
<code>\normalunexpanded</code>	1353		18696, 18698, 18700, 18702, 18730,
<code>\normalvcenter</code>	1352		22399, 22400, 22401, 22650, 22665,
<code>\normalvoffset</code>	1359		22666, 23037, 23038, 23063, 24346,
<code>\nospaces</code>	870		24347, 24348, 24384, 25069, 25070,
<code>\notexpanded: <token></code>	196		25071, 25194, 25279, 25365, 25366,
<code>\novrule</code>	871		25367, 25368, 25369, 25370, 25371,
<code>\nulldelimiterspace</code>	384		25372, 25373, 25452, 25455, 25791,
<code>\nullfont</code>	385		25792, 25806, 25807, 25821, 26105,
<code>\num</code>	239		26328, 26353, 26359, 26360, 26361,
<code>\number</code>	386		26362, 26363, 26512, 26547, 26549,
<code>\numexpr</code>	551		26557, 26750, 26800, 26803, 26812,
			26927, 26950, 26951, 26983, 26984,
			26988, 27041, 27042, 27082, 27087,
			27097, 27102, 27112, 27117, 27127,
			27132, 27142, 27147, 27157, 27162,
			27689, 27690, 27735, 27820, 27823,
			27835, 27841, 27888, 27890, 27891,
			27901, 27907, 27984, 27985, 27992,
			28038, 28039, 28046, 28112, 28113,
			28333, 28616, 28617, 28618, 28695,
			28696, 28697, 36302, 36303, 36304
		<code>\oradical</code>	1182
		<code>\orieveryjob</code>	1337, 1338
		<code>\oripdfoutput</code>	1340, 1341
		<code>\outer</code>	7, 807, 391
		<code>\output</code>	392
		<code>\outputbox</code>	872
		<code>\outputmode</code>	925
O			
<code>\O</code>	29400, 31299, 31659, 31949		
<code>\o</code>	29400, 31299, 31671, 31950		
<code>\odelcode</code>	1176		
<code>\odelimiter</code>	1177		
<code>\OE</code>	29401, 31300, 31660		
<code>\oe</code>	29401, 31300, 31672		
<code>\omathaccent</code>	1178		
<code>\omathchar</code>	1179		
<code>\omathchardef</code>	1180		
<code>\omathcode</code>	1181		
<code>\omit</code>	387		
<code>\c_one</code>	36401		
<code>\c_one_hundred</code>	36435		
<code>\openin</code>	388		
<code>\openout</code>	389		

- \outputpenalty 393
- \over 394
- \overfullrule 395
- \overline 396
- \overwithdelims 397
- P**
- \PackageError 88, 96
- \pagebottomoffset 873
- \pagedepth 398
- \pagedir 874
- \pagedirection 875
- \pagediscards 552
- \pagefilllstretch 399
- \pagefillstretch 400
- \pagefilstretch 401
- \pagefistretch 1143
- \pagegoal 402
- \pageheight 926
- \pageleftoffset 876
- \pagerightoffset 877
- \pageshrink 403
- \pagestretch 404
- \pagetopoffset 878
- \pagetotal 405
- \pagewidth 927
- \par .. 14–19, 87, 370, 1265, 406, 32197,
32199, 32203, 32208, 32213, 32218,
32225, 32230, 32237, 32242, 32262
- \pardir 879
- \pardirection 880
- \parfillskip 407
- \parindent 408
- \parshape 409
- \parshapedimen 553
- \parshapeindent 554
- \parshapelength 555
- \parskip 410
- \partokencontext 1183
- \partokenname 1184
- \patterns 411
- \pausing 412
- pc 257
- pdf commands:
 - \pdf_destination:nn 297, 35685
 - \pdf_destination:nnnn 297, 298, 35687
 - \pdf_object_if_exist:nTF . 295, 35608
 - \pdf_object_if_exist_p:n . 295, 35608
 - \pdf_object_new:nn 295, 35608
 - \pdf_object_ref:n 296, 35608
 - \pdf_object_ref_last: . . . 296, 35608
 - \pdf_object_unnamed_write:nn . . .
..... 296, 35608
 - \pdf_object_write:nn 295, 35608
 - \pdf_pageobject_ref:n . . . 296, 35630
 - \pdf_pagobject_ref:n 296
 - \pdf_uncompress: 297, 35600
 - \pdf_version: 296, 35681
 - \pdf_version_compare:Nn . . 296, 35632
 - \pdf_version_compare:NnTF 296, 35670
 - \pdf_version_compare_p:Nn 296
 - \pdf_version_gset:n 296, 35666
 - \pdf_version_major: 296, 35681
 - \pdf_version_min_gset:n . . 296, 35666
 - \pdf_version_minor: 296, 35681
- pdf internal commands:
 - __pdf_backend_compress_objects:n
..... 35605
 - __pdf_backend_compresslevel:n 35604
 - __pdf_backend_destination:nn . 35686
 - __pdf_backend_destination:nnnn .
..... 35690
 - __pdf_backend_object_last: . . 35629
 - __pdf_backend_object_new:nn . 35609
 - __pdf_backend_object_now:nn . 35625
 - __pdf_backend_object_ref:n . . 35622
 - __pdf_backend_object_write:nn 35618
 - __pdf_backend_pageobject_ref:n .
..... 35631
 - __pdf_backend_version_major: . . .
..... 35637, 35645,
35648, 35657, 35660, 35682, 35683
 - __pdf_backend_version_major_-
gset:n 35677
 - __pdf_backend_version_minor: . . .
.. 35638, 35649, 35661, 35682, 35684
 - __pdf_backend_version_minor_-
gset:n 35678
 - \g_pdf_init_bool
.. 35589, 35602, 35619, 35626, 35675
 - __pdf_version_compare_<:w 35632
 - __pdf_version_compare_=:w 35632
 - __pdf_version_compare_>:w 35632
 - __pdf_version_gset:w 35666
 - \pdfadjustspacing 650
 - \pdfannot 578
 - \pdfcatalog 579
 - \pdfcolorstack 581
 - \pdfcolorstackinit 582
 - \pdfcompresslevel 580
 - \pdfcopyfont 651
 - \pdfcreationdate 583
 - \pdfdecimaldigits 584
 - \pdfdest 585
 - \pdfdestmargin 586
 - \pdfdraftmode 652
 - \pdfeachlinedepth 653
 - \pdfeachlineheight 654

<code>\pdfelapsedtime</code>	655	<code>\pdfpageheight</code>	668
<code>\pdfendlink</code>	587	<code>\pdfpageref</code>	625
<code>\pdfendthread</code>	588	<code>\pdfpageresources</code>	626
<code>\pdfextension</code>	881	<code>\pdfpagesattr</code>	623, 627
<code>\pdffeedback</code>	882	<code>\pdfpagewidth</code>	669
<code>\pdffiledump</code>	712	<code>\pdfpkmode</code>	670
<code>\pdffilemoddate</code>	711	<code>\pdfpkresolution</code>	671
<code>\pdffilesize</code>	709	<code>\pdfprimitive</code>	672
<code>\pdffirstlineheight</code>	656	<code>\pdfprotrudechars</code>	673
<code>\pdffontattr</code>	589	<code>\pdfpxdimen</code>	674
<code>\pdffontexpand</code>	657	<code>\pdfrandomseed</code>	675
<code>\pdffontname</code>	590	<code>\pdfrefobj</code>	628
<code>\pdffontobjnum</code>	591	<code>\pdfrefxform</code>	629
<code>\pdffontsize</code>	658	<code>\pdfrefximage</code>	630
<code>\pdfgamma</code>	592	<code>\pdfresettimer</code>	676
<code>\pdfgentounicode</code>	595	<code>\pdfrestore</code>	631
<code>\pdfglyphtounicode</code>	596	<code>\pdfretval</code>	632
<code>\pdfhorigin</code>	597	<code>\pdfsave</code>	633
<code>\pdfignoreddimen</code>	659	<code>\pdfsavepos</code>	677
<code>\pdfimageapplygamma</code>	593	<code>\pdfsetmatrix</code>	634
<code>\pdfimagegamma</code>	594	<code>\pdfsetrandomseed</code>	678
<code>\pdfimagehicolor</code>	598	<code>\pdfshellescape</code>	679
<code>\pdfimageresolution</code>	599	<code>\pdfstartlink</code>	635
<code>\pdfincludechars</code>	600	<code>\pdfstartthread</code>	636
<code>\pdfinclusioncopyfonts</code>	601	<code>\pdfstrcmp</code>	124, 704, 22, 708
<code>\pdfinclusionerrorlevel</code>	602	<code>\pdfsuppressptexinfo</code>	637
<code>\pdfinfo</code>	604	pdfTeX commands:	
<code>\pdfinsertht</code>	660	<code>\pdfTeX_if_engine:TF</code>	
<code>\pdflastannot</code>	605 36511, 36513, 36515	
<code>\pdflastlinedepth</code>	661	<code>\pdfTeX_if_engine_p:</code>	36509
<code>\pdflastlink</code>	606	<code>\pdfTeXbanner</code>	682
<code>\pdflastobj</code>	607	<code>\pdfTeXrevision</code>	683
<code>\pdflastxform</code>	608	<code>\pdfTeXversion</code>	684
<code>\pdflastximage</code>	609	<code>\pdfthread</code>	638
<code>\pdflastximagecolordepth</code>	610	<code>\pdfthreadmargin</code>	639
<code>\pdflastximagepages</code>	612	<code>\pdftracingfonts</code>	680, 1232, 1233
<code>\pdflastxpos</code>	662	<code>\pdftrailer</code>	640
<code>\pdflastypos</code>	663	<code>\pdfuniformdeviate</code>	681
<code>\pdflinkmargin</code>	613	<code>\pdfuniqueresname</code>	641
<code>\pdfliteral</code>	614	<code>\pdfvariable</code>	883
<code>\pdfmajorversion</code>	615	<code>\pdfvorigin</code>	642
<code>\pdfmapfile</code>	664	<code>\pdfxform</code>	643
<code>\pdfmapline</code>	665	<code>\pdfxformname</code>	644
<code>\pdfmdfivesum</code>	710	<code>\pdfximage</code>	645
<code>\pdfminorversion</code>	616	<code>\pdfximagebbox</code>	646
<code>\pdfnames</code>	617	peek commands:	
<code>\pdfnoligatures</code>	666	<code>\peek_after:Nw</code> 71, 191, 4209,	
<code>\pdfnormaldeviate</code>	667	19341, 19354, 19379, 19420, 36291	
<code>\pdfobj</code>	618	<code>\peek_analysis_map_break:</code>	
<code>\pdfobjcompresslevel</code>	619 194, 4174, 4196	
<code>\pdfoutline</code>	620	<code>\peek_analysis_map_break:n</code>	
<code>\pdfoutput</code>	621 194, 4174, 7926	
<code>\pdfpageattr</code>	622	<code>\peek_analysis_map_inline:n</code>	
<code>\pdfpagebox</code>	624	... 45, 191, 194, 427, 550, 4186, 7919	

- \peek_catcode:NTF .. [192](#), [19475](#), [34457](#)
- \peek_catcode_collect_inline:Nn .
..... [309](#), [36271](#)
- \peek_catcode_ignore_spaces:NTF .
..... [36653](#)
- \peek_catcode_remove:NTF
..... [192](#), [19475](#), [34517](#)
- \peek_catcode_remove_ignore_
spaces:NTF [36653](#)
- \peek_charcode:NTF [192](#), [194](#), [195](#), [19475](#)
- \peek_charcode_collect_inline:Nn
..... [309](#), [36271](#)
- \peek_charcode_ignore_spaces:NTF
..... [36653](#)
- \peek_charcode_remove:NTF
..... [192](#), [195](#), [19475](#)
- \peek_charcode_remove_ignore_
spaces:NTF [36653](#)
- \peek_gafter:Nw [191](#), [19341](#)
- \peek_meaning:NTF [192](#), [19475](#)
- \peek_meaning_collect_inline:Nn .
..... [309](#), [36271](#)
- \peek_meaning_ignore_spaces:NTF .
..... [36653](#)
- \peek_meaning_remove:NTF . [192](#), [19475](#)
- \peek_meaning_remove_ignore_
spaces:NTF [36653](#)
- \peek_N_type:TF
..... [193](#), [19489](#), [19526](#), [19528](#)
- \peek_regex:NTF [194](#), [7863](#), [7878](#), [7880](#)
- \peek_regex:nTF [194](#),
[501](#), [549](#), [551](#), [552](#), [7863](#), [7869](#), [7871](#)
- \peek_regex_remove_once:n [194](#)
- \peek_regex_remove_once:NTF
..... [195](#), [7863](#), [7897](#), [7899](#)
- \peek_regex_remove_once:nTF
..... [195](#), [551](#), [7863](#), [7887](#), [7889](#)
- \peek_regex_replace_once:Nn [195](#), [7955](#)
- \peek_regex_replace_once:nn [195](#), [7955](#)
- \peek_regex_replace_once:NnTF ...
..... [195](#), [7955](#), [7965](#), [7967](#)
- \peek_regex_replace_once:nnTF ...
..... [195](#),
[523](#), [526](#), [549](#), [553](#), [7955](#), [7957](#), [7959](#)
- \peek_remove_filler:n
..... [193](#), [19366](#), [34455](#), [34515](#)
- \peek_remove_spaces:n
..... [192](#), [19350](#), [36662](#),
[36665](#), [36668](#), [36671](#), [36674](#), [36677](#)
- \g_peek_token [191](#), [19330](#), [19344](#)
- \l_peek_token [191](#), [194](#),
[445-447](#), [876](#), [878](#), [879](#), [1325](#), [1372](#),
[4214](#), [4215](#), [4216](#), [4259](#), [4319](#), [4322](#),
[4367](#), [19330](#), [19342](#), [19359](#), [19384](#),
[19387](#), [19398](#), [19436](#), [19448](#), [19468](#),
[19495](#), [19496](#), [19497](#), [19500](#), [34464](#),
[34471](#), [34485](#), [36297](#), [36298](#), [36299](#)
- peek internal commands:
 __peek_collect:N [1372](#), [36271](#)
- __peek_collect:NNn [36271](#)
- __peek_collect_remove:nw [36271](#)
- \l__peek_collect_tl [1372](#),
[36270](#), [36282](#), [36284](#), [36309](#), [36314](#)
- __peek_collect_true:w .. [1372](#), [36271](#)
- __peek_execute_branches_.... : [1372](#)
- __peek_execute_branches_
 catcode: [879](#), [19442](#), [36272](#)
- __peek_execute_branches_
 catcode_aux: [19442](#)
- __peek_execute_branches_
 catcode_auxii:N [19442](#)
- __peek_execute_branches_
 catcode_auxiii: [19442](#)
- __peek_execute_branches_
 charcode: [879](#), [19442](#), [36274](#)
- __peek_execute_branches_
 meaning: [879](#), [19434](#), [36276](#)
- __peek_execute_branches_N_type:
 [19489](#)
- __peek_false:w
 [879](#), [1372](#), [19334](#), [19352](#),
[19363](#), [19369](#), [19399](#), [19415](#), [19439](#),
[19462](#), [19472](#), [19506](#), [19519](#), [36283](#)
- __peek_false_aux:n [1372](#), [36284](#), [36285](#)
- __peek_N_type:w [19489](#)
- __peek_N_type_aux:nnw [19489](#)
- __peek_remove_filler: [19366](#)
- __peek_remove_filler:w [19366](#)
- __peek_remove_filler_expand:w [19366](#)
- __peek_remove_spaces: [19350](#)
- \l__peek_search_tl [874](#), [878](#), [1372](#),
[19333](#), [19408](#), [19459](#), [19469](#), [36281](#)
- \l__peek_search_token
[874](#), [1372](#), [19332](#), [19407](#), [19436](#), [36280](#)
- __peek_tmp:w
 [19334](#), [19348](#), [19490](#), [19512](#)
- __peek_token_generic:NNTF
 [879](#), [19422](#),
[19424](#), [19426](#), [19523](#), [19527](#), [19529](#)
- __peek_token_generic_aux:NNNTF .
 [19404](#), [19423](#), [19429](#)
- __peek_token_remove_generic:NNTF
 [879](#), [19422](#), [19430](#), [19432](#)
- __peek_true:w [879](#),
[1372](#), [19334](#), [19414](#), [19437](#), [19460](#),
[19470](#), [19504](#), [19518](#), [19519](#), [36290](#)

- `__peek_true_aux:w` 875, 876,
19334, 19347, 19354, 19355, 19368,
19409, 19423, 36291, 36292, 36310
- `__peek_true_remove:w`
875, 876, 19345,
19360, 19385, 19389, 19429, 36315
- `__peek_use_none_delimit_by_s_-
stop:w` 879, 19340, 19502
- `\penalty` 413
- `\pi` 23348, 23349
- `pi` 256
- `\pm` 24819, 24820
- `\postbreakpenalty` 1144
- `\postdisplaypenalty` 414
- `\postexhyphenchar` 884
- `\posthyphenchar` 885
- `\prebinoppenalty` 886
- `\prebreakpenalty` 1145
- `\predisplaydirection` 556
- `\predisplaygapfactor` 887
- `\predisplaypenalty` 415
- `\predisplaysize` 416
- `\preexhyphenchar` 888
- `\prehyphenchar` 889
- `\prerelpenalty` 890
- `\pretolerance` 417
- `\prevdepth` 418
- `\prevgraf` 419
- prg commands:
 - `\prg_break:` 71, 521, 745, 795,
796, 2255, 3597, 3672, 4025, 4106,
4136, 4137, 4138, 4139, 4140, 4141,
4454, 4722, 4726, 5984, 5994, 5999,
6008, 6032, 6060, 6926, 6954, 7742,
8686, 11071, 12902, 13873, 13889,
14007, 14039, 14145, 14148, 14289,
14336, 14389, 14395, 14628, 14709,
14881, 15022, 16626, 16659, 16710,
16763, 17303, 17821, 18303, 22461,
22470, 24470, 24490, 24491, 24715,
24716, 24729, 24829, 24830, 24831,
28222, 28274, 28498, 35905, 35911
 - `\prg_break:n`
. 71, 2255, 6462, 8686, 12904,
13775, 13783, 13795, 16500, 16639,
17313, 22199, 22218, 22232, 22477
 - `\prg_break_point:`
. 71, 414, 421, 640, 1362,
2255, 3425, 3466, 3590, 3597, 3948,
4107, 4143, 4450, 4700, 5980, 6029,
6463, 6796, 6948, 7736, 7742, 8686,
11044, 12892, 13776, 13784, 13874,
13890, 14008, 14040, 14146, 14149,
14290, 14337, 14390, 14396, 14629,
14829, 14986, 16497, 16627, 16661,
16712, 16763, 16808, 16815, 17308,
17821, 18303, 22193, 22212, 22227,
22462, 22471, 24471, 24492, 24717,
24833, 28223, 28274, 28506, 35906
 - `\prg_break_point:Nn`
. 70, 71, 140, 378, 442, 452,
796, 816, 902, 2246, 4077, 4196,
6687, 6701, 6747, 7925, 8686, 10268,
10287, 12448, 12477, 12488, 13316,
13342, 13362, 13384, 16662, 16703,
16713, 16736, 16743, 16752, 17355,
18176, 18198, 18220, 18247, 18269,
19850, 19872, 19888, 20217, 24902
 - `\prg_do_nothing:` 13, 71, 471,
524, 544, 545, 647, 670, 671, 726,
786, 832, 849, 884, 986, 1163, 2244,
2255, 2579, 2971, 2998, 3097, 3098,
3099, 3512, 3670, 3671, 3919, 3968,
4236, 4548, 5030, 5073, 5074, 5081,
5082, 7009, 7237, 7695, 7712, 7716,
8883, 10612, 10907, 11340, 11377,
11379, 12121, 12751, 12786, 13144,
13146, 13939, 14879, 16162, 16163,
16299, 16306, 16592, 16594, 17814,
17820, 17828, 17983, 18196, 18205,
18268, 18279, 18354, 18366, 18420,
18424, 18431, 21592, 22743, 22777,
22803, 22811, 24355, 28203, 28877
 - `\prg_generate_conditional_-
variant:Nnn` 64, 3191,
8428, 10111, 12277, 12287, 12311,
12321, 12337, 12354, 12365, 12439,
12689, 12707, 13221, 13233, 13241,
13272, 15983, 16005, 16433, 16501,
16595, 16597, 16611, 16613, 16615,
16617, 18001, 18015, 18016, 18165,
18167, 19765, 19766, 19814, 19827,
19838, 21735, 32061, 32063, 32067,
32694, 35957, 35958, 36031, 36032
 - `\prg_map_break:Nn` 70, 71,
378, 442, 684, 845, 893, 2246, 4175,
4177, 4445, 8686, 10251, 10253,
12515, 12517, 13375, 13377, 16650,
16652, 18282, 18284, 19904, 19906
 - `\prg_new_conditional:Nnn`
. 62, 1596, 8378
 - `\prg_new_conditional:Npnn`
. 62–64, 348, 690, 865, 878, 1579,
2131, 2850, 4823, 4843, 4865, 4911,
4935, 8378, 8420, 8463, 8523, 8538,
8549, 8564, 8574, 8670, 8672, 8674,
8676, 9170, 10185, 11216, 11880,
12269, 12279, 12294, 12303, 12313,

12369, 12385, 12396, 12677, 12691,
 12709, 12748, 12768, 12783, 13204,
 13211, 13216, 13223, 13228, 13772,
 13781, 13796, 13804, 14476, 14510,
 14529, 15967, 15975, 15985, 15995,
 16144, 16425, 17124, 17177, 17215,
 17223, 17771, 17776, 17830, 18118,
 18982, 18987, 18992, 18997, 19004,
 19010, 19016, 19021, 19026, 19031,
 19036, 19041, 19046, 19051, 19058,
 19073, 19078, 19114, 19222, 19231,
 19809, 19816, 20033, 20038, 20332,
 20340, 21728, 21736, 22645, 23790,
 24619, 24627, 24643, 29372, 30356,
 30376, 30404, 30429, 32057, 32059,
 32065, 32684, 33933, 35610, 35632
 \prg_new_eq_conditional:Nnn
 64, 1715, 8378, 8459,
 8461, 11965, 11966, 13195, 13197,
 13199, 13201, 16330, 16332, 16880,
 16881, 16882, 16883, 16884, 16885,
 17073, 17075, 17919, 17921, 18114,
 18116, 19805, 19807, 19964, 19966,
 20306, 20308, 20399, 20401, 24617,
 24618, 28935, 28937, 32002, 32004
 \prg_new_protected_conditional:Nnn
 63, 1596, 8378
 \prg_new_protected_conditional:Npnn
 63,
 1579, 5278, 7383, 7388, 7415, 7417,
 8378, 8861, 10102, 10205, 10225,
 10886, 11018, 11165, 11167, 11169,
 11171, 11206, 11268, 12325, 12338,
 12356, 13235, 13243, 13913, 13922,
 16482, 16591, 16593, 16599, 16602,
 16605, 16608, 17992, 18002, 18004,
 18132, 18136, 19745, 19755, 19829,
 28939, 35953, 35955, 36027, 36029
 \prg_replicate:nn
 69, 111, 148, 548, 571, 959, 4407,
 5036, 5791, 6393, 6419, 6565, 6573,
 6736, 6902, 7014, 7654, 7662, 7722,
 7838, 7840, 8622, 9284, 9467, 10444,
 22146, 22306, 26047, 26899, 27207,
 27463, 27509, 27546, 28069, 28077,
 28536, 28639, 35025, 35033, 35101,
 35135, 35147, 35150, 35230, 35231
 \prg_return_false:
 63, 64, 359, 539, 791,
 811, 841, 1573, 1639, 1647, 1801,
 1806, 1819, 1824, 1832, 1849, 2134,
 2860, 4837, 4848, 4851, 4856, 4860,
 4861, 4869, 4872, 4877, 4880, 4917,
 4920, 4941, 4944, 5285, 5290, 7493,
 8378, 8425, 8468, 8528, 8544, 8554,
 8570, 8580, 8671, 8673, 8675, 8677,
 8865, 8873, 9173, 10109, 10192,
 10208, 10228, 10895, 11023, 11048,
 11178, 11198, 11212, 11231, 11240,
 11251, 11265, 11272, 11885, 12274,
 12284, 12299, 12308, 12318, 12334,
 12348, 12362, 12375, 12392, 12407,
 12686, 12704, 12722, 12730, 12740,
 12756, 12779, 12790, 13209, 13214,
 13219, 13226, 13231, 13239, 13247,
 13777, 13785, 13801, 13817, 13920,
 13929, 14480, 14483, 14486, 14513,
 14516, 14533, 14536, 14539, 15972,
 15980, 15991, 16001, 16148, 16430,
 16496, 16515, 17122, 17154, 17159,
 17182, 17220, 17228, 17774, 17781,
 17846, 17849, 17995, 18009, 18121,
 18156, 18162, 18985, 18990, 18995,
 19000, 19007, 19014, 19019, 19024,
 19029, 19034, 19039, 19044, 19049,
 19054, 19071, 19076, 19081, 19086,
 19120, 19123, 19135, 19235, 19260,
 19277, 19286, 19753, 19763, 19812,
 19820, 19836, 20036, 20055, 20070,
 20071, 20336, 20343, 21733, 21742,
 22656, 22658, 23803, 23813, 24624,
 24638, 24651, 28949, 28955, 29382,
 29385, 30359, 30363, 30366, 30396,
 30422, 30443, 32058, 32060, 32066,
 32690, 32692, 33938, 33941, 35614,
 35640, 35652, 35664, 35985, 36043
 \prg_return_true: . . . 63, 64, 359,
 536, 539, 637, 677, 690, 791, 889,
 1365, 1573, 1639, 1647, 1804, 1821,
 1829, 1834, 1847, 1852, 2134, 2852,
 2860, 4826, 4840, 4848, 4851, 4856,
 4860, 4872, 4877, 4880, 4915, 4939,
 5281, 5287, 7491, 8378, 8423, 8466,
 8526, 8542, 8552, 8568, 8578, 8671,
 8673, 8675, 8677, 8886, 9173, 10107,
 10190, 10195, 10198, 10211, 10231,
 10893, 11024, 11062, 11179, 11213,
 11229, 11238, 11249, 11271, 11883,
 12272, 12282, 12297, 12306, 12316,
 12332, 12346, 12362, 12373, 12390,
 12405, 12684, 12702, 12720, 12738,
 12754, 12777, 12788, 13209, 13214,
 13219, 13226, 13231, 13239, 13247,
 13795, 13799, 13807, 13820, 13920,
 13929, 14480, 14486, 14518, 14533,
 14539, 15970, 15978, 15989, 15999,
 16148, 16428, 16500, 16518, 17154,
 17180, 17218, 17226, 17774, 17779,

- 17842, 17845, 17851, 17998, 18012,
 18122, 18152, 18162, 18985, 18990,
 18995, 19000, 19007, 19014, 19019,
 19024, 19029, 19034, 19039, 19044,
 19049, 19054, 19070, 19076, 19084,
 19134, 19258, 19284, 19751, 19761,
 19812, 19825, 19834, 20036, 20071,
 20335, 20344, 21732, 21741, 22649,
 22654, 23798, 23819, 24622, 24640,
 24649, 28945, 29383, 30368, 30372,
 30379, 30382, 30385, 30388, 30391,
 30394, 30408, 30411, 30414, 30417,
 30420, 30432, 30435, 30438, 30441,
 32058, 32060, 32066, 32689, 33939,
 35613, 35639, 35651, 35663, 36005
 \prg_set_conditional:Nnn
 62, 1596, 8378
 \prg_set_conditional:Npnn
 62–64, 1579,
 1798, 1810, 1826, 1838, 8378, 11259
 \prg_set_eq_conditional:NNn
 64, 1715, 8378
 \prg_set_protected_conditional:Nnn
 63, 1596, 8378
 \prg_set_protected_conditional:Npnn
 63, 1579, 8378, 11031
 prg internal commands:
 __prg_break_point:Nn 378
 __prg_F_true:w 1668
 __prg_generate_conditional:nnNNNnnn
 1591, 1616, 1625
 __prg_generate_conditional:NNnnnnNw
 1625
 __prg_generate_conditional_-
 count:NNNnn 1596
 __prg_generate_conditional_-
 count:nnNNNnn 1596
 __prg_generate_conditional_-
 fast:nw 359, 360, 1625
 __prg_generate_conditional_-
 parm:NNNpnn 1579
 __prg_generate_conditional_-
 test:w 1625
 __prg_generate_F_form:wNNnnnnN 1668
 __prg_generate_p_form:wNNnnnnN .
 359, 1668
 __prg_generate_T_form:wNNnnnnN 1668
 __prg_generate_TF_form:wNNnnnnN
 1668
 __prg_p_true:w 1668
 __prg_replicate:N 8622
 __prg_replicate 8622
 __prg_replicate_0:n 8622
 __prg_replicate_1:n 8622
 __prg_replicate_2:n 8622
 __prg_replicate_3:n 8622
 __prg_replicate_4:n 8622
 __prg_replicate_5:n 8622
 __prg_replicate_6:n 8622
 __prg_replicate_7:n 8622
 __prg_replicate_8:n 8622
 __prg_replicate_9:n 8622
 __prg_replicate_first:N 8622
 __prg_replicate_first -:n ... 8622
 __prg_replicate_first_0:n ... 8622
 __prg_replicate_first_1:n ... 8622
 __prg_replicate_first_2:n ... 8622
 __prg_replicate_first_3:n ... 8622
 __prg_replicate_first_4:n ... 8622
 __prg_replicate_first_5:n ... 8622
 __prg_replicate_first_6:n ... 8622
 __prg_replicate_first_7:n ... 8622
 __prg_replicate_first_8:n ... 8622
 __prg_replicate_first_9:n ... 8622
 __prg_set_eq_conditional:NNNn 1715
 __prg_set_eq_conditional:nnNNnNw
 1723, 1731
 __prg_set_eq_conditional_F_-
 form:nnn 1731
 __prg_set_eq_conditional_F_-
 form:wNnnnn 1768
 __prg_set_eq_conditional_-
 loop:nnnnNw 1731
 __prg_set_eq_conditional_p_-
 form:nnn 1731
 __prg_set_eq_conditional_p_-
 form:wNnnnn 1762
 __prg_set_eq_conditional_T_-
 form:nnn 1731
 __prg_set_eq_conditional_T_-
 form:wNnnnn 1766
 __prg_set_eq_conditional_TF_-
 form:nnn 1731
 __prg_set_eq_conditional_TF_-
 form:wNnnnn 1764
 __prg_T_true:w 1668
 __prg_TF_true:w 360, 1668
 __prg_use_none_delimit_by_q_-
 recursion_stop:w
 1577, 1654, 1736, 1741, 1748
 \primitive 780
 prop commands:
 \c_empty_prop 205,
 883, 19538, 19548, 19552, 19555, 19811
 \prop_clear:N 198, 19551,
 19558, 19590, 19604, 33472, 34376
 \prop_clear_new:N
 198, 19557, 34566, 34597, 34638

- \prop_concat:NNN [200](#), [883](#), [19576](#)
- \prop_const_from_keyval:Nn
 - [199](#), [19588](#), [32653](#), [32660](#), [35343](#)
- \prop_count:N [201](#), [19699](#), [35887](#)
- \prop_gclear:N [198](#), [19551](#), [19561](#), [19596](#)
- \prop_gclear_new:N
 - [198](#), [1281](#), [19557](#), [32727](#), [32728](#)
- \prop_gconcat:NNN [200](#), [19576](#)
- \prop_get:Nn [71](#), [36517](#), [36519](#)
- \prop_get:NnN [137](#), [138](#), [200](#),
 - [201](#), [19658](#), [33712](#), [33716](#), [33785](#),
 - [33789](#), [33948](#), [34057](#), [34145](#), [35140](#)
- \prop_get:NnNTF [200](#), [202](#), [9575](#),
 - [9595](#), [9650](#), [10164](#), [10378](#), [13980](#),
 - [19829](#), [21066](#), [28808](#), [32918](#), [34051](#),
 - [34150](#), [34576](#), [34834](#), [34847](#), [34862](#),
 - [34941](#), [34969](#), [34985](#), [35356](#), [35369](#)
- \prop_gpop:NnN [201](#), [19666](#)
- \prop_gpop:NnNTF [201](#), [203](#), [19745](#)
- .prop_gput:N [226](#), [21220](#)
- \prop_gput:Nnn [199](#),
 - [3724](#), [3725](#), [8336](#), [8337](#), [9365](#), [9367](#),
 - [10058](#), [10059](#), [10066](#), [10067](#), [10069](#),
 - [10070](#), [10071](#), [10072](#), [10092](#), [10138](#),
 - [10323](#), [10353](#), [13743](#), [13744](#), [13745](#),
 - [13746](#), [13747](#), [13748](#), [13749](#), [13750](#),
 - [13751](#), [13752](#), [13753](#), [13754](#), [13755](#),
 - [13756](#), [13757](#), [13764](#), [13767](#), [14616](#),
 - [14617](#), [19620](#), [19767](#), [20632](#), [20633](#),
 - [21810](#), [21811](#), [22716](#), [22717](#), [28761](#),
 - [29090](#), [29091](#), [32944](#), [32962](#), [32997](#),
 - [33028](#), [34759](#), [34760](#), [34761](#), [34762](#),
 - [34763](#), [34764](#), [34765](#), [34766](#), [34777](#),
 - [34779](#), [34781](#), [34782](#), [34783](#), [34784](#),
 - [34785](#), [34786](#), [34787](#), [34887](#), [34888](#),
 - [34902](#), [34916](#), [34926](#), [35576](#), [35577](#)
 - \prop_gput_from_keyval:Nn [200](#), [19588](#)
 - \prop_gput_if_new:Nnn [200](#), [19788](#)
 - \prop_gremove:Nn
 - [201](#), [10149](#), [10363](#), [19642](#), [28759](#)
 - \prop_gset_eq:NN [199](#), [19555](#), [19563](#),
 - [19580](#), [32729](#), [32731](#), [32896](#), [32898](#),
 - [32935](#), [32937](#), [33184](#), [33350](#), [33391](#)
 - \prop_gset_from_keyval:Nn [199](#), [19588](#)
 - \prop_hput:Nnn [19767](#)
 - \prop_if_empty:NTF
 - [202](#), [19714](#), [19734](#), [19809](#), [33937](#), [35884](#)
 - \prop_if_empty_p:N [202](#), [19809](#)
 - \prop_if_exist:NTF
 - [201](#), [19558](#), [19561](#), [19805](#), [20963](#), [33935](#)
 - \prop_if_exist_p:N [201](#), [19805](#)
 - \prop_if_in:NnTF
 - [202](#), [9370](#), [9376](#), [19816](#), [34578](#)
 - \prop_if_in_p:Nn [202](#), [19816](#)
 - \prop_item:Nn [201](#), [203](#), [9371](#), [9377](#),
 - [19688](#), [35203](#), [35242](#), [36518](#), [36520](#)
 - \prop_log:N [205](#), [19907](#)
 - \prop_map_break:
 - [204](#), [892](#), [893](#), [19846](#), [19847](#),
 - [19848](#), [19849](#), [19850](#), [19872](#), [19884](#),
 - [19885](#), [19886](#), [19887](#), [19888](#), [19903](#)
 - \prop_map_break:n
 - [204](#), [19696](#), [19825](#), [19903](#)
 - \prop_map_function:NN [203](#), [303](#),
 - [893](#), [10179](#), [10393](#), [19704](#), [19722](#),
 - [19737](#), [19840](#), [19922](#), [33860](#), [35431](#)
 - \prop_map_inline:Nn
 - [203](#), [19585](#), [19865](#),
 - [33194](#), [33196](#), [33199](#), [33217](#), [33219](#),
 - [33293](#), [33310](#), [33371](#), [33373](#), [33377](#),
 - [33379](#), [33559](#), [33578](#), [33759](#), [33768](#)
 - \prop_map_tokens:Nn
 - [203](#), [797](#), [887](#), [891](#), [19690](#), [19818](#), [19880](#)
 - \prop_new:N
 - [198](#), [9364](#), [9366](#), [9388](#), [9547](#),
 - [10080](#), [10311](#), [13742](#), [19545](#), [19558](#),
 - [19561](#), [19571](#), [19572](#), [19573](#), [19574](#),
 - [19575](#), [20705](#), [20706](#), [20963](#), [28712](#),
 - [33172](#), [33173](#), [33174](#), [33642](#), [33683](#),
 - [34663](#), [34753](#), [34758](#), [34775](#), [34780](#)
 - \prop_pop:NnN [200](#), [19666](#)
 - \prop_pop:NnNTF [200](#), [202](#), [19745](#)
 - .prop_put:N [226](#), [21220](#)
 - \prop_put:Nnn
 - [199](#), [200](#), [400](#), [881](#), [882](#), [9623](#),
 - [9639](#), [9656](#), [19585](#), [19612](#), [19767](#),
 - [21073](#), [32941](#), [32959](#), [32978](#), [32995](#),
 - [33026](#), [33228](#), [33230](#), [33236](#), [33238](#),
 - [33247](#), [33253](#), [33261](#), [33320](#), [33328](#),
 - [33418](#), [33424](#), [33432](#), [33439](#), [33583](#),
 - [33643](#), [33645](#), [33647](#), [33649](#), [33651](#),
 - [33653](#), [33655](#), [33657](#), [33659](#), [33661](#),
 - [33663](#), [33665](#), [33667](#), [33669](#), [33671](#),
 - [33673](#), [33675](#), [33677](#), [34029](#), [34377](#),
 - [34567](#), [34585](#), [34628](#), [34643](#), [34824](#)
 - \prop_put_from_keyval:Nn [200](#), [19588](#)
 - \prop_put_if_new:Nnn [200](#), [19788](#)
 - \prop_rand_key_value:N [304](#), [35882](#)
 - \prop_remove:Nn [201](#),
 - [9620](#), [9635](#), [19642](#), [33754](#), [33757](#), [33761](#)
 - \prop_set_eq:NN
 - [199](#), [19552](#), [19563](#), [19577](#),
 - [19584](#), [32884](#), [32886](#), [32928](#), [32930](#),
 - [33181](#), [33190](#), [33192](#), [33343](#), [33367](#),
 - [33369](#), [33388](#), [33516](#), [33749](#), [34648](#)
 - \prop_set_from_keyval:Nn
 - [199](#), [200](#), [883](#), [19588](#), [34803](#)
 - \prop_show:N [204](#), [19907](#)

- `\prop_to_keyval:N` [201](#), [19709](#)
 - `\g_tmpa_prop` [205](#), [19571](#)
 - `\l_tmpa_prop` [205](#), [19571](#)
 - `\g_tmpb_prop` [205](#), [19571](#)
 - `\l_tmpb_prop` [205](#), [19571](#)
 - prop internal commands:
 - `__prop_concat:NNNN` [19576](#)
 - `__prop_count:nn` [19699](#)
 - `__prop_from_keyval_key:w` [884](#)
 - `__prop_from_keyval_value:w` ... [884](#)
 - `__prop_if_in:nnn` [19816](#)
 - `__prop_if_recursion_tail_stop:n`
..... [19543](#)
 - `\l__prop_internal_prop`
..... [19575](#), [19584](#),
[19585](#), [19586](#), [19602](#), [19603](#), [19604](#)
 - `\l__prop_internal_tl`
..... [889](#), [19534](#), [19537](#),
[19771](#), [19777](#), [19778](#), [19794](#), [19801](#)
 - `__prop_item:nnn` [887](#), [19688](#)
 - `__prop_keyval_parse:NNNn`
.. [19610](#), [19611](#), [19618](#), [19619](#), [19625](#)
 - `__prop_map_function:Nw` .. [892](#), [19840](#)
 - `__prop_map_tokens:nw` [19880](#)
 - `__prop_missing_eq:n` [19588](#)
 - `__prop_pair:wn` ... [881](#), [885](#), [892](#),
[893](#), [19534](#), [19535](#), [19636](#), [19639](#),
[19773](#), [19796](#), [19846](#), [19847](#), [19848](#),
[19849](#), [19853](#), [19854](#), [19855](#), [19856](#),
[19868](#), [19870](#), [19875](#), [19884](#), [19885](#),
[19886](#), [19887](#), [19891](#), [19892](#), [19893](#),
[19894](#), [19917](#), [19926](#), [19929](#), [35892](#)
 - `__prop_put:NNnn` [19767](#)
 - `__prop_put_if_new:NNnn` [19788](#)
 - `__prop_rand_item:w` [35882](#)
 - `__prop_show:NN` [19907](#)
 - `__prop_show_validate:w` [19907](#)
 - `__prop_split:NnTF`
..... [882](#), [889–891](#), [19631](#),
[19644](#), [19650](#), [19660](#), [19668](#), [19677](#),
[19747](#), [19757](#), [19776](#), [19799](#), [19831](#)
 - `__prop_split_aux:NnTF` [19631](#)
 - `__prop_split_aux:w` [885](#), [19631](#)
 - `__prop_to_keyval:nn` [19709](#)
 - `__prop_to_keyval:nnw` [19709](#)
 - `__prop_to_keyval_exp_after:wN` ..
..... [888](#), [19709](#)
 - `__prop_use_i_delimit_by_s_-`
`stop:nw` [35881](#), [35895](#)
 - `\protect` [1208](#), [10511](#),
[23283](#), [29665](#), [29688](#), [29690](#), [31511](#)
 - `\protected` [117](#), [119](#), [143](#), [557](#), [19148](#), [19150](#)
 - `\protrudechars` [928](#)
 - `\ProvidesExplClass` [9](#)
 - `\ProvidesExplFile` [9](#)
 - `\ProvidesExplPackage` [9](#)
 - `pt` [257](#)
 - `\ptexminorversion` [1146](#)
 - `\ptexrevision` [1147](#)
 - `\ptexversion` [1148](#)
 - `\pxdimen` [929](#)
- Q**
- quark commands:
 - `\q_mark` [138](#), [427](#), [15915](#)
 - `\q_nil` [25](#),
[118](#), [138](#), [355](#), [773](#), [775](#), [777](#), [1535](#),
[1538](#), [15915](#), [15969](#), [15988](#), [15994](#),
[16009](#), [16010](#), [16016](#), [16040](#), [16044](#),
[21677](#), [21678](#), [21680](#), [21682](#), [21684](#),
[21686](#), [21692](#), [35086](#), [35092](#), [35093](#)
 - `\q_no_value`
.... [74](#), [87](#), [88](#), [94–96](#), [137](#), [138](#),
[144](#), [145](#), [151](#), [178](#), [200](#), [201](#), [306](#),
[773](#), [775](#), [792](#), [793](#), [836](#), [886](#), [1365](#),
[8859](#), [10202](#), [10215](#), [10884](#), [11015](#),
[11043](#), [11158](#), [11160](#), [11162](#), [11164](#),
[11204](#), [15915](#), [15977](#), [15998](#), [16004](#),
[16506](#), [16514](#), [16526](#), [16552](#), [17958](#),
[17973](#), [19662](#), [19673](#), [19682](#), [36044](#)
 - `\quark_if_nil:n` [775](#), [776](#)
 - `\quark_if_nil:NnTF` [138](#), [348](#), [777](#), [15967](#)
 - `\quark_if_nil:NnTF` [138](#),
[680](#), [774](#), [775](#), [777](#), [15985](#), [35100](#), [35113](#)
 - `\quark_if_nil_p:N` [138](#), [15967](#)
 - `\quark_if_nil_p:n` [138](#), [15985](#)
 - `\quark_if_no_value:NnTF` [138](#), [11045](#),
[15967](#), [33714](#), [33718](#), [33787](#), [33791](#)
 - `\quark_if_no_value:NnTF` ... [138](#), [15985](#)
 - `\quark_if_no_value_p:N` ... [138](#), [15967](#)
 - `\quark_if_no_value_p:n` ... [138](#), [15985](#)
 - `\quark_if_recursion_tail_break:N`
..... [36521](#)
 - `\quark_if_recursion_tail_break:n`
..... [36523](#)
 - `\quark_if_recursion_tail_-`
`break:NN` [140](#), [776](#), [15955](#)
 - `\quark_if_recursion_tail_-`
`break:nN` [140](#), [776](#), [15955](#)
 - `\quark_if_recursion_tail_stop:N` .
..... [139](#), [347](#),
[776](#), [1205](#), [15923](#), [29231](#), [31284](#),
[31317](#), [31622](#), [31634](#), [31696](#), [31747](#)
 - `\quark_if_recursion_tail_stop:n` .
..... [139](#), [347](#), [775](#), [776](#),
[5973](#), [6075](#), [15937](#), [16907](#), [19928](#), [30803](#)
 - `\quark_if_recursion_tail_stop_-`
`do:Nn` [139](#), [347](#), [776](#), [15923](#)

- \quark_if_recursion_tail_stop_-
 - do:nn [139](#), [347](#), [776](#), [15937](#)
- \quark_new:N [138](#),
 - [347](#), [348](#), [779](#), [4478](#), [4479](#), [8415](#),
[8416](#), [10328](#), [10792](#), [10794](#), [10795](#),
[12083](#), [12084](#), [12085](#), [12086](#), [12087](#),
[13088](#), [13089](#), [13741](#), [15910](#), [15915](#),
[15916](#), [15917](#), [15918](#), [15919](#), [15920](#),
[15922](#), [16926](#), [16927](#), [18490](#), [19541](#),
[19542](#), [20710](#), [29248](#), [29250](#), [29251](#)
- \q_recursion_stop [25](#), [139](#), [140](#), [355](#),
[773](#), [1537](#), [1541](#), [5969](#), [6070](#), [15919](#),
[16896](#), [19917](#), [29239](#), [31052](#), [31249](#),
[31304](#), [31339](#), [31676](#), [31744](#), [31962](#)
- \q_recursion_tail
 - [139](#), [140](#), [773](#), [774](#),
[5969](#), [6069](#), [15919](#), [15925](#), [15931](#),
[15940](#), [15947](#), [15952](#), [15957](#), [15964](#),
[16896](#), [19917](#), [29239](#), [31051](#), [31248](#),
[31303](#), [31338](#), [31675](#), [31743](#), [31961](#)
- \q_stop [25](#),
[38](#), [113](#), [137](#), [138](#), [355](#), [773](#), [1536](#),
[1539](#), [12665](#), [15915](#), [29117](#), [29121](#),
[29132](#), [29151](#), [29156](#), [29167](#), [29171](#),
[29183](#), [29184](#), [29190](#), [29192](#), [29193](#),
[29195](#), [29198](#), [29214](#), [29222](#), [35113](#)
- quark internal commands:
 - \q_bool_recursion_stop
[8415](#), [8418](#), [8522](#), [8548](#)
 - \q_bool_recursion_tail
[8415](#), [8522](#), [8548](#)
 - \q_char_no_value [18490](#), [18866](#)
 - \q_cs_nil [3214](#)
 - \q_cs_recursion_stop
[2882](#), [2886](#), [2897](#), [3207](#)
 - \q_file_nil
[10792](#), [10859](#), [10873](#), [10965](#), [10971](#)
 - \q_file_recursion_stop
[10794](#), [10838](#), [10849](#)
 - \q_file_recursion_tail
[10794](#), [10838](#), [10842](#)
 - \q_int_recursion_stop
[16926](#), [17624](#), [17641](#), [17684](#), [17711](#)
 - \q_int_recursion_tail
[16926](#), [17624](#), [17641](#), [17684](#)
 - \q_iow_nil [10328](#), [10540](#), [10547](#)
 - \q_keys_no_value
[944](#), [20695](#), [20710](#), [21284](#),
[21308](#), [21325](#), [21350](#), [21367](#), [21396](#)
 - \q_prg_recursion_stop
[361](#), [1578](#), [1643](#), [1728](#)
 - \q_prg_recursion_tail
[361](#), [1643](#), [1653](#), [1728](#), [1747](#)
 - \q_prop_recursion_stop [19541](#)
 - \q__prop_recursion_tail [19541](#)
 - __quark_if_empty_if:n . [15985](#), [16146](#)
 - __quark_if_nil:w [775](#), [15985](#)
 - __quark_if_no_value:w [15985](#)
 - __quark_if_recursion_tail:w ...
[774](#), [779](#), [15937](#), [15964](#)
 - __quark_module_name:N
[780](#), [16013](#), [16036](#), [16151](#)
 - __quark_module_name:w [16151](#)
 - __quark_module_name_end:w ... [16151](#)
 - __quark_module_name_loop:w .. [16151](#)
 - __quark_new_conditional:Nnnn . [16012](#)
 - __quark_new_conditional_aux_-
do:Nnnnn . [780](#), [16133](#), [16135](#), [16136](#)
 - __quark_new_conditional_-
define:Nnnnn [780](#), [16136](#)
 - __quark_new_conditional_N:Nnnn .
[16132](#)
 - __quark_new_conditional_n:Nnnn .
[16132](#)
 - __quark_new_test:Nnnn [16012](#)
 - __quark_new_test_aux:Nn
[16013](#), [16014](#), [16024](#)
 - __quark_new_test_aux:nnNNnnnn [16012](#)
 - __quark_new_test_aux_do:nnNNnnnnNNn
[778](#), [779](#), [16067](#), [16072](#),
[16077](#), [16082](#), [16087](#), [16093](#), [16096](#)
 - __quark_new_test_define_break_-
ifx:nnNNnn [16094](#), [16109](#)
 - __quark_new_test_define_break_-
tl:nnNNnn [16078](#), [16109](#)
 - __quark_new_test_define_-
ifx:nNnnnn
[778](#), [779](#), [16083](#), [16088](#), [16109](#)
 - __quark_new_test_define_-
tl:nNnnnn
[778](#), [779](#), [16068](#), [16073](#), [16109](#)
 - __quark_new_test_N:Nnnn [16065](#)
 - __quark_new_test_n:Nnnn [16065](#)
 - __quark_new_test_NN:Nnnn [16065](#)
 - __quark_new_test_Nn:Nnnn [16065](#)
 - __quark_new_test_nN:Nnnn [16075](#)
 - __quark_new_test_nn:Nnnn [16065](#)
 - \q__quark_nil [15922](#)
 - __quark_quark_conditional_-
name:N [781](#), [16035](#), [16173](#)
 - __quark_quark_conditional_-
name:w [781](#), [16173](#)
 - __quark_test_define_aux:NNNNnnNNn
[779](#), [16096](#)
 - __quark_tmp:w
[781](#), [16151](#), [16172](#), [16173](#), [16183](#)

- \q__regex_nil
 - ... [4449](#), [4454](#), [4479](#), [4484](#), [5091](#),
 - [5095](#), [5753](#), [5771](#), [5772](#), [5867](#), [5877](#)
 - \q__regex_recursion_stop
 - .. [4478](#), [4481](#), [4483](#), [5753](#), [5772](#), [7737](#)
 - \q__str_nil
 - [751](#), [13741](#), [14828](#), [14835](#), [14850](#), [14877](#)
 - \q__str_recursion_stop
 - [13088](#), [13672](#), [13680](#), [13685](#)
 - \q__str_recursion_tail
 - [707](#), [13088](#), [13315](#),
 - [13324](#), [13341](#), [13361](#), [13383](#), [13672](#)
 - \q__text_nil [29248](#), [29681](#), [29682](#)
 - \q__text_recursion_stop
 - [29250](#), [29253](#),
 - [29433](#), [29447](#), [29456](#), [29535](#), [29551](#),
 - [29560](#), [29603](#), [29627](#), [29647](#), [29747](#),
 - [29755](#), [29811](#), [29825](#), [29834](#), [29836](#),
 - [29903](#), [29919](#), [29928](#), [29972](#), [30042](#),
 - [30051](#), [30078](#), [30086](#), [30258](#), [30266](#),
 - [30314](#), [30320](#), [30336](#), [30341](#), [30454](#),
 - [30459](#), [30475](#), [30484](#), [30556](#), [30561](#),
 - [30609](#), [30614](#), [30637](#), [30642](#), [30678](#),
 - [30687](#), [31361](#), [31374](#), [31383](#), [31401](#),
 - [31414](#), [31416](#), [31433](#), [31442](#), [31470](#)
 - \q__text_recursion_tail
 - [29250](#), [29380](#), [29433](#), [29534](#),
 - [29603](#), [29627](#), [29647](#), [29811](#), [29836](#),
 - [29902](#), [29972](#), [31361](#), [31400](#), [31470](#)
 - \q__tl_mark
 - [674](#), [12083](#), [12190](#), [12192](#), [12194](#), [12196](#)
 - \q__tl_nil [675](#), [12083](#), [12216](#)
 - \q__tl_recursion_stop [12086](#)
 - \q__tl_recursion_tail . [12086](#), [12891](#)
 - \q__tl_stop [675](#), [12083](#), [12215](#)
 - \quitvmode [690](#)
- R**
- \r [29389](#), [31735](#), [31759](#), [31785](#), [31909](#), [31910](#)
 - \radical [420](#)
 - \raise [421](#)
 - rand [256](#)
 - randint [256](#)
 - \randomseed [930](#)
 - \read [422](#)
 - \readline [558](#)
 - \readpapersizespecial [1149](#)
 - \ref [29406](#), [29413](#)
 - regex commands:
 - \regex_case_replace_all:nN ... [7452](#)
 - \regex_case_replace_all:nNTF . [7452](#)
 - \regex_const:Nn [54](#), [7345](#)
 - \regex_count:NnN [55](#), [7393](#)
 - \regex_count:nnN [55](#), [538](#), [7393](#)
 - \regex_extract_all:NnN [56](#), [7411](#)
 - \regex_extract_all:nnN
 - [47](#), [56](#), [450](#), [7411](#)
 - \regex_extract_all:NnNTF ... [56](#), [7411](#)
 - \regex_extract_all:nnNTF ... [56](#), [7411](#)
 - \regex_extract_once:NnN [56](#), [7411](#)
 - \regex_extract_once:nnN [56](#), [7411](#)
 - \regex_extract_once:NnNTF .. [56](#), [7411](#)
 - \regex_extract_once:nnNTF [50](#), [56](#), [7411](#)
 - \regex_gset:Nn [54](#), [7345](#)
 - \regex_log:N [54](#), [493](#), [7360](#)
 - \regex_log:n [54](#), [7360](#)
 - \regex_match:NnTF [55](#), [7383](#)
 - \regex_match:nnTF .. [55](#), [540](#), [550](#), [7383](#)
 - \regex_match_case:nn
 - [55](#), [58](#), [472](#), [501](#), [7397](#), [7531](#)
 - \regex_match_case:nnTF
 - [55](#), [7397](#), [7407](#), [7409](#)
 - \regex_new:N
 - [54](#), [453](#), [7339](#), [7341](#), [7342](#), [7343](#), [7344](#)
 - \regex_replace:nnN [182](#)
 - \regex_replace_all:NnN [57](#), [7411](#)
 - \regex_replace_all:nnN
 - [47](#), [57](#), [537](#), [7411](#)
 - \regex_replace_all:NnNTF ... [57](#), [7411](#)
 - \regex_replace_all:nnNTF ... [57](#), [7411](#)
 - \regex_replace_case_all:nN
 - [58](#), [7457](#), [7469](#)
 - \regex_replace_case_all:nNTF ...
 - [58](#), [7452](#), [7470](#), [7471](#), [7472](#), [7473](#), [7474](#)
 - \regex_replace_case_once:nN [58](#), [7429](#)
 - \regex_replace_case_once:nNTF ...
 - [58](#), [7429](#), [7448](#), [7450](#)
 - \regex_replace_once:NnN [57](#), [7411](#)
 - \regex_replace_once:nnN
 - [56-58](#), [195](#), [537](#), [7411](#)
 - \regex_replace_once:NnNTF .. [57](#), [7411](#)
 - \regex_replace_once:nnNTF
 - [57](#), [553](#), [7411](#)
 - \regex_set:Nn [46](#), [54](#), [55](#), [7345](#)
 - \regex_show:N [54](#), [493](#), [7360](#)
 - \regex_show:n [47](#), [52](#), [54](#), [7360](#)
 - \regex_split:NnN [57](#), [7411](#)
 - \regex_split:nnN [57](#), [7411](#)
 - \regex_split:NnNTF [57](#), [7411](#)
 - \regex_split:nnNTF [57](#), [7411](#)
 - \g_tmpa_regex [59](#), [7341](#)
 - \l_tmpa_regex [59](#), [7341](#)
 - \g_tmpb_regex [59](#), [7341](#)
 - \l_tmpb_regex [59](#), [7341](#)
 - regex internal commands:
 - __regex_A_test: [465](#), [5361](#),
 - [5383](#), [5999](#), [6002](#), [6008](#), [6108](#), [6592](#)

```

__regex_action_cost:n .....
    500, 504, 6381, 6382, 6390, 6840, 6866
__regex_action_free:n ..... 500,
    513, 6404, 6410, 6411, 6422, 6480,
    6484, 6509, 6534, 6538, 6541, 6569,
    6577, 6587, 6601, 6644, 6838, 6842
__regex_action_free_aux:nn .. 6842
__regex_action_free_group:n ...
    .... 500, 513, 6430, 6549, 6552, 6842
__regex_action_start_wildcard:N
    ..... 500, 6262, 6282, 6835
__regex_action_submatch:nN ....
    ..... 500, 6286,
    6311, 6503, 6504, 6642, 6891, 6893
__regex_action_submatch_aux:w 6893
__regex_action_submatch_auxii:w
    ..... 6893
__regex_action_submatch_-
    auxiii:w ..... 6893
__regex_action_submatch_auxiv:w
    ..... 6893
__regex_action_success: .....
    ..... 500, 6265, 6314, 6332, 6914
__regex_action_wildcard: ..... 518
\l__regex_added_begin_int .....
    ..... 7486, 7625, 7633, 7637,
    7678, 7819, 7824, 7827, 7838, 7853
\l__regex_added_end_int .....
    ..... 7486, 7627, 7633, 7638,
    7679, 7821, 7824, 7828, 7840, 7854
\c__regex_all_catcodes_int .....
    ..... 4895, 5017, 5121, 5719
\c__regex_ascii_lower_int .....
    ..... 4477, 4539, 4545
\c__regex_ascii_max_control_int .
    ..... 4474, 4656
\c__regex_ascii_max_int .....
    ..... 4474, 4649, 4657, 4847
\c__regex_ascii_min_int .....
    ..... 4474, 4648, 4655
__regex_assertion:Nn ..... 465,
    479, 511, 5357, 5379, 5988, 6101, 6592
__regex_b_test: .....
    465, 511, 5369, 5371, 6005, 6106, 6592
\l__regex_balance_int .....
    ..... 453, 524, 547, 4473,
    6991, 7023, 7282, 7299, 7498, 7511,
    7513, 7514, 7766, 7796, 7820, 7822
\g__regex_balance_intarray .....
    .... 450, 539, 6970, 6977, 7485, 7510
\g__regex_balance_tl .....
    ..... 524, 528, 6932,
    6992, 7022, 7048, 7065, 7075, 7150
__regex_begin ..... 7476
__regex_branch:n .....
    ..... 465, 483, 507, 4470,
    5022, 5097, 5529, 5582, 5767, 5877,
    5885, 5969, 5971, 5974, 6083, 6475
__regex_break_point:TF 454, 478,
    504, 4486, 4492, 6381, 6382, 6598, 6615
__regex_break_true:w .....
    ..... 454, 455, 4486, 4492, 4497,
    4504, 4511, 4515, 4522, 4528, 4577,
    4589, 4605, 5332, 6622, 6628, 6634
__regex_build:N .....
    ... 536, 6245, 7390, 7396, 7414, 7418
__regex_build:n .....
    502, 536, 6245, 7385, 7394, 7413, 7416
__regex_build_aux:NN .....
    ..... 549, 6245, 7875, 7894, 7964
__regex_build_aux:Nn .....
    ..... 549, 6245, 7866, 7884, 7956
__regex_build_for_cs:n .. 4600, 6321
__regex_build_new_state: .....
    ..... 6259, 6260,
    6279, 6280, 6284, 6324, 6325, 6354,
    6363, 6395, 6429, 6433, 6477, 6492,
    6497, 6536, 6555, 6590, 6594, 6639
\l__regex_build_tl .... 483, 553,
    4467, 5014, 5021, 5039, 5044, 5047,
    5048, 5051, 5052, 5055, 5115, 5118,
    5158, 5172, 5176, 5301, 5315, 5356,
    5378, 5391, 5423, 5436, 5440, 5522,
    5525, 5528, 5534, 5535, 5538, 5581,
    5871, 5875, 5882, 5888, 5909, 5925,
    5943, 6082, 6139, 6142, 6153, 6183,
    6198, 6202, 6205, 6211, 6990, 7013,
    7024, 7027, 7078, 7147, 7204, 7207,
    7221, 7289, 8023, 8026, 8034, 8037
__regex_build_transition_-
    left:NNN ... 6350, 6538, 6552, 6569
__regex_build_transition_-
    right:nNn .....
    ... 6350, 6396, 6430, 6480, 6484,
    6509, 6534, 6541, 6549, 6577, 6587
__regex_build_transitions_-
    laziness:NNNN .....
    ..... 6361, 6403, 6409, 6421
\l__regex_capturing_group_int ...
    ..... 450, 500, 547,
    6244, 6257, 6295, 6300, 6305, 6446,
    6448, 6459, 6460, 6468, 6469, 6472,
    6736, 6810, 6811, 6884, 6903, 7138,
    7142, 7722, 7743, 7748, 7801, 7809
\g__regex_case_balance_tl .....
    ..... 7053, 7056, 7062, 7066, 7074
__regex_case_build:n .....
    ..... 540, 6269, 7440, 7463, 7537

```

| | | | |
|--|---|---|---|
| <code>__regex_case_build_aux:Nn</code> ... | 6269 | <code>__regex_clean_assertion:Nn</code> .. | 5946 |
| <code>__regex_case_build_loop:n</code> ... | 6269 | <code>__regex_clean_bool:n</code> | 5946 |
| <code>\l__regex_case_changed_char_int</code> . | | <code>__regex_clean_branch:n</code> | 5946 |
| | 455 , 4514 , | <code>__regex_clean_branch_loop:n</code> . | 5946 |
| 4526 , 4527 , 4534 , 4538 , 4544 , 6657 | | <code>__regex_clean_class:n</code> | 5946 |
| <code>\g__regex_case_int</code> | | <code>__regex_clean_class:NnnnN</code> ... | 5946 |
| | 536 , 537 , 6267 , 6272 , 6289 , | <code>__regex_clean_class_loop:nnn</code> . | 5946 |
| 6292 , 6312 , 6313 , 7401 , 7441 , 7733 | | <code>__regex_clean_exact_cs:n</code> | 5946 |
| <code>\l__regex_case_max_group_int</code> ... | | <code>__regex_clean_exact_cs:w</code> | 5946 |
| | 6268 , 6288 , 6295 , 6302 , 6304 | <code>__regex_clean_group:nnnN</code> | 5946 |
| <code>__regex_case_replacement:n</code> | | <code>__regex_clean_int:n</code> | 5946 |
| | 7052 , 7464 | <code>__regex_clean_int_aux:N</code> | 5946 |
| <code>__regex_case_replacement_aux:n</code> . | | <code>__regex_clean_regex:n</code> ... | 5946 , 7375 |
| | 7064 , 7071 | <code>__regex_clean_regex_loop:w</code> .. | 5946 |
| <code>\g__regex_case_replacement_tl</code> ... | | <code>__regex_command_K:</code> | |
| | 7052 , 7062 , 7068 , 7073 | | 465 , 5943 , 5987 , 6099 , 6637 |
| <code>\c__regex_catcode_A_int</code> | 4895 | <code>__regex_compile:n</code> | 5057 , |
| <code>\c__regex_catcode_B_int</code> | 4895 | 5093 , 6251 , 7347 , 7352 , 7357 , 7364 | |
| <code>\c__regex_catcode_C_int</code> | 4895 | <code>__regex_compile:w</code> | |
| <code>\c__regex_catcode_D_int</code> | 4895 | | 471 , 5011 , 5059 , 5724 |
| <code>\c__regex_catcode_E_int</code> | 4895 | <code>__regex_compile_\$:</code> | 5352 |
| <code>\c__regex_catcode_in_class_mode_</code> - | | <code>__regex_compile(:</code> | 5546 |
| <code>int</code> 4885 , 5006 , 5390 , 5551 , 5644 , 5673 | | <code>__regex_compile):</code> | 5585 |
| <code>\c__regex_catcode_L_int</code> | 4895 | <code>__regex_compile.:</code> | 5323 |
| <code>\c__regex_catcode_M_int</code> | 4895 | <code>__regex_compile/A:</code> | 5352 |
| <code>\c__regex_catcode_mode_int</code> | | <code>__regex_compile/B:</code> | 5352 |
| .. | 4885 , 5002 , 5075 , 5422 , 5642 , 5671 | <code>__regex_compile/b:</code> | 5352 |
| <code>\c__regex_catcode_O_int</code> | 4895 | <code>__regex_compile/c:</code> | 5630 |
| <code>\c__regex_catcode_P_int</code> | 4895 | <code>__regex_compile/D:</code> | 5335 |
| <code>\c__regex_catcode_S_int</code> | 4895 | <code>__regex_compile/d:</code> | 5335 |
| <code>\c__regex_catcode_T_int</code> | 4895 | <code>__regex_compile/G:</code> | 5352 |
| <code>\c__regex_catcode_U_int</code> | 4895 | <code>__regex_compile/H:</code> | 5335 |
| <code>\l__regex_catcodes_bool</code> | | <code>__regex_compile/h:</code> | 5335 |
| | 4892 , 5678 , 5682 , 5717 | <code>__regex_compile/K:</code> | 5940 |
| <code>\l__regex_catcodes_int</code> | | <code>__regex_compile/N:</code> | 5335 |
| | 466 , 4892 , 5018 , 5120 , | <code>__regex_compile/S:</code> | 5335 |
| 5122 , 5128 , 5409 , 5426 , 5526 , 5539 , | | <code>__regex_compile/s:</code> | 5335 |
| 5638 , 5675 , 5710 , 5712 , 5718 , 5719 | | <code>__regex_compile/u:</code> | 5804 |
| <code>__regex_char_if_alphanumeric:N</code> 4865 | | <code>__regex_compile/V:</code> | 5335 |
| <code>__regex_char_if_alphanumeric:NTF</code> | | <code>__regex_compile/v:</code> | 5335 |
| | 4843 , 5068 , 7256 | <code>__regex_compile/W:</code> | 5335 |
| <code>__regex_char_if_special:N</code> ... | 4843 | <code>__regex_compile/w:</code> | 5335 |
| <code>__regex_char_if_special:NTF</code> ... | | <code>__regex_compile/Z:</code> | 5352 |
| | 4843 , 5064 | <code>__regex_compile/z:</code> | 5352 |
| <code>__regex_chk_c_allowed:TF</code> 4987 , 5631 | | <code>__regex_compile[:</code> | 5401 |
| <code>__regex_class:NnnnN</code> 465 , 473 , 474 , | | <code>__regex_compile]:</code> | 5385 |
| 480 , 4471 , 5116 , 5417 , 5418 , 5424 , | | <code>__regex_compile^:</code> | 5352 |
| 5784 , 5917 , 5927 , 5989 , 6098 , 6375 | | <code>__regex_compile_abort_tokens:n</code> . | |
| <code>\c__regex_class_mode_int</code> | | | 5131 , 5165 , 5506 , 5516 |
| | 4885 , 4992 , 5007 | <code>__regex_compile_anchor_letter:NNN</code> | |
| <code>__regex_class_repeat:n</code> | | | 5352 |
| | 505 , 6385 , 6391 , 6407 , 6416 | <code>__regex_compile_c[:w</code> | 5667 |
| <code>__regex_class_repeat:nN</code> . | 6386 , 6400 | <code>__regex_compile_c_C:NN</code> .. | 5646 , 5655 |
| <code>__regex_class_repeat:nnN</code> 6387 , 6414 | | <code>__regex_compile_c_lbrack_add:N</code> 5667 | |

- `__regex_compile_c_lbrack_end:` [5667](#)
- `__regex_compile_c_lbrack_-`
 - `loop:NN` [5667](#)
- `__regex_compile_c_test:NN` ... [5630](#)
- `__regex_compile_class:NN` [5431](#)
- `__regex_compile_class:TFNN`
 - [480](#), [5416](#), [5427](#), [5431](#)
- `__regex_compile_class_catcode:w`
 - [5408](#), [5420](#)
- `__regex_compile_class_normal:w` .
 - [5411](#), [5414](#)
- `__regex_compile_class_posix:NNNNw`
 - [5450](#)
- `__regex_compile_class_posix_-`
 - `end:w` [5450](#)
- `__regex_compile_class_posix_-`
 - `loop:w` [5450](#)
- `__regex_compile_class_posix_-`
 - `test:w` [5404](#), [5450](#)
- `__regex_compile_cs_aux:Nn` ... [5739](#)
- `__regex_compile_cs_aux:NNnnN` [5739](#)
- `__regex_compile_end:`
 - [471](#), [5011](#), [5084](#), [5748](#)
- `__regex_compile_end_cs:` . [5080](#), [5739](#)
- `__regex_compile_escaped:N` [5069](#), [5100](#)
- `__regex_compile_group_begin:N` ..
 - [5520](#), [5568](#), [5573](#), [5591](#), [5593](#)
- `__regex_compile_group_end:`
 - [5520](#), [5588](#)
- `__regex_compile_if_quantifier:TFw`
 - [5140](#), [5868](#), [5880](#)
- `__regex_compile_lparen:w` [5555](#), [5559](#)
- `__regex_compile_one:n`
 - [5110](#), [5267](#), [5273](#), [5327](#), [5338](#), [5341](#), [5351](#), [5497](#), [5755](#)
- `__regex_compile_quantifier:w` ...
 - .. [5129](#), [5147](#), [5396](#), [5540](#), [5873](#), [5889](#)
- `__regex_compile_quantifier*:w` [5181](#)
- `__regex_compile_quantifier+:w` [5181](#)
- `__regex_compile_quantifier?:w` [5181](#)
- `__regex_compile_quantifier_-`
 - `abort:nnN`
 - [5156](#), [5191](#), [5210](#), [5223](#), [5246](#)
- `__regex_compile_quantifier_-`
 - `braced_auxi:w` [5187](#)
- `__regex_compile_quantifier_-`
 - `braced_auxii:w` [5187](#)
- `__regex_compile_quantifier_-`
 - `braced_auxiii:w` [5187](#)
- `__regex_compile_quantifier_-`
 - `lazyness:nnNN` [474](#), [5168](#), [5182](#), [5184](#), [5186](#), [5199](#), [5219](#), [5241](#)
- `__regex_compile_quantifier_-`
 - `none:` [5152](#), [5154](#), [5156](#)
- `__regex_compile_range:Nw` [5265](#), [5278](#)
- `__regex_compile_raw:N` [4937](#), [5065](#), [5069](#), [5071](#), [5103](#), [5108](#), [5136](#), [5258](#), [5260](#), [5280](#), [5326](#), [5376](#), [5399](#), [5447](#), [5467](#), [5485](#), [5543](#), [5548](#), [5553](#), [5569](#), [5579](#), [5587](#), [5605](#), [5606](#), [5607](#), [5613](#), [5624](#), [5625](#), [5626](#), [5634](#), [5689](#), [5737](#), [5744](#), [5809](#), [5825](#), [5826](#), [5832](#)
- `__regex_compile_raw_error:N` ...
 - [5255](#), [5354](#), [5807](#), [5944](#)
- `__regex_compile_special:N`
 - [467](#), [5065](#), [5100](#), [5142](#), [5149](#), [5170](#), [5197](#), [5202](#), [5217](#), [5230](#), [5264](#), [5283](#), [5434](#), [5452](#), [5471](#), [5491](#), [5492](#), [5561](#), [5596](#), [5614](#), [5657](#), [5676](#), [5816](#), [5835](#)
- `__regex_compile_special_group_-`
 - `-:w` [5594](#)
- `__regex_compile_special_group_-`
 - `::w` [5590](#)
- `__regex_compile_special_group_-`
 - `i:w` [5594](#)
- `__regex_compile_special_group_-`
 - `l:w` [5590](#)
- `__regex_compile_u_brace:NNN` ...
 - [5810](#), [5811](#), [5814](#)
- `__regex_compile_u_end:` .. [5811](#), [5878](#)
- `__regex_compile_u_in_cs:` [5899](#), [5902](#)
- `__regex_compile_u_in_cs_aux:n` ..
 - [5912](#), [5915](#)
- `__regex_compile_u_loop:NN` [5820](#), [5830](#)
- `__regex_compile_u_not_cs:` [5897](#), [5921](#)
- `__regex_compile_u_payload:` [491](#), [5878](#)
- `__regex_compile_ur:n` [491](#), [5856](#)
- `__regex_compile_ur_aux:w` [5856](#)
- `__regex_compile_ur_end:`
 - [5810](#), [5824](#), [5856](#)
- `__regex_compile_use:n` ... [5086](#), [6301](#)
- `__regex_compile_use_aux:w` [5090](#), [5095](#)
- `__regex_compile_|:` [5577](#)
- `__regex_compute_case_changed_-`
 - `char:` [4532](#), [6781](#)
- `__regex_count:nnN` .. [7394](#), [7396](#), [7543](#)
- `\c_regex_cs_in_class_mode_int` ..
 - [4885](#), [5730](#)
- `\c_regex_cs_mode_int` ... [4885](#), [5728](#)
- `\l_regex_curr_analysis_tl`
 - [514](#), [6671](#), [6717](#), [6745](#), [6752](#), [6786](#), [6787](#)
- `\l_regex_curr_catcode_int`
 - .. [4556](#), [4575](#), [4583](#), [4595](#), [6657](#), [6784](#)
- `\l_regex_curr_char_int`
 - [516](#), [4496](#), [4502](#), [4503](#), [4510](#), [4520](#), [4521](#), [4534](#), [4535](#), [4536](#), [4537](#), [4543](#), [4576](#), [5331](#), [6331](#), [6613](#), [6621](#), [6657](#), [6741](#), [6780](#), [6783](#), [6799](#)

__regex_curr_cs_to_str:
 [4428](#), [4586](#), [4603](#)
 \l__regex_curr_pos_int
 . [451](#), [516](#), [6633](#), [6652](#), [6728](#), [6739](#),
 [6779](#), [6913](#), [6921](#), [7499](#), [7504](#), [7508](#),
 [7509](#), [7511](#), [8008](#), [8013](#), [8017](#), [8018](#)
 \l__regex_curr_state_int ... [513](#),
 [519](#), [6663](#), [6817](#), [6818](#), [6820](#), [6825](#),
 [6828](#), [6850](#), [6855](#), [6860](#), [6861](#), [6869](#)
 \l__regex_curr_submatches_tl ...
 [6664](#), [6735](#), [6830](#),
 [6862](#), [6863](#), [6874](#), [6896](#), [6900](#), [6925](#)
 \l__regex_curr_token_tl
 [4431](#), [6657](#), [6782](#)
 \l__regex_default_catcodes_int ..
 [466](#), [4892](#),
 [5016](#), [5018](#), [5128](#), [5426](#), [5526](#), [5539](#)
 __regex_disable_submatches: ...
 .. [4599](#), [5725](#), [6888](#), [7520](#), [7546](#), [7905](#)
 \l__regex_empty_success_bool ...
 [6674](#), [6720](#), [6724](#), [6919](#), [7607](#)
 __regex_end [7476](#)
 __regex_escape_\u:w [4722](#)
 __regex_escape_\/scan_stop:w [4722](#)
 __regex_escape_\a:w [4722](#)
 __regex_escape_\e:w [4722](#)
 __regex_escape_\f:w [4722](#)
 __regex_escape_\n:w [4722](#)
 __regex_escape_\r:w [4722](#)
 __regex_escape_\t:w [4722](#)
 __regex_escape_\x:w [4741](#)
 __regex_escape_\:w [4706](#)
 __regex_escape_\scan_stop:w . [4722](#)
 __regex_escape_escaped:N
 [4692](#), [4716](#), [4719](#)
 __regex_escape_loop:N
 [460](#), [4699](#), [4706](#),
 [4741](#), [4780](#), [4791](#), [4792](#), [4812](#), [4821](#)
 __regex_escape_raw:N
 [461](#), [4693](#), [4719](#), [4730](#),
 [4732](#), [4734](#), [4736](#), [4738](#), [4740](#), [4754](#)
 __regex_escape_unescaped:N
 [4691](#), [4709](#), [4719](#)
 __regex_escape_use:nnnn
 [459](#), [471](#), [4687](#), [5062](#), [6993](#)
 __regex_escape_x:N . [461](#), [4779](#), [4783](#)
 __regex_escape_x_end:w ... [461](#), [4741](#)
 __regex_escape_x_large:n [4741](#)
 __regex_escape_x_loop:N
 [461](#), [4776](#), [4795](#)
 __regex_escape_x_loop_error: . [4795](#)
 __regex_escape_x_loop_error:n ..
 [4801](#), [4813](#), [4818](#)
 __regex_escape_x_test:N
 [461](#), [4744](#), [4758](#)
 __regex_escape_x_testii:N ... [4758](#)
 \l__regex_every_match_tl
 [6673](#), [6756](#), [6766](#), [6803](#)
 __regex_extract: . [541](#), [552](#), [7561](#),
 [7568](#), [7581](#), [7718](#), [7763](#), [7791](#), [7980](#)
 __regex_extract_all:nnN . [7422](#), [7555](#)
 __regex_extract_aux:w [7718](#)
 __regex_extract_check:n [7669](#)
 __regex_extract_check:w
 [542](#), [544](#), [7629](#), [7669](#)
 __regex_extract_check_end:w ...
 [544](#), [7669](#)
 __regex_extract_check_loop:w . [7669](#)
 __regex_extract_map:N ... [7643](#), [7705](#)
 __regex_extract_map_aux:NNn . [7705](#)
 __regex_extract_map_loop:w
 [545](#), [7641](#), [7705](#)
 __regex_extract_once:nnN [7420](#), [7555](#)
 __regex_extract_seq_aux:n [7622](#), [7645](#)
 __regex_extract_seq_aux:ww .. [7645](#)
 \l__regex_fresh_thread_bool
 [515](#), [519](#), [6643](#),
 [6649](#), [6674](#), [6797](#), [6837](#), [6839](#), [6920](#)
 __regex_G_test:
 [465](#), [5363](#), [6003](#), [6109](#), [6592](#)
 __regex_get_digits:NtFw
 [4923](#), [5189](#), [5204](#)
 __regex_get_digits_loop:nw
 [4926](#), [4929](#), [4932](#)
 __regex_get_digits_loop:w ... [4923](#)
 __regex_group:nnnN [465](#), [483](#), [5568](#),
 [5573](#), [5859](#), [5990](#), [6092](#), [6263](#), [6443](#)
 __regex_group_aux:nnnnN
 [507](#), [6426](#), [6445](#), [6453](#), [6456](#)
 __regex_group_aux:nnnnnN [507](#)
 __regex_group_end_extract_seq:N
 [544](#), [7563](#), [7572](#), [7612](#), [7614](#)
 __regex_group_end_replace:N ...
 [7782](#), [7815](#), [7817](#)
 __regex_group_end_replace_
 check:n [548](#), [7817](#)
 __regex_group_end_replace_
 check:w [548](#), [7817](#)
 __regex_group_end_replace_try: .
 [548](#), [7817](#)
 \l__regex_group_level_int . [4884](#),
 [5015](#), [5033](#), [5035](#), [5037](#), [5527](#), [5533](#)
 __regex_group_no_capture:nnnN ..
 [465](#), [5591](#), [5859](#),
 [5860](#), [5872](#), [5884](#), [5991](#), [6094](#), [6443](#)
 __regex_group_repeat:nn . [6438](#), [6487](#)
 __regex_group_repeat:nnN [6439](#), [6527](#)

__regex_group_repeat:nnnN [6440](#), [6558](#)
 __regex_group_repeat_aux:n
 [508](#), [509](#), [6494](#), [6507](#), [6545](#), [6562](#)
 __regex_group_resetting:nnnN . . .
 . . . [465](#), [5593](#), [5860](#), [5992](#), [6096](#), [6454](#)
 __regex_group_resetting_
 loop:nnNn [6454](#)
 __regex_group_submatches:nnN . . .
 [6495](#), [6500](#), [6530](#), [6546](#), [6560](#)
 __regex_hexadecimal_use:N . . . [4823](#)
 __regex_hexadecimal_use:Ntf . . .
 [4778](#), [4790](#), [4803](#), [4823](#)
 __regex_if_end_range:NN [5278](#)
 __regex_if_end_range:NNTF . . . [5278](#)
 __regex_if_in_class:TF . . . [4947](#),
 [5026](#), [5113](#), [5129](#), [5262](#), [5325](#), [5387](#),
 [5403](#), [5548](#), [5579](#), [5587](#), [8097](#), [8110](#)
 __regex_if_in_class_or_catcode:TF
 [4967](#), [5354](#), [5376](#), [5806](#)
 __regex_if_in_cs:TF
 [4955](#), [5735](#), [5742](#), [8095](#), [8104](#)
 __regex_if_match:nn
 [7385](#), [7390](#), [7517](#), [7536](#)
 __regex_if_raw_digit:NN [4935](#)
 __regex_if_raw_digit:NNTF
 [4925](#), [4931](#), [4935](#)
 __regex_if_two_empty_matches:TF
 [515](#), [6674](#), [6725](#), [6731](#), [6916](#)
 __regex_if_within_catcode:TF . . .
 [4979](#), [5406](#)
 __regex_input_item:n . [549](#), [553](#),
 [554](#), [7861](#), [7922](#), [7944](#), [7985](#), [8009](#), [8018](#)
 \l__regex_input_tl
 [550](#), [551](#), [553](#), [7861](#),
 [7917](#), [7921](#), [7943](#), [7945](#), [8007](#), [8011](#)
 __regex_int_eval:w [4396](#),
 [6882](#), [6946](#), [6947](#), [6958](#), [7745](#), [7748](#)
 __regex_intarray_item:NnTF
 [4433](#), [6970](#), [6977](#)
 __regex_intarray_item_aux:NnTF [4433](#)
 \l__regex_internal_a_int [475](#), [528](#),
 [4459](#), [5189](#), [5200](#), [5211](#), [5220](#), [5224](#),
 [5232](#), [5235](#), [5239](#), [5242](#), [5249](#), [6408](#),
 [6411](#), [6417](#), [6422](#), [6496](#), [6511](#), [6517](#),
 [6523](#), [6532](#), [6535](#), [6539](#), [6542](#), [6547](#),
 [6550](#), [6553](#), [6568](#), [6576](#), [6585](#), [7157](#),
 [7178](#), [7734](#), [7743](#), [7745](#), [7748](#), [7750](#)
 \l__regex_internal_a_tl [459](#),
 [491](#), [492](#), [496](#), [547](#), [548](#), [4459](#), [4585](#),
 [4588](#), [4690](#), [4697](#), [4704](#), [5474](#), [5479](#),
 [5495](#), [5500](#), [5505](#), [5509](#), [5515](#), [5516](#),
 [5750](#), [5761](#), [5819](#), [5863](#), [5895](#), [5907](#),
 [5923](#), [6086](#), [6089](#), [6142](#), [6163](#), [6205](#),
 [6212](#), [6307](#), [6308](#), [6345](#), [6346](#), [6347](#),
 [6348](#), [6478](#), [6479](#), [6483](#), [6485](#), [6742](#),
 [6745](#), [7368](#), [7380](#), [7771](#), [7805](#), [7839](#)
 \l__regex_internal_b_int
 [4459](#), [5204](#),
 [5233](#), [5236](#), [5237](#), [5239](#), [5243](#), [5250](#),
 [6512](#), [6517](#), [6522](#), [6568](#), [6576](#), [6585](#)
 \l__regex_internal_b_tl
 [4459](#), [5818](#), [5838](#), [5851](#)
 \l__regex_internal_bool
 [4459](#), [5473](#), [5478](#), [5499](#), [5508](#)
 \l__regex_internal_c_int
 [4459](#), [6514](#), [6519](#), [6520](#), [6524](#)
 \l__regex_internal_regex
 [470](#), [4908](#), [5055](#), [5093](#), [5752](#),
 [5758](#), [6252](#), [7348](#), [7353](#), [7358](#), [7365](#)
 \l__regex_internal_seq [4459](#), [6218](#),
 [6219](#), [6224](#), [6231](#), [6232](#), [6233](#), [6235](#)
 \g__regex_internal_tl
 [542](#), [544](#), [545](#), [4459](#),
 [4695](#), [4699](#), [5904](#), [5911](#), [7619](#), [7630](#),
 [7631](#), [7681](#), [7684](#), [7709](#), [7831](#), [7836](#)
 __regex_item_caseful_equal:n . . .
 [465](#),
 [4494](#), [4616](#), [4617](#), [4621](#), [4622](#), [4623](#),
 [4624](#), [4625](#), [4634](#), [4639](#), [4657](#), [4675](#),
 [5019](#), [5618](#), [5786](#), [5918](#), [6037](#), [6110](#)
 __regex_item_caseful_range:nn . .
 [465](#), [4494](#), [4613](#), [4628](#), [4631](#), [4632](#),
 [4633](#), [4647](#), [4654](#), [4661](#), [4663](#), [4665](#),
 [4668](#), [4669](#), [4670](#), [4671](#), [4676](#), [4679](#),
 [4684](#), [4685](#), [5020](#), [5620](#), [6045](#), [6112](#)
 __regex_item_caseless_equal:n . .
 [465](#), [4508](#), [5599](#), [6038](#), [6117](#)
 __regex_item_caseless_range:nn . .
 [465](#), [4508](#), [5601](#), [6046](#), [6119](#)
 __regex_item_catcode: [4553](#)
 __regex_item_catcode:Ntf
 [465](#), [480](#), [4553](#), [5122](#), [5428](#), [6053](#), [6124](#)
 __regex_item_catcode_reverse:Ntf
 [465](#), [4553](#), [5429](#), [6054](#), [6126](#)
 __regex_item_cs:n
 [465](#), [4593](#), [5758](#), [6035](#), [6133](#)
 __regex_item_equal:n
 [4551](#), [5019](#), [5268](#),
 [5274](#), [5304](#), [5317](#), [5318](#), [5598](#), [5617](#)
 __regex_item_exact:nn
 [465](#), [492](#), [4573](#), [5933](#), [6047](#), [6130](#)
 __regex_item_exact_cs:n
 [465](#), [488](#), [4573](#), [5760](#), [5930](#), [6036](#), [6132](#)
 __regex_item_range:nn
 [4551](#), [5020](#), [5306](#), [5600](#), [5619](#)
 __regex_item_reverse:n
 [465](#), [481](#), [4489](#), [4572](#),
 [4638](#), [5342](#), [5499](#), [6039](#), [6128](#), [6616](#)

- \l_regex_last_char_int 547, 6718, 6719, 7478, 7621, 7800, 7808
- 6613, 6627, 6657, 6780, 6922
- \l_regex_last_char_success_int .
- 6657, 6715, 6741, 6922
- \l_regex_left_state_int .. 6240,
- 6261, 6281, 6285, 6339, 6346, 6357,
- 6364, 6367, 6368, 6370, 6371, 6397,
- 6405, 6408, 6431, 6479, 6481, 6491,
- 6511, 6531, 6533, 6561, 6564, 6567,
- 6570, 6582, 6595, 6604, 6640, 6647
- \l_regex_left_state_seq
- 6240, 6338, 6345, 6478
- __regex_maplike_break:
- 452, 550, 4444,
- 6687, 6701, 6747, 6761, 6769, 7925
- _regex_match:n 6680, 7523,
- 7550, 7560, 7570, 7596, 7760, 7793
- __regex_match_case:nnTF . 7399, 7526
- __regex_match_case_aux:nn ... 7526
- \l_regex_match_count_int
- 538, 540, 7475, 7547, 7548, 7553
- _regex_match_cs:n 4603, 6680
- _regex_match_init: 6680, 7916
- _regex_match_once_init:
- 6683, 6693, 6722, 6773, 7918
- __regex_match_once_init_aux: ...
- 6743, 6749
- _regex_match_one_active:n .. 6776
- _regex_match_one_token:nnN ...
- 516, 519, 550, 6685, 6686,
- 6697, 6698, 6700, 6746, 6776, 7923
- \l_regex_match_success_bool ...
- ... 515, 6677, 6734, 6760, 6768, 6918
- \l_regex_matched_analysis_tl ...
- 514, 6671, 6716, 6742, 6751, 6785, 6923
- \l_regex_max_pos_int
- 523, 6652, 7504,
- 7602, 7608, 7780, 7813, 7999, 8013
- \l_regex_max_state_int
- . 499, 503, 561, 6237, 6258, 6278,
- 6316, 6318, 6319, 6323, 6356, 6358,
- 6359, 6418, 6490, 6510, 6512, 6520,
- 6564, 6570, 6578, 6588, 6707, 8369
- \l_regex_max_thread_int
- 6667, 6691,
- 6737, 6790, 6793, 6798, 6875, 6883
- _regex_maybe_compute_ccc:
- 4513, 4525, 4548, 4550, 6781
- \l_regex_min_pos_int
- 523, 6652, 6713, 6714
- \l_regex_min_state_int 503, 6237,
- 6258, 6278, 6323, 6707, 6738, 8368
- \l_regex_min_submatch_int
- 538, 542,
- 547, 6718, 6719, 7478, 7621, 7800, 7808
- \l_regex_min_thread_int
- .. 6667, 6691, 6737, 6790, 6792, 6798
- \l_regex_mode_int 4885,
- 4949, 4957, 4960, 4969, 4972, 4981,
- 4989, 4992, 5002, 5003, 5005, 5007,
- 5061, 5075, 5077, 5389, 5393, 5394,
- 5395, 5422, 5433, 5550, 5640, 5641,
- 5669, 5670, 5726, 5727, 5896, 5942
- _regex_mode_quit_c: 5000, 5112, 5523
- _regex_msg_repeated:nnN
- 6178, 6199, 6209, 8338
- _regex_multi_match:n
- ... 514, 6754, 7548, 7568, 7577, 7791
- \c_regex_no_match_regex
- 4468, 4908, 7340
- \c_regex_outer_mode_int
- 4885, 4960, 4972, 4981,
- 4989, 5003, 5061, 5077, 5896, 5942
- _regex_peek:nnTF
- ... 552, 7865, 7874, 7883, 7893, 7901
- _regex_peek_aux:nnTF ... 7901, 7974
- _regex_peek_end:
- 549, 551, 7867, 7876, 7929
- \l_regex_peek_false_tl
- 7858, 7913, 7933, 7939, 8003
- _regex_peek_reinsert:N
- 551, 552, 7932, 7933, 7939, 7941, 8003
- _regex_peek_remove_end:n
- 549, 551, 7885, 7895, 7929
- _regex_peek_replace:nnTF
- 7956, 7964, 7971
- _regex_peek_replace_end: 7974, 7976
- _regex_peek_replacement_put:n .
- 7982, 8020
- _regex_peek_replacement_put_-
- submatch_aux:n 7984, 8031
- _regex_peek_replacement_-
- token:n 554, 7986, 8029
- _regex_peek_replacement_var:N .
- 7987, 8041
- \l_regex_peek_true_tl
- 551, 552, 7858, 7912, 7932, 7938, 7991
- _regex_pop_lr_states:
- 6296, 6328, 6336, 6436
- _regex_posix_alnum: 4641
- _regex_posix_alpha: 4641
- _regex_posix_ascii: 4641
- _regex_posix_blank: 4641
- _regex_posix_cntrl: 4641
- _regex_posix_digit: 4641
- _regex_posix_graph: 4641
- _regex_posix_lower: 4641
- _regex_posix_print: 4641

- _regex_posix_punct: [4641](#)
- _regex_posix_space: [4641](#)
- _regex_posix_upper: [4641](#)
- _regex_posix_word: [4641](#)
- _regex_posix_xdigit: [4641](#)
- _regex_prop_: [478](#), [5323](#)
- _regex_prop_d: [478](#), [4612](#), [4659](#)
- _regex_prop_h: [4612](#), [4651](#)
- _regex_prop_N: [4612](#), [5351](#)
- _regex_prop_s: [4612](#)
- _regex_prop_v: [4612](#)
- _regex_prop_w: [4612](#), [4680](#), [6614](#), [6616](#), [6617](#)
- _regex_push_lr_states: [6287](#), [6326](#), [6336](#), [6434](#)
- _regex_quark_if_nil:N [4485](#)
- _regex_quark_if_nil:NTF [5776](#), [5796](#)
- _regex_quark_if_nil:nTF [4485](#)
- _regex_quark_if_nil_p:n [4485](#)
- _regex_query_range:nn [523](#), [552](#), [6937](#), [6943](#), [6962](#), [7034](#), [7775](#), [7812](#), [7994](#)
- _regex_query_range_loop:ww . [6943](#)
- _regex_query_set:n [7496](#), [7562](#), [7571](#), [7597](#), [7764](#), [7794](#)
- _regex_query_set_aux:nN [7496](#)
- _regex_query_set_from_input_-tl: [7981](#), [8005](#)
- _regex_query_set_item:n [8005](#)
- _regex_query_submatch:n [6960](#), [7148](#), [7660](#), [8035](#), [8038](#)
- _regex_reinsert_item:n [551](#), [552](#), [7941](#), [7985](#), [8024](#)
- _regex_replace_all:nnN . [7426](#), [7786](#)
- _regex_replace_all_aux:nnN ... [7462](#), [7787](#), [7788](#)
- _regex_replace_once:nnN [7424](#), [7753](#)
- _regex_replace_once_aux:nnN ... [7439](#), [7753](#)
- _regex_replacement:n [552](#), [6985](#), [7441](#), [7754](#), [7787](#), [7988](#)
- _regex_replacement_apply:Nn ... [6985](#), [7064](#)
- _regex_replacement_balance_-one_match:n [522](#), [523](#), [6933](#), [7046](#), [7768](#), [7803](#)
- _regex_replacement_c:w [7187](#)
- _regex_replacement_c_A:w [526](#), [7119](#), [7275](#)
- _regex_replacement_c_B:w [7107](#), [7278](#)
- _regex_replacement_c_C:w ... [7287](#)
- _regex_replacement_c_D:w [7114](#), [7292](#)
- _regex_replacement_c_E:w [7108](#), [7295](#)
- _regex_replacement_c_L:w [7117](#), [7304](#)
- _regex_replacement_c_M:w [7109](#), [7307](#)
- _regex_replacement_c_O:w [7106](#), [7111](#), [7115](#), [7118](#), [7120](#), [7310](#)
- _regex_replacement_c_P:w [7112](#), [7313](#)
- _regex_replacement_c_S:w [7102](#), [7116](#), [7319](#)
- _regex_replacement_c_T:w [7110](#), [7327](#)
- _regex_replacement_c_U:w [7113](#), [7330](#)
- _regex_replacement_cat:NNN ... [7192](#), [7235](#)
- \l_regex_replacement_category_-seq [6930](#), [7016](#), [7019](#), [7020](#), [7089](#), [7249](#)
- \l_regex_replacement_category_-tl [527](#), [6930](#), [7084](#), [7090](#), [7093](#), [7250](#), [7251](#)
- _regex_replacement_char:nnN ... [534](#), [7270](#), [7277](#), [7284](#), [7294](#), [7301](#), [7306](#), [7309](#), [7312](#), [7316](#), [7329](#), [7332](#)
- \l_regex_replacement_csnames_-int [522](#), [6929](#), [7010](#), [7012](#), [7014](#), [7081](#), [7149](#), [7203](#), [7210](#), [7220](#), [7222](#), [7229](#), [7240](#), [7281](#), [7298](#), [8022](#), [8033](#)
- _regex_replacement_cu_aux:Nw ... [7197](#), [7201](#), [7215](#)
- _regex_replacement_do_one_-match:n [552](#), [553](#), [6935](#), [7032](#), [7773](#), [7811](#), [7992](#)
- _regex_replacement_error:NNN ... [7158](#), [7170](#), [7181](#), [7193](#), [7198](#), [7216](#), [7334](#)
- _regex_replacement_escaped:N ... [7006](#), [7125](#), [7254](#)
- _regex_replacement_exp_not:N ... [530](#), [6941](#), [7197](#), [7290](#), [7986](#)
- _regex_replacement_exp_not:n ... [6942](#), [7215](#), [7987](#)
- _regex_replacement_g:w [7154](#)
- _regex_replacement_g_digits:NN [7154](#)
- _regex_replacement_lbrace:N ... [6999](#), [7156](#), [7196](#), [7214](#), [7227](#)
- _regex_replacement_normal:n ... [7001](#), [7007](#), [7079](#), [7132](#), [7162](#), [7189](#), [7224](#), [7232](#), [7247](#)
- _regex_replacement_normal_-aux:N [7079](#)
- _regex_replacement_put:n [7077](#), [7082](#), [7273](#), [7325](#), [7982](#)
- _regex_replacement_put_-submatch:n [7130](#), [7136](#), [7177](#)
- _regex_replacement_put_-submatch_aux:n [7136](#), [7983](#)
- _regex_replacement_rbrace:N ... [6996](#), [7176](#), [7218](#)

```

\__regex_replacement_set:n 6985, 7067
\l__regex_replacement_tl .....
..... 7860, 7973, 7988
\__regex_replacement_u:w ..... 7212
\__regex_return: .....
... 536, 7386, 7391, 7416, 7418, 7488
\l__regex_right_state_int .....
..... 6240, 6264, 6308, 6309,
6329, 6341, 6348, 6357, 6358, 6397,
6404, 6410, 6423, 6431, 6481, 6485,
6496, 6510, 6519, 6531, 6535, 6539,
6542, 6547, 6550, 6553, 6561, 6575,
6578, 6581, 6584, 6588, 6604, 6647
\l__regex_right_state_seq .....
.. 6240, 6307, 6317, 6340, 6347, 6483
\l__regex_saved_success_bool ...
..... 515, 4601, 4608, 6677
\__regex_show:N 535, 6079, 7365, 7377
\__regex_show:NN ..... 7360
\__regex_show:Nn ..... 7360
\__regex_show_char:n .....
.. 6111, 6115, 6118, 6122, 6131, 6144
\__regex_show_class:NnnnN 6098, 6180
\__regex_show_group_aux:nnnnN ...
..... 6093, 6095, 6097, 6171
\__regex_show_item_catcode:NnTF .
..... 6125, 6127, 6216
\__regex_show_item_exact_cs:n ...
..... 6132, 6229
\l__regex_show_lines_int .....
..... 4910, 6152, 6184, 6187, 6194
\__regex_show_one:n .....
..... 6087, 6100, 6103, 6111,
6114, 6118, 6121, 6131, 6135, 6150,
6166, 6173, 6177, 6190, 6206, 6234
\__regex_show_pop: ..... 6160, 6176
\l__regex_show_prefix_seq . 4909,
6085, 6088, 6136, 6156, 6161, 6163
\__regex_show_push:n .....
..... 6137, 6160, 6174, 6185
\__regex_show_scope:nn .....
..... 6129, 6134, 6160, 6221
\__regex_single_match: .....
514, 4598, 6754, 7521, 7558, 7758, 7914
\__regex_split:nnN ..... 7428, 7574
\__regex_standard_escapechar: ...
..... 4397, 4694, 5060, 6256, 6277
\l__regex_start_pos_int .....
... 6633, 6652, 6728, 6733, 6740,
7580, 7592, 7605, 7608, 7731, 7813
\g__regex_state_active_intarray .
..... 450, 503, 514,
515, 6669, 6710, 6816, 6819, 6827, 6854
\l__regex_step_int 450, 6666, 6712,
6778, 6817, 6821, 6829, 6843, 6845
\__regex_store_state:n .....
..... 513, 6738, 6868, 6871
\__regex_store_submatches: ... 6871
\__regex_store_submatches:n .. 6890
\__regex_store_submatches:nn ...
..... 6873, 6877
\__regex_submatch_balance:n ....
..... 6934, 6966, 7049, 7151, 7649
\g__regex_submatch_begin_-
intarray .....
. 450, 522, 545, 6939, 6963, 6980,
7041, 7481, 7587, 7590, 7603, 7744
\g__regex_submatch_case_intarray
..... 7060, 7481, 7726, 7732
\g__regex_submatch_end_intarray .
..... 450, 545, 6964, 6973,
7481, 7584, 7600, 7747, 7777, 7996
\l__regex_submatch_int .....
450, 538, 541, 542, 547, 6719, 7478,
7599, 7601, 7604, 7606, 7609, 7622,
7721, 7725, 7727, 7728, 7802, 7810
\g__regex_submatch_prev_intarray
..... 450, 538, 545, 6938,
7037, 7481, 7582, 7598, 7724, 7730
\g__regex_success_bool .....
.... 515, 4602, 4604, 4607, 6677,
6705, 6759, 6771, 7443, 7466, 7490,
7539, 7720, 7761, 7931, 7937, 7978
\l__regex_success_pos_int .....
..... 6652, 6714, 6733, 6921, 7580
\l__regex_success_submatches_tl .
..... 513, 545, 6664, 6924, 7735
\__regex_tests_action_cost:n ...
..... 6375, 6396, 6405, 6423
\g__regex_thread_info_intarray ..
. 450, 512, 514, 520, 6669, 6809, 6880
\__regex_tl_even_items:n . 4446, 7464
\__regex_tl_even_items_loop:nn 4446
\__regex_tl_odd_items:n .....
..... 4446, 7440, 7463, 7537
\__regex_tmp:w ..... 542, 545,
4416, 4418, 4422, 4424, 4458, 5335,
5345, 5346, 5347, 5348, 5349, 5372,
5383, 5384, 7411, 7420, 7422, 7424,
7426, 7428, 7618, 7623, 7641, 7647,
7685, 7690, 7694, 7698, 7703, 7713
\__regex_toks_clear:N 4400, 6316, 6356
\__regex_toks_memcpy:NNn . 4405, 6521
\__regex_toks_put_left:Nn .....
.. 4414, 6285, 6309, 6351, 6503, 6504
\__regex_toks_put_right:Nn .....
..... 451, 4414, 6261, 6264,

```

- 6281, 6329, 6353, 6364, 6595, 6640
- _regex_toks_set:Nn [4400](#), [7509](#), [8018](#)
- _regex_toks_use:w [4399](#), [6818](#), [6956](#), [8372](#)
- _regex_trace:nnn [8354](#), [8371](#)
- _regex_trace_pop:nnN [8354](#)
- _regex_trace_push:nnN [8354](#)
- \g_regex_trace_regex_int [8364](#)
- _regex_trace_states:n [8365](#)
- _regex_two_if_eq:NNNN [4911](#)
- _regex_two_if_eq:NNNTF [4911](#), [5170](#), [5217](#), [5230](#), [5264](#), [5434](#), [5471](#), [5491](#), [5492](#), [5561](#), [5596](#), [5613](#), [5614](#), [5676](#), [5809](#), [5816](#), [7247](#)
- _regex_use_i_delimit_by_q-recursion_stop:nw [4480](#), [5799](#)
- _regex_use_none_delimit_by_q-nil:w [4454](#), [4480](#)
- _regex_use_none_delimit_by_q-recursion_stop:w [4480](#), [5777](#), [5801](#), [7736](#)
- _regex_use_state: . [6814](#), [6831](#), [6857](#)
- _regex_use_state_and_submatches:w [517](#), [6807](#), [6823](#)
- _regex_Z_test: [465](#), [5365](#), [5367](#), [5384](#), [6004](#), [6107](#), [6592](#)
- \l_regex_zeroth_submatch_int [538](#), [545](#), [7478](#), [7583](#), [7585](#), [7588](#), [7591](#), [7721](#), [7731](#), [7733](#), [7745](#), [7748](#), [7769](#), [7774](#), [7778](#), [7993](#), [7997](#)
- register commands:
 - register_lua_data [11843](#)
- \relax [4](#), [21](#), [25](#), [30](#), [34](#), [70](#), [71](#), [73](#), [82](#), [107](#), [120](#), [121](#), [122](#), [123](#), [124](#), [125](#), [126](#), [127](#), [128](#), [129](#), [130](#), [132](#), [133](#), [134](#), [135](#), [136](#), [137](#), [138](#), [139](#), [140](#), [141](#), [142](#), [423](#)
- \relpenalty [424](#)
- \RequirePackage [110](#)
- \resettimer [781](#)
- reverse commands:
 - \reverse_if:N [27](#), [716](#), [811](#), [812](#), [1008](#), [1416](#), [4227](#), [4502](#), [4503](#), [4520](#), [4521](#), [4526](#), [4527](#), [8576](#), [13649](#), [17014](#), [17164](#), [17166](#), [17168](#), [17170](#), [17225](#), [20058](#), [20063](#), [20067](#), [20069](#), [23224](#), [26804](#), [27626](#), [27649](#), [29135](#), [29137](#), [29159](#), [29183](#)
- \right [425](#)
- \rightghost [891](#)
- \righthyphenmin [426](#)
- \rightmarginkern [691](#)
- \rightskip [427](#)
- \rmfamily [31574](#)
- \romannumeral [428](#)
- round [253](#)
- \rPCODE [692](#)
- S**
- \saveboxresource [934](#)
- \savecatcodetable [892](#)
- \saveimageresource [935](#)
- \savepos [933](#)
- \savingshyphcodes [559](#)
- \savingsvdiscards [560](#)
- scan commands:
 - \scan_align_safe_stop: [36525](#)
 - \scan_new:N [141](#), [700](#), [782](#), [3320](#), [3321](#), [3729](#), [9165](#), [9166](#), [10325](#), [10326](#), [10791](#), [12798](#), [13073](#), [13074](#), [13075](#), [13084](#), [13085](#), [13740](#), [16186](#), [16211](#), [16212](#), [16213](#), [16923](#), [16924](#), [17805](#), [17806](#), [18489](#), [18958](#), [18959](#), [19338](#), [19339](#), [19534](#), [19539](#), [19540](#), [19938](#), [19939](#), [20283](#), [20446](#), [20447](#), [20448](#), [20449](#), [20707](#), [20708](#), [20709](#), [22340](#), [22343](#), [22344](#), [22345](#), [22346](#), [22348](#), [22349](#), [22350](#), [22351](#), [22352](#), [22451](#), [29247](#), [33918](#), [33932](#), [35588](#), [35843](#), [35844](#)
 - \scan_stop: [13](#), [22](#), [23](#), [141](#), [156](#), [193](#), [345](#), [348](#), [365](#), [366](#), [368](#), [377](#), [383](#), [394](#), [443](#), [457](#), [465](#), [488](#), [563](#), [666](#), [670](#), [677](#), [678](#), [680](#), [685](#), [691](#), [716](#), [781](#), [811](#), [816](#), [867](#), [876](#), [878](#), [879](#), [881](#), [892](#), [896](#), [897](#), [902](#), [1004](#), [1008-1010](#), [1013](#), [1210](#), [159](#), [173](#), [1445](#), [1800](#), [1818](#), [1828](#), [1846](#), [1872](#), [2201](#), [2224](#), [2233](#), [2242](#), [2314](#), [2756](#), [2880](#), [2881](#), [2896](#), [2936](#), [2962](#), [2986](#), [3003](#), [3207](#), [3213](#), [3354](#), [3735](#), [3868](#), [3908](#), [3912](#), [3918](#), [3920](#), [3967](#), [3969](#), [4229](#), [4235](#), [4237](#), [4253](#), [4255](#), [4265](#), [4266](#), [4267](#), [4273](#), [4282](#), [4586](#), [4587](#), [4700](#), [4760](#), [4785](#), [4797](#), [4933](#), [5769](#), [6069](#), [6073](#), [6076](#), [6231](#), [6833](#), [7272](#), [7324](#), [7618](#), [8880](#), [8884](#), [9136](#), [10137](#), [10142](#), [10242](#), [10352](#), [10355](#), [10904](#), [10911](#), [11306](#), [12111](#), [12281](#), [12291](#), [12358](#), [12387](#), [12389](#), [12753](#), [12773](#), [12787](#), [13650](#), [13850](#), [14839](#), [15921](#), [16195](#), [16198](#), [16571](#), [17355](#), [19003](#), [19075](#), [19387](#), [19448](#), [19524](#), [19527](#), [19529](#), [19950](#), [19969](#), [19971](#), [19975](#), [19978](#), [19981](#), [19985](#), [19990](#), [19994](#), [20217](#), [20293](#), [20311](#), [20313](#), [20321](#), [20323](#), [20327](#), [20329](#), [20350](#), [20355](#), [20358](#), [20384](#), [20404](#), [20406](#)

- 20414, 20416, 20420, 20422, 20426,
 21840, 21923, 22024, 22062, 22069,
 22231, 22326, 22515, 23222, 23226,
 23427, 23444, 23745, 23792, 23793,
 24048, 24091, 24119, 24133, 24902,
 26715, 26723, 27468, 27471, 27474,
 27477, 27480, 27483, 27486, 27489,
 27492, 28757, 28784, 29260, 29261,
 31969, 32117, 33695, 36326, 36329
 \s_stop [141](#), [782](#), [16198](#), 16207
 scan internal commands:
 \s__bool_mark ... [35843](#), 35864, 35878
 \s__bool_stop ... [35843](#), 35864, 35878
 \s__char_stop
 .. [18489](#), 18820, 18825, 18866, 18868
 \s__clist_mark [836](#), [839-841](#),
[846](#), [17805](#), 17807, 17835, 17836,
 17853, 17975, 17985, 17989, 18011,
 18073, 18079, 18093, 18105, 18106,
 18107, 18110, 18111, 18112, 18121,
 18122, 18131, 18327, 18328, 18340,
 18341, 18354, 18362, 18368, 18371
 \s__clist_stop
[840](#), [842](#), [846](#), [17805](#), 17808, 17809,
 17821, 17825, 17960, 17963, 17975,
 17978, 17986, 17989, 17997, 18011,
 18079, 18107, 18110, 18111, 18123,
 18131, 18174, 18175, 18182, 18186,
 18188, 18190, 18197, 18202, 18218,
 18219, 18245, 18246, 18253, 18258,
 18260, 18262, 18268, 18275, 18303,
 18308, 18329, 18340, 18341, 18342,
 18355, 18368, 18371, 18401, 18435
 \s__color_mark
[33932](#), 34172, 34174, 34177, 34184,
 34390, 34395, 34401, 34404, 34411,
 34442, 34535, 34541, 34617, 34620,
 34630, 34679, 35047, 35089, 35092,
 35110, 35116, 35232, 35261, 35264,
 35278, 35289, 35293, 35296, 35304,
 35310, 35314, 35317, 35330, 35340
 \s__color_stop [1313](#), [33918](#), 33960,
 33966, 33967, 33974, 33978, 33979,
 33982, 33988, 33990, 33992, 33994,
 33996, 33998, 34010, 34012, 34018,
 34039, 34068, 34085, 34091, 34113,
 34172, 34174, 34177, 34184, 34191,
 34193, 34202, 34213, 34214, 34216,
 34218, 34220, 34260, 34267, 34268,
 34269, 34278, 34287, 34390, 34395,
 34401, 34404, 34411, 34419, 34442,
 34535, 34541, 34569, 34572, 34605,
 34611, 34617, 34620, 34630, 34679,
 34698, 34702, 34727, 34729, 34731,
 34733, 34751, 34866, 34879, 34883,
 34894, 34901, 34909, 34915, 34923,
 34925, 34931, 34935, 34936, 34957,
 34959, 35038, 35044, 35047, 35055,
 35069, 35083, 35086, 35089, 35093,
 35096, 35106, 35110, 35116, 35143,
 35172, 35227, 35228, 35235, 35246,
 35247, 35261, 35264, 35278, 35289,
 35293, 35296, 35304, 35310, 35314,
 35317, 35330, 35340, 35384, 35385
 \s__cs_mark [365](#),
[399](#), [400](#), 1788, 1789, 1792, 1793,
 1794, [2880](#), 2910, 2911, 2913, 2919,
 2923, 2945, 2954, 2973, 3001, 3004,
 3012, 3027, 3059, 3073, 3077, 3086,
 3105, 3114, 3119, 3208, 3211, 3227
 \s__cs_stop [365](#), [400](#),
 1789, 1792, 1793, 1794, [2880](#), 2883,
 2884, 2914, 2923, 2949, 3001, 3004,
 3008, 3016, 3022, 3031, 3037, 3039,
 3059, 3081, 3086, 3116, 3119, 3208
 \s__dim_mark [19938](#), 20099, 20106
 \s__dim_stop [19938](#),
 19940, 20046, 20070, 20099, 20106
 \s__file_stop .. [647](#), 10764, 10769,
[10791](#), 10859, 10860, 10864, 10871,
 10873, 10874, 10965, 10966, 10971,
 10973, 10975, 11350, 11352, 11355,
 11356, 11358, 11370, 11446, 11449,
 11456, 11458, 11474, 11475, 11478
 \s__fp [965-967](#), [972](#), [973](#), [998](#), [1004](#),
[1006](#), [1008](#), [1022](#), [1024](#), [1025](#), [1055](#),
[1059](#), [1061](#), [1063](#), [1069](#), [1072](#), [1163](#),
[22340](#), 22353, 22354, 22355, 22356,
 22357, 22367, 22372, 22374, 22375,
 22390, 22403, 22406, 22408, 22418,
 22430, 22450, 22467, 22470, 22477,
 22484, 22500, 22527, 22633, 22635,
 22637, 22638, 22639, 22641, 22642,
 22643, 22645, 22661, 22821, 22826,
 23053, 23107, 23116, 23118, 23794,
 23949, 24431, 24446, 24470, 24490,
 24491, 24586, 24592, 24595, 24596,
 24637, 24662, 24663, 24677, 24678,
 24715, 24716, 24829, 24830, 24831,
 24840, 24856, 24860, 24924, 24925,
 24928, 24939, 24940, 24948, 24949,
 24951, 24952, 24953, 24955, 24956,
 24957, 24969, 24972, 24976, 24979,
 24999, 25049, 25052, 25055, 25075,
 25076, 25078, 25079, 25080, 25088,
 25091, 25102, 25103, 25105, 25114,
 25190, 25342, 25376, 25377, 25380,
 25461, 25599, 25607, 25609, 25786,

- 25795, 25797, 25802, 25810, 25812,
 25814, 25817, 26320, 26332, 26334,
 26543, 26560, 26562, 26743, 26762,
 26764, 26765, 26768, 26785, 26788,
 26791, 26815, 26816, 26818, 26834,
 26923, 26936, 26938, 26941, 26946,
 26979, 26995, 27078, 27091, 27093,
 27106, 27108, 27121, 27123, 27136,
 27138, 27151, 27153, 27166, 27176,
 27677, 27693, 27694, 27698, 27709,
 27816, 27829, 27831, 27847, 27850,
 27860, 27883, 27894, 27896, 27910,
 27912, 27917, 27979, 28000, 28003,
 28033, 28054, 28057, 28107, 28123,
 28126, 28201, 28202, 28312, 28314,
 28346, 28612, 28620, 28623, 28702
 \s__fp<type> 998
 \s__fp_division 22348
 \s__fp_exact 22348, 22353,
 22354, 22355, 22356, 22357, 24924
 \s__fp_expr_mark
 ... 1004, 1005, 1008, 1030, 1033,
 22343, 23998, 24011, 24092, 24134
 \s__fp_expr_stop 974,
 22343, 22541, 23900, 23999, 24003,
 24012, 25006, 25017, 25027, 25035
 \s__fp_invalid 22348
 \s__fp_mark 22345, 22490, 22491, 22495
 \s__fp_overflow 22348, 22374
 \s__fp_stop 972, 22345,
 22347, 22391, 22467, 22478, 22485,
 22491, 22495, 22509, 22528, 23322,
 23326, 23830, 23835, 24431, 24453,
 24586, 24592, 24636, 24637, 24662,
 24663, 24829, 24830, 24831, 24998,
 24999, 26619, 26634, 27953, 27957
 \s__fp_tuple 971, 22451,
 22457, 22458, 22535, 22537, 24211,
 24423, 24438, 24463, 24465, 24482,
 24483, 24485, 24583, 24707, 24708,
 25848, 25849, 25855, 25856, 27929
 \s__fp_underflow 22348, 22372
 \s__int_mark
 .. 16923, 17137, 17140, 17206, 17213
 \s__int_stop
 811, 822, 823, 16923, 16925,
 17116, 17132, 17134, 17138, 17151,
 17206, 17213, 17617, 17623, 17640
 \s__iow_mark . 10325, 10654, 10661,
 10673, 10747, 10748, 10749, 10750
 \s__iow_stop
 10325, 10327, 10540, 10581,
 10639, 10677, 10690, 10747, 10750
 \s__kernel_stop 2214, 2222, 2231, 2240
 \s__keys_mark
 20707, 20753, 20756, 20764, 20771,
 21454, 21457, 21462, 21468, 21702,
 21705, 21714, 21716, 21721, 21724
 \s__keys_nil
 20707, 20748, 20749, 20751,
 20753, 20756, 20761, 20770, 20771,
 20779, 21449, 21450, 21452, 21454,
 21457, 21460, 21468, 21469, 21701,
 21704, 21710, 21712, 21720, 21723
 \s__keys_stop
 20707, 20789, 20799, 20964, 21069,
 21076, 21437, 21447, 21634, 21654
 \s__keyval_mark 913-915,
 918, 20446, 20462, 20472, 20483,
 20484, 20485, 20486, 20492, 20493,
 20495, 20500, 20501, 20504, 20505,
 20506, 20511, 20512, 20516, 20517,
 20520, 20521, 20524, 20525, 20528,
 20531, 20532, 20537, 20540, 20541,
 20545, 20546, 20549, 20552, 20556,
 20557, 20558, 20559, 20566, 20567,
 20576, 20577, 20579, 20583, 20595,
 20596, 20604, 20605, 20616, 20617,
 20618, 20619, 20621, 20642, 20643,
 20648, 20652, 20654, 20656, 20668
 \s__keyval_nil 914,
 20446, 20491, 20499, 20504, 20506,
 20507, 20508, 20510, 20516, 20519,
 20524, 20528, 20530, 20537, 20539,
 20545, 20549, 20551, 20556, 20558,
 20566, 20577, 20595, 20604, 20641,
 20645, 20661, 20664, 20668, 20669
 \s__keyval_stop ... 20446, 20484,
 20486, 20497, 20505, 20517, 20525,
 20528, 20534, 20546, 20549, 20551,
 20552, 20556, 20566, 20595, 20596,
 20604, 20605, 20616, 20619, 20621
 \s__keyval_tail 914, 20446,
 20462, 20472, 20480, 20481, 20490,
 20574, 20576, 20582, 20583, 20618
 \s__msg_mark
 .. 9165, 9515, 9580, 9581, 9586, 9589
 \s__msg_stop 9165,
 9167, 9517, 9521, 9523, 9582, 10028
 \s__pdf_stop . 35588, 35633, 35634,
 35642, 35654, 35667, 35671, 35673
 \s__peek_mark
 19338, 19501, 19502, 19509
 \s__peek_stop
 .. 19338, 19340, 19490, 19503, 19512
 \s__prg_mark 1637, 1639, 1647
 \s__prg_stop 1664, 1669, 1688, 1696,
 1704, 1758, 1762, 1764, 1766, 1768

- \s__prop
 . [881](#), [885](#), [892](#), [893](#), [1361](#), [19534](#),
 [19534](#), [19535](#), [19538](#), [19636](#), [19639](#),
 [19774](#), [19797](#), [19846](#), [19847](#), [19848](#),
 [19849](#), [19853](#), [19854](#), [19855](#), [19856](#),
 [19870](#), [19884](#), [19885](#), [19886](#), [19887](#),
 [19891](#), [19892](#), [19893](#), [19894](#), [19915](#),
 [19917](#), [19926](#), [19929](#), [35892](#), [35897](#)
- \s__prop_mark
 [884](#), [885](#), [19539](#), [19636](#), [19638](#), [19639](#)
- \s__prop_stop [884](#),
 [885](#), [19539](#), [19636](#), [19639](#), [35881](#), [35888](#)
- \s__quark [782](#),
 [15921](#), [16156](#), [16158](#), [16159](#), [16170](#),
 [16173](#), [16178](#), [16181](#), [16183](#), [16204](#)
- \g__scan_marks_tl
 [782](#), [16185](#), [16188](#), [16194](#), [16199](#), [16201](#)
- \s__seq [783](#), [787](#),
 [793](#), [797](#), [800](#), [1361](#), [1364](#), [16211](#),
 [16222](#), [16252](#), [16257](#), [16262](#), [16267](#),
 [16278](#), [16310](#), [16338](#), [16346](#), [16350](#),
 [16573](#), [16621](#), [16841](#), [16894](#), [35902](#),
 [35908](#), [35939](#), [36000](#), [36012](#), [36014](#)
- \s__seq_mark
 . [16212](#), [16829](#), [16830](#), [16844](#), [16847](#)
- \s__seq_stop
 [16212](#), [16526](#), [16529](#), [16537](#), [16539](#),
 [16620](#), [16621](#), [16831](#), [16844](#), [16847](#),
 [16849](#), [35901](#), [35902](#), [35904](#), [35908](#),
 [35912](#), [35914](#), [35919](#), [36003](#), [36014](#)
- \s__skip_stop ... [20283](#), [20344](#), [20346](#)
- \s__sort_mark [417](#), [420-422](#),
 [3320](#), [3515](#), [3519](#), [3525](#), [3529](#), [3535](#),
 [3538](#), [3603](#), [3604](#), [3606](#), [3643](#), [3645](#),
 [3648](#), [3652](#), [3655](#), [3658](#), [3660](#), [3663](#)
- \s__sort_stop [419](#), [421](#), [422](#), [3320](#),
 [3591](#), [3600](#), [3604](#), [3606](#), [3643](#), [3644](#),
 [3645](#), [3650](#), [3652](#), [3656](#), [3658](#), [3666](#)
- \s__str [725](#),
 [733](#), [751](#), [754](#), [13740](#), [13889](#), [13893](#),
 [14075](#), [14122](#), [14190](#), [14193](#), [14637](#),
 [14649](#), [14654](#), [14664](#), [14669](#), [14674](#),
 [14677](#), [14692](#), [14705](#), [14708](#), [14843](#),
 [14844](#), [14861](#), [14867](#), [14883](#), [14889](#),
 [14890](#), [14995](#), [15010](#), [15019](#), [15020](#)
- \s__str_mark
 [703](#), [709](#), [716](#), [13084](#), [13270](#), [13301](#),
 [13308](#), [13391](#), [13408](#), [13656](#), [13658](#)
- \s__str_stop ... [709](#), [713](#), [749](#), [753](#),
 [754](#), [13084](#), [13086](#), [13087](#), [13179](#),
 [13270](#), [13301](#), [13308](#), [13391](#), [13400](#),
 [13406](#), [13408](#), [13414](#), [13431](#), [13450](#),
 [13512](#), [13569](#), [13581](#), [13619](#), [13635](#),
 [13642](#), [13650](#), [13652](#), [13656](#), [13658](#),
 [13889](#), [13895](#), [13937](#), [13942](#), [13950](#),
 [14145](#), [14148](#), [14167](#), [14173](#), [14552](#),
 [14554](#), [14562](#), [14650](#), [14686](#), [14800](#),
 [14802](#), [14806](#), [14818](#), [14958](#), [14960](#),
 [14964](#), [14976](#), [14985](#), [14992](#), [15013](#)
- \s__text_stop
 . [29247](#), [29322](#), [29324](#), [29681](#), [29682](#)
- \s__tl [426-429](#), [436](#), [437](#),
 [3728](#), [3729](#), [3965](#), [3996](#), [4002](#), [4027](#),
 [4045](#), [4050](#), [4064](#), [4076](#), [4106](#), [4109](#)
- \s__tl_act_stop [693](#), [12798](#),
 [12804](#), [12805](#), [12808](#), [12811](#), [12815](#),
 [12824](#), [12827](#), [12830](#), [12833](#), [12836](#),
 [12838](#), [12840](#), [12844](#), [12847](#), [12853](#)
- \s__tl_mark [682](#), [12431](#), [12441](#), [12560](#),
 [12561](#), [12564](#), [12567](#), [12568](#), [13073](#)
- \s__tl_nil [687](#), [12595](#),
 [12599](#), [12618](#), [12621](#), [12624](#), [13073](#)
- \s__tl_stop
 . [672](#), [682](#), [686](#), [12173](#), [12175](#),
 [12389](#), [12395](#), [12431](#), [12441](#), [12446](#),
 [12447](#), [12456](#), [12460](#), [12462](#), [12464](#),
 [12466](#), [12475](#), [12476](#), [12486](#), [12487](#),
 [12496](#), [12501](#), [12503](#), [12505](#), [12562](#),
 [12564](#), [12569](#), [12571](#), [12601](#), [12624](#),
 [12646](#), [12648](#), [12666](#), [12681](#), [12695](#),
 [12719](#), [12744](#), [13038](#), [13048](#), [13073](#)
- \s__token_mark
 . [874](#), [18958](#), [19317](#), [19318](#), [19327](#)
- \s__token_stop ... [868](#), [870](#), [18958](#),
 [19063](#), [19066](#), [19096](#), [19131](#), [19239](#),
 [19243](#), [19249](#), [19272](#), [19319](#), [19327](#)
- \scantextokens [893](#)
- \scantokens [561](#)
- \scriptbaselineshiftfactor [1150](#)
- \scriptfont [429](#)
- \scriptscriptbaselineshiftfactor . [1152](#)
- \scriptscriptfont [430](#)
- \scriptscriptstyle [431](#)
- \scriptsize [31591](#)
- \scriptspace [432](#)
- \scriptstyle [433](#)
- \scrollmode [434](#)
- \scshape [31580](#)
- sec [254](#)
- secd [254](#)
- \selectfont [31552](#)
- seq commands:
 \c_empty_seq [153](#), [784](#), [16222](#), [16226](#),
 [16230](#), [16233](#), [16427](#), [16505](#), [16513](#)
 \seq_clear:N
 . [142](#), [153](#), [6136](#), [7020](#), [9578](#), [9641](#),
 [11398](#), [11491](#), [16229](#), [16236](#), [16371](#)
 \seq_clear_new:N [142](#), [16235](#)

- \seq_concat:NNN [143](#), [153](#), [11404](#), [16324](#)
- \seq_const_from_clist:Nn . [143](#), [16275](#)
- \seq_count:N [145](#), [150](#), [152](#),
[236](#), [7019](#), [11516](#), [16442](#), [16632](#),
[16646](#), [16793](#), [16821](#), [35963](#), [36038](#)
- \seq_elt:w [783](#)
- \seq_elt_end: [783](#)
- \seq_gclear:N
. [142](#), [414](#), [3465](#), [16229](#), [16239](#), [16459](#)
- \seq_gclear_new:N [142](#), [16235](#)
- \seq_gconcat:NNN . . . [143](#), [11417](#), [16324](#)
- \seq_get:NN
. [151](#), [6478](#), [6483](#), [16874](#), [34511](#)
- \seq_get:NNTF [151](#), [16880](#)
- \seq_get_left:NN
[144](#), [16521](#), [16874](#), [16875](#), [16880](#), [16881](#)
- \seq_get_left:NNTF [146](#), [16591](#)
- \seq_get_right:NN [144](#), [16546](#)
- \seq_get_right:NNTF [146](#), [16591](#)
- \seq_gpop:NN
. [151](#), [11327](#), [16874](#), [28881](#), [34504](#)
- \seq_gpop:NNTF
[152](#), [10126](#), [10341](#), [16880](#), [28852](#), [28864](#)
- \seq_gpop_item:NnN [306](#), [36021](#)
- \seq_gpop_item:NnNTF [306](#), [36021](#)
- \seq_gpop_left:NN
[145](#), [16532](#), [16878](#), [16879](#), [16884](#), [16885](#)
- \seq_gpop_left:NNTF [146](#), [16599](#)
- \seq_gpop_right:NN [145](#), [16564](#)
- \seq_gpop_right:NNTF [146](#), [16599](#)
- \seq_gpush:Nn
. [30](#), [152](#), [10151](#), [10365](#), [11312](#),
[16854](#), [28856](#), [28866](#), [28875](#), [34447](#)
- \seq_gput_left:Nn [144](#), [16334](#),
[16864](#), [16865](#), [16866](#), [16867](#), [16868](#),
[16869](#), [16870](#), [16871](#), [16872](#), [16873](#)
- \seq_gput_right:Nn
. [144](#), [10766](#), [10773](#), [11301](#), [16355](#)
- \seq_gremove_all:Nn [147](#), [16381](#)
- \seq_gremove_duplicates:N . [147](#), [16365](#)
- \seq_greverse:N [147](#), [16407](#)
- \seq_gset_eq:NN
[142](#), [3439](#), [16233](#), [16241](#), [16368](#), [16439](#)
- \seq_gset_filter:NNn [305](#), [35922](#)
- \seq_gset_from_clist:NN . . [143](#), [16249](#)
- \seq_gset_from_clist:Nn . . [143](#), [16249](#)
- \seq_gset_from_function:NnN
. [305](#), [35942](#)
- \seq_gset_from_inline_x:Nnn
. [305](#), [3457](#), [16454](#), [35932](#), [35945](#)
- \seq_gset_item:Nnn [306](#), [35947](#)
- \seq_gset_item:NnnTF [306](#), [35947](#)
- \seq_gset_map:Nnn [150](#), [16783](#)
- \seq_gset_map_x:Nnn [150](#), [16773](#)
- \seq_gset_split:Nnn [143](#), [16281](#)
- \seq_gset_split_keep_spaces:Nnn
. [143](#), [16281](#)
- \seq_gshuffle:N [147](#), [16435](#)
- \seq_gsort:Nn [147](#), [3435](#), [16425](#)
- \seq_if_empty:NNTF
[147](#), [7016](#), [16425](#), [16645](#), [17883](#), [28915](#)
- \seq_if_empty_p:N [147](#), [16425](#)
- \seq_if_exist:NNTF
. [144](#), [16236](#), [16239](#), [16330](#), [16819](#)
- \seq_if_exist_p:N [144](#), [16330](#)
- \seq_if_in:Nn [841](#)
- \seq_if_in:NnTF [148](#),
[152](#), [153](#), [10150](#), [10364](#), [16374](#), [16482](#)
- \seq_indexed_map_function:NN . [36607](#)
- \seq_indexed_map_inline:Nn . . . [36607](#)
- \seq_item:Nn [56](#), [145](#),
[796](#), [9659](#), [9660](#), [9665](#), [16619](#), [16646](#)
- \seq_log:N [154](#), [16886](#)
- \seq_map_break:
[149](#), [150](#), [305](#), [16649](#), [16662](#), [16703](#),
[16713](#), [16736](#), [16743](#), [16752](#), [21497](#)
- \seq_map_break:n . . . [149](#), [796](#), [3436](#),
[3439](#), [9598](#), [9612](#), [10937](#), [11037](#), [16649](#)
- \seq_map_function:NN
. [6](#), [148](#), [303](#), [798](#), [6156](#),
[6224](#), [9663](#), [11407](#), [16653](#), [16901](#), [17889](#)
- \seq_map_indexed_function:NN
. [148](#), [16740](#), [36609](#), [36610](#)
- \seq_map_indexed_inline:Nn
. [149](#), [16740](#), [36607](#), [36608](#)
- \seq_map_inline:Nn
. [148](#), [153](#), [1362](#), [3436](#),
[3439](#), [9593](#), [11036](#), [16372](#), [16699](#), [21490](#)
- \seq_map_tokens:Nn
. [148](#), [10936](#), [11520](#), [16706](#)
- \seq_map_variable:Nnn . . . [148](#), [16728](#)
- \seq_mapthread_function:Nnn
. [305](#), [35900](#)
- \seq_new:N [6](#), [142](#),
[3307](#), [4465](#), [4909](#), [6242](#), [6243](#), [6931](#),
[9548](#), [9549](#), [10078](#), [10309](#), [10758](#),
[10783](#), [10789](#), [10790](#), [16223](#), [16236](#),
[16239](#), [16364](#), [16437](#), [16911](#), [16912](#),
[16913](#), [16914](#), [18034](#), [18586](#), [18589](#),
[20699](#), [28706](#), [28707](#), [28708](#), [34431](#)
- \seq_pop:NN
. [151](#), [6307](#), [6345](#), [6347](#), [7089](#), [16874](#)
- \seq_pop:NNTF [152](#), [16880](#)
- \seq_pop_item:NnN [306](#), [36021](#)
- \seq_pop_item:NnNTF [306](#), [36021](#)
- \seq_pop_left:NN
[144](#), [16532](#), [16876](#), [16877](#), [16882](#), [16883](#)
- \seq_pop_left:NNTF [146](#), [16599](#)

- \seq_pop_right:NN [145](#), [6085](#), [6163](#), [16564](#)
- \seq_pop_right:NNTF [146](#), [16599](#)
- \seq_push:Nn [152](#),
[6317](#), [6338](#), [6340](#), [7249](#), [16854](#), [16861](#)
- \seq_put_left:Nn [144](#), [9588](#), [16334](#),
[16854](#), [16855](#), [16856](#), [16857](#), [16858](#),
[16859](#), [16860](#), [16861](#), [16862](#), [16863](#)
- \seq_put_right:Nn
..... [144](#), [152](#), [153](#), [6088](#),
[6161](#), [9649](#), [11493](#), [16355](#), [16375](#), [36458](#)
- \seq_rand_item:N [145](#), [16643](#)
- \seq_remove_all:Nn [143](#),
[147](#), [152](#), [153](#), [16381](#), [18066](#), [36460](#)
- \seq_remove_duplicates:N
..... [147](#), [152](#), [153](#), [11405](#), [16365](#)
- \seq_reverse:N [147](#), [789](#), [16407](#)
- \seq_set_eq:NN [142](#),
[153](#), [3436](#), [16230](#), [16241](#), [16366](#), [16438](#)
- \seq_set_filter:Nnn
..... [305](#), [799](#), [6219](#), [35922](#)
- \seq_set_from_clist:NN
..... [143](#), [16249](#), [18065](#)
- \seq_set_from_clist:Nnn [143](#),
[172](#), [785](#), [11401](#), [11415](#), [16249](#), [21413](#)
- \seq_set_from_function:Nnn
..... [305](#), [7642](#), [35942](#)
- \seq_set_from_inline_x:Nnn
..... [305](#), [1363](#), [35932](#), [35943](#)
- \seq_set_item:Nnn ... [306](#), [1365](#), [35947](#)
- \seq_set_item:NnnTF [306](#), [35947](#)
- \seq_set_map:Nnn [150](#), [16783](#)
- \seq_set_map_x:Nnn
..... [150](#), [799](#), [6232](#), [16773](#)
- \seq_set_split:Nnn
..... [143](#), [6218](#), [6231](#), [16281](#), [18587](#), [18590](#)
- \seq_set_split_keep_spaces:Nnn ..
..... [143](#), [16281](#)
- \seq_show:N [154](#), [597](#), [699](#), [16886](#)
- \seq_shuffle:N [147](#), [16435](#)
- \seq_sort:Nn [44](#), [147](#), [3435](#), [16425](#)
- \seq_use:Nn [151](#), [6235](#), [16817](#)
- \seq_use:Nnnn [150](#), [16817](#)
- \g_tmpa_seq [154](#), [16911](#)
- \l_tmpa_seq [154](#), [16911](#)
- \g_tmpb_seq [154](#), [16911](#)
- \l_tmpb_seq [154](#), [16911](#)
- seq internal commands:
- _seq_count:w [800](#), [16793](#)
- _seq_count_end:w [799](#), [16793](#)
- _seq_get_left:wnw [16521](#)
- _seq_get_right_end:NnN [16546](#)
- _seq_get_right_loop:nw . [793](#), [16546](#)
- _seq_if_in: [16482](#)
- _seq_int_eval:w [35946](#), [36002](#), [36012](#)
- _l_seq_internal_a_int
 .. [16449](#), [16455](#), [16464](#), [16466](#), [16467](#)
- _l_seq_internal_a_tl [786](#),
[1364](#), [1365](#), [16219](#), [16293](#), [16297](#),
[16303](#), [16308](#), [16310](#), [16396](#), [16401](#),
[16486](#), [16490](#), [35961](#), [36017](#), [36035](#)
- _l_seq_internal_b_int
 [16465](#), [16468](#), [16469](#)
- _l_seq_internal_b_tl
 [1365](#), [16219](#), [16392](#), [16396](#),
[16489](#), [16490](#), [36036](#), [36044](#), [36051](#)
- _g_seq_internal_seq [16435](#)
- _seq_item:n [783](#), [788](#),
[791](#), [792](#), [794](#)–[797](#), [800](#), [1361](#), [1362](#),
[16214](#), [16338](#), [16346](#), [16356](#), [16358](#),
[16363](#), [16413](#), [16414](#), [16416](#), [16421](#),
[16451](#), [16487](#), [16526](#), [16529](#), [16539](#),
[16554](#), [16557](#), [16570](#), [16571](#), [16582](#),
[16626](#), [16635](#), [16660](#), [16665](#), [16666](#),
[16667](#), [16668](#), [16680](#), [16685](#), [16691](#),
[16695](#), [16711](#), [16717](#), [16718](#), [16719](#),
[16720](#), [16763](#), [16765](#), [16779](#), [16789](#),
[16800](#), [16801](#), [16802](#), [16803](#), [16804](#),
[16805](#), [16806](#), [16807](#), [16812](#), [16813](#),
[16828](#), [16843](#), [16846](#), [16849](#), [35938](#),
[35939](#), [35961](#), [36007](#), [36010](#), [36048](#)
- _seq_item:nN [16619](#)
- _seq_item:nwn [16619](#)
- _seq_item:wNn [16619](#)
- _seq_map_function:Nw ... [796](#), [16653](#)
- _seq_map_indexed:NN
 [16742](#), [16750](#), [16755](#)
- _seq_map_indexed:nNN [16740](#)
- _seq_map_indexed:Nw ... [798](#), [16740](#)
- _seq_map_tokens:nw [16706](#)
- _seq_mapthread_function:Nnnwnn
 [35900](#)
- _seq_mapthread_function:wNN . [35900](#)
- _seq_mapthread_function:wNw . [35900](#)
- _seq_pop:Nnnn
 .. [16503](#), [16533](#), [16535](#), [16565](#), [16567](#)
- _seq_pop_item:nn [1365](#), [36039](#), [36041](#)
- _seq_pop_item:Nnnn
 [36028](#), [36030](#), [36033](#)
- _seq_pop_item:Nnnnn [36021](#)
- _seq_pop_item_aux:w
 [1365](#), [36039](#), [36048](#)
- _seq_pop_item_def:
 . [783](#), [16403](#), [16453](#), [16677](#), [16703](#),
[16736](#), [16781](#), [16791](#), [35930](#), [35940](#)
- _seq_pop_left:Nnn
 [16532](#), [16601](#), [16604](#)
- _seq_pop_left:wnwnnn [16532](#)

- __seq_pop_right:NNN [788](#), [16564](#), [16607](#), [16610](#)
- __seq_pop_right_loop:nn [16564](#)
- __seq_pop_TF:NNNN [794](#), [16503](#), [16592](#), [16594](#), [16601](#), [16604](#), [16607](#), [16610](#)
- __seq_push_item_def: . [16450](#), [16677](#)
- __seq_push_item_def:n [783](#), [16387](#), [16677](#), [16701](#), [16730](#), [16779](#), [16789](#), [35928](#), [35938](#)
- __seq_put_left_aux:w ... [787](#), [16334](#)
- __seq_remove_all_aux:NNn [16381](#)
- __seq_remove_duplicates:NN . [16365](#)
- \l__seq_remove_seq [16364](#), [16371](#), [16374](#), [16375](#), [16377](#)
- __seq_reverse:NN [16407](#)
- __seq_reverse_item:nw [789](#), [790](#)
- __seq_reverse_item:nwn [16407](#)
- __seq_set_filter:NNNn [35922](#)
- __seq_set_from_inline_x:NNnn . [35932](#)
- __seq_set_item:NnnNN [35947](#)
- __seq_set_item:nnNNNN [1365](#), [35947](#), [36037](#)
- __seq_set_item:nNnnNNNN [1364](#), [35947](#)
- __seq_set_item:wn [35947](#)
- __seq_set_item_end:w .. [1364](#), [35947](#)
- __seq_set_item_false:nnNNNN ... [1364](#), [35947](#)
- __seq_set_map:NNNn [16783](#)
- __seq_set_map_x:NNNn [16773](#)
- __seq_set_split:NNnn [16281](#)
- __seq_set_split:NNNnn [16282](#), [16284](#), [16286](#), [16288](#), [16289](#)
- __seq_set_split:Nw [16281](#)
- __seq_set_split:w [16281](#)
- __seq_set_split_auxi:w [786](#)
- __seq_set_split_auxii:w [786](#)
- __seq_set_split_end: ... [786](#), [16281](#)
- __seq_show:NN [16886](#)
- __seq_show_validate:nn [16886](#)
- __seq_shuffle:NN [16435](#)
- __seq_shuffle_item:n [16435](#)
- __seq_tmp:w [16221](#), [16413](#), [16416](#), [16570](#), [16582](#)
- __seq_use:NNnNnn [16817](#)
- __seq_use:nwn [16817](#)
- __seq_use:nwwwnwn [16817](#)
- __seq_use_setup:w [16817](#)
- __seq_wrap_item:n [786](#), [1362](#), [16252](#), [16257](#), [16262](#), [16267](#), [16278](#), [16294](#), [16319](#), [16363](#), [16399](#), [16908](#), [35928](#)
- \setbox [435](#)
- \setfontid [894](#)
- \setlanguage [436](#)
- \setrandomseed [936](#)
- \c_seven [36413](#)
- \sfcode [437](#)
- \sffamily [31575](#), [33690](#)
- \shapemode [895](#)
- \shellescape [782](#)
- \Shipout [1218](#)
- \shipout [438](#), [1205](#), [1206](#)
- \ShortText [54](#), [96](#)
- \show [439](#)
- \showbox [440](#)
- \showboxbreadth [441](#)
- \showboxdepth [442](#)
- \showgroups [562](#)
- \showifs [563](#)
- \showlists [443](#)
- \showmode [1154](#)
- \showthe [444](#)
- \showtokens [564](#)
- sign [253](#)
- sin [254](#)
- sind [254](#)
- \c_six [36411](#)
- \c_sixteen [36431](#)
- \sjis [1155](#)
- \skewchar [445](#)
- \skip [446](#), [19155](#)
- skip commands:
 - \c_max_skip [217](#), [20369](#)
 - \skip_add:Nn [215](#), [20320](#)
 - \skip_const:Nn [215](#), [910](#), [20290](#), [20369](#), [20370](#)
 - \skip_eval:n [216](#), [20293](#), [20334](#), [20349](#), [20364](#), [20368](#)
 - \skip_gadd:Nn [215](#), [20320](#)
 - \skip_gset:N [226](#), [21228](#)
 - \skip_gset:Nn [215](#), [906](#), [20310](#)
 - \skip_gset_eq:NN [215](#), [20316](#)
 - \skip_gsub:Nn [215](#), [20320](#)
 - \skip_gzero:N [215](#), [20296](#), [20303](#)
 - \skip_gzero_new:N [215](#), [20300](#)
 - \skip_horizontal:N [217](#), [20353](#)
 - \skip_horizontal:n [217](#), [20353](#)
 - \skip_if_eq:nnTF [216](#), [20332](#)
 - \skip_if_eq_p:nn [216](#), [20332](#)
 - \skip_if_exist:NTF [215](#), [20301](#), [20303](#), [20306](#)
 - \skip_if_exist_p:N [215](#), [20306](#)
 - \skip_if_finite:nTF [216](#), [20338](#)
 - \skip_if_finite_p:n [216](#), [20338](#)
 - \skip_log:N [217](#), [20365](#)
 - \skip_log:n [217](#), [20365](#)

- \skip_new:N 214, 215, 20284, 20292, 20301, 20303, 20371, 20372, 20373, 20374
- .skip_set:N 226, 21228
- \skip_set:Nn 215, 20310
- \skip_set_eq:NN 215, 20316
- \skip_show:N 216, 20361
- \skip_show:n 216, 909, 20363
- \skip_sub:Nn 215, 20320
- \skip_use:N .. 216, 20343, 20350, 20351
- \skip_vertical:N 218, 20353
- \skip_vertical:n 218, 20353
- \skip_zero:N 215, 218, 896, 20296, 20301
- \skip_zero_new:N 215, 20300
- \g_tmpa_skip 217, 20371
- \l_tmpa_skip 217, 20371
- \g_tmpb_skip 217, 20371
- \l_tmpb_skip 217, 20371
- \c_zero_skip 217, 896, 19953, 19955, 20369
- skip internal commands:
 - _skip_if_finite:wwNw 20338
 - _skip_tmp:w 20338, 20348
- \skipdef 447
- \slshape 31581
- \small 31592
- sort commands:
 - \sort_ordered: 36527
 - \sort_return_same: 43, 44, 417, 3518, 36528
 - \sort_return_swapped: 43, 44, 417, 3518, 36530
 - \sort_reversed: 36529
- sort internal commands:
 - __sort:nnNnn 418, 419
 - \l__sort_A_int .. 416, 3317, 3324, 3331, 3334, 3343, 3482, 3487, 3490, 3510, 3542, 3549, 3564, 3566, 3567
 - \l__sort_B_int 416, 3317, 3487, 3491, 3499, 3501, 3502, 3554, 3555, 3564, 3565, 3574, 3575, 3577
 - \l__sort_begin_int 411, 416, 3315, 3479, 3567, 3577
 - \l__sort_block_int . 410, 411, 415, 3314, 3326, 3331, 3335, 3338, 3343, 3344, 3409, 3470, 3473, 3480, 3483
 - \l__sort_C_int 416, 3317, 3488, 3492, 3499, 3500, 3511, 3543, 3550, 3554, 3556, 3557, 3574, 3576
 - __sort_compare:nn 413, 417, 3408, 3509
 - __sort_compute_range: 410-412, 3348, 3396
 - __sort_copy_block: . 415, 3489, 3497
 - __sort_disable_toksdef: . 3395, 3674
 - __sort_disabled_toksdef:n ... 3674
 - \l__sort_end_int 411, 415, 416, 3315, 3471, 3479, 3480, 3481, 3482, 3483, 3484, 3485, 3502
 - __sort_error: 3668, 3680, 3698
 - __sort_i:nnnnNn 420
 - \g__sort_internal_seq 413, 414, 3307, 3457, 3464, 3465
 - \g__sort_internal_tl 3307, 3420, 3423, 3424
 - \l__sort_length_int 410, 411, 3309, 3406, 3470
 - __sort_level: 413, 423, 3410, 3468, 3672
 - __sort_loop:wNn 419, 420
 - __sort_main:NNNn 414, 3393, 3419, 3456
 - \l__sort_max_int 410, 411, 3309, 3328, 3400
 - \c__sort_max_length_int 3348
 - __sort_merge_blocks: 3472, 3477, 3671
 - __sort_merge_blocks_aux: 415, 3493, 3507, 3560, 3570, 3670
 - __sort_merge_blocks_end: 418, 3568, 3572
 - \l__sort_min_int 410, 411, 413, 3309, 3325, 3333, 3350, 3366, 3374, 3387, 3397, 3407, 3421, 3460, 3471, 3696, 3697
 - __sort_quick_cleanup:w 3582
 - __sort_quick_end:nnTFNn 421, 422, 3602, 3642
 - __sort_quick_only_i:NnnnnNn . 3607
 - __sort_quick_only_i_end:nnnwnw . 3618, 3642
 - __sort_quick_only_ii:NnnnnNn . 3607
 - __sort_quick_only_ii_end:nnnwnw 3625, 3642
 - __sort_quick_prepare:Nnnn ... 3582
 - __sort_quick_prepare_end:NNNnw 3582
 - __sort_quick_single_end:nnnwnw . 3611, 3642
 - __sort_quick_split:NnNn ... 420, 421, 3602, 3607, 3647, 3654, 3660, 3662
 - __sort_quick_split_end:nnnwnw . 3632, 3639, 3642
 - __sort_quick_split_i:NnnnnNn ... 419, 3607
 - __sort_quick_split_ii:NnnnnNn 3607
 - __sort_redefine_compute_range: 3348
 - __sort_return_mark:w 417, 3513, 3514, 3518
 - __sort_return_none_error: 417, 3516, 3518, 3552, 3562

- __sort_return_same:w 417, 3526, 3544, 3552
- __sort_return_swapped:w . 3536, 3562
- __sort_return_two_error: . 417, 3518
- __sort_seq:NNNNn 413, 3435
- __sort_shrink_range: 411, 412, 3322, 3352, 3368, 3376, 3389
- __sort_shrink_range_loop: ... 3322
- __sort_tl:NNn 413, 3412
- __sort_tl_toks:w 413, 3412
- __sort_too_long_error:NNw 3401, 3691
- \l__sort_top_int 410, 413, 416, 3309, 3397, 3400, 3403, 3404, 3407, 3429, 3460, 3481, 3484, 3485, 3488, 3557, 3697
- \l__sort_true_max_int 410, 411, 3309, 3325, 3338, 3351, 3367, 3375, 3388, 3696
- sp 257
- spac commands:
 - \spac_directions_normal_body_dir 1361
 - \spac_directions_normal_page_dir 1362
 - \spacefactor 448
 - \spaceskip 449
 - \span 450
 - \special 451
 - \splitbotmark 452
 - \splitbotmarks 565
 - \splitdiscards 566
 - \splitfirstmark 453
 - \splitfirstmarks 567
 - \splitmaxdepth 454
 - \splittopskip 455
 - sqr 255
 - \SS 29402, 31301, 31677
 - \ss 29402, 31301, 31673
- str commands:
 - \c_ampersand_str 132, 13701
 - \c_at_sign_str 132, 13701
 - \c_backslash_str 132, 4712, 5314, 13701, 14403, 14405, 14428, 14457, 14459, 14491, 14500, 14504
 - \c_circumflex_str 132, 13701
 - \c_colon_str ... 132, 13701, 19066, 19243, 19249, 20799, 34605, 34610
 - \c_dollar_str 132, 13701
 - \c_hash_str 132, 13701, 14371, 14474, 15049, 15050, 15053, 15056, 29183, 29214, 29215, 29219
 - \c_left_brace_str 132, 467, 4775, 5187, 5191, 5211, 5224, 5248, 5722, 5733, 5737, 5816, 5840, 6998, 13701, 29500
 - \c_percent_str 132, 13701, 14373, 14527
 - \c_right_brace_str 132, 4811, 5197, 5217, 5230, 5740, 5744, 5837, 6995, 13701, 29508
 - str_byte 13770
 - \str_case:nn ... 123, 5454, 10001, 11190, 13249, 35123, 35161, 36056
 - \str_case:nnn 36531, 36533
 - \str_case:nnTF ... 123, 899, 8798, 8833, 9140, 9837, 13249, 13254, 13259, 20987, 21033, 36532, 36534
 - \str_case_e:nn ... 124, 13249, 36536
 - \str_case_e:nnTF 124, 2691, 5195, 13249, 13285, 13290, 14426, 36538, 36540, 36542, 36544
 - \str_case_x:nn 36535
 - \str_case_x:nnn 36537
 - \str_case_x:nnTF . 36539, 36541, 36543
 - \c_str_cctab 262, 1184, 28986
 - \str_clear:N 121, 13092, 20781, 20924, 21435, 21436
 - \str_clear_new:N 121, 13092
 - \str_compare:nNnTF 124, 13204
 - \str_compare_p:nNn 124, 13204
 - \str_concat:NNN 121, 13092
 - \str_const:Nn 121, 8706, 8725, 8743, 8789, 9102, 11575, 11582, 11586, 11590, 13119, 13701, 13702, 13703, 13704, 13705, 13706, 13707, 13708, 13709, 13710, 13711, 13712, 13713, 14467, 14468, 14490, 20676, 20677, 20678, 20679, 20680, 20681, 20682, 36054
 - \str_convert_pdfname:n 135, 15025, 34889
 - \str_count:N ... 126, 4167, 9286, 9287, 9469, 9470, 10440, 10518, 13591
 - \str_count:n 126, 4161, 13591
 - \str_count_ignore_spaces:n 126, 714, 3775, 13591
 - \str_count_spaces:N 126, 13571
 - \str_count_spaces:n 126, 714, 13571, 13597
 - \str_declare_eight_bit_encoding:nnn 36604
 - str_end 14928
 - str_error 13770
 - \str_fold_case:n 36592
 - \str_foldcase:n 130, 131, 183, 266, 13659, 23413, 34792, 36600, 36601, 36602, 36603
 - \str_gclear:N 121, 13092

- `\str_gclear_new:N` [13092](#)
- `\str_gconcat:NNN` [121](#), [13092](#)
- `\str_gput_left:Nn` [122](#), [13119](#)
- `\str_gput_right:Nn` [122](#), [13119](#)
- `\str_gremove_all:Nn` [129](#), [13189](#)
- `\str_gremove_once:Nn` [129](#), [13183](#)
- `\str_greplace_all:Nnn`
..... [129](#), [13143](#), [13192](#)
- `\str_greplace_once:Nnn`
..... [129](#), [13143](#), [13186](#)
- `.str_gset:N` [226](#), [21236](#)
- `\str_gset:Nn`
..... [122](#), [11333](#), [11334](#), [11335](#), [13119](#)
- `\str_gset_convert:Nnnn` ... [135](#), [13909](#)
- `\str_gset_convert:NnnnTF` . [135](#), [13909](#)
- `\str_gset_eq:NN`
..... [121](#), [11320](#), [11321](#), [11322](#), [13092](#)
- `.str_gset_x:N` [226](#), [21236](#)
- `\str_head:N` [127](#), [715](#), [13629](#)
- `\str_head:n`
.. [127](#), [689](#), [715](#), [12682](#), [12727](#), [13629](#)
- `\str_head_ignore_spaces:n` [127](#), [13629](#)
- `\str_if_empty:NTF`
[122](#), [11053](#), [13195](#), [20737](#), [20760](#), [21660](#)
- `\str_if_empty_p:N` [122](#), [13195](#)
- `\str_if_eq:nn` [198](#), [882](#), [891](#)
- `\str_if_eq:NNTF` [122](#), [705](#), [13228](#)
- `\str_if_eq:nnTF` [104](#),
[105](#), [123](#), [124](#), [202](#), [203](#), [788](#), [868](#),
[2712](#), [5097](#), [8196](#), [8340](#), [8761](#), [8782](#),
[8813](#), [8975](#), [9609](#), [9652](#), [9928](#), [9931](#),
[11376](#), [11439](#), [11454](#), [13216](#), [13276](#),
[13304](#), [14810](#), [14813](#), [14968](#), [14971](#),
[16389](#), [19069](#), [19126](#), [19695](#), [19824](#),
[20334](#), [20832](#), [21494](#), [23283](#), [23356](#),
[24583](#), [29194](#), [29213](#), [29217](#), [29609](#),
[29665](#), [29978](#), [30013](#), [31511](#), [34123](#),
[34406](#), [34560](#), [34594](#), [34639](#), [34697](#),
[34991](#), [35001](#), [36548](#), [36550](#), [36552](#)
- `\str_if_eq_p:NN` [122](#), [13228](#)
- `\str_if_eq_p:nn`
[123](#), [8721](#), [8748](#), [8749](#), [8821](#), [8822](#),
[9110](#), [9112](#), [11594](#), [13216](#), [29670](#),
[32803](#), [33687](#), [34400](#), [35591](#), [36546](#)
- `\str_if_eq_x:nnTF` [36547](#), [36549](#), [36551](#)
- `\str_if_eq_x_p:nn` [36545](#)
- `\str_if_exist:NTF` ... [121](#), [8780](#), [13195](#)
- `\str_if_exist_p:N` [121](#), [13195](#)
- `\str_if_in:NnTF` [123](#), [13235](#)
- `\str_if_in:nnTF` [123](#), [3243](#), [13235](#), [28975](#)
- `\str_item:Nn` [127](#), [13433](#)
- `\str_item:nn` [127](#), [710](#), [714](#), [13433](#)
- `\str_item_ignore_spaces:nn`
..... [127](#), [710](#), [13433](#)
- `\str_log:N` [131](#), [13718](#)
- `\str_log:n` [131](#), [13718](#)
- `\str_lower_case:n` [36592](#)
- `\str_lowercase:n` [130](#),
[266](#), [13659](#), [36592](#), [36593](#), [36594](#), [36595](#)
- `\str_map_break:` [125](#), [5963](#), [13310](#), [13384](#)
- `\str_map_break:n` [125](#), [126](#), [3247](#), [13310](#)
- `\str_map_function:NN` [124](#), [709](#), [13310](#)
- `\str_map_function:nN`
... [124](#), [125](#), [707](#), [5956](#), [13310](#), [15028](#)
- `\str_map_inline:Nn` [125](#), [13310](#)
- `\str_map_inline:nn`
..... [125](#), [3241](#), [6694](#), [13310](#)
- `\str_map_tokens:Nn` [125](#), [13378](#)
- `\str_map_tokens:nn` [125](#), [13378](#)
- `\str_map_variable:NNn` ... [125](#), [13310](#)
- `\str_map_variable:nNn` ... [125](#), [13310](#)
- `\str_new:N` . [121](#), [9163](#), [9164](#), [10755](#),
[10756](#), [10757](#), [10786](#), [10787](#), [10788](#),
[13092](#), [13714](#), [13715](#), [13716](#), [13717](#),
[20686](#), [20688](#), [20691](#), [20693](#), [20696](#)
- `str_overflow` [14928](#)
- `\str_put_left:Nn` [122](#), [13119](#)
- `\str_put_right:Nn` [122](#), [13119](#)
- `\str_range:Nnn` [128](#), [13494](#)
- `\str_range:nnn` [95](#), [128](#), [714](#), [4164](#), [13494](#)
- `\str_range_ignore_spaces:nnn` ...
..... [128](#), [13494](#)
- `\str_remove_all:Nn` [129](#), [13189](#)
- `\str_remove_once:Nn` [129](#), [13183](#)
- `\str_replace_all:Nnn` [129](#), [13143](#), [13190](#)
- `\str_replace_once:Nnn`
..... [129](#), [13143](#), [13184](#)
- `.str_set:N` [226](#), [21236](#)
- `\str_set:Nn` [122](#), [129](#),
[226](#), [9249](#), [9250](#), [9457](#), [9458](#), [11389](#),
[11390](#), [11391](#), [13119](#), [13369](#), [20716](#),
[20718](#), [21301](#), [21303](#), [21444](#), [21569](#)
- `\str_set_convert:Nnnn`
..... [135](#), [136](#), [724](#), [735](#), [13909](#)
- `\str_set_convert:NnnnTF`
..... [135](#), [724](#), [13909](#)
- `\str_set_eq:NN` [121](#), [13092](#)
- `.str_set_x:N` [226](#), [21236](#)
- `\str_show:N` [131](#), [13718](#)
- `\str_show:n` [131](#), [13718](#)
- `\str_tail:N` [127](#), [13644](#)
- `\str_tail:n` ... [127](#), [431](#), [13644](#), [29296](#)
- `\str_tail_ignore_spaces:n` [127](#), [13644](#)
- `\str_upper_case:n` [36592](#)
- `\str_uppercase:n` [130](#),
[266](#), [13659](#), [36596](#), [36597](#), [36598](#), [36599](#)
- `\str_use:N` [126](#), [13092](#)
- `\c_tilde_str` [132](#), [13701](#)

- \g_tmpa_str [132](#), [13714](#)
- \l_tmpa_str [129](#), [132](#), [13714](#)
- \g_tmpb_str [132](#), [13714](#)
- \l_tmpb_str [132](#), [13714](#)
- \c_underscore_str [132](#), [13701](#)
- \c_zero_str [132](#), [13701](#)
- str internal commands:
- __str_alias_prop . [727](#), [13742](#), [13980](#)
- \c_str_byte_1_t1 [13823](#)
- \c_str_byte_0_t1 [13823](#)
- \c_str_byte_1_t1 [13823](#)
- \c__str_byte_255_t1 [13823](#)
- \c__str_byte_<number>_t1 [722](#)
- __str_case:nnTF [13249](#)
- __str_case:nw [13249](#)
- __str_case_e:nnTF [13249](#)
- __str_case_e:nw [13249](#)
- __str_case_end:nw [13249](#)
- __str_change_case:nn [13659](#)
- __str_change_case_aux:nn [13659](#)
- __str_change_case_char:nN ... [13659](#)
- __str_change_case_end:nw [13659](#)
- __str_change_case_end:wn [13678](#), [13696](#)
- __str_change_case_loop:nw ... [13659](#)
- __str_change_case_output:nw . [13659](#)
- __str_change_case_result:n .. [13659](#)
- __str_change_case_space:n ... [13659](#)
- __str_collect_delimit_by_q-
stop:w [13522](#), [13545](#)
- __str_collect_end:nnnnnnnw ... [713](#), [13545](#)
- __str_collect_end:wn [13545](#)
- __str_collect_loop:wn [13545](#)
- __str_collect_loop:wnNNNNNNN . [13545](#)
- __str_convert:nnn [726](#), [727](#), [13952](#), [13953](#), [13967](#)
- __str_convert:nnnn [727](#), [13967](#)
- __str_convert:NNnNN [13949](#)
- __str_convert:nNNnnn [13909](#)
- __str_convert:wwwnn [726](#), [13936](#), [13941](#), [13949](#)
- __str_convert_decode: . [13940](#), [14072](#)
- __str_convert_decode_clist: . [14112](#)
- __str_convert_decode_eight_
bit:n [14133](#), [14177](#)
- __str_convert_decode_utf16: . [14799](#)
- __str_convert_decode_utf16be: [14799](#)
- __str_convert_decode_utf16le: [14799](#)
- __str_convert_decode_utf32: . [14957](#)
- __str_convert_decode_utf32be: [14957](#)
- __str_convert_decode_utf32le: [14957](#)
- __str_convert_decode_utf8: .. [14618](#)
- __str_convert_encode: . [13945](#), [14076](#)
- __str_convert_encode_clist: . [14123](#)
- __str_convert_encode_eight_
bit:n [14135](#), [14204](#)
- __str_convert_encode_utf16: . [14714](#)
- __str_convert_encode_utf16be: [14714](#)
- __str_convert_encode_utf16le: [14714](#)
- __str_convert_encode_utf32: . [14897](#)
- __str_convert_encode_utf32be: [14897](#)
- __str_convert_encode_utf32le: [14897](#)
- __str_convert_encode_utf8: .. [14543](#)
- __str_convert_escape: [14070](#)
- __str_convert_escape_bytes: . [14070](#)
- __str_convert_escape_hex: ... [14463](#)
- __str_convert_escape_name: [741](#), [14467](#)
- __str_convert_escape_string: . [14490](#)
- __str_convert_escape_url: ... [14522](#)
- __str_convert_gmap:N [13867](#), [14073](#),
[14185](#), [14464](#), [14470](#), [14493](#), [14523](#)
- __str_convert_gmap_internal:N .. [13883](#), [14083](#), [14091](#), [14125](#), [14214](#),
[14544](#), [14727](#), [14899](#), [14903](#), [14905](#)
- __str_convert_gmap_internal_
loop:Nw [13883](#)
- __str_convert_gmap_internal_
loop:Nww [13887](#), [13893](#), [13897](#)
- __str_convert_gmap_loop:NN .. [13867](#)
- __str_convert_lowercase_
alphanum:n [13972](#), [14004](#)
- __str_convert_lowercase_
alphanum_loop:N [14004](#)
- __str_convert_pdfname:n [15025](#)
- __str_convert_pdfname_bytes:n [15025](#)
- __str_convert_pdfname_bytes_
aux:n [15025](#)
- __str_convert_pdfname_bytes_
aux:nnn [15025](#)
- __str_convert_pdfname_bytes_
aux:nnnn [15046](#), [15047](#)
- __str_convert_unescape: [14054](#)
- __str_convert_unescape_bytes: [14054](#)
- __str_convert_unescape_hex: . [14279](#)
- __str_convert_unescape_name: ... [737](#), [14325](#)
- __str_convert_unescape_string: . [14375](#)
- __str_convert_unescape_url: . [14325](#)
- __str_count:n [714](#), [13449](#), [13509](#), [13591](#)
- __str_count_aux:n [13591](#)
- __str_count_loop:NNNNNNNNN .. [13591](#)
- __str_count_spaces_loop:w ... [13571](#)
- __str_declare_eight_bit_
aux:NNnnn [14129](#)

- __str_declare_eight_bit_-
 encoding:nnnn [731](#), [14129](#), [15064](#),
 [15071](#), [15135](#), [15177](#), [15234](#), [15335](#),
 [15422](#), [15508](#), [15582](#), [15595](#), [15648](#),
 [15746](#), [15809](#), [15847](#), [15862](#), [36606](#)
- __str_declare_eight_bit_loop:Nn
 [14129](#)
- __str_declare_eight_bit_-
 loop:Nnn [14129](#)
- __str_decode_clist_char:n ... [14112](#)
- __str_decode_eight_bit_aux:n . [14177](#)
- __str_decode_eight_bit_aux:Nn [14177](#)
- __str_decode_native_char:N . [14072](#)
- __str_decode_utf_viii_aux:wNnnwN
 [14618](#)
- __str_decode_utf_viii_continuation:wwN
 [14618](#)
- __str_decode_utf_viii_end: . [14618](#)
- __str_decode_utf_viii_overflow:w
 [14618](#)
- __str_decode_utf_viii_start:N [14618](#)
- __str_decode_utf_xvi:Nw . [749](#), [14799](#)
- __str_decode_utf_xvi_bom:NN . [14799](#)
- __str_decode_utf_xvi_error:nnN .
 [14833](#)
- __str_decode_utf_xvi_extra:NNw .
 [14833](#)
- __str_decode_utf_xvi_pair:NN ...
 [749](#), [751](#), [14827](#), [14833](#)
- __str_decode_utf_xvi_pair_-
 end:Nw [14833](#)
- __str_decode_utf_xvi_quad:NNwNN
 [14833](#)
- __str_decode_utf_xxxii:Nw [753](#), [14957](#)
- __str_decode_utf_xxxii_bom:NNNN
 [14957](#)
- __str_decode_utf_xxxii_end:w . [14957](#)
- __str_decode_utf_xxxii_loop:NNNN
 [14957](#)
- \g__str_deprecation_bool [36614](#), [36616](#)
- __str_encode_clist_char:n ... [14123](#)
- __str_encode_eight_bit_aux:NNn .
 [14204](#)
- __str_encode_eight_bit_aux:nnN .
 [14204](#)
- __str_encode_native_char:n . [14076](#)
- __str_encode_utf_vii_loop:wwnnw [742](#)
- __str_encode_utf_viii_char:n . [14543](#)
- __str_encode_utf_viii_loop:wwnnw
 [14543](#)
- __str_encode_utf_xvi_aux:N . [14714](#)
- __str_encode_utf_xvi_be:nn ... [747](#)
- __str_encode_utf_xvi_char:n . [14714](#)
- __str_encode_utf_xxxii_be:n . [14897](#)
- __str_encode_utf_xxxii_be_-
 aux:nn [14897](#)
- __str_encode_utf_xxxii_le:n . [14897](#)
- __str_encode_utf_xxxii_le_-
 aux:nn [14897](#)
- \l__str_end_flag [14748](#)
- \g__str_error_bool [13769](#),
 [13906](#), [13916](#), [13920](#), [13925](#), [13929](#)
- __str_escape_hex_char:N [14463](#)
- __str_escape_name_char:n
 [14467](#), [15038](#), [15061](#)
- \c__str_escape_name_not_str
 [740](#), [14467](#)
- \c__str_escape_name_str .. [740](#), [14467](#)
- __str_escape_string_char:N . [14490](#)
- \c__str_escape_string_str [14490](#)
- __str_escape_url_char:n [14522](#)
- \l__str_extra_flag [14565](#), [14748](#)
- __str_filter_bytes:n
 [14030](#), [14064](#), [14345](#), [14407](#)
- __str_filter_bytes_aux:N [14030](#)
- __str_head:w [715](#), [13629](#)
- __str_hexadecimal_use:N [13804](#)
- __str_hexadecimal_use:NTF
 [737](#), [13804](#), [14299](#), [14309](#), [14348](#), [14350](#)
- __str_if_contains_char:Nn ... [13772](#)
- __str_if_contains_char:nn ... [13781](#)
- __str_if_contains_char:NnTF ...
 [13772](#), [14479](#), [14485](#), [14498](#)
- __str_if_contains_char:nnTF ...
 [721](#), [13772](#), [14532](#), [14538](#)
- __str_if_contains_char_aux:nn [13772](#)
- __str_if_contains_char_auxi:nN .
 [13772](#)
- __str_if_contains_char_true: . [13772](#)
- __str_if_eq:nn . [704](#), [705](#), [13203](#),
 [13207](#), [13213](#), [13218](#), [13225](#), [13230](#)
- __str_if_escape_name:n [14476](#)
- __str_if_escape_name:nTF [14467](#)
- __str_if_escape_string:N [14510](#)
- __str_if_escape_string:NTF .. [14490](#)
- __str_if_escape_url:n [14529](#)
- __str_if_escape_url:nTF [14522](#)
- __str_if_flag_error:nnn
 [724](#), [725](#), [13899](#),
 [13918](#), [13927](#), [14065](#), [14092](#), [14186](#),
 [14215](#), [14293](#), [14339](#), [14340](#), [14398](#),
 [14399](#), [14631](#), [14728](#), [14831](#), [14988](#)
- __str_if_flag_no_error:nnn
 [724](#), [13899](#), [13918](#), [13927](#)
- __str_if_flag_times:nTF
 [13907](#), [14574](#), [14575](#), [14576](#), [14577](#),
 [14763](#), [14764](#), [14765](#), [14935](#), [14936](#)

- __str_if_recursion_tail_-
 - break:NN [13090](#), [13350](#), [13368](#)
- __str_if_recursion_tail_stop_-
 - do:Nn [13090](#), [13695](#)
- \l_str_internal_tl
 - [727](#), [13735](#), [13824](#),
[13825](#), [13827](#), [13980](#), [13981](#), [13982](#),
[13984](#), [13988](#), [13992](#), [13999](#), [14131](#)
- __str_item:nn [710](#), [13433](#)
- __str_item:w [710](#), [13433](#)
- __str_map_function:nn
 - [707](#), [13310](#), [13381](#)
- __str_map_function:w
 - [707](#), [13310](#), [13381](#)
- __str_map_inline:NN [13310](#)
- __str_map_variable:NnN [13310](#)
- \c_str_max_byte_int .. [13739](#), [14097](#)
- \l_str_missing_flag .. [14565](#), [14748](#)
- \l_str_modulo_int [14204](#)
- __str_octal_use:N [13796](#)
- __str_octal_use:NTF
 - [721](#), [722](#), [13796](#), [14410](#), [14412](#), [14414](#)
- __str_output_byte:n
 - [752](#), [13835](#), [13864](#), [13865](#), [14025](#),
[14230](#), [14557](#), [14563](#), [14915](#), [14924](#)
- __str_output_byte:w
 - [737](#), [13835](#), [14286](#), [14312](#), [14347](#), [14409](#)
- __str_output_byte_pair:nnN .. [13851](#)
- __str_output_byte_pair_be:n ..
 - [13851](#), [14716](#), [14720](#), [14914](#)
- __str_output_byte_pair_le:n ...
 - [13851](#), [14722](#), [14925](#)
- __str_output_end: ... [737](#), [13835](#),
[14291](#), [14311](#), [14361](#), [14443](#), [14447](#)
- __str_output_hexadecimal:n
 - [13835](#), [14466](#), [14474](#),
[14527](#), [15049](#), [15050](#), [15053](#), [15056](#)
- \l_str_overflow_flag [14565](#)
- \l_str_overlong_flag [14565](#)
- __str_range:nnn [13494](#)
- __str_range:nw [13494](#)
- __str_range:w [13494](#)
- __str_range_normalize:nn
 - [13517](#), [13518](#), [13526](#)
- __str_replace:NNNnn [13143](#)
- __str_replace_aux:NNNnnn [13143](#)
- __str_replace_next:w [13143](#)
- \c_str_replacement_char_int ...
 - [13738](#), [14199](#),
[14643](#), [14667](#), [14681](#), [14701](#), [14708](#),
[14738](#), [14890](#), [14999](#), [15004](#), [15020](#)
- \g_str_result_tl
 - [719](#), [723-725](#), [729](#), [731](#), [737](#),
[749](#), [752](#), [754](#), [13737](#), [13869](#), [13873](#),
[13885](#), [13889](#), [13935](#), [13947](#), [14063](#),
[14064](#), [14114](#), [14115](#), [14118](#), [14126](#),
[14284](#), [14288](#), [14333](#), [14335](#), [14386](#),
[14389](#), [14392](#), [14395](#), [14625](#), [14627](#),
[14717](#), [14800](#), [14802](#), [14806](#), [14825](#),
[14900](#), [14958](#), [14960](#), [14963](#), [14982](#)
- __str_skip_end:NNNNNNNN . [711](#), [13473](#)
- __str_skip_end:w [13473](#)
- __str_skip_exp_end:w
 - [711](#), [713](#), [13460](#), [13469](#), [13473](#), [13524](#)
- __str_skip_loop:wNNNNNNNN ... [13473](#)
- __str_tail_auxi:w [13644](#)
- __str_tail_auxii:w [716](#), [13644](#)
- __str_tmp:n [13093](#),
[13099](#), [13102](#), [13120](#), [13130](#), [13133](#)
- __str_tmp:w [733](#),
[737](#), [747](#), [749](#), [754](#), [13735](#), [14179](#),
[14185](#), [14207](#), [14214](#), [14325](#), [14371](#),
[14373](#), [14378](#), [14403](#), [14726](#), [14733](#),
[14738](#), [14740](#), [14743](#), [14744](#), [14824](#),
[14839](#), [14844](#), [14855](#), [14858](#), [14864](#),
[14865](#), [14981](#), [14996](#), [15001](#), [15007](#)
- __str_to_other_end:w ... [709](#), [13388](#)
- __str_to_other_fast_end:w ... [13411](#)
- __str_to_other_fast_loop:w
 - [13413](#), [13422](#), [13429](#)
- __str_to_other_loop:w ... [709](#), [13388](#)
- __str_unescape_hex_auxi:N ... [14279](#)
- __str_unescape_hex_auxii:N .. [14279](#)
- __str_unescape_name_loop:wNN . [14325](#)
- __str_unescape_string_loop:wNNN
 - [14375](#)
- __str_unescape_string_newlines:wN
 - [14375](#)
- __str_unescape_string_repeat:NNNNNN
 - [14375](#)
- __str_unescape_url_loop:wNN . [14325](#)
- __str_use_i_delimit_by_s_-
 - stop:nw [715](#), [13086](#),
[13459](#), [13468](#), [13587](#), [13638](#), [13641](#)
- __str_use_none_delimit_by_s_-
 - stop:w
[13086](#), [13177](#), [13457](#), [13466](#), [13625](#),
[13895](#), [14167](#), [14173](#), [14558](#), [14650](#)
- \strcmp [22](#)
- \string [456](#)
- \sum [1327](#)
- \suppressfontnotfounderror [713](#)
- \suppressifcsnameerror [896](#)
- \suppresslongerror [897](#)
- \suppressmathparerror [898](#)
- \suppressoutererror [899](#)
- \suppressprimitiveerror [900](#)
- \synctex [693](#)

sys commands:

`\c_sys_backend_str` [75](#), [8777](#)
`\c_sys_day_int` [72](#), [8970](#)
`\c_sys_engine_exec_str`
 [73](#), [576](#), [8723](#), [11484](#)
`\c_sys_engine_format_str`
 [73](#), [576](#), [8723](#), [11485](#)
`\c_sys_engine_str`
 [73](#), [576](#), [651](#), [8706](#), [36056](#)
`\c_sys_engine_version_str` [306](#), [36054](#)
`\sys_everyjob:` [8960](#), [9093](#)
`\sys_finalise:` [76](#), [8779](#), [9091](#)
`\sys_get_shell:nnN` [74](#), [8856](#)
`\sys_get_shell:nnNTF`
 [74](#), [302](#), [8856](#), [8858](#)
`\sys_gset_rand_seed:n` . [74](#), [256](#), [9016](#)
`\c_sys_hour_int` [72](#), [8970](#)
`\sys_if_engine luatex:TF` [73](#),
 [98](#), [2698](#), [8706](#), [8731](#), [8756](#), [8815](#),
 [8825](#), [8826](#), [8896](#), [8918](#), [8950](#), [9042](#),
 [9072](#), [10140](#), [10909](#), [11142](#), [11304](#),
 [11573](#), [11612](#), [18654](#), [19158](#), [28713](#),
 [28751](#), [28790](#), [28803](#), [28829](#), [28897](#),
 [28921](#), [28958](#), [36498](#), [36500](#), [36502](#)
`\sys_if_engine luatex:p:`
 [73](#), [8706](#), [10304](#),
 [13760](#), [14032](#), [14056](#), [14078](#), [14248](#),
 [15031](#), [29008](#), [29034](#), [29096](#), [30033](#),
 [30107](#), [30147](#), [30208](#), [30244](#), [30450](#),
 [30498](#), [30505](#), [30583](#), [30661](#), [30739](#),
 [30762](#), [30798](#), [31617](#), [31702](#), [36496](#)
`\sys_if_engine pdftex:TF`
 [73](#), [8706](#), [8727](#),
 [8751](#), [9124](#), [30163](#), [36512](#), [36514](#), [36516](#)
`\sys_if_engine pdftex:p:`
 [73](#), [8706](#), [8765](#), [36510](#)
`\sys_if_engine ptex:TF`
 [73](#), [8706](#), [8729](#), [8754](#)
`\sys_if_engine ptex:p:` [73](#), [3793](#), [8706](#)
`\sys_if_engine uptex:TF`
 [73](#), [8706](#), [8730](#), [8755](#)
`\sys_if_engine uptex:p:` [73](#), [3794](#), [8706](#)
`\sys_if_engine xetex:TF`
 ... [6](#), [73](#), [2697](#), [8706](#), [8728](#), [8753](#),
 [8796](#), [9119](#), [18655](#), [36586](#), [36588](#), [36590](#)
`\sys_if_engine xetex:p:`
 [73](#), [8706](#), [8981](#),
 [13761](#), [14033](#), [14057](#), [14079](#), [14249](#),
 [15032](#), [29008](#), [29034](#), [29096](#), [30034](#),
 [30108](#), [30148](#), [30209](#), [30245](#), [30451](#),
 [30499](#), [30506](#), [30584](#), [30662](#), [30740](#),
 [30763](#), [30799](#), [31618](#), [31703](#), [36584](#)
`\sys_if_output_dvi:TF` [73](#), [9100](#)
`\sys_if_output_dvi:p:` [73](#), [9100](#)

`\sys_if_output_pdf:TF`
 [73](#), [8811](#), [9100](#), [9122](#)
`\sys_if_output_pdf:p:` [73](#), [9100](#)
`\sys_if_platform_unix:TF`
 [74](#), [8777](#), [11591](#)
`\sys_if_platform_unix:p:`
 [74](#), [8777](#), [11591](#)
`\sys_if_platform_windows:TF`
 [74](#), [8777](#), [11591](#)
`\sys_if_platform_windows:p:`
 [74](#), [8777](#), [11591](#)
`\sys_if_rand_exist:TF` . [306](#), [577](#),
 [8775](#), [9004](#), [9018](#), [22257](#), [28241](#), [28265](#)
`\sys_if_rand_exist:p:` [306](#), [8775](#)
`\sys_if_shell:` [75](#)
`\sys_if_shell:TF` [75](#), [8863](#), [9080](#), [36110](#)
`\sys_if_shell:p:` [75](#), [9080](#)
`\sys_if_shell_restricted:TF` [75](#), [9080](#)
`\sys_if_shell_restricted:p:` [75](#), [9080](#)
`\sys_if_shell_unrestricted:TF` ...
 [75](#), [9080](#)
`\sys_if_shell_unrestricted:p:` ...
 [75](#), [9080](#)
`\sys_if_timer_exist:TF` [9031](#)
`\sys_if_timer_exist:p:` [9031](#)
`\c_sys_jobname_str`
 [72](#), [93](#), [585](#), [8968](#), [36396](#)
`\sys_load_backend:n` [75](#), [8777](#)
`\sys_load_debug:` [76](#), [8848](#)
`\sys_load_deprecation:` [36611](#)
`\c_sys_minute_int` [72](#), [8970](#)
`\c_sys_month_int` [72](#), [8970](#)
`\c_sys_output_str` [73](#), [9100](#)
`\c_sys_platform_str`
 [74](#), [8777](#), [11573](#), [11594](#)
`\sys_rand_seed:` ... [74](#), [147](#), [256](#), [9002](#)
`\c_sys_shell_escape_int`
 [75](#), [9068](#), [9083](#), [9085](#), [9087](#)
`\sys_shell_now:n` [75](#), [8898](#)
`\sys_shell_shipout:n` [75](#), [8929](#)
`\sys_timer:` [73](#), [9031](#)
`\c_sys_year_int` [72](#), [8970](#)

sys internal commands:

`\g__sys_backend_tl`
 [8787](#), [8788](#), [8789](#), [9114](#)
`__sys_const:nn` . [8690](#), [8720](#), [8775](#),
 [9066](#), [9082](#), [9084](#), [9086](#), [9109](#), [9111](#)
`\g__sys_debug_bool` .. [8847](#), [8850](#), [8852](#)
`__sys_elapsedtime:` [9031](#)
`__sys_everyjob:n` [8960](#), [8968](#),
 [8970](#), [9002](#), [9016](#), [9068](#), [9080](#), [9089](#)
`\g__sys_everyjob_tl` [8960](#)
`__sys_finalise:n`
 [9091](#), [9100](#), [9115](#), [9132](#)

- \g__sys_finalise_tl 9091
- __sys_get:nnN 8856
- __sys_get_do:Nw 8856
- \l_sys_internal_tl 8854
- __sys_load_backend_check:N .. 8777
- \c__sys_marker_tl ... 8855, 8879, 8891
- __sys_shell_now:n 8898
- __sys_shell_shipout:n 8929
- \c__sys_shell_stream_int
..... 8896, 8925, 8957
- __sys_tmp:w
.. 8973, 8994, 8996, 8997, 8998, 8999
- syst commands:
- \c_syst_catcodes_n 28964, 28968
- \c_syst_last_allocated_toks .. 3381
- T**
- \T 86
- \t 29389, 31742
- \tabskip 457
- \tagcode 694
- \tan 254
- \tand 254
- \tate 1156
- \tbaselineshift 1157
- \c_ten 36419
- TEX and L^AT_EX 2_ε commands:
- \@ 13702
- \@@@hyph 337
- \@end 1191, 1192
- \@hyph 1195, 1198
- \@input 1193
- \@italiccorr 1199
- \@shipout 1201, 1202
- \@tracingfonts 338, 1237
- \@underline 1200
- \@addtofilelist 11300
- \@changed@cmd 29708, 31524
- \@classoptionslist .. 9134, 9136, 9138
- \@current@cmd 29707, 31523
- \@currnamestack
..... 632, 10777, 10779, 10780
- \@expl@finalise@setup@@ .. 28999,
29000, 31308, 31310, 35592, 35594
- \@expl@luadata@bytecode 36
- \@filelist 97, 633, 646,
649, 11299, 11399, 11402, 11411, 11416
- \@firstofone 23
- \@firstoftwo 24, 355
- \@gobbbbletwo 25
- \@gobble 25
- \@kernel@after@begindocument ...
..... 31312, 35596
- \@protected@testopt 1208, 29695
- \@secondoftwo 24, 355
- \@tempa 104, 106, 1209, 1223, 1226
- \@tfor 337, 1209
- \@uclclist 1243, 31337
- \@unexpandable@protect 1009
- \@unusedoptionlist 9153
- \afterassignment 445, 544
- \AtBeginDocument 337
- \botmark 870
- \box 278
- \catcodetable 1182, 1185, 1188
- \char 197
- \chardef 189, 190, 563, 808, 1199
- \conditionally@traceoff
..... 625, 9552, 10498
- \conditionally@traceon 9570
- \copy 271
- \count 197, 412
- \cr 573
- \CROP@shipout 1210
- \csname 21, 344, 634, 635
- \csstring 364
- \currentgrouplevel .. 376, 1184, 1359
- \currentgrouptype 376, 1359
- \def 197
- \detokenize 107
- \development@branch@name 11487, 11488
- \dimen 868
- \dimendef 868
- \dimexpr 1259
- \directlua 98
- \dp 272, 1010, 1011
- \dup@shipout 1211
- \e@alloc@ccodetable@count 28962
- \e@alloc@top 412, 3367
- \edef 3, 6, 665
- \end 337, 595
- \endcsname 21
- \endinput 81
- \endlinechar 87, 117,
118, 671, 672, 870, 1182, 1183, 1185
- \endtemplate 71, 573
- \errhelp 591
- \errmessage 591, 592
- \errorcontextlines 346, 592, 699, 1262
- \escapechar ... 107, 364, 376, 450, 624
- \everyeof 672
- \everyjob 582
- \everypar 28, 193, 379, 397
- \expandafter 38, 40
- \expanded 6, 25, 33, 35,
381, 384, 391, 393, 398, 404, 671, 688
- \fi 196
- \firstmark 399, 870

- \fmtname 73
- \font 196, 868
- \fontdimen 60, 237, 957–960
- \frozen@everydisplay 1196
- \frozen@everymath 1197
- \futurelet
427, 430, 431, 442, 445, 573, 875, 878
- \global 317
- \GPTorg@shipout 1212
- \halign 71, 379, 573, 858
- \hskip 217
- \ht 272, 1010, 1011
- \hyphen 870
- \hyphenchar 957
- \ifcase 167
- \ifdim 220
- \ifeof 93
- \iffalse 65
- \ifhbox 281
- \ifnum 167
- \ifodd 168, 879
- \iftrue 65
- \ifvbox 281
- \ifvoid 281
- \ifx 27, 314
- \indent 379
- \infty 249
- \input 337
- \input@path
. 94, 638, 10938, 10940, 11038, 11040
- \italiccorr 870
- \jobname 72, 582
- \lastnamedcs 367
- \lccode 430, 434, 826, 1194
- \leavevmode 28
- \let 317
- \letcharcode 855
- \LL@shipout 1213
- \loctoks 412
- \long 5, 197, 392, 699
- \lower 1357
- \lowercase 531–533
- \luaescapestring 99
- \makeatletter 9
- \MakeUppercase 1236
- \mathchar 197
- \mathchardef 190, 808, 1199
- \mathop 1325
- \mathord 292
- \meaning 20,
187, 196, 197, 429, 868, 869, 878, 879
- \mem@oldshipout 1214
- \message 33
- \newcatcodetable 1182
- \newif 65, 100
- \newlinechar 117,
118, 346, 367, 592, 622, 671, 672, 699
- \newread 613
- \newtoks 43, 423, 450
- \newwrite 619
- \noexpand 39, 196, 391–394
- \nullfont 870
- \number 167, 805, 1065
- \numexpr 345, 395
- \opem@shipout 1215
- \or 167
- \outer 197,
314, 442, 443, 613, 619, 879, 1374, 1376
- \parindent 28
- \pdfescapehex 735
- \pdfescapename 134, 735
- \pdfescapestring 134, 735
- \pdffilesize 637, 638
- \pdfmapfile 339
- \pdfmapline 339
- \pdfstrcmp 311, 312, 314, 328
- \pdfuniformdeviate 256
- \pgfpages@originalshipout 1216
- \pi 249
- \pr@shipout 1217
- \primitive 337, 391–393, 582
- \protect 625, 1008, 1009, 1247
- \protected 197, 392, 699
- \ProvidesClass 9
- \ProvidesFile 9
- \ProvidesPackage 9
- \quitvmode 379
- \read 87, 617
- \readline 88, 617
- \relax 27, 196, 314, 360,
365, 376, 634, 828, 965, 967, 992, 1024
- \RequirePackage 10, 314, 632
- \reserveinserts 314
- \romannumeral 41, 965
- \savecatcodetable 1184
- \scantokens 118, 136, 637, 670
- \shipout 337
- \show 20, 109, 376
- \showbox 1262
- \showgroups 14, 377
- \showthe 376, 825, 905, 909, 911
- \showtokens 109, 597, 699
- \sin 249
- \skip 435, 436
- \space 870
- \splitbotmark 870
- \splitfirstmark 870
- \SS 1250

- `\strcmp` 311, 328
- `\string` 187, 430–432
- `\tenrm` 196
- `\the` 158, 196, 212, 216,
219, 383, 391–393, 395, 396, 805, 1259
- `\toks` 43, 147, 167, 395–397,
410–418, 423, 429, 431, 433, 434,
436, 438, 450, 451, 500, 508, 513,
514, 523, 532, 539, 552, 553, 561, 790
- `\toks@` 397
- `\toksdef` 423
- `\topmark` 197, 870
- `\tracingfonts` 338
- `\tracingnesting` 636, 670
- `\tracingonline` 1262
- `\typeout` 625
- `\uccode` 1194
- `\Ucharcat` 857
- `\unexpanded` . 39, 108, 109, 113–115,
145, 150, 151, 174, 177–180, 201,
304, 307, 391–394, 665, 688, 689, 811
- `\unhbox` 278
- `\unhcopy` 275
- `\uniformdeviate` 256
- `\unless` 27
- `\unvbox` 278
- `\unvcopy` 277
- `\uppercase` 531
- `\usepackage` 632
- `\valign` 573
- `\value` 156
- `\verb` 118
- `\verso@orig@shipout` 1219
- `\vskip` 218
- `\vtop` 1282
- `\wd` 272, 1010, 1011
- `\write` 90, 622
- tex commands:
 - `\tex_above:D` 190
 - `\tex_abovedisplayshortskip:D` .. 191
 - `\tex_abovedisplayskip:D` 192
 - `\tex_abovewithdelims:D` 193
 - `\tex_accent:D` 194
 - `\tex_adjdemerits:D` 195
 - `\tex_adjustspacing:D` 650, 908
 - `\tex_advance:D` .. 196, 3473, 3480,
3483, 3902, 3904, 3937, 3939, 5393,
17078, 17080, 17082, 17084, 17090,
17092, 17094, 17096, 19981, 19984,
19990, 19993, 20321, 20323, 20327,
20329, 20414, 20416, 20420, 20422
 - `\tex_afterassignment:D` . 197, 3843,
3886, 4283, 7629, 7680, 7835, 19347
 - `\tex_aftergroup:D` ... 1187, 198, 1451
 - `\tex_alignmark:D` 783, 1242
 - `\tex_aligntab:D` 784, 1243
 - `\tex_atop:D` 199
 - `\tex_atopwithdelims:D` 200
 - `\tex_attribute:D` 785, 1244
 - `\tex_attributedef:D` 786, 1245
 - `\tex_automaticdiscretionary:D` .. 788
 - `\tex_automatichyphenmode:D` 789
 - `\tex_automatichyphenpenalty:D` .. 791
 - `\tex_autospacing:D` 1107
 - `\tex_autoxspacing:D` 1108
 - `\tex_badness:D` 201
 - `\tex_baselineskip:D` 202
 - `\tex_batchmode:D` 203, 9434
 - `\tex_begincsname:D` 792
 - `\tex_begingroup:D` 204, 1204, 1306, 1446
 - `\tex_beginL:D` 512
 - `\tex_beginR:D` 513
 - `\tex_belowdisplayshortskip:D` .. 205
 - `\tex_belowdisplayskip:D` 206
 - `\tex_binoppenalty:D` 207
 - `\tex_bodydir:D` 793, 1281, 1361
 - `\tex_bodydirection:D` 794
 - `\tex_botmark:D` 208
 - `\tex_botmarks:D` 514
 - `\tex_box:D` ... 209, 31997, 31999, 32042
 - `\tex_boxdir:D` 795, 1282
 - `\tex_boxdirection:D` 796
 - `\tex_boxmaxdepth:D` 210
 - `\tex_breakafterdirmode:D` 797
 - `\tex_brokenpenalty:D` 211
 - `\tex_catcode:D` 212, 2594,
7105, 8679, 11969, 18493, 18495, 29107
 - `\tex_catcodetable:D`
..... 798, 1246, 28807, 28814
 - `\tex_char:D` 213
 - `\tex_chardef:D` 353,
214, 1439, 1468, 1470, 1770, 1771,
8384, 8406, 8411, 10137, 10352,
17053, 19228, 29415, 29417, 29419
 - `\tex_cleaders:D` 215
 - `\tex_clearmarks:D` 799, 1247
 - `\tex_closein:D` 216, 10148
 - `\tex_closeout:D` 217, 10362
 - `\tex_clubpenalties:D` 515
 - `\tex_clubpenalty:D` 218
 - `\tex_copy:D` 219, 31991,
31993, 32017, 32026, 32035, 32043
 - `\tex_copyfont:D` 651, 909
 - `\tex_count:D` 220, 3350, 3366,
3374, 3375, 10085, 10087, 10316, 10318
 - `\tex_countdef:D` 221
 - `\tex_cr:D` 222
 - `\tex_crampeddisplaystyle:D` 800, 1248

- `\tex_crampedscriptscriptstyle:D` 802, 1249
- `\tex_crampedscriptstyle:D` 803, 1251
- `\tex_crampedtextstyle:D` 804, 1252
- `\tex_crcr:D` 223
- `\tex_creationdate:D` 773
- `\tex_csname:D` 224, 1433
- `\tex_csstring:D` 805
- `\tex_currentcjktoken:D` 1109, 1166
- `\tex_currentgrouplevel:D` 1187, 516, 28796, 28876, 28885, 28892, 34826
- `\tex_currentgrouptype:D` 517
- `\tex_currentifbranch:D` 518
- `\tex_currentiflevel:D` 519
- `\tex_currentiftype:D` 520
- `\tex_currentspacingmode:D` 1110
- `\tex_currentxspacingmode:D` 1111
- `\tex_day:D` 225, 1313, 1317
- `\tex_deadcycles:D` 226
- `\tex_def:D` 227, 697, 698, 699, 1452, 1454, 1456, 1457, 1478, 1480, 1481, 1482, 1484, 1485, 1486, 1488, 1489, 1490
- `\tex_defaultthyphenchar:D` 228
- `\tex_defaultskewchar:D` 229
- `\tex_delcode:D` 230
- `\tex_delimiter:D` 231
- `\tex_delimiterfactor:D` 232
- `\tex_delimitershortfall:D` 233
- `\tex_detokenize:D` 521, 1442, 1444, 29100
- `\tex_dimen:D` 234
- `\tex_dimendef:D` 235
- `\tex_dimexpr:D` 522, 19936, 31967
- `\tex_directlua:D` 806, 1235, 1236, 8724, 9074, 11576, 11601
- `\tex_disablecjktoken:D` 1167
- `\tex_discretionary:D` 236
- `\tex_disinhibitglue:D` 1112
- `\tex_displayindent:D` 237
- `\tex_displaylimits:D` 238, 34475
- `\tex_displaystyle:D` 239
- `\tex_displaywidowpenalties:D` 523
- `\tex_displaywidowpenalty:D` 240
- `\tex_displaywidth:D` 241
- `\tex_divide:D` 242, 3344, 5394
- `\tex_doublehyphenemerits:D` 243
- `\tex_dp:D` 244, 32007
- `\tex_draftmode:D` 652, 910
- `\tex_dtou:D` 1113
- `\tex_dump:D` 245
- `\tex_dviextension:D` 807
- `\tex_dvifedback:D` 808
- `\tex_dvivariable:D` 809
- `\tex_eachlinedepth:D` 653
- `\tex_eachlineheight:D` 654
- `\tex_edef:D` 246, 1205, 1206, 1222, 1307, 1308, 1313, 1314, 1319, 1320, 1325, 1326, 1479, 1483, 1487, 1491, 10590, 10648, 36362
- `\tex_efcode:D` 685
- `\tex_elapsedtime:D` 655, 774, 9048, 9051, 9067
- `\tex_else:D` 247, 1208, 1234, 1310, 1316, 1322, 1328, 1419, 1471, 1474, 1516
- `\tex_emergencystretch:D` 248
- `\tex_enablecjktoken:D` 1168, 8712
- `\tex_end:D` 249, 1192, 1344, 1881
- `\tex_endcsname:D` 250, 1434
- `\tex_endgroup:D` 251, 1190, 1230, 1331, 1447
- `\tex_endinput:D` 252, 9443, 11285, 11497
- `\tex_endL:D` 524
- `\tex_endlinechar:D` 158, 159, 173, 253, 10220, 10222, 10223, 12108, 12109, 12110, 12144, 28758, 28762, 28775, 28809, 28810, 28825, 28990, 29005, 29041
- `\tex_endR:D` 525
- `\tex_epTeXinputencoding:D` 1114
- `\tex_epTeXversion:D` 1115, 36074, 36097
- `\tex_eqno:D` 254
- `\tex_errhelp:D` 255, 9302
- `\tex_errmessage:D` 256, 1873, 9322
- `\tex_errorcontextlines:D` 257, 2200, 2207, 2208, 9317, 9336, 9532, 13041, 32106
- `\tex_errorstopmode:D` 258
- `\tex_escapechar:D` 635, 259, 2185, 3808, 3870, 3871, 4208, 4262, 4263, 4266, 4301, 4398, 10242, 10452, 10499, 10505, 14283, 14332, 14385
- `\tex_eTeXglueshrinkorder:D` 810
- `\tex_eTeXgluestretchorder:D` 811
- `\tex_eTeXrevision:D` 526
- `\tex_eTeXversion:D` 527
- `\tex_etoksapp:D` 812
- `\tex_etokspre:D` 813
- `\tex_euc:D` 1116
- `\tex_everycr:D` 260
- `\tex_everydisplay:D` 261, 1196
- `\tex_everyeof:D` 528, 4281, 4282, 8879, 10902, 12117, 12169
- `\tex_everyhbox:D` 262
- `\tex_everyjob:D` 263, 1338, 1345
- `\tex_everymath:D` 264, 1197
- `\tex_everypar:D` 265

- `\tex_everyvbox:D` 266
- `\tex_exceptionpenalty:D` 814
- `\tex_exhyphenpenalty:D` 267
- `\tex_expandafter:D` 268, 702,
1209, 1223, 1225, 1226, 1435, 29100
- `\tex_expanded:D` ... 404, 888, 816,
1354, 1515, 1516, 2277, 2280, 2347,
2350, 2383, 2389, 2480, 2483, 2504,
2507, 2572, 2575, 2603, 3080, 3120,
3128, 9970, 12629, 12633, 18662,
19709, 19719, 20454, 20458, 20612
- `\tex_explicitdiscretionary:D` .. 817
- `\tex_explicitthyphenpenalty:D` .. 815
- `\tex_fam:D` 269
- `\tex_fi:D` 270, 703, 1194,
1203, 1227, 1229, 1238, 1239, 1240,
1298, 1300, 1301, 1305, 1312, 1318,
1324, 1330, 1336, 1339, 1342, 1355,
1363, 1368, 1420, 1476, 1477, 1518
- `\tex_filedump:D`
. 712, 775, 1406, 11135, 11147, 11154
- `\tex_filemoddate:D`
..... 711, 776, 1402, 11256
- `\tex_filesize:D`
..... 639, 640, 709, 777, 1389,
10921, 10997, 11029, 11154, 11181
- `\tex_finalhyphenemerits:D` 271
- `\tex_firstlineheight:D` 656
- `\tex_firstmark:D` 272
- `\tex_firstmarks:D` 529
- `\tex_firstvalidlanguage:D` 818
- `\tex_floatingpenalty:D` 273
- `\tex_font:D` 274, 22061
- `\tex_fontchardp:D` 530
- `\tex_fontcharht:D` 531
- `\tex_fontcharic:D` 532
- `\tex_fontcharwd:D` 533
- `\tex_fontdimen:D` 275, 22053
- `\tex_fontexpand:D` 657, 911
- `\tex_fontid:D` 819, 1253
- `\tex_fontname:D` 276
- `\tex_fontsize:D` 658
- `\tex_forcecjktoken:D` 1169
- `\tex_formatname:D` 820, 1254
- `\tex_futurelet:D`
..... 277, 3816, 3874, 4268,
4284, 4288, 4349, 4361, 19342, 19344
- `\tex_gdef:D` 278, 1492, 1495, 1499, 1503
- `\tex_gleaders:D` 826, 1255
- `\tex_global:D`
897, 179, 183, 279, 704, 1225, 1311,
1317, 1323, 1329, 1950, 1957, 8384,
8411, 10137, 10352, 11932, 11944,
17031, 17037, 17041, 17060, 17071,
17082, 17084, 17094, 17096, 17104,
18962, 18964, 18974, 19344, 19950,
19955, 19971, 19978, 19984, 19993,
20293, 20313, 20318, 20323, 20329,
20384, 20390, 20406, 20411, 20416,
20422, 22061, 31993, 31999, 32072,
32125, 32137, 32150, 32170, 32217,
32229, 32241, 32254, 32275, 32290
- `\tex_globaldefs:D` 280
- `\tex_glueexpr:D` 534, 20311,
20313, 20321, 20323, 20327, 20329,
20343, 20350, 20355, 20358, 28175
- `\tex_glueshrink:D` 535
- `\tex_glueshrinkorder:D` 536
- `\tex_gluestretch:D` ... 537, 4028, 4034
- `\tex_gluestretchorder:D` 538
- `\tex_gluetomu:D` 539
- `\tex_halign:D` 281
- `\tex_hangafter:D` 282
- `\tex_hangindent:D` 283
- `\tex_hbadness:D` 284
- `\tex_hbox:D` 285, 32117, 32120,
32125, 32132, 32137, 32144, 32150,
32164, 32170, 32178, 32183, 33630
- `\tex_hfi:D` 1117
- `\tex_hfil:D` 286
- `\tex_hfill:D` 287
- `\tex_hfilneg:D` 288
- `\tex_hfuzz:D` 289
- `\tex_hjcode:D` 821
- `\tex_hoffset:D` 290, 1357
- `\tex_holdinginserts:D` 291
- `\tex_hpack:D` 822
- `\tex_hrule:D` 292
- `\tex_hsize:D`
.... 293, 32780, 32805, 32806, 32856
- `\tex_hskip:D` 294, 20353
- `\tex_hss:D`
295, 32187, 32189, 32191, 32634, 32643
- `\tex_ht:D` 296, 32006
- `\tex_hyphen:D` 189, 1198
- `\tex_hyphenation:D` 297
- `\tex_hyphenationbounds:D` 823
- `\tex_hyphenationmin:D` 824
- `\tex_hyphenchar:D` 298, 22054
- `\tex_hyphenpenalty:D` 299
- `\tex_hyphenpenaltymode:D` 825
- `\tex_if:D` 300, 1422, 1423
- `\tex_ifabsdim:D` 647, 912
- `\tex_ifabsnum:D`
..... 956, 648, 913, 22120, 22124
- `\tex_ifcase:D` 301, 16922
- `\tex_ifcat:D` 302, 1424
- `\tex_ifcondition:D` 827

- `\tex_ifcsname:D` 540, 1432
- `\tex_ifdbox:D` 1118
- `\tex_ifddir:D` 1119
- `\tex_ifdefined:D` 541, 701,
1191, 1195, 1201, 1232, 1235, 1241,
1300, 1301, 1332, 1337, 1340, 1343,
1356, 1364, 1431, 1469, 1472, 1516
- `\tex_ifdim:D` 303, 19935
- `\tex_ifeof:D` 304, 10184
- `\tex_iffalse:D` 305, 1417
- `\tex_iffontchar:D` 542
- `\tex_ifhbox:D` 306, 32054
- `\tex_ifhmode:D` 307, 1428
- `\tex_ifincsname:D` 686
- `\tex_ifinner:D` 308, 1430
- `\tex_ifjfont:D` 1120
- `\tex_ifmbbox:D` 1121
- `\tex_ifmdir:D` 1122
- `\tex_ifmmode:D` 309, 1427
- `\tex_ifnum:D` 310, 1299, 1449
- `\tex_ifodd:D` .. 311, 1426, 8377, 16921
- `\tex_ifprimitive:D` 649, 779
- `\tex_iftbox:D` 1123
- `\tex_iftdir:D` 1125
- `\tex_iftfont:D` 1124
- `\tex_iftrue:D` 312, 1416
- `\tex_ifvbox:D` 313, 32055
- `\tex_ifvmode:D` 314, 1429
- `\tex_ifvoid:D` 315, 32056
- `\tex_ifx:D` 316, 1207,
1224, 1309, 1315, 1321, 1327, 1425
- `\tex_ifybox:D` 1126
- `\tex_ifydir:D` 1127
- `\tex_ignoreddimen:D` 659
- `\tex_ignoreligaturesinfont:D` .. 914
- `\tex_ignorespaces:D` 317
- `\tex_immediate:D`
. 318, 1890, 1892, 10354, 10362, 10419
- `\tex_immediateassigned:D` 828
- `\tex_immediateassignment:D` 829
- `\tex_indent:D` 319, 2261
- `\tex_inhibitglue:D` 1128
- `\tex_inhibitxspcode:D` 1129
- `\tex_initcatcodetable:D`
..... 830, 1256, 28723
- `\tex_input:D`
. 320, 1193, 1346, 8884, 10908, 11303
- `\tex_inputlineno:D` ... 321, 1888, 9240
- `\tex_insert:D` 322
- `\tex_insertht:D` 660, 915
- `\tex_insertpenalties:D` 323
- `\tex_interactionmode:D` 543,
2198, 2203, 2204, 32090, 32093, 32095
- `\tex_interlinepenalties:D` 544
- `\tex_interlinepenalty:D` 324
- `\tex_italiccorrection:D`
..... 188, 1199, 1358
- `\tex_jcharwidowpenalty:D` 1130
- `\tex_jfam:D` 1131
- `\tex_jfont:D` 1132
- `\tex_jis:D` 1133
- `\tex_jobname:D`
..... 325, 8969, 9090, 10768, 10769
- `\tex_kanjiskip:D` 1134, 8710
- `\tex_kansuji:D` 1135
- `\tex_kansujichar:D` 1136
- `\tex_kcatcode:D` 1137
- `\tex_kchar:D` 1170
- `\tex_kchardef:D` 1171
- `\tex_kern:D` 326, 31971
- `\tex_kuten:D` 1138, 1172
- `\tex_language:D` 327, 1347
- `\tex_lastbox:D` 328, 32070, 32072
- `\tex_lastkern:D` 329
- `\tex_lastlinedepth:D` 661
- `\tex_lastlinefit:D` 545
- `\tex_lastnamedcs:D` 831
- `\tex_lastnodechar:D` 1139
- `\tex_lastnodesubtype:D` 1140
- `\tex_lastnodetype:D` 546
- `\tex_lastpenalty:D` 330
- `\tex_lastskip:D` 331
- `\tex_lastxpos:D` 662, 922
- `\tex_lastypos:D` 663, 923
- `\tex_latelua:D` 832, 1257, 11602
- `\tex_lateluafunction:D` 833
- `\tex_lccode:D` 332, 3789, 3799, 3868,
3870, 3873, 3903, 7272, 7324, 13394,
13395, 13417, 13418, 18569, 18571
- `\tex_leaders:D` 333
- `\tex_left:D` 334, 1365
- `\tex_leftghost:D` 834, 1283
- `\tex_leftthyphenmin:D` 335
- `\tex_leftmarginkern:D` 687
- `\tex_leftskip:D` 336
- `\tex_leqno:D` 337
- `\tex_let:D` 1374, 180, 183, 338,
704, 1192, 1193, 1196, 1197, 1198,
1199, 1200, 1202, 1225, 1231, 1233,
1237, 1242, 1243, 1244, 1245, 1246,
1247, 1248, 1249, 1251, 1252, 1253,
1254, 1255, 1256, 1257, 1258, 1259,
1260, 1261, 1262, 1263, 1264, 1265,
1266, 1267, 1268, 1269, 1270, 1271,
1272, 1274, 1275, 1277, 1278, 1279,
1281, 1282, 1283, 1284, 1285, 1287,
1288, 1289, 1290, 1291, 1292, 1293,
1294, 1295, 1296, 1297, 1303, 1304,

- 1311, 1317, 1323, 1329, 1333, 1334,
 1335, 1338, 1341, 1344, 1345, 1346,
 1347, 1348, 1349, 1350, 1351, 1352,
 1353, 1354, 1357, 1358, 1359, 1360,
 1361, 1362, 1365, 1366, 1367, 1416,
 1417, 1418, 1419, 1420, 1421, 1422,
 1423, 1424, 1425, 1426, 1427, 1428,
 1429, 1430, 1431, 1432, 1433, 1434,
 1435, 1436, 1437, 1438, 1440, 1441,
 1442, 1443, 1444, 1445, 1446, 1447,
 1449, 1450, 1451, 1467, 1478, 1479,
 1492, 1493, 1946, 3790, 3800, 4229,
 4259, 11930, 11932, 11942, 11944,
 18962, 18964, 18974, 36326, 36329
 \tex_letcharcode:D 835
 \tex_letterspacefont:D 688
 \tex_limits:D 339, 34473
 \tex_linedir:D 836
 \tex_linedirection:D 837
 \tex_linepenalty:D 340
 \tex_lineskip:D 341
 \tex_lineskiplimit:D 342
 \tex_localbrokenpenalty:D . 838, 1284
 \tex_localinterlinepenalty:D ...
 839, 1285
 \tex_localleftbox:D 844, 1287
 \tex_localrightbox:D 845, 1288
 \tex_long:D 343, 697, 698,
 699, 1452, 1454, 1457, 1480, 1481,
 1482, 1483, 1484, 1486, 1488, 1489,
 1490, 1491, 1495, 1497, 1503, 1505
 \tex_looseness:D 344
 \tex_lower:D 345, 32053
 \tex_lowercase:D
 858, 1371, 346, 3790, 3800,
 3869, 4249, 7273, 7325, 9309, 13396,
 13419, 18600, 18707, 36267, 36570
 \tex_lpcode:D 689
 \tex_luabytecode:D 840
 \tex_luabytecodecall:D 841
 \tex_luacopyinputnodes:D 842
 \tex_luadef:D 843
 \tex_luaescapestring:D
 846, 1258, 11600
 \tex_luafunction:D 847, 1259
 \tex_luafunctioncall:D 848
 \tex_luatexbanner:D 849
 \tex_luatexrevision:D ... 850, 36081
 \tex_luatexversion:D
 851, 1301, 1332, 1469, 8708,
 10305, 17048, 17726, 18950, 36079
 \tex_mag:D 347
 \tex_mapfile:D 664, 1303
 \tex_mapline:D 665, 1304
 \tex_mark:D 348
 \tex_marks:D 547
 \tex_mathaccent:D 349
 \tex_mathbin:D 350
 \tex_mathchar:D 351
 \tex_mathchardef:D 353, 352, 1475,
 17056, 17057, 29416, 29418, 29420
 \tex_mathchoice:D 353
 \tex_mathclose:D 354
 \tex_mathcode:D ... 355, 18563, 18565
 \tex_mathdelimitersmode:D 852
 \tex_mathdir:D 853, 1289
 \tex_mathdirection:D 854
 \tex_mathdisplayskipmode:D 855
 \tex_matheqnogapstep:D 856
 \tex_mathinner:D 356
 \tex_mathnolimitsmode:D 857
 \tex_mathop:D 357, 1348
 \tex_mathopen:D 358
 \tex_mathoption:D 858
 \tex_mathord:D 359
 \tex_mathpenaltiesmode:D 859
 \tex_mathpunct:D 360
 \tex_mathrel:D 361
 \tex_mathrulesfam:D 860
 \tex_mathscriptboxmode:D 862
 \tex_mathscriptcharmode:D 863
 \tex_mathscriptsmode:D 861
 \tex_mathstyle:D 864, 1260
 \tex_mathsurround:D 362
 \tex_mathsurroundmode:D 865
 \tex_mathsurroundskip:D 866
 \tex_maxdeadcycles:D 363
 \tex_maxdepth:D 364
 \tex_mdffivesum:D 710, 778, 1393, 11091
 \tex_meaning:D .. 365, 1206, 1223,
 1307, 1313, 1319, 1325, 1440, 1441
 \tex_medmuskip:D 366
 \tex_message:D 367
 \tex_middle:D 548, 1366
 \tex_mkern:D 368
 \tex_month:D ... 369, 1319, 1323, 1349
 \tex_moveleft:D 370, 32047
 \tex_moveright:D 371, 32049
 \tex_mskip:D 372
 \tex_muexpr:D .. 549, 20404, 20406,
 20414, 20416, 20420, 20422, 20426
 \tex_multiply:D 373
 \tex_muskip:D 374
 \tex_muskipdef:D 375
 \tex_mutogluue:D 343, 550
 \tex_newlinechar:D
 376, 1872, 9315, 9530,
 10418, 12110, 12136, 12140, 13039

| | | | |
|--|---------------------------------------|---|----------------|
| <code>\tex_noalign:D</code> | 377 | <code>\tex_pageshrink:D</code> | 403 |
| <code>\tex_noautospaceing:D</code> | 1141 | <code>\tex_pagestretch:D</code> | 404 |
| <code>\tex_noautoxspacing:D</code> | 1142 | <code>\tex_pagetopoffset:D</code> | 878, 1265 |
| <code>\tex_noboundary:D</code> | 378 | <code>\tex_pagetotal:D</code> | 405 |
| <code>\tex_noexpand:D</code> | 379, 1436 | <code>\tex_pagewidth:D</code> | 669, 927, 1294 |
| <code>\tex_nohrule:D</code> | 867 | <code>\tex_par:D</code> | 406 |
| <code>\tex_noindent:D</code> | 380 | <code>\tex_pardir:D</code> | 879, 1295 |
| <code>\tex_nokerns:D</code> | 868, 1261 | <code>\tex_pardirection:D</code> | 880 |
| <code>\tex_noligatures:D</code> | 666 | <code>\tex_parfillskip:D</code> | 407 |
| <code>\tex_noligs:D</code> | 869, 1262 | <code>\tex_parindent:D</code> | 408 |
| <code>\tex_nolimits:D</code> | 381, 34474 | <code>\tex_parshape:D</code> | 409 |
| <code>\tex_nonscript:D</code> | 382 | <code>\tex_parshapedimen:D</code> | 553 |
| <code>\tex_nonstopmode:D</code> | 383 | <code>\tex_parshapeindent:D</code> | 554 |
| <code>\tex_normaldeviate:D</code> | 667, 924 | <code>\tex_parshapelength:D</code> | 555 |
| <code>\tex_nospaces:D</code> | 870 | <code>\tex_parskip:D</code> | 410 |
| <code>\tex_novrule:D</code> | 871 | <code>\tex_partokencontext:D</code> | 1183 |
| <code>\tex_nulldelimiterspace:D</code> | 384 | <code>\tex_partokenname:D</code> | 1184 |
| <code>\tex_nullfont:D</code> | 385, 19257 | <code>\tex_patterns:D</code> | 411 |
| <code>\tex_number:D</code> | 386, 16918, 32683 | <code>\tex_pausing:D</code> | 412 |
| <code>\tex_numexpr:D</code> | 551, 4396, 16919, 22325, 35946 | <code>\tex_pdfannot:D</code> | 578 |
| <code>\tex_odelcode:D</code> | 1176 | <code>\tex_pdfcatalog:D</code> | 579 |
| <code>\tex_odelimiter:D</code> | 1177 | <code>\tex_pdfcolorstack:D</code> | 581 |
| <code>\tex_omathaccent:D</code> | 1178 | <code>\tex_pdfcolorstackinit:D</code> | 582 |
| <code>\tex_omathchar:D</code> | 1179 | <code>\tex_pdfcompresslevel:D</code> | 580 |
| <code>\tex_omathchardef:D</code> | 1180, 1472, 1473, 17049, 17051, 17052 | <code>\tex_pdfcreationdate:D</code> | 583 |
| <code>\tex_omathcode:D</code> | 1181 | <code>\tex_pdfdecimaldigits:D</code> | 584 |
| <code>\tex_omit:D</code> | 387 | <code>\tex_pdfdest:D</code> | 585 |
| <code>\tex_openin:D</code> | 388, 10139 | <code>\tex_pdfdestmargin:D</code> | 586 |
| <code>\tex_openout:D</code> | 389, 10354 | <code>\tex_pdfendlink:D</code> | 587 |
| <code>\tex_or:D</code> | 390, 1418 | <code>\tex_pdfendthread:D</code> | 588 |
| <code>\tex_oradical:D</code> | 1182 | <code>\tex_pdfextension:D</code> | 881 |
| <code>\tex_outer:D</code> | 391, 1350, 36362 | <code>\tex_pdffeedback:D</code> | 882 |
| <code>\tex_output:D</code> | 392 | <code>\tex_pdffontattr:D</code> | 589 |
| <code>\tex_outputbox:D</code> | 872, 1263 | <code>\tex_pdffontname:D</code> | 590 |
| <code>\tex_outputpenalty:D</code> | 393 | <code>\tex_pdffontobjnum:D</code> | 591 |
| <code>\tex_over:D</code> | 394, 1351 | <code>\tex_pdfgamma:D</code> | 592 |
| <code>\tex_overfullrule:D</code> | 395 | <code>\tex_pdfgentounicode:D</code> | 595 |
| <code>\tex_overline:D</code> | 396 | <code>\tex_pdfglyptounicode:D</code> | 596 |
| <code>\tex_overwithdelims:D</code> | 397 | <code>\tex_pdfhorigin:D</code> | 597 |
| <code>\tex_pagebottomoffset:D</code> ... | 873, 1290 | <code>\tex_pdfimageapplygamma:D</code> | 593 |
| <code>\tex_pagedepth:D</code> | 398 | <code>\tex_pdfimagegamma:D</code> | 594 |
| <code>\tex_pagedir:D</code> | 874, 1291, 1362 | <code>\tex_pdfimagehicolor:D</code> | 598 |
| <code>\tex_pagedirection:D</code> | 875 | <code>\tex_pdfimageresolution:D</code> | 599 |
| <code>\tex_pagediscards:D</code> | 552 | <code>\tex_pdfincludechars:D</code> | 600 |
| <code>\tex_pagefillllstretch:D</code> | 399 | <code>\tex_pdfinclusioncopyfonts:D</code> .. | 601 |
| <code>\tex_pagefillstretch:D</code> | 400 | <code>\tex_pdfinclusionerrorlevel:D</code> .. | 603 |
| <code>\tex_pagefilstretch:D</code> | 401 | <code>\tex_pdfinfo:D</code> | 604 |
| <code>\tex_pagefistretch:D</code> | 1143 | <code>\tex_pdflastannot:D</code> | 605 |
| <code>\tex_pagegoal:D</code> | 402 | <code>\tex_pdflastlink:D</code> | 606 |
| <code>\tex_pageheight:D</code> | 668, 926, 1292 | <code>\tex_pdflastobj:D</code> | 607 |
| <code>\tex_pageleftoffset:D</code> | 876, 1264 | <code>\tex_pdflastxform:D</code> | 608, 917 |
| <code>\tex_pagerightoffset:D</code> | 877, 1293 | <code>\tex_pdflastximage:D</code> | 609, 919 |
| | | <code>\tex_pdflastximagecolordepth:D</code> . | 611 |
| | | <code>\tex_pdflastximagepages:D</code> .. | 612, 921 |

| | | | |
|---|-----------|---|-------------------|
| <code>\tex_pdflinkmargin:D</code> | 613 | <code>\tex_prerelpenalty:D</code> | 890 |
| <code>\tex_pdfliteral:D</code> | 614 | <code>\tex_pretolerance:D</code> | 417 |
| <code>\tex_pdfmajorversion:D</code> | 615 | <code>\tex_prevdepth:D</code> | 418 |
| <code>\tex_pdfminorversion:D</code> | 616 | <code>\tex_prevgraf:D</code> | 419 |
| <code>\tex_pdfnames:D</code> | 617 | <code>\tex_primitive:D</code> | |
| <code>\tex_pdfobj:D</code> | 618 | 393, 672, 780, 2654, 8978, 8988 | |
| <code>\tex_pdfobjcompresslevel:D</code> | 619 | <code>\tex_protected:D</code> | |
| <code>\tex_pdfoutline:D</code> | 620 | 557, 1480, 1482, 1484, | |
| <code>\tex_pdfoutput:D</code> | 576, | 1485, 1486, 1487, 1488, 1489, 1490, | |
| 621, 925, 1341, 8752, 8758, 8766, 9105 | | 1491, 1499, 1501, 1503, 1505, 36362 | |
| <code>\tex_pdfpageattr:D</code> | 622 | <code>\tex_protrudechars:D</code> | 673, 928 |
| <code>\tex_pdfpagebox:D</code> | 624 | <code>\tex_ptexminorversion:D</code> | |
| <code>\tex_pdfpageref:D</code> | 625 | 1146, 36071, 36090 | |
| <code>\tex_pdfpageresources:D</code> | 626 | <code>\tex_ptexrevision:D</code> 1147, 36072, 36091 | |
| <code>\tex_pdfpagesattr:D</code> | 623, 627 | <code>\tex_ptexversion:D</code> | |
| <code>\tex_pdfrefobj:D</code> | 628 | ... 1148, 36066, 36069, 36085, 36088 | |
| <code>\tex_pdfrefxform:D</code> | 629, 931 | <code>\tex_pxdimen:D</code> | 674, 929 |
| <code>\tex_pdfrefximage:D</code> | 630, 932 | <code>\tex_quitvmode:D</code> | 690 |
| <code>\tex_pdfrestore:D</code> | 631 | <code>\tex_radical:D</code> | 420 |
| <code>\tex_pdfretval:D</code> | 632 | <code>\tex_raise:D</code> | 421, 32051 |
| <code>\tex_pdfsave:D</code> | 633 | <code>\tex_randomseed:D</code> | 675, 930, 9005 |
| <code>\tex_pdfsetmatrix:D</code> | 634 | <code>\tex_read:D</code> | 422, 9435, 10204 |
| <code>\tex_pdfstartlink:D</code> | 635 | <code>\tex_readline:D</code> | 558, 10221 |
| <code>\tex_pdfstartthread:D</code> | 636 | <code>\tex_readpapersizespecial:D</code> .. | 1149 |
| <code>\tex_pdfsuppressptexinfo:D</code> | 637 | <code>\tex_relax:D</code> | |
| <code>\tex_pdftexbanner:D</code> | 682, 1333 | ... 343, 967, 423, 1445, 16920, 19937 | |
| <code>\tex_pdftexrevision:D</code> 683, 1334, 36062 | | <code>\tex_relpnalty:D</code> | 424 |
| <code>\tex_pdftexversion:D</code> | | <code>\tex_resettimer:D</code> | 676, 781 |
| ... 340, 684, 1300, 1335, 8709, 36060 | | <code>\tex_right:D</code> | 425, 1367 |
| <code>\tex_pdfthread:D</code> | 638 | <code>\tex_rightghost:D</code> | 891, 1296 |
| <code>\tex_pdfthreadmargin:D</code> | 639 | <code>\tex_rightthyphenmin:D</code> | 426 |
| <code>\tex_pdftrailer:D</code> | 640 | <code>\tex_rightmarginkern:D</code> | 691 |
| <code>\tex_pdfuniqueresname:D</code> | 641 | <code>\tex_rightskip:D</code> | 427 |
| <code>\tex_pdfvariable:D</code> | 883 | <code>\tex_romannumeral:D</code> 364, 390, 428, | |
| <code>\tex_pdfvorigin:D</code> | 642 | 1438, 1450, 1775, 18612, 22327, 28796 | |
| <code>\tex_pdfxform:D</code> | 643, 934 | <code>\tex_rpcode:D</code> | 692 |
| <code>\tex_pdfxformname:D</code> | 644 | <code>\tex_savecatcodetable:D</code> | |
| <code>\tex_pdfximage:D</code> | 645, 935 | 892, 1270, 28757, 28813 | |
| <code>\tex_pdfximagebbox:D</code> | 646 | <code>\tex_savepos:D</code> | 677, 933 |
| <code>\tex_penalty:D</code> | 413 | <code>\tex_savinghyphcodes:D</code> | 559 |
| <code>\tex_pkmode:D</code> | 670 | <code>\tex_savingvdiscards:D</code> | 560 |
| <code>\tex_pkresolution:D</code> | 671 | <code>\tex_scantextokens:D</code> | 893, 1271 |
| <code>\tex_postbreakpenalty:D</code> | 1144 | <code>\tex_scantokens:D</code> .. | 561, 12122, 12183 |
| <code>\tex_postdisplaypenalty:D</code> | 414 | <code>\tex_scriptbaselineshiftfactor:D</code> | |
| <code>\tex_posttexhyphenchar:D</code> ... | 884, 1266 | 1151 | |
| <code>\tex_postthyphenchar:D</code> | 885, 1267 | <code>\tex_scriptfont:D</code> | 429 |
| <code>\tex_prebinoppenalty:D</code> | 886 | <code>\tex_scriptscriptbaselineshiftfactor:D</code> | |
| <code>\tex_prebreakpenalty:D</code> | 1145 | 1153 | |
| <code>\tex_predisplaydirection:D</code> | 556 | <code>\tex_scriptscriptfont:D</code> | 430 |
| <code>\tex_predisplaygapfactor:D</code> | 887 | <code>\tex_scriptscriptstyle:D</code> | 431 |
| <code>\tex_predisplaypenalty:D</code> | 415 | <code>\tex_scriptspace:D</code> | 432 |
| <code>\tex_predisplaysize:D</code> | 416 | <code>\tex_scriptstyle:D</code> | 433 |
| <code>\tex_preexhyphenchar:D</code> | 888, 1268 | <code>\tex_scrollmode:D</code> | 434 |
| <code>\tex_prehyphenchar:D</code> | 889, 1269 | | |

- \tex_setbox:D 435,
31991, 31993, 31997, 31999, 32017,
32026, 32035, 32070, 32072, 32120,
32125, 32132, 32137, 32144, 32150,
32164, 32170, 32212, 32217, 32224,
32229, 32236, 32241, 32248, 32254,
32269, 32275, 32286, 32290, 33630
- \tex_setfontid:D 894
- \tex_setlanguage:D 436
- \tex_setrandomseed:D . 678, 936, 9021
- \tex_sfcode:D 437, 18581, 18583
- \tex_shapemode:D 895
- \tex_shellescape:D ... 679, 782, 9077
- \tex_shipout:D 438, 1202, 1226
- \tex_show:D 439
- \tex_showbox:D 440, 32107
- \tex_showboxbreadth:D ... 441, 32103
- \tex_showboxdepth:D 442, 32104
- \tex_showgroups:D 562, 2202
- \tex_showifs:D 563
- \tex_showlists:D 443
- \tex_showmode:D 1154
- \tex_showthe:D 444
- \tex_showtokens:D
..... 699, 564, 1360, 9534, 13043
- \tex_sjis:D 1155
- \tex_skewchar:D 445
- \tex_skip:D
446, 3906, 3935, 3954, 4012, 4028, 4034
- \tex_skipdef:D 447
- \tex_space:D 187
- \tex_spacefactor:D 448
- \tex_spaceskip:D 449
- \tex_span:D 450
- \tex_special:D 451
- \tex_splitbotmark:D 452
- \tex_splitbotmarks:D 565
- \tex_splitdiscards:D 566
- \tex_splitfirstmark:D 453
- \tex_splitfirstmarks:D 567
- \tex_splitmaxdepth:D 454
- \tex_splittopskip:D 455
- \tex_strcmp:D
..... 708, 1371, 11215, 13203, 22698
- \tex_string:D 456, 1205,
1209, 1308, 1314, 1320, 1326, 1443
- \tex_suppressfontnotfounderror:D
..... 714, 1279
- \tex_suppressifcsnameerror:D ...
..... 896, 1272
- \tex_suppresslongerror:D .. 897, 1274
- \tex_suppressmathparerror:D 898, 1275
- \tex_suppressoutererror:D . 899, 1277
- \tex_suppressprimitiveerror:D .. 901
- \tex_synctex:D 693
- \tex_tabskip:D 457
- \tex_tagcode:D 694
- \tex_tate:D 1156
- \tex_tbaselineshift:D 1157
- \tex_textbaselineshiftfactor:D 1159
- \tex_textdir:D 902, 1297
- \tex_textdirection:D 903
- \tex_textfont:D 458
- \tex_textstyle:D 459
- \tex_TeXTeXstate:D 568
- \tex_tfont:D 1160
- \tex_the:D 343,
383, 1004, 1010, 1011, 159, 460,
1888, 2171, 2320, 2324, 2653, 2695,
2786, 2805, 2820, 2825, 3430, 3462,
3510, 3511, 3542, 3543, 3549, 3550,
4027, 4144, 4281, 4399, 4418, 4424,
4427, 9005, 16456, 16931, 16932,
17107, 17108, 18495, 18565, 18571,
18577, 18583, 20167, 20168, 20169,
20239, 20240, 23318, 23806, 32090
- \tex_thickmuskip:D 461
- \tex_thinmuskip:D 462
- \tex_time:D 463, 1307, 1311
- \tex_toks:D 464, 2796,
2825, 3403, 3430, 3462, 3499, 3510,
3511, 3542, 3543, 3549, 3550, 3554,
3564, 3574, 3851, 3869, 4027, 4399,
4402, 4404, 4409, 4417, 4418, 4423,
4424, 4427, 16456, 16467, 16468, 16469
- \tex_toksapp:D 904
- \tex_toksdef:D 465, 3681
- \tex_tokspre:D 905
- \tex_tolerance:D 466
- \tex_topmark:D 467
- \tex_topmarks:D 569
- \tex_topskip:D 468
- \tex_tpack:D 906
- \tex_tracingassigns:D 570
- \tex_tracingcommands:D 469
- \tex_tracingfonts:D
..... 680, 937, 1231, 1233, 1237
- \tex_tracinggroups:D 571
- \tex_tracingifs:D 572
- \tex_tracinglostchars:D 470
- \tex_tracingmacros:D 471
- \tex_tracingnesting:D
..... 573, 8878, 10901, 12107
- \tex_tracingonline:D
..... 472, 2199, 2205, 2206, 32105
- \tex_tracingoutput:D 473
- \tex_tracingpages:D 474
- \tex_tracingparagraphs:D 475

| | | | |
|--|-------------------|---|------|
| <code>\tex_tracingrestores:D</code> | 476 | <code>\tex_Umathinnerordspacing:D</code> . . . | 994 |
| <code>\tex_tracingscantokens:D</code> | 574 | <code>\tex_Umathinnerpunctspacing:D</code> . . | 996 |
| <code>\tex_tracingstacklevels:D</code> | 1185 | <code>\tex_Umathinnerrelspacing:D</code> . . . | 997 |
| <code>\tex_tracingstats:D</code> | 477 | <code>\tex_Umathlimitabovebgap:D</code> | 998 |
| <code>\tex_uccode:D</code> | 478, 18575, 18577 | <code>\tex_Umathlimitabovekern:D</code> | 999 |
| <code>\tex_Uchar:D</code> | 939, 1278, 29100 | <code>\tex_Umathlimitabovevgap:D</code> . . . | 1000 |
| <code>\tex_Ucharcat:D</code> 940, 1383, 18660, 29105 | | <code>\tex_Umathlimitbelowbgap:D</code> . . . | 1001 |
| <code>\tex_uchyph:D</code> | 479 | <code>\tex_Umathlimitbelowkern:D</code> . . . | 1002 |
| <code>\tex_ucs:D</code> | 1173 | <code>\tex_Umathlimitbelowvgap:D</code> . . . | 1003 |
| <code>\tex_Udelcode:D</code> | 941 | <code>\tex_Umathnolimitsubfactor:D</code> . | 1004 |
| <code>\tex_Udelcodenum:D</code> | 942 | <code>\tex_Umathnolimitsupfactor:D</code> . | 1005 |
| <code>\tex_Udelimiter:D</code> | 943 | <code>\tex_Umathopbinspacing:D</code> | 1006 |
| <code>\tex_Udelimiterover:D</code> | 944 | <code>\tex_Umathopclosespacing:D</code> . . . | 1007 |
| <code>\tex_Udelimiterunder:D</code> | 945 | <code>\tex_Umathopenbinspacing:D</code> . . . | 1008 |
| <code>\tex_Uhextensible:D</code> | 946 | <code>\tex_Umathopenclosespacing:D</code> . | 1009 |
| <code>\tex_Umathaccent:D</code> | 947 | <code>\tex_Umathopeninnerspacing:D</code> . | 1010 |
| <code>\tex_Umathaxis:D</code> | 948 | <code>\tex_Umathopenopenspacing:D</code> . . | 1011 |
| <code>\tex_Umathbinbinspacing:D</code> | 949 | <code>\tex_Umathopenopspacing:D</code> | 1012 |
| <code>\tex_Umathbinclosespacing:D</code> . . . | 950 | <code>\tex_Umathopenordspacing:D</code> . . . | 1013 |
| <code>\tex_Umathbininnerspacing:D</code> . . . | 951 | <code>\tex_Umathopenpunctspacing:D</code> . | 1014 |
| <code>\tex_Umathbinopenspacing:D</code> | 952 | <code>\tex_Umathopenrelspacing:D</code> . . . | 1015 |
| <code>\tex_Umathbinopspacing:D</code> | 953 | <code>\tex_Umathoperatorsize:D</code> | 1016 |
| <code>\tex_Umathbinordspacing:D</code> | 954 | <code>\tex_Umathopinnerspacing:D</code> . . . | 1017 |
| <code>\tex_Umathbinpunctspacing:D</code> . . . | 955 | <code>\tex_Umathopopenspacing:D</code> | 1018 |
| <code>\tex_Umathbinrelspacing:D</code> | 956 | <code>\tex_Umathopopspacing:D</code> | 1019 |
| <code>\tex_Umathchar:D</code> | 957 | <code>\tex_Umathopordspacing:D</code> | 1020 |
| <code>\tex_Umathcharclass:D</code> | 958 | <code>\tex_Umathoppunctspacing:D</code> . . . | 1021 |
| <code>\tex_Umathchardef:D</code> | 959 | <code>\tex_Umathoprelspacing:D</code> | 1022 |
| <code>\tex_Umathcharfam:D</code> | 960 | <code>\tex_Umathordbinspacing:D</code> | 1023 |
| <code>\tex_Umathcharnum:D</code> | 961 | <code>\tex_Umathordclosespacing:D</code> . . | 1024 |
| <code>\tex_Umathcharnumdef:D</code> | 962 | <code>\tex_Umathordinnerspacing:D</code> . . | 1025 |
| <code>\tex_Umathcharslot:D</code> | 963 | <code>\tex_Umathordopenspacing:D</code> . . . | 1026 |
| <code>\tex_Umathclosebinspacing:D</code> . . . | 964 | <code>\tex_Umathordopspacing:D</code> | 1027 |
| <code>\tex_Umathcloseclosespacing:D</code> . . | 966 | <code>\tex_Umathordordspacing:D</code> | 1028 |
| <code>\tex_Umathcloseinnerspacing:D</code> . . | 968 | <code>\tex_Umathordpunctspacing:D</code> . . | 1029 |
| <code>\tex_Umathcloseopenspacing:D</code> . . | 969 | <code>\tex_Umathordrelspacing:D</code> | 1030 |
| <code>\tex_Umathcloseopspacing:D</code> | 970 | <code>\tex_Umathoverbarkern:D</code> | 1031 |
| <code>\tex_Umathcloseordspacing:D</code> . . . | 971 | <code>\tex_Umathoverbarrule:D</code> | 1032 |
| <code>\tex_Umathclosepunctspacing:D</code> . . | 973 | <code>\tex_Umathoverbarvgap:D</code> | 1033 |
| <code>\tex_Umathcloserelspacing:D</code> . . . | 974 | <code>\tex_Umathoverdelimiterbgap:D</code> . | 1035 |
| <code>\tex_Umathcode:D</code> | 975 | <code>\tex_Umathoverdelimitervgap:D</code> . | 1037 |
| <code>\tex_Umathcodenum:D</code> | 976 | <code>\tex_Umathpunctbinspacing:D</code> . . | 1038 |
| <code>\tex_Umathconnectoroverlapmin:D</code> . | 978 | <code>\tex_Umathpunctclosespacing:D</code> . | 1040 |
| <code>\tex_Umathfractiondelsize:D</code> . . . | 979 | <code>\tex_Umathpunctinnerspacing:D</code> . | 1042 |
| <code>\tex_Umathfractiondenomdown:D</code> . . | 981 | <code>\tex_Umathpunctopenspacing:D</code> . | 1043 |
| <code>\tex_Umathfractiondenomvgap:D</code> . . | 983 | <code>\tex_Umathpunctopspacing:D</code> . . . | 1044 |
| <code>\tex_Umathfractionnumup:D</code> | 984 | <code>\tex_Umathpunctordspacing:D</code> . . | 1045 |
| <code>\tex_Umathfractionnumvgap:D</code> . . . | 985 | <code>\tex_Umathpunctpunctspacing:D</code> . | 1047 |
| <code>\tex_Umathfractionrule:D</code> | 986 | <code>\tex_Umathpunctrelspacing:D</code> . . | 1048 |
| <code>\tex_Umathinnerbinspacing:D</code> . . . | 987 | <code>\tex_Umathquad:D</code> | 1049 |
| <code>\tex_Umathinnerclosespacing:D</code> . . | 989 | <code>\tex_Umathradicaldegreeafter:D</code> . | 1051 |
| <code>\tex_Umathinnerinnerspacing:D</code> . . | 991 | <code>\tex_Umathradicaldegreebefore:D</code> . | 1053 |
| <code>\tex_Umathinneropenspacing:D</code> . . | 992 | <code>\tex_Umathradicaldegreeraise:D</code> . | 1055 |
| <code>\tex_Umathinneropspacing:D</code> | 993 | <code>\tex_Umathradicalkern:D</code> | 1056 |

| | | | |
|---|---|--|--|
| <code>\tex_Umathradicalrule:D</code> | 1057 | <code>\tex_uptexversion:D</code> | 1175, 36094 |
| <code>\tex_Umathradicalvgap:D</code> | 1058 | <code>\tex_Uradical:D</code> | 1094 |
| <code>\tex_Umathrelbinspacing:D</code> | 1059 | <code>\tex_Uroot:D</code> | 1095 |
| <code>\tex_Umathrelclosespacing:D</code> | 1060 | <code>\tex_Uskewed:D</code> | 1096 |
| <code>\tex_Umathrelinnerspacing:D</code> | 1061 | <code>\tex_Uskewedwithdelims:D</code> | 1097 |
| <code>\tex_Umathreloppenspacing:D</code> | 1062 | <code>\tex_Ustack:D</code> | 1098 |
| <code>\tex_Umathreloppspacing:D</code> | 1063 | <code>\tex_Ustartdisplaymath:D</code> | 1099 |
| <code>\tex_Umathrelordspacing:D</code> | 1064 | <code>\tex_Ustartmath:D</code> | 1100 |
| <code>\tex_Umathrelpunctspacing:D</code> | 1065 | <code>\tex_Ustopdisplaymath:D</code> | 1101 |
| <code>\tex_Umathrelrelspacing:D</code> | 1066 | <code>\tex_Ustopmath:D</code> | 1102 |
| <code>\tex_Umathskewedfractionhgap:D</code> | 1068 | <code>\tex_Usubscript:D</code> | 1103 |
| <code>\tex_Umathskewedfractionvgap:D</code> | 1070 | <code>\tex_Usuperscript:D</code> | 1104 |
| <code>\tex_Umathspaceafterscript:D</code> | 1071 | <code>\tex_Uunderdelimiter:D</code> | 1105 |
| <code>\tex_Umathstackdenomdown:D</code> | 1072 | <code>\tex_Uvextensible:D</code> | 1106 |
| <code>\tex_Umathstacknumup:D</code> | 1073 | <code>\tex_vadjust:D</code> | 489 |
| <code>\tex_Umathstackvgap:D</code> | 1074 | <code>\tex_valign:D</code> | 490 |
| <code>\tex_Umathsubshiftdown:D</code> | 1075 | <code>\tex_vbadness:D</code> | 491 |
| <code>\tex_Umathsubshiftdrop:D</code> | 1076 | <code>\tex_vbox:D</code> | 492, 32197,
32202, 32207, 32212, 32217, 32236,
32241, 32248, 32254, 32269, 32275 |
| <code>\tex_Umathsubsupshiftdown:D</code> | 1077 | <code>\tex_vcenter:D</code> | 493, 1352 |
| <code>\tex_Umathsubsupvgap:D</code> | 1078 | <code>\tex_vfi:D</code> | 1165 |
| <code>\tex_Umathsubtopmax:D</code> | 1079 | <code>\tex_vfil:D</code> | 494 |
| <code>\tex_Umathsupbottommin:D</code> | 1080 | <code>\tex_vfill:D</code> | 495 |
| <code>\tex_Umathsupshiftdrop:D</code> | 1081 | <code>\tex_vfilneg:D</code> | 496 |
| <code>\tex_Umathsupshiftup:D</code> | 1082 | <code>\tex_vfuzz:D</code> | 497 |
| <code>\tex_Umathsupsubbottommax:D</code> | 1083 | <code>\tex_vffset:D</code> | 498, 1359 |
| <code>\tex_Umathunderbarkern:D</code> | 1084 | <code>\tex_vpack:D</code> | 907 |
| <code>\tex_Umathunderbarrule:D</code> | 1085 | <code>\tex_vrule:D</code> | 499, 33695 |
| <code>\tex_Umathunderbarvgap:D</code> | 1086 | <code>\tex_vsize:D</code> | 500 |
| <code>\tex_Umathunderdelimiterbgap:D</code> | 1088 | <code>\tex_vskip:D</code> | 501, 20356 |
| <code>\tex_Umathunderdelimitervgap:D</code> | 1090 | <code>\tex_vsplit:D</code> | 502, 32286, 32291 |
| <code>\tex_undefined:D</code> | | <code>\tex_vss:D</code> | 503 |
| | 870, 1231, 1303, 1304, 1311, 1317,
1323, 1329, 1333, 1334, 1335, 1963,
1971, 3353, 3790, 3800, 3878, 3978,
17751, 20891, 20906, 20982, 21003 | <code>\tex_vtop:D</code> | 504, 32199, 32224, 32229 |
| <code>\tex_underline:D</code> | 480, 1200 | <code>\tex_wd:D</code> | 505, 32008 |
| <code>\tex_unexpanded:D</code> | | <code>\tex_widowpenalties:D</code> | 577 |
| | 389, 575, 1353, 1437, 2569, 29103 | <code>\tex_widowpenalty:D</code> | 506 |
| <code>\tex_unhbox:D</code> | 481, 32193 | <code>\tex_write:D</code> | |
| <code>\tex_unhcopy:D</code> | 482, 32192 | | 507, 1890, 1892, 10399, 10402, 10419 |
| <code>\tex_uniformdeviate:D</code> | 790,
1167, 1168, 681, 938, 8776, 16435,
16466, 28270, 28271, 28452, 28455 | <code>\tex_xdef:D</code> | 508, 1493, 1497, 1501, 1505 |
| <code>\tex_unkern:D</code> | 483 | <code>\tex_XeTeXcharclass:D</code> | 715 |
| <code>\tex_unless:D</code> | 576, 1421 | <code>\tex_XeTeXcharglyph:D</code> | 716 |
| <code>\tex_Unosubscript:D</code> | 1091 | <code>\tex_XeTeXcountfeatures:D</code> | 717 |
| <code>\tex_Unosuperscript:D</code> | 1092 | <code>\tex_XeTeXcountglyphs:D</code> | 718 |
| <code>\tex_unpenalty:D</code> | 484 | <code>\tex_XeTeXcountselectors:D</code> | 719 |
| <code>\tex_unskip:D</code> | 485 | <code>\tex_XeTeXcountvariations:D</code> | 720 |
| <code>\tex_unvbox:D</code> | 486, 32282 | <code>\tex_XeTeXdashbreakstate:D</code> | 722 |
| <code>\tex_unvcopy:D</code> | 487, 32281 | <code>\tex_XeTeXdefaultencoding:D</code> | 721 |
| <code>\tex_Uoverdelimiter:D</code> | 1093 | <code>\tex_XeTeXfeaturecode:D</code> | 723 |
| <code>\tex_uppercase:D</code> | 488, 36572 | <code>\tex_XeTeXfeaturename:D</code> | 724 |
| <code>\tex_uptexrevision:D</code> | 1174, 36095 | <code>\tex_XeTeXfindfeaturebyname:D</code> | 726 |
| | | <code>\tex_XeTeXfindselectorbyname:D</code> | 728 |
| | | <code>\tex_XeTeXfindvariationbyname:D</code> | 730 |
| | | <code>\tex_XeTeXfirstfontchar:D</code> | 731 |

- `\tex_XeTeXfonttype:D` 732
- `\tex_XeTeXgenerateactualtext:D` . 734
- `\tex_XeTeXglyph:D` 735
- `\tex_XeTeXglyphbounds:D` 736
- `\tex_XeTeXglyphindex:D` 737
- `\tex_XeTeXglyphname:D` 738
- `\tex_XeTeXinputencoding:D` 739
- `\tex_XeTeXinputnormalization:D` . 741
- `\tex_XeTeXinterchartokenstate:D` 743
- `\tex_XeTeXinterchartoks:D` 744
- `\tex_XeTeXisdefaultselector:D` . 746
- `\tex_XeTeXisexclusivefeature:D` . 748
- `\tex_XeTeXlastfontchar:D` 749
- `\tex_XeTeXlinebreaklocale:D` ... 751
- `\tex_XeTeXlinebreakpenalty:D` .. 752
- `\tex_XeTeXlinebreakskip:D` 750
- `\tex_XeTeXOTcountfeatures:D` ... 753
- `\tex_XeTeXOTcountlanguages:D` .. 754
- `\tex_XeTeXOTcountscripts:D` 755
- `\tex_XeTeXOTfeaturetag:D` 756
- `\tex_XeTeXOTlanguagetag:D` 757
- `\tex_XeTeXOTscripttag:D` 758
- `\tex_XeTeXpdfpfile:D` 759
- `\tex_XeTeXpdfpagecount:D` 760
- `\tex_XeTeXpicfile:D` 761
- `\tex_XeTeXrevision:D` 762, 8984, 36103
- `\tex_XeTeXselectorname:D` 763
- `\tex_XeTeXtracingfonts:D` 764
- `\tex_XeTeXupwardsmode:D` 765
- `\tex_XeTeXuseglyphmetrics:D` ... 766
- `\tex_XeTeXvariation:D` 767
- `\tex_XeTeXvariationdefault:D` .. 768
- `\tex_XeTeXvariationmax:D` 769
- `\tex_XeTeXvariationmin:D` 770
- `\tex_XeTeXvariationname:D` 771
- `\tex_XeTeXversion:D` 772, 8716, 17050, 17727, 18951, 36102
- `\tex_xkanjiskip:D` 1161
- `\tex_xleaders:D` 509
- `\tex_xspaceskip:D` 510
- `\tex_xspcode:D` 1162
- `\tex_ybaselineshift:D` 1163
- `\tex_year:D` 511, 1325, 1329
- `\tex_yoko:D` 1164
- `\text` 31557
- text commands:
 - `\l_text_accents_tl` 265, 267, 29387, 29626, 31691
 - `\l_text_case_exclude_arg_tl` 266, 268, 29405, 29965
 - `\text_declare_expand_equivalent:Nn` 265, 29769
 - `\text_declare_purify_equivalent:Nn` 267, 31537, 31550, 31551, 31552, 31553, 31570, 31595, 31596, 31598, 31599, 31601, 31604, 31605, 31611, 31613, 31614, 31615, 31623, 31635, 31677, 31692
 - `\text_expand` 266
 - `\text_expand:n` 265, 267, 29421, 29802, 31354
 - `\l_text_expand_exclude_tl` 265, 268, 29411, 29596
 - `\l_text_letterlike_tl` 265, 268, 29387, 29646
 - `\text_lowercase:n` 130, 184, 266, 29780, 36619, 36621
 - `\text_lowercase:nn` 266, 29780, 36622, 36624
 - `\l_text_math_arg_tl` 265, 267, 268, 29407, 29595, 29964, 31470
 - `\l_text_math_delims_tl` 265, 267, 268, 29409, 29533, 29897, 31399
 - `\text_purify:n` 267, 31348
 - `\text_titlecase:n` 130, 183, 266, 267, 29780, 36631, 36633
 - `\text_titlecase:nn` 266, 29780, 36634, 36636
 - `\l_text_titlecase_check_letter_`
 `bool` 267, 268, 29778, 30104
 - `\text_titlecase_first:n` 266, 267, 29780
 - `\text_titlecase_first:nn` . 266, 29780
 - `\text_uppercase:n` 130, 183, 185, 266, 29780, 36625, 36627
 - `\text_uppercase:nn` 266, 29780, 36628, 36630
- text internal commands:
 - `__text_change_case:nnn` 29781, 29783, 29785, 29787, 29789, 29791, 29793, 29795, 29796
 - `__text_change_case_aux:nnn` .. 29796
 - `__text_change_case_boundary_`
 `upper_el:nnN` 30449
 - `__text_change_case_boundary_`
 `upper_el:NnnN` 30449
 - `__text_change_case_boundary_`
 `upper_el:Nnnw` 30449
 - `__text_change_case_boundary_`
 `upper_el:nnNw` 30449
 - `__text_change_case_break:w` .. 29796
 - `__text_change_case_char:nnnN` ... 29796, 30220, 30231, 30240, 30254, 30502, 30602, 30635, 30717, 30731, 30754
 - `__text_change_case_char_`
 `auxi:nnnN` 29796

| | | | |
|--|-----------------------|--|---|
| _text_change_case_char_-
auxii:nnnN | 29796 | _text_change_case_group_-
upper:nnn | 29796 |
| _text_change_case_char_-
lower:nnN | 29796 | _text_change_case_if_greek:nTF | 30243 |
| _text_change_case_char_next_-
end:nn | 29796 | _text_change_case_if_greek_-
accent:nTF | 30243 |
| _text_change_case_char_next_-
lower:nn | 29796 | _text_change_case_if_greek_-
accent_p:n | 30243 |
| _text_change_case_char_next_-
title:nn | 29796 | _text_change_case_if_greek_-
diacritic:nTF | 30243 |
| _text_change_case_char_next_-
titleonly:nn | 29796 | _text_change_case_if_greek_-
diacritic_p:n | 30243 |
| _text_change_case_char_next_-
upper:nn | 29796 | _text_change_case_if_greek_p:n | 30243 |
| _text_change_case_char_-
title:nN | 29796 | _text_change_case_if_takes_-
dialytika:nTF | 30243 |
| _text_change_case_char_-
title:nnN | 29796 | _text_change_case_letterlike:nnnnN | 29796 |
| _text_change_case_char_-
title:nnnN | 29796 | _text_change_case_letterlike_-
lower:nnN | 29796 |
| _text_change_case_char_-
titleonly:nN | 29796 | _text_change_case_letterlike_-
title:nnN | 29796 |
| _text_change_case_char_-
titleonly:nnN | 29796 | _text_change_case_letterlike_-
titleonly:nnN | 29796 |
| _text_change_case_char_-
upper:nnN | 29796 | _text_change_case_letterlike_-
upper:nnN | 29796 |
| _text_change_case_char_-
UTFviii:nnnn | 29796 | _text_change_case_loop:nnw | 29796 , 30264 , 30273 , 30285 ,
30289 , 30319 , 30325 , 30332 , 30340 ,
30352 , 30458 , 30470 , 30482 , 30489 ,
30493 , 30543 , 30560 , 30579 , 30673 ,
30685 , 30697 , 30702 , 30712 , 30729 |
| _text_change_case_char_-
UTFviii:nnnNN | 29796 | _text_change_case_lower_-
az:nnnN | 30756 |
| _text_change_case_char_-
UTFviii:nnnNNN | 29796 | _text_change_case_lower_lt:nnN | 30504 |
| _text_change_case_cs_check:nnN | 29796 | _text_change_case_lower_-
lt:nnnN | 30508 |
| _text_change_case_end:w | 29796 | _text_change_case_lower_lt:nnw | 30504 |
| _text_change_case_exclude:nnN | 29796 | _text_change_case_lower_lt_-
auxi:nnnN | 30510 , 30521 |
| _text_change_case_exclude:nnNN | 29796 | _text_change_case_lower_lt_-
auxii:nnnN | 30525 , 30546 |
| _text_change_case_exclude:nnnN | 29796 | _text_change_case_lower_-
sigma:nnN | 29796 |
| _text_change_case_exclude:nnNnn | 29796 | _text_change_case_lower_-
sigma:NnnN | 29796 |
| _text_change_case_exclude:nnNw | 29796 | _text_change_case_lower_-
sigma:nnnN | 29796 , 30549 , 30675 |
| _text_change_case_group_-
lower:nnn | 29796 | _text_change_case_lower_-
sigma:nnnNN | 29796 |
| _text_change_case_group_-
title:nnn | 29796 | _text_change_case_lower_-
sigma:nnNw | 29796 |
| _text_change_case_group_-
titleonly:nnn | 29796 | | |

| | | | |
|--|---|--|---|
| <code>_text_change_case_lower-</code> | | <code>_text_change_case_upper-</code> | |
| <code>sigma:nnw</code> | 29796 | <code>el:NnnN</code> | 30243 |
| <code>_text_change_case_lower-</code> | | <code>_text_change_case_upper-</code> | |
| <code>tr:NnnN</code> | 30660 | <code>el:nnnN</code> | 30243 |
| <code>_text_change_case_lower-</code> | | <code>_text_change_case_upper-</code> | |
| <code>tr:nnnN</code> | 30660 , 30757 | <code>el:nnNw</code> | 30243 |
| <code>_text_change_case_lower-</code> | | <code>_text_change_case_upper_el-</code> | |
| <code>tr:nnnNN</code> | 30660 | <code>dialytika:N</code> | 30243 |
| <code>_text_change_case_lower-</code> | | <code>_text_change_case_upper_el-</code> | |
| <code>tr:nnNw</code> | 30660 | <code>dialytika:nnN</code> | 30243 |
| <code>_text_change_case_math-</code> | | <code>_text_change_case_upper_el-</code> | |
| <code>group:nnNn</code> | 29796 | <code>gobble:nnN</code> | 30243 |
| <code>_text_change_case_math-</code> | | <code>_text_change_case_upper_el-</code> | |
| <code>loop:nnNw</code> | 29796 | <code>gobble:nnw</code> | 30243 |
| <code>_text_change_case_math_N-</code> | | <code>_text_change_case_upper_el-</code> | |
| <code>type:nnNN</code> | 29796 | <code>hiatus:nnN</code> | 30243 |
| <code>_text_change_case_math-</code> | | <code>_text_change_case_upper_el-</code> | |
| <code>search:nnNNN</code> | 29796 | <code>hiatus:nnNw</code> | 30243 |
| <code>_text_change_case_math-</code> | | <code>_text_change_case_upper_lt:nnN</code> | |
| <code>space:nnNw</code> | 29796 | | 30582 |
| <code>_text_change_case_N_type:nnN</code> | 29796 | <code>_text_change_case_upper-</code> | |
| <code>_text_change_case_N_type:nnnN</code> | 29796 | <code>lt:nnnN</code> | 30586 |
| | 29796 | <code>_text_change_case_upper_lt:nnw</code> | |
| <code>_text_change_case_N_type-</code> | | | 30582 |
| <code>aux:nnN</code> | 29796 | <code>_text_change_case_upper_lt-</code> | |
| <code>_text_change_case_result:n</code> | 29796 | <code>aux:nnnN</code> | 30588 , 30599 |
| <code>_text_change_case_setup:NN</code> | 31282 , 31289 , 31291 | <code>_text_change_case_upper-</code> | |
| <code>_text_change_case_setup:Nn</code> | 31315 , 31335 , 31337 | <code>tr:nnnN</code> | 30734 , 30759 |
| <code>_text_change_case_space:nnw</code> | 29796 | <code>_text_change_cases_lower-</code> | |
| <code>_text_change_case_store:n</code> | 29796 , 30215 , 30237 , 30263 , 30272 , 30284 , 30288 , 30299 , 30304 , 30316 , 30480 , 30491 , 30537 , 30551 , 30576 , 30604 , 30631 , 30654 , 30671 , 30683 , 30695 , 30700 , 30711 , 30724 , 30742 , 30749 | <code>lt:nnnN</code> | 30504 |
| <code>_text_change_case_store:nw</code> | 29796 | <code>_text_change_cases_lower_lt-</code> | |
| <code>_text_change_case_title-</code> | | <code>auxi:nnnN</code> | 30504 |
| <code>el:nnnN</code> | 30497 | <code>_text_change_cases_lower_lt-</code> | |
| <code>_text_change_case_title_nl:nnN</code> | 30625 | <code>auxii:nnnN</code> | 30504 |
| | 30625 | <code>_text_change_cases_upper-</code> | |
| <code>_text_change_case_title-</code> | | <code>lt:nnnN</code> | 30582 |
| <code>nl:nnnN</code> | 30625 | <code>_text_change_cases_upper_lt-</code> | |
| <code>_text_change_case_title_nl:nnw</code> | 30625 | <code>aux:nnnN</code> | 30582 |
| | 30625 | <code>_text_char_catcode:N</code> | |
| <code>_text_change_case_upper-</code> | | | 29340 , 30048 , 30058 , 30059 , 30216 , 30309 , 30310 , 30481 , 30492 , 30539 , 30540 , 30541 , 30552 , 30577 , 30605 , 30632 , 30655 , 30672 , 30684 , 30696 , 30701 , 30745 |
| <code>az:nnnN</code> | 30756 | <code>\c_text_chardef_group_begin-</code> | |
| <code>_text_change_case_upper-</code> | | <code>token</code> | 29415 , 29497 |
| <code>de-alt:nnnN</code> | 30207 | <code>\c_text_chardef_group_end_token</code> | |
| <code>_text_change_case_upper-</code> | | | 29415 , 29505 |
| <code>de-alt:nnnNN</code> | 30207 | <code>\c_text_chardef_space_token</code> | |
| <code>_text_change_case_upper_el:nnn</code> | 30243 | | 29415 , 29484 |
| | 30243 | <code>\c_text_dotless_i_tl</code> | 30711 , 30760 |
| | | <code>\c_text_dotted_I_tl</code> | 30750 , 30760 |
| | | <code>_text_end_env:n</code> | 31598 |
| | | <code>_text_expand:n</code> | 29421 |

| | | | |
|---|---|--|---|
| <code>__text_expand_accent:N</code> | 29421 | <code>__text_if_expandable:NTF</code> | |
| <code>__text_expand_accent:NN</code> | 29421 | | 29372 , 29731 , 31527 |
| <code>__text_expand_cs:N</code> | 29421 | <code>__text_if_recursion_tail_stop:N</code> | |
| <code>__text_expand_cs_expand:N</code> | 29421 | | 31347 |
| <code>__text_expand_encoding:N</code> | 29421 | <code>__text_if_recursion_tail_stop-</code> | |
| <code>__text_expand_encoding_escape:N</code> | | do:Nn | 29254 , |
| | 29421 | | 29477 , 29539 , 29564 , 29607 , 29631 , |
| <code>__text_expand_encoding_escape:NN</code> | | | 29651 , 29890 , 29907 , 29932 , 29976 , |
| | 29709 , 29712 | | 31393 , 31405 , 31446 , 31474 , 31517 |
| <code>__text_expand_end:w</code> | 29421 | <code>__text_loop:Nn</code> 31620 , 31628 , 31632 , | |
| <code>__text_expand_exclude:N</code> | 29421 | | 31640 , 31651 , 31694 , 31699 , 31726 |
| <code>__text_expand_exclude:NN</code> | 29421 | <code>__text_loop:n</code> | |
| <code>__text_expand_exclude:nN</code> | 29421 | | 30801 , 30810 , 30846 , 31151 |
| <code>__text_expand_exclude:Nnn</code> | 29421 | <code>__text_loop:NNn</code> | 31745 , 31751 , 31753 |
| <code>__text_expand_exclude:Nw</code> | 29421 | <code>\l__text_math_mode_tl</code> | 29414 |
| <code>__text_expand_explicit:N</code> | 29421 | <code>\c__text_mathchardef_group-</code> | |
| <code>__text_expand_group:n</code> | 29421 | begin_token | 29415 , 29498 |
| <code>__text_expand_implicit:N</code> | 29421 | <code>\c__text_mathchardef_group_end-</code> | |
| <code>__text_expand_letterlike:N</code> | 29421 | token | 29415 , 29506 |
| <code>__text_expand_letterlike:NN</code> | 29421 | <code>\c__text_mathchardef_space_token</code> | |
| <code>__text_expand_loop:w</code> | 29421 | | 29415 , 29488 |
| <code>__text_expand_math_group:Nn</code> | 29421 | <code>__text_purify:n</code> | 31348 |
| <code>__text_expand_math_loop:Nw</code> | 29421 | <code>__text_purify_accent:NN</code> | 31678 |
| <code>__text_expand_math_N_type:NN</code> | 29421 | <code>__text_purify_encoding:N</code> | 31348 |
| <code>__text_expand_math_search:NNN</code> | 29421 | <code>__text_purify_encoding_escape:NN</code> | |
| <code>__text_expand_math_space:Nw</code> | 29421 | | 31348 |
| <code>__text_expand_N_type:N</code> | 29421 | <code>__text_purify_end:w</code> | 31348 |
| <code>__text_expand_N_type_auxi:N</code> | 29421 | <code>__text_purify_expand:N</code> | 31348 |
| <code>__text_expand_N_type_auxii:N</code> | 29421 | <code>__text_purify_group:n</code> | 31348 |
| <code>__text_expand_N_type_auxiii:N</code> | 29421 | <code>__text_purify_loop:w</code> | 31348 |
| <code>__text_expand_protect:N</code> | 29421 | <code>__text_purify_math_cmd:N</code> | 31348 |
| <code>__text_expand_protect:nN</code> | 29421 | <code>__text_purify_math_cmd:n</code> | |
| <code>__text_expand_protect:Nw</code> | 29421 | | 31479 , 31483 |
| <code>__text_expand_replace:N</code> | 29421 | <code>__text_purify_math_cmd:NN</code> | 31348 |
| <code>__text_expand_replace:n</code> | 29421 | <code>__text_purify_math_cmd:Nn</code> | 31348 |
| <code>__text_expand_result:n</code> | 29421 | <code>__text_purify_math_end:w</code> | 31348 |
| <code>__text_expand_space:w</code> | 29421 | <code>__text_purify_math_group:NNn</code> | 31348 |
| <code>__text_expand_store:n</code> | 29421 | <code>__text_purify_math_loop:NNw</code> | 31348 |
| <code>__text_expand_store:nw</code> | 29421 | <code>__text_purify_math_N_type:NNN</code> | 31348 |
| <code>__text_expand_testopt:N</code> | 29421 | <code>__text_purify_math_result:n</code> | |
| <code>__text_expand_testopt:NNn</code> | 29421 | | 31417 , |
| <code>__text_expand_unexpanded:N</code> | | | 31421 , 31422 , 31423 , 31428 , 31484 |
| | 1210 , 29421 | <code>__text_purify_math_search:NNN</code> | 31348 |
| <code>__text_expand_unexpanded:n</code> | 29421 | <code>__text_purify_math_space:NNw</code> | 31348 |
| <code>__text_expand_unexpanded:w</code> | 29421 | <code>__text_purify_math_start:NNw</code> | 31348 |
| <code>__text_expand_unexpanded_test:w</code> | | <code>__text_purify_math_stop:Nw</code> | |
| | 29421 | | 31428 , 31447 |
| <code>\c__text_final_sigma_tl</code> | | <code>__text_purify_math_store:n</code> | 31348 |
| | 30083 , 30098 , 30760 | <code>__text_purify_math_store:nw</code> | 31348 |
| <code>\c__text_grosses_Eszett_tl</code> | | <code>__text_purify_N_type:N</code> | 31348 |
| | 30237 , 30760 | <code>__text_purify_N_type_aux:N</code> | 31348 |
| <code>\c__text_I_ogonek_tl</code> | 30760 | <code>__text_purify_protect:N</code> | 31348 |
| <code>\c__text_i_ogonek_tl</code> | 30760 | <code>__text_purify_replace:N</code> | 31348 |
| | | <code>__text_purify_replace:n</code> | 31348 |

- _text_purify_result:n 31362, 31366, 31367, 31368
- _text_purify_space:w 31348
- _text_purify_store:n 31348
- _text_purify_store:nw 31348
- _text_quark_if_nil:nTF 29249, 29684
- _text_quark_if_nil_p:n 29249
- \c_text_sigma_tl 30097, 30760
- _text_tmp:n 31637, 31642, 31698, 31705, 31712, 31750
- _text_tmp:nnnn 30812, 30843, 30844, 31642, 31643, 31717, 31718
- _text_tmp:nnnnnn 31111, 31148, 31149
- _text_tmp:w 30765, 30768, 30784, 30787, 30788, 30789, 30790, 30791, 30792, 30793, 30806, 30828, 31053, 31056, 31074, 31080, 31081, 31082, 31083, 31084, 31085, 31086, 31087, 31088, 31089, 31090, 31093, 31105, 31108, 31109, 31110, 31130, 31250, 31253, 31272, 31278
- _text_tmp_aux:n 31714, 31717
- _text_token_to_explicit:N 29255, 31503
- _text_token_to_explicit:n .. 29255
- _text_token_to_explicit_auxi:w 29255
- _text_token_to_explicit_auxii:w 29255
- _text_token_to_explicit_auxiii:w 29255
- _text_token_to_explicit_char:N 29255
- _text_token_to_explicit_cs:N 29255
- _text_token_to_explicit_cs_aux:N 29255
- _text_use_i_delimit_by_q_recursion_stop:nw 29252, 29543, 29611, 29635, 29655, 29911, 29980, 31409, 31478
- \textbaselineshiftfactor 1158
- \textbf 31562
- \textdir 902
- \textdirection 903
- \textfont 458
- \textit 31564
- \textmd 31563
- \textnormal 31558
- \textrm 31559
- \textsc 31567
- \textsf 31560
- \textsl 31565
- \textstyle 459
- \texttt 24820, 31561
- \textulc 31568
- \textup 31566
- \TeXeTstate 568
- \tfont 1160
- \TH 29403, 31302, 31661
- \th 29403, 31302, 31674
- \the 46, 120, 121, 122, 123, 124, 125, 126, 127, 128, 129, 460
- \thickmuskip 461
- \thinmuskip 462
- \c_thirteen 36425
- \c_thirty_two 36433
- thousand commands:
 - \c_one_thousand 36441
 - \c_ten_thousand 36443
 - \c_three 36405
 - \time 463, 1308, 8994, 8996
 - \tiny 31593, 33690
- tl commands:
 - \c_catcode_active_space_tl 308, 36264
 - \c_catcode_active_tl 186, 866, 18978, 19038
 - \c_catcode_other_space_tl 183, 627, 10463, 10507, 10587, 10676, 10752, 18956
 - \c_empty_tl 118, 819, 832, 2677, 9453, 11914, 11930, 11932, 11967, 12271, 17450, 17456, 17803, 17819, 29135, 36174, 36211, 36230
 - \c_job_name_tl 36395
 - \c_novalue_tl . 105, 118, 11968, 12380
 - \c_space_tl 118, 3743, 9244, 11485, 11972, 12844, 13688, 18313, 21835, 29148, 29219, 29470, 29491, 29577, 29881, 29947, 31386, 31462, 31611, 33475, 33519, 33586, 34005, 34006, 34007, 34014, 34015, 34179, 34180, 34186, 34187, 34188, 34936, 35049, 35092, 35093, 35115, 35819, 35821
 - \tl_analysis_log:N 45, 4080
 - \tl_analysis_log:n 45, 4094
 - \tl_analysis_map_inline:Nn 45, 4053, 5923
 - \tl_analysis_map_inline:nn .. 45, 194, 514, 549, 550, 4053, 6684, 7501
 - \tl_analysis_show:N . 45, 4080, 36554
 - \tl_analysis_show:n . 45, 4094, 36556
 - \tl_build_begin:N 307, 308, 496, 1367, 5014, 5525, 6082, 6183, 6716, 6990, 36130, 36145
 - \tl_build_clear:N 307, 6751, 6923, 7917, 36145

- \tl_build_end:N 307, 308,
496, 1367, 1368, 5044, 5052, 5535,
6139, 6202, 7024, 7943, 8007, 36218
- \tl_build_gbegin:N
..... 307, 308, 36130, 36146
- \tl_build_gclear:N 307, 36145
- \tl_build_gend:N 308, 36218
- \tl_build_get:N 308
- \tl_build_get:NN 308, 6742, 36204
- \tl_build_gput_left:Nn ... 308, 36187
- \tl_build_gput_right:Nn .. 308, 36147
- \tl_build_put_left:Nn ... 308, 36187
- \tl_build_put_right:Nn 308,
526, 1369, 5021, 5039, 5047, 5051,
5115, 5118, 5158, 5172, 5176, 5301,
5315, 5356, 5378, 5391, 5423, 5436,
5440, 5522, 5528, 5534, 5538, 5581,
5871, 5875, 5882, 5888, 5909, 5925,
5943, 6153, 6198, 6211, 6785, 7013,
7078, 7147, 7204, 7207, 7221, 7289,
7921, 8023, 8026, 8034, 8037, 36147
- \tl_case:Nn 106, 12410
- \tl_case:Nnn 36557, 36559
- \tl_case:NnTF
106, 12410, 12415, 12420, 36558, 36560
- \tl_case:nnTF 706, 813, 1360
- \tl_clear:N
103, 4348, 4690, 6717, 6752, 10556,
10557, 10560, 10569, 10679, 10682,
10742, 11185, 11929, 11936, 12098,
13992, 17861, 17862, 21067, 21524,
34554, 34982, 36035, 36221, 36282
- \tl_clear_new:N
..... 103, 11425, 11426, 11427,
11428, 11429, 11935, 17865, 17866,
29771, 31539, 34563, 34596, 34637
- \tl_concat:NNN 103, 884, 11947, 13115
- \tl_const:Nn
. 103, 562, 3736, 4384, 4468, 7358,
8855, 9168, 9169, 9208, 9213, 9215,
9217, 9219, 9221, 9226, 9227, 9234,
10453, 10459, 10880, 11917, 11967,
11970, 11972, 12090, 13829, 13834,
16222, 16277, 17859, 18709, 18956,
18980, 19538, 19603, 22353, 22354,
22355, 22356, 22357, 22365, 22454,
24547, 25864, 26311, 26312, 26313,
26314, 26315, 26316, 26317, 26318,
26319, 29123, 29138, 29161, 29173,
29202, 29232, 29233, 29234, 30770,
30814, 30830, 31058, 31095, 31113,
31132, 31255, 31285, 31287, 31305,
31306, 31323, 31330, 31697, 31748,
34705, 34706, 34707, 34708, 34709,
34755, 34756, 34757, 34767, 34768,
34769, 34770, 34771, 34772, 34773,
34774, 34893, 34908, 34922, 34956,
35212, 35217, 35222, 35383, 36267
- \tl_count:N 32, 105, 108, 12530
- \tl_count:n 32, 105, 108, 372,
714, 806, 972, 1617, 1621, 2012,
2063, 5787, 7431, 7435, 7454, 7458,
7528, 7532, 12530, 12897, 12912, 12924
- \tl_count_tokens:n 108, 12543
- \tl_gclear:N
. 103, 414, 3424, 6992, 8963, 9095,
11929, 11938, 17863, 17864, 36226
- \tl_gclear_new:N
..... 103, 11935, 17867, 17868
- \tl_gconcat:NNN ... 103, 11947, 13116
- \tl_gput_left:Nn
..... 103, 11993, 14717, 14900
- \tl_gput_right:Nn 104, 1569,
1570, 7022, 7073, 7074, 7150, 8966,
9098, 12041, 16194, 16358, 28539,
29000, 31310, 31312, 35594, 35596
- \tl_gremove_all:Nn 117, 12263
- \tl_gremove_once:Nn 116, 12257
- \tl_greplace_all:NNn 116, 12189, 12266
- \tl_greplace_once:NNn
..... 116, 12189, 12260
- \tl_greverse:N 108, 12880
- .tl_gset:N 226, 21252
- \tl_gset:Nn
103, 144, 308, 665, 674, 1370, 7056,
7065, 8787, 8801, 8807, 8816, 8817,
8827, 8828, 8842, 9143, 9145, 9147,
9149, 9151, 11973, 12102, 16327,
16535, 16604, 19651, 19680, 19760
- \tl_gset_eq:NN 103, 3414,
7062, 7353, 8400, 11941, 13112,
13912, 13928, 16245, 16246, 16247,
16248, 17873, 17874, 17875, 17876,
19567, 19568, 19569, 19570, 24552
- \tl_gset_from_file:NNn 36561
- \tl_gset_from_file_x:NNn 36563
- \tl_gset_rescan:NNn 117, 12091
- .tl_gset_x:N 227, 21252
- \tl_gsort:Nn 116, 3412, 12629
- \tl_gtrim_spaces:N 109, 12573
- \tl_head:N 112, 12629, 24583
- \tl_head:n . 112, 113, 392, 688, 689,
696, 2635, 12629, 12921, 29158, 29166
- \tl_head:w 113,
689, 690, 12629, 29183, 29214, 35113
- \tl_if_blank:nTF 104, 112,
113, 3586, 6696, 8786, 9361, 10243,
10762, 10929, 10934, 10953, 10988,

- 11085, 11146, 11153, 11226, 11235,
[12302](#), [12671](#), [12911](#), [13095](#), [13122](#),
15051, 15054, 18309, 18423, 18829,
21431, 21637, 29114, 29127, 29177,
29206, 30523, 30548, 30601, 30776,
34072, 34099, 34108, 34112, 34215,
34409, 34614, 34622, 35250, 35253,
35256, 35280, 35306, 35332, 36374
- \tl_if_blank_p:n
..... [104](#), [11103](#), [12302](#), [34094](#)
- \tl_if_empty:N
..... [13199](#), [13201](#), [18114](#), [18116](#)
- \tl_if_empty:NTF
... [104](#), [7084](#), [9260](#), [9270](#), [10573](#),
[10663](#), [10698](#), [10779](#), [11022](#), [11177](#),
[11211](#), [12269](#), [21459](#), [21464](#), [21613](#),
[34044](#), [34355](#), [34626](#), [34988](#), [35009](#)
- \tl_if_empty:nTF [104](#), [491](#),
[677](#), [679](#), [680](#), [832](#), [841](#), [847](#), [1657](#),
[1751](#), [2057](#), [2614](#), [2733](#), [4281](#), [4388](#),
[4389](#), [5867](#), [7676](#), [7851](#), [8333](#), [8474](#),
[9527](#), [9619](#), [9634](#), [9722](#), [9726](#), [9784](#),
[9891](#), [9924](#), [9927](#), [9977](#), [9978](#), [9987](#),
[9994](#), [10000](#), [10007](#), [10567](#), [10814](#),
[11354](#), [11360](#), [11362](#), [11364](#), [11571](#),
[12203](#), [12279](#), [12289](#), [12361](#), [12401](#),
[12761](#), [13022](#), [13055](#), [13157](#), [13960](#),
[15015](#), [15939](#), [15946](#), [15963](#), [16113](#),
[16291](#), [17813](#), [17832](#), [17841](#), [17844](#),
[18127](#), [18160](#), [18273](#), [18359](#), [19245](#),
[19519](#), [20800](#), [20887](#), [21806](#), [23057](#),
[23910](#), [28218](#), [28286](#), [29200](#), [35581](#)
- \tl_if_empty_p:N
..... [104](#), [11488](#), [12269](#), [34399](#)
- \tl_if_empty_p:n ... [104](#), [12279](#), [12289](#)
- \tl_if_eq:NNTF
[104](#)–[106](#), [122](#), [137](#), [705](#), [788](#), [9600](#),
[9654](#), [12313](#), [12434](#), [13059](#), [16396](#),
[19811](#), [33753](#), [33756](#), [34121](#), [34358](#)
- \tl_if_eq:NnTF [104](#), [12325](#)
- \tl_if_eq:nnTF
[105](#), [123](#), [147](#), [174](#), [788](#), [12338](#), [18148](#)
- \tl_if_eq_p:NN [104](#), [12313](#)
- \tl_if_exist:N [13195](#), [13197](#)
- \tl_if_exist:NTF
..... [103](#), [4086](#), [11507](#), [11936](#),
[11938](#), [11965](#), [12523](#), [18841](#), [18852](#),
[18931](#), [18939](#), [31308](#), [34669](#), [34953](#)
- \tl_if_exist_p:N
..... [103](#), [11487](#), [11965](#), [35592](#)
- \tl_if_head_eq_catcode:nN .. [690](#), [691](#)
- \tl_if_head_eq_catcode:nNTF
..... [106](#), [12677](#)
- \tl_if_head_eq_catcode_p:nN
..... [106](#), [12677](#)
- \tl_if_head_eq_charcode:nN .. [689](#), [691](#)
- \tl_if_head_eq_charcode:nNTF ...
..... [106](#), [12677](#), [29116](#)
- \tl_if_head_eq_charcode_p:nN ...
..... [106](#), [12677](#)
- \tl_if_head_eq_meaning:nN [690](#)
- \tl_if_head_eq_meaning:nNTF
..... [106](#), [5954](#), [12677](#)
- \tl_if_head_eq_meaning_p:nN
..... [106](#), [5786](#), [12677](#)
- \tl_if_head_is_group:nTF
... [106](#), [2611](#), [2730](#), [12697](#), [12735](#),
[12768](#), [12820](#), [17839](#), [29452](#), [29556](#),
[29749](#), [29830](#), [29924](#), [31379](#), [31438](#)
- \tl_if_head_is_group_p:n .. [106](#), [12768](#)
- \tl_if_head_is_N_type:n [690](#)
- \tl_if_head_is_N_type:nTF .. [107](#),
[2608](#), [2672](#), [2718](#), [2724](#), [2769](#), [2784](#),
[2829](#), [12398](#), [12680](#), [12694](#), [12711](#),
[12748](#), [12982](#), [29449](#), [29553](#), [29753](#),
[29827](#), [29921](#), [30044](#), [30080](#), [30260](#),
[30317](#), [30338](#), [30456](#), [30477](#), [30558](#),
[30611](#), [30639](#), [30680](#), [31376](#), [31435](#)
- \tl_if_head_is_N_type_p:n [107](#), [12748](#)
- \tl_if_head_is_space:nTF
..... [107](#), [12783](#), [12964](#), [12973](#), [13682](#)
- \tl_if_head_is_space_p:n .. [107](#), [12783](#)
- \tl_if_in:Nn [841](#)
- \tl_if_in:NnTF
[105](#), [12225](#), [12351](#), [12351](#), [12352](#), [16188](#)
- \tl_if_in:nnTF
..... [105](#), [679](#), [680](#), [705](#), [4587](#),
[8869](#), [9513](#), [9515](#), [12139](#), [12209](#),
[12211](#), [12351](#), [12352](#), [12353](#), [12356](#),
[13238](#), [13246](#), [19517](#), [33566](#), [36116](#)
- \tl_if_noalue:nTF [105](#), [12367](#)
- \tl_if_noalue_p:n [105](#), [12367](#)
- \tl_if_single:NNTF
..... [105](#), [12381](#), [12382](#), [12383](#)
- \tl_if_single:nTF
..... [105](#), [566](#), [680](#), [5948](#), [5984](#),
[5999](#), [6032](#), [12382](#), [12383](#), [12384](#), [12385](#)
- \tl_if_single_p:N [105](#), [12381](#)
- \tl_if_single_p:n .. [105](#), [12381](#), [12385](#)
- \tl_if_single_token:nTF
..... [105](#), [5088](#), [12396](#), [31318](#)
- \tl_if_single_token_p:n .. [105](#), [12396](#)
- \tl_item:Nn [113](#), [12886](#)
- \tl_item:nn
[113](#), [537](#), [696](#), [7401](#), [7441](#), [12886](#), [12912](#)
- \tl_log:N [110](#), [4083](#), [13014](#), [13729](#)

- \tl_log:n 110, 376, 1044, 2163,
2179, 8438, 10158, 10372, 13016,
13049, 13725, 17762, 24577, 35816
- \tl_lower_case:n 36619
- \tl_lower_case:nn 36619
- \tl_map_break: 60,
111, 439, 452, 4076, 4077, 12448,
12462, 12477, 12488, 12503, 12514
- \tl_map_break:n 111,
112, 3419, 10941, 11041, 12514, 34487
- \tl_map_function:NN
. . . 110, 111, 305, 5911, 12443, 12538
- \tl_map_function:nN
. . 110, 2056, 5135, 12443, 12533, 16294
- \tl_map_inline:Nn
. 110, 111, 3419, 11040,
12467, 13825, 13827, 31691, 34483
- \tl_map_inline:nn
. 110, 111, 140, 452, 3381,
6290, 6744, 8718, 10456, 12467,
19475, 19477, 19479, 23963, 27051,
31543, 31554, 31571, 31602, 36653
- \tl_map_tokens:Nn . . 111, 10940, 12483
- \tl_map_tokens:nn
. 111, 7063, 12483, 12508
- \tl_map_variable:NNn 111, 12507
- \tl_map_variable:nNn 111, 684, 12507
- \tl_mixed_case:n 36619
- \tl_mixed_case:nn 36619
- \tl_new:N . 102, 103, 187, 667, 3308,
3733, 3753, 4459, 4460, 4466, 4467,
6659, 6664, 6665, 6671, 6672, 6673,
6930, 6932, 7052, 7053, 7858, 7859,
7860, 7861, 8854, 8967, 9099, 9114,
9162, 9545, 9546, 10076, 10079,
10101, 10299, 10310, 10335, 10430,
10432, 10445, 10447, 10448, 10450,
10754, 10784, 10785, 11911, 11936,
11938, 12323, 12324, 13076, 13077,
13078, 13079, 13736, 13737, 16219,
16220, 17804, 17856, 17857, 18648,
19333, 19537, 20684, 20687, 20692,
20694, 20700, 20701, 20703, 20704,
28535, 28710, 28711, 29387, 29390,
29405, 29407, 29409, 29411, 29414,
32652, 32677, 32678, 33684, 33928,
33931, 34020, 34021, 34022, 34023,
34352, 34429, 34546, 34664, 36270
- \tl_put_left:Nn 103, 11993
- \tl_put_right:Nn 104, 1368,
4230, 4232, 4307, 4317, 4356, 4376,
4697, 10704, 10707, 10712, 11058,
12041, 16356, 18046, 18679, 18681,
18684, 18686, 18687, 18689, 18691,
18693, 18694, 18696, 18698, 18700,
18702, 35145, 35191, 36309, 36314
- \tl_rand_item:N 114, 12909
- \tl_rand_item:n 114, 12909
- \tl_range:Nnn 115, 12916
- \tl_range:nnn . . . 115, 128, 307, 12916
- \tl_range_braced:Nnn 307, 36236
- \tl_range_braced:nnn 115, 307, 36236
- \tl_range_unbraced:Nnn . . . 307, 36236
- \tl_range_unbraced:nnn 115, 307, 36236
- \tl_remove_all:Nn . . . 116, 117, 12263
- \tl_remove_once:Nn 116, 12257
- \tl_replace_all:Nnn
. . 116, 786, 839, 12189, 12264, 16303
- \tl_replace_once:Nnn
. 116, 12189, 12258, 18716
- \tl_rescan:nn 117, 118, 262, 671, 12091
- \tl_reverse:N 108, 12880
- \tl_reverse:n
. 108, 109, 12861, 12881, 12883
- \tl_reverse_items:n . . 108, 109, 12557
- .tl_set:N 226, 21252
- \tl_set:Nn
103, 117, 118, 144, 227, 308, 398,
599, 665, 667, 674, 923, 1370, 4203,
5055, 5818, 5895, 6142, 6205, 6735,
6756, 6766, 6782, 6787, 6830, 6862,
6900, 7251, 7912, 7913, 7973, 8859,
8894, 9299, 9529, 9557, 9640, 10202,
10215, 10248, 10521, 10558, 10884,
10919, 11015, 11043, 11158, 11160,
11162, 11164, 11204, 11438, 11441,
11442, 11443, 11444, 11451, 11452,
11453, 11455, 11459, 11973, 12100,
12328, 12341, 12342, 12510, 13038,
13058, 13981, 14131, 16293, 16297,
16325, 16392, 16401, 16486, 16489,
16506, 16514, 16533, 16542, 16601,
16732, 17347, 17958, 17964, 17973,
17980, 18234, 18677, 19408, 19645,
19661, 19662, 19670, 19671, 19673,
19679, 19682, 19749, 19750, 19759,
19771, 19794, 19833, 20209, 20695,
20868, 21068, 21283, 21292, 21322,
21334, 21342, 21364, 21376, 21384,
21395, 21404, 21415, 21530, 24894,
28834, 28844, 29388, 29391, 29406,
29408, 29410, 29412, 29772, 31540,
32923, 33567, 33568, 33689, 33929,
33957, 34027, 34054, 34061, 34078,
34086, 34089, 34127, 34142, 34363,
34369, 34370, 34374, 34418, 34426,
34430, 34556, 34564, 34581, 34584,
34625, 34641, 34665, 34674, 34694,

- 34728, 34730, 34732, 34735, 34752,
 34946, 35134, 35961, 36036, 36281
 \tl_set_eq:NN 103, 172, 563,
 3412, 6924, 7348, 7831, 8399, 9603,
 9611, 11061, 11941, 13111, 13910,
 13919, 16241, 16242, 16243, 16244,
 17869, 17870, 17871, 17872, 19563,
 19564, 19565, 19566, 20776, 21439,
 21519, 21540, 24551, 33947, 34056,
 34141, 34627, 34647, 34670, 34989
 \tl_set_from_file:Nnn 36565
 \tl_set_from_file_x:Nnn 36567
 \tl_set_rescan:Nnn
 117, 118, 262, 636, 671, 12091
 .tl_set_x:N 227, 21252
 \tl_show:N
 109, 110, 172, 439, 4081, 13014, 13722
 \tl_show:n 83, 109,
 110, 376, 699, 1044, 1359, 2159,
 2176, 8436, 10156, 10370, 13014,
 13034, 13718, 17761, 18497, 18567,
 18573, 18579, 18585, 24575, 35814
 \tl_show_analysis:N 36553
 \tl_show_analysis:n 36555
 \tl_sort:Nn 116, 3412, 12629
 \tl_sort:nN 116, 419, 420, 3582, 12629
 \tl_tail:N 113, 731, 5761, 12629, 14126
 \tl_tail:n 113, 12629
 \tl_to_lowercase:n 36569
 \tl_to_str:N
 92, 107, 120, 530, 626, 703, 10516,
 10527, 11402, 11416, 12519, 13063,
 13064, 13165, 13230, 13238, 13721,
 13728, 14288, 18470, 18932, 18940
 \tl_to_str:n 51,
 53, 74, 92, 107, 117, 118, 120, 130,
 131, 198, 199, 222, 349, 359, 530,
 677, 681, 703, 709, 715, 868, 890,
 1442, 1465, 1556, 1561, 1642, 1727,
 2214, 2895, 2909, 2912, 2919, 2923,
 3207, 3239, 3257, 5135, 6076, 7208,
 7367, 7435, 7458, 7532, 8864, 9396,
 9397, 10039, 10040, 10041, 10042,
 10438, 10454, 10819, 10948, 10980,
 11069, 12112, 12206, 12281, 12518,
 13035, 13050, 13127, 13166, 13238,
 13246, 13391, 13413, 13437, 13444,
 13498, 13505, 13579, 13598, 13609,
 13634, 13642, 13650, 13656, 13668,
 13679, 13824, 13937, 13942, 14007,
 15027, 16172, 16183, 17622, 17639,
 17683, 17768, 18946, 18954, 19062,
 19066, 19096, 19097, 19131, 19146,
 19148, 19150, 19239, 19512, 19517,
 19632, 19691, 19773, 19796, 19819,
 19929, 20049, 20249, 20348, 21700,
 22231, 22491, 22495, 22512, 22706,
 22707, 23320, 23321, 23326, 23330,
 27974, 28028, 28102, 28609, 28908,
 29338, 30188, 30191, 33710, 33783,
 34033, 34598, 34791, 35397, 35612,
 35819, 35821, 35825, 35827, 35832,
 35834, 36111, 36344, 36366, 36369
 \tl_to_uppercase:n 36571
 \tl_trim_spaces:N 109, 12573
 \tl_trim_spaces:n 109, 687,
 946, 10867, 12573, 16282, 16284, 36374
 \tl_trim_spaces_apply:nN
 109, 919, 10864,
 12573, 17815, 18276, 18363, 18436
 \tl_upper_case:n 36619
 \tl_upper_case:nn 36619
 \tl_use:N 108,
 212, 216, 219, 8962, 9094, 12521, 20981
 \g_tmpa_tl 119, 13076
 \l_tmpa_tl 6, 58, 117, 118,
 1205, 1207, 1224, 1307, 1309, 1313,
 1315, 1319, 1321, 1325, 1327, 13078
 \g_tmpb_tl 119, 13076
 \l_tmpb_tl 118, 1206,
 1207, 1222, 1224, 1308, 1309, 1314,
 1315, 1320, 1321, 1326, 1327, 13078
 tl internal commands:
 __tl_act:NNNn
 693, 694, 12547, 12799, 12866
 __tl_act_count_group:n 12549, 12556
 __tl_act_count_group:nn 12543
 __tl_act_count_normal:N 12548, 12554
 __tl_act_count_normal:nN 12543
 __tl_act_count_space: . 12550, 12555
 __tl_act_count_space:n 12543
 __tl_act_end:wn 685, 12799
 __tl_act_group:nwNNN 12799
 __tl_act_if_head_is_space:nTF ..
 693, 12799
 __tl_act_if_head_is_space:w . 12799
 __tl_act_if_head_is_space_-
 true:w 12799
 __tl_act_loop:w 693, 12799
 __tl_act_normal:NwNNN 12799
 __tl_act_output:n 694, 12799
 __tl_act_result:n ... 694, 12834,
 12855, 12857, 12858, 12859, 12860
 __tl_act_reverse 694
 __tl_act_reverse_output:n
 12799, 12875, 12877, 12879
 __tl_act_space:wwNNN ... 693, 12799

__tl_analysis:n
 [430](#), [439](#), [3776](#), [4055](#), [4088](#), [4100](#)
 __tl_analysis_a:n [3780](#), [3805](#)
 __tl_analysis_a_bgroup:w [3836](#), [3858](#)
 __tl_analysis_a_cs:ww [3915](#)
 __tl_analysis_a_egroup:w [3838](#), [3858](#)
 __tl_analysis_a_group:nw [3858](#)
 __tl_analysis_a_group_aux:w . [3858](#)
 __tl_analysis_a_group_auxii:w [3858](#)
 __tl_analysis_a_group_test:w . [3858](#)
 __tl_analysis_a_loop:w
 .. [3812](#), [3815](#), [3856](#), [3898](#), [3912](#), [3930](#)
 __tl_analysis_a_safe:N
 [3837](#), [3879](#), [3915](#)
 __tl_analysis_a_space:w . [3835](#), [3841](#)
 __tl_analysis_a_space_test:w ...
 [432](#), [3841](#)
 __tl_analysis_a_store:
 [433](#), [3852](#), [3894](#), [3900](#)
 __tl_analysis_a_type:w .. [3816](#), [3817](#)
 __tl_analysis_b:n [3781](#), [3943](#)
 __tl_analysis_b_char:Nww [3970](#), [3976](#)
 __tl_analysis_b_cs:Nww .. [3972](#), [4000](#)
 __tl_analysis_b_cs_test:ww .. [4000](#)
 __tl_analysis_b_loop:w
 [438](#), [3943](#), [4046](#), [4051](#)
 __tl_analysis_b_normal:wwN
 [3956](#), [4021](#)
 __tl_analysis_b_normals:ww
 [437](#), [3953](#), [3956](#), [3997](#), [4007](#)
 __tl_analysis_b_special:w [3959](#), [4018](#)
 __tl_analysis_b_special_char:wN
 [4018](#)
 __tl_analysis_b_special_space:w
 [4018](#)
 __tl_analysis_char_arg:Nw
 [4179](#), [4294](#), [4353](#)
 __tl_analysis_char_arg_aux:Nw [4179](#)
 \l__tl_analysis_char_token . [427](#),
 [432](#), [433](#), [3730](#), [3845](#), [3850](#), [3888](#), [3893](#)
 __tl_analysis_cs_space_count:NN
 [3760](#), [3929](#), [4003](#)
 __tl_analysis_cs_space_count:w [3760](#)
 __tl_analysis_cs_space_count_-
 end:w [3760](#)
 __tl_analysis_disable:n
 [3785](#), [3807](#), [3873](#), [3926](#)
 __tl_analysis_extract_charcode:
 [3754](#), [3868](#), [4261](#)
 __tl_analysis_extract_charcode_-
 aux:w [3754](#)
 \l__tl_analysis_index_int
 . [434](#), [435](#), [3750](#), [3810](#), [3813](#), [3851](#),
 [3869](#), [3906](#), [3909](#), [3935](#), [3937](#), [4024](#)
 __tl_analysis_map_inline_aux:Nn
 [4053](#)
 __tl_analysis_map_inline_-
 aux:nnn [4053](#)
 \l__tl_analysis_nesting_int
 [431](#), [3751](#), [3811](#), [3902](#), [3911](#)
 \l__tl_analysis_next_token
 [427](#), [445](#), [3730](#), [4284](#), [4366](#)
 \l__tl_analysis_normal_int
 [3749](#), [3809](#), [3854](#),
 [3896](#), [3907](#), [3910](#), [3927](#), [3936](#), [3941](#)
 \g__tl_analysis_result_tl
 [439](#), [3753](#), [3945](#), [4075](#), [4105](#)
 __tl_analysis_show: [4090](#), [4101](#), [4103](#)
 __tl_analysis_show:Nn [4094](#)
 __tl_analysis_show:NNN [4080](#)
 __tl_analysis_show_active:n ...
 [4118](#), [4147](#)
 __tl_analysis_show_cs:n . [4114](#), [4147](#)
 \c__tl_analysis_show_etc_str ...
 [441](#), [4167](#), [4169](#), [4384](#)
 __tl_analysis_show_long:nn .. [4147](#)
 __tl_analysis_show_long_-
 aux:nnnn [4147](#), [4153](#)
 __tl_analysis_show_loop:wNw . [4103](#)
 __tl_analysis_show_normal:n ...
 [4121](#), [4127](#)
 __tl_analysis_show_value:N
 [4132](#), [4156](#)
 \l__tl_analysis_token
 [427](#), [428](#), [432](#), [433](#),
 [442](#), [3730](#), [3757](#), [3816](#), [3820](#), [3823](#),
 [3826](#), [3874](#), [3878](#), [3893](#), [4181](#), [4259](#),
 [4268](#), [4273](#), [4289](#), [4349](#), [4361](#), [4366](#)
 \l__tl_analysis_type_int [432](#), [434](#),
 [435](#), [3752](#), [3819](#), [3834](#), [3902](#), [3904](#), [3908](#)
 __tl_build_begin:NN .. [36130](#), [36175](#)
 __tl_build_begin:NNN .. [1368](#), [36130](#)
 __tl_build_end_loop:NN [36218](#)
 __tl_build_get:NNN
 [36204](#), [36220](#), [36225](#)
 __tl_build_get:w [36204](#)
 __tl_build_get_end:w [36204](#)
 __tl_build_last:NNn
 [1367](#), [1368](#), [36142](#), [36147](#), [36208](#)
 __tl_build_put:nn [1368](#), [36147](#), [36199](#)
 __tl_build_put:nw [1368](#), [36147](#)
 __tl_build_put_left:NNn [36187](#)
 __tl_case:NnTF
 .. [12413](#), [12418](#), [12423](#), [12428](#), [12430](#)
 __tl_case:nnTF [12410](#)
 __tl_case:Nw [12410](#)
 __tl_case_end:nw [12410](#)
 __tl_count:n [685](#), [12530](#)

- __tl_head_aux:n [12634](#), [12636](#)
- __tl_head_auxi:nw [12629](#)
- __tl_head_auxii:n [12629](#)
- __tl_head_exp_not:w [691](#), [12677](#)
- __tl_if_blank_p:NNw [12302](#)
- __tl_if_empty_if:n
 - [677](#), [678](#), [776](#), [12289](#), [12305](#),
 - [12372](#), [12399](#), [12403](#), [12656](#), [12795](#)
- __tl_if_head_eq_empty_arg:w ...
 - [689](#), [691](#), [12677](#)
- __tl_if_head_eq_meaning_-
 - normal:nN [12712](#), [12716](#)
- __tl_if_head_eq_meaning_-
 - special:nN [12713](#), [12725](#)
- __tl_if_head_is_group_fi_-
 - false:w [12768](#)
- __tl_if_head_is_N_type_auxi:w ..
 - [691](#), [12748](#)
- __tl_if_head_is_N_type_auxii:n .
 - [12763](#), [12766](#)
- __tl_if_head_is_N_type_auxiii:nn
 - [12748](#)
- __tl_if_head_is_N_type_auxiiii:n
 - [12748](#)
- __tl_if_head_is_space:w [12783](#)
- __tl_if_novalue:w [680](#), [12367](#)
- __tl_if_recursion_tail_break:nN
 - [695](#), [12088](#), [12902](#)
- __tl_if_recursion_tail_stop:nTF
 - [12088](#)
- __tl_if_recursion_tail_stop_p:n
 - [12088](#)
- __tl_if_single:nnw [680](#), [12385](#)
- \l_tl_internal_a_tl
 - [699](#), [4348](#), [4356](#), [4367](#), [4378](#), [12093](#),
 - [12095](#), [12098](#), [12323](#), [12341](#), [12345](#),
 - [13038](#), [13044](#), [13058](#), [13059](#), [13064](#)
- \l_tl_internal_b_tl
 - .. [12323](#), [12328](#), [12331](#), [12342](#), [12345](#)
- __tl_item:nn [12886](#)
- __tl_item_aux:nn [12886](#)
- __tl_map_function:Nnnnnnnnn ...
 - [683](#), [12443](#), [12472](#)
- __tl_map_function_end:w . [682](#), [12443](#)
- __tl_map_tokens:nnnnnnnnnn ... [12483](#)
- __tl_map_tokens_end:w [12483](#)
- __tl_map_variable:Nnn [12507](#)
- __tl_peek_analysis_active_str:n
 - [4186](#)
- __tl_peek_analysis_char:N ... [4186](#)
- __tl_peek_analysis_char:nN .. [4186](#)
- __tl_peek_analysis_collect:n . [4186](#)
- __tl_peek_analysis_collect:w . [4186](#)
- __tl_peek_analysis_collect_-
 - end:NNN [4186](#)
- __tl_peek_analysis_collect_-
 - loop: [4186](#)
- __tl_peek_analysis_collect_-
 - test: [4186](#)
- __tl_peek_analysis_cs: [4186](#)
- __tl_peek_analysis_escape: .. [4186](#)
- __tl_peek_analysis_explicit:n [4186](#)
- __tl_peek_analysis_loop:NNn . [4186](#)
- __tl_peek_analysis_next: [4186](#)
- __tl_peek_analysis_normal:N . [4186](#)
- __tl_peek_analysis_retest: [444](#), [4186](#)
- __tl_peek_analysis_special: . [4186](#)
- __tl_peek_analysis_str: [4186](#)
- __tl_peek_analysis_str:n [4186](#)
- __tl_peek_analysis_str:w [4186](#)
- __tl_peek_analysis_test: [4186](#)
- \c__tl_peek_catcodes_tl .. [3734](#), [4255](#)
- \l__tl_peek_charcode_int
 - [4178](#), [4260](#), [4262](#), [4265](#), [4299](#)
- \l__tl_peek_code_tl
 - [3733](#), [4203](#), [4230](#), [4232](#), [4245](#), [4254](#),
 - [4307](#), [4313](#), [4317](#), [4344](#), [4376](#), [4382](#)
- __tl_quark_if_nil:n [12089](#)
- __tl_quark_if_nil:nTF [12214](#)
- __tl_range:Nnnn . [12916](#), [36238](#), [36243](#)
- __tl_range:nnNn [12916](#)
- __tl_range:nnnNn [12916](#)
- __tl_range:w [696](#), [12916](#)
- __tl_range_braced:w [696](#), [1371](#), [36236](#)
- __tl_range_collect:nn .. [1371](#), [12916](#)
- __tl_range_collect_braced:w ...
 - [696](#), [1371](#), [36236](#)
- __tl_range_collect_group:nN . [12916](#)
- __tl_range_collect_group:nn ...
 - [12984](#), [12993](#)
- __tl_range_collect_N:nN [12916](#)
- __tl_range_collect_space:nw . [12916](#)
- __tl_range_collect_unbraced:w [36236](#)
- __tl_range_items:nnNn [696](#)
- __tl_range_normalize:nn
 - [12931](#), [12935](#), [12995](#)
- __tl_range_skip:w [696](#), [12916](#)
- __tl_range_skip_spaces:n [12916](#)
- __tl_range_unbraced:w [36236](#)
- __tl_replace:NnNNNnn [673](#),
 - [674](#), [12190](#), [12192](#), [12194](#), [12196](#), [12201](#)
- __tl_replace_auxi:NnnNNNnn
 - [675](#), [12201](#)
- __tl_replace_auxii:nNNNnn
 - [674](#), [675](#), [12201](#)
- __tl_replace_next:w
 - .. [673](#), [675](#), [676](#), [12194](#), [12196](#), [12201](#)

- __tl_replace_next_aux:w [12201](#)
- __tl_replace_wrap:w
- [673](#), [675](#), [676](#), [12190](#), [12192](#), [12201](#)
- __tl_rescan:NNw
- [670](#), [12091](#), [12175](#), [12180](#)
- __tl_rescan_aux: [12091](#)
- \c__tl_rescan_marker_tl
- [672](#), [12090](#), [12117](#), [12125](#), [12155](#), [12187](#)
- __tl_reverse_group_preserve:n
- [12868](#), [12876](#)
- __tl_reverse_group_preserve:nn
- [12861](#)
- __tl_reverse_items:nwNwn [12557](#)
- __tl_reverse_items:wn [12557](#)
- __tl_reverse_normal:N [12867](#), [12874](#)
- __tl_reverse_normal:nN [12861](#)
- __tl_reverse_space: [12869](#), [12878](#)
- __tl_reverse_space:n [12861](#)
- __tl_set_rescan:nNN
- [670](#), [672](#), [12112](#), [12134](#)
- __tl_set_rescan:NNnn [671](#), [12091](#)
- __tl_set_rescan_multi:nNN
- [670](#), [672](#), [12091](#), [12142](#), [12164](#)
- __tl_set_rescan_single:nnNN
- [672](#), [12134](#)
- __tl_set_rescan_single:NNww [672](#)
- __tl_set_rescan_single_aux:nnnNN
- [12134](#)
- __tl_set_rescan_single_aux:w
- [672](#), [12134](#)
- __tl_show:n [13034](#)
- __tl_show:NN [13014](#)
- __tl_show:w [13034](#)
- __tl_tl_head:w [12629](#), [12719](#)
- __tl_tmp:w [679](#), [686](#), [4253](#),
 [4255](#), [12360](#), [12361](#), [12367](#), [12380](#),
 [12589](#), [12628](#), [12799](#), [12814](#), [13080](#)
- __tl_trim_mark:
- [686](#), [687](#), [12576](#), [12581](#), [12589](#)
- __tl_trim_spaces:nn
- [919](#), [12575](#), [12581](#), [12589](#)
- __tl_trim_spaces_auxi:w [686](#), [12589](#)
- __tl_trim_spaces_auxii:w [687](#), [12589](#)
- __tl_trim_spaces_auxiii:w [687](#), [12589](#)
- __tl_trim_spaces_auxiv:w [687](#), [12589](#)
- __tl_use_none_delimit_by_q_act_-
 stop:w [12799](#)
- __tl_use_none_delimit_by_s_act_-
 stop:w [12833](#), [12838](#)
- __tl_use_none_delimit_by_s_-
 stop:w [682](#), [12443](#), [12495](#), [12503](#)
- token commands:
- \c_alignment_token [186](#), [864](#), [3986](#),
 [18898](#), [18960](#), [18999](#), [29345](#), [34457](#)
- \c_catcode_letter_token [186](#),
 [865](#), [3982](#), [18910](#), [18960](#), [19028](#), [29357](#)
- \c_catcode_other_token [186](#),
 [866](#), [3980](#), [18913](#), [18960](#), [19033](#), [29360](#)
- \c_group_begin_token [186](#)
- \c_group_end_token [186](#)
- \c_math_subscript_token
- [186](#), [865](#), [3990](#), [18904](#),
 [18960](#), [19018](#), [29316](#), [29351](#), [34466](#)
- \c_math_superscript_token
- [186](#), [865](#), [3988](#),
 [18901](#), [18960](#), [19013](#), [29348](#), [34467](#)
- \c_math_toggle_token
- [186](#), [864](#), [3984](#),
 [18895](#), [18960](#), [18994](#), [29313](#), [29342](#)
- \c_parameter_token
- [186](#), [523](#), [865](#), [18960](#), [19003](#), [19006](#)
- \c_space_token [39](#), [107](#), [118](#),
 [186](#), [193](#), [309](#), [690](#), [865](#), [2753](#), [3820](#),
 [3850](#), [3992](#), [4181](#), [4216](#), [4319](#), [4765](#),
 [4806](#), [7101](#), [10628](#), [12699](#), [12737](#),
 [18907](#), [18960](#), [19023](#), [19359](#), [19384](#),
 [19497](#), [29317](#), [29354](#), [29481](#), [36299](#)
- \token_case_catcode:Nn [191](#), [19290](#)
- \token_case_catcode:NnTF
- [191](#), [19290](#), [19292](#), [19294](#)
- \token_case_charcode:Nn [191](#), [19290](#)
- \token_case_charcode:NnTF
- [191](#), [19290](#), [19300](#), [19302](#)
- \token_case_meaning:Nn [191](#), [19290](#)
- \token_case_meaning:NnTF [191](#),
 [5985](#), [6000](#), [6033](#), [6043](#), [6051](#), [8444](#),
 [19290](#), [19308](#), [19310](#), [34464](#), [34471](#)
- \token_get_arg_spec:N [36573](#)
- \token_get_prefix_spec:N [36575](#)
- \token_get_replacement_spec:N [36577](#)
- \token_if_active:NTF [188](#), [19036](#), [30138](#)
- \token_if_active_p:N
- [188](#), [19036](#), [29720](#), [30094](#), [30113](#), [31492](#)
- \token_if_alignment:NTF [188](#), [18997](#)
- \token_if_alignment_p:N [188](#), [18997](#)
- \token_if_chardef:NTF [189](#), [4136](#), [19109](#)
- \token_if_chardef_p:N
- [189](#), [19109](#), [29285](#)
- \token_if_cs:NTF [189](#), [19073](#),
 [29584](#), [29955](#), [30270](#), [30324](#), [31499](#)
- \token_if_cs_p:N
- [189](#), [19073](#), [29719](#), [30346](#), [30464](#),
 [30566](#), [30619](#), [30647](#), [30692](#), [31491](#)
- \token_if_dim_register:NTF
- [190](#), [4138](#), [19109](#)
- \token_if_dim_register_p:N [190](#), [19109](#)

- \token_if_eq_catcode:NNTF
..... [188](#), [191](#), [192](#), [309](#), [2753](#),
[19046](#), [19291](#), [19293](#), [19295](#), [19297](#)
- \token_if_eq_catcode_p:NN [188](#), [19046](#)
- \token_if_eq_charcode:NNTF
.. [188](#), [191](#), [192](#), [309](#), [4806](#), [4811](#),
[5446](#), [5646](#), [5659](#), [5661](#), [5699](#), [5837](#),
[7087](#), [7101](#), [9839](#), [10628](#), [19051](#),
[19299](#), [19301](#), [19303](#), [19305](#), [26574](#)
- \token_if_eq_charcode_p:NN [188](#), [19051](#)
- \token_if_eq_meaning:NNTF
..... [189](#), [191](#),
[192](#), [309](#), [2756](#), [2767](#), [5142](#), [5149](#),
[5452](#), [5485](#), [5634](#), [5657](#), [5689](#), [5824](#),
[5832](#), [5835](#), [7156](#), [7162](#), [7189](#), [7196](#),
[7214](#), [7237](#), [7254](#), [10680](#), [19041](#),
[19307](#), [19309](#), [19311](#), [19313](#), [22827](#),
[23838](#), [23897](#), [24594](#), [24842](#), [24844](#),
[24849](#), [24913](#), [25099](#), [27171](#), [29516](#),
[29522](#), [29541](#), [29567](#), [29695](#), [29733](#),
[29909](#), [29935](#), [31407](#), [31448](#), [34485](#)
- \token_if_eq_meaning_p:NN
. [189](#), [19041](#), [29380](#), [29481](#), [29483](#),
[29487](#), [29497](#), [29498](#), [29505](#), [29506](#)
- \token_if_expandable:NNTF
..... [189](#), [4134](#), [19078](#), [29374](#)
- \token_if_expandable_p:N . [189](#), [19078](#)
- \token_if_font_selection:NNTF ...
..... [190](#), [19109](#)
- \token_if_font_selection_p:N ...
..... [190](#), [19109](#)
- \token_if_group_begin:NNTF [187](#), [18982](#)
- \token_if_group_begin_p:N [187](#), [18982](#)
- \token_if_group_end:NNTF .. [187](#), [18987](#)
- \token_if_group_end_p:N .. [187](#), [18987](#)
- \token_if_int_register:NNTF
..... [190](#), [4139](#), [19109](#)
- \token_if_int_register_p:N [190](#), [19109](#)
- \token_if_letter:N [867](#)
- \token_if_letter:NNTF
.... [188](#), [19026](#), [30057](#), [30109](#), [30488](#)
- \token_if_letter_p:N
..... [188](#), [19026](#), [30091](#), [30112](#)
- \token_if_long_macro:NNTF . [189](#), [19109](#)
- \token_if_long_macro_p:N . [189](#), [19109](#)
- \token_if_macro:NNTF
. [189](#), [2219](#), [2228](#), [2237](#), [19056](#), [19234](#)
- \token_if_macro_p:N [189](#), [19056](#)
- \token_if_math_subscript:NNTF ...
..... [188](#), [19016](#)
- \token_if_math_subscript_p:N ...
..... [188](#), [19016](#)
- \token_if_math_superscript:NNTF ..
..... [188](#), [19010](#)
- \token_if_math_superscript_p:N ..
..... [188](#), [19010](#)
- \token_if_math_toggle:NNTF [187](#), [18992](#)
- \token_if_math_toggle_p:N [187](#), [18992](#)
- \token_if_mathchardef:NNTF
..... [189](#), [4137](#), [19109](#)
- \token_if_mathchardef_p:N
..... [189](#), [19109](#), [29286](#)
- \token_if_muskip_register:NNTF ...
..... [190](#), [19109](#)
- \token_if_muskip_register_p:N ...
..... [190](#), [19109](#)
- \token_if_other:NNTF [188](#), [19031](#)
- \token_if_other_p:N [188](#), [19031](#)
- \token_if_parameter:NNTF .. [188](#), [19002](#)
- \token_if_parameter_p:N .. [188](#), [19002](#)
- \token_if_primitive:NNTF .. [190](#), [19158](#)
- \token_if_primitive_p:N .. [190](#), [19158](#)
- \token_if_protected_long_
macro:NNTF [189](#), [2650](#), [19109](#)
- \token_if_protected_long_macro_
p:N [189](#), [19109](#), [29379](#)
- \token_if_protected_macro:NNTF ...
..... [189](#), [2649](#), [19109](#)
- \token_if_protected_macro_p:N ...
..... [189](#), [19109](#), [29378](#)
- \token_if_skip_register:NNTF
..... [190](#), [4140](#), [19109](#)
- \token_if_skip_register_p:N
..... [190](#), [19109](#)
- \token_if_space:NNTF [188](#), [19021](#)
- \token_if_space_p:N [188](#), [19021](#)
- \token_if_toks_register:NNTF
..... [190](#), [397](#), [2852](#), [4141](#), [19109](#)
- \token_if_toks_register_p:N
..... [190](#), [19109](#)
- \token_new:Nn [36579](#)
- \token_to_meaning:N
..... [20](#), [187](#), [196](#), [866](#),
[870](#), [1440](#), [1456](#), [1898](#), [2222](#), [2231](#),
[2240](#), [2858](#), [2909](#), [3757](#), [4130](#), [4155](#),
[5091](#), [8451](#), [13030](#), [13069](#), [18960](#),
[19062](#), [19130](#), [19238](#), [19500](#), [29322](#)
- \token_to_str:N
..... [7](#), [21](#), [92](#), [120](#), [187](#), [196](#),
[364](#), [444](#), [445](#), [635](#), [691](#), [692](#), [811](#),
[868](#), [1005](#), [1007](#), [1442](#), [1456](#), [1456](#),
[1620](#), [1629](#), [1661](#), [1684](#), [1735](#), [1740](#),
[1755](#), [1776](#), [1777](#), [1797](#), [1898](#), [2024](#),
[2060](#), [2067](#), [2155](#), [2175](#), [2188](#), [2835](#),
[2929](#), [3014](#), [3029](#), [3044](#), [3051](#), [3077](#),
[3086](#), [3137](#), [3203](#), [3224](#), [3679](#), [3695](#),
[3743](#), [3744](#), [3745](#), [3746](#), [3765](#), [3846](#),
[3889](#), [3919](#), [3968](#), [3979](#), [3981](#), [3983](#),

- 3993, 4029, 4040, 4090, 4129, 4154,
 4236, 4291, 4311, 4320, 4385, 4708,
 4715, 4825, 4829, 5564, 7082, 7379,
 7434, 7457, 7531, 8151, 8355, 8357,
 8446, 8447, 8451, 8508, 8855, 8975,
 10167, 10169, 10381, 10383, 10500,
 10501, 10502, 10503, 10504, 10511,
 10816, 10833, 10880, 11007, 11189,
 11970, 12090, 12752, 12772, 13026,
 13030, 13063, 13069, 13351, 13798,
 13806, 13809, 16018, 16042, 16046,
 16061, 16191, 16445, 16900, 17145,
 18464, 18470, 18815, 18819, 18841,
 18844, 18852, 18855, 18931, 18932,
 18939, 18940, 18960, 19144, 19145,
 19150, 19152, 19153, 19154, 19155,
 19156, 19921, 21929, 21977, 22098,
 22140, 22248, 22490, 22505, 22712,
 22713, 23204, 23205, 23234, 23401,
 23452, 23484, 23504, 23519, 23531,
 23532, 23545, 23546, 23571, 23580,
 23582, 23607, 23610, 23635, 23637,
 23651, 23667, 23685, 23755, 23765,
 23766, 23781, 23782, 24109, 24151,
 24343, 24589, 28583, 28903, 28948,
 28954, 29296, 29297, 29716, 29724,
 29771, 29772, 30002, 30005, 30014,
 31285, 31287, 31305, 31306, 31321,
 31324, 31328, 31331, 31488, 31496,
 31539, 31540, 31681, 31684, 31688,
 31697, 31749, 32112, 32702, 32922,
 33853, 35983, 36343, 36366, 36369
- token internal commands:
- \c_token_A_int 19228, 19265
 - _token_case:NNnTF 19290
 - _token_case:NNw 19290
 - _token_case_end:nw 19290
 - _token_delimit_by_ufont:w . . . 19090
 - _token_delimit_by_char":w . . . 19090
 - _token_delimit_by_count:w . . . 19090
 - _token_delimit_by_dimen:w . . . 19090
 - _token_delimit_by_macro:w . . . 19090
 - _token_delimit_by_muskip:w . . . 19090
 - _token_delimit_by_skip:w 19090
 - _token_delimit_by_toks:w 19090
 - _token_if_macro_p:w 19056
 - _token_if_primitive:NNw 19158
 - _token_if_primitive:Nw 19158
 - _token_if_primitive_loop:N . . . 19158
 - _token_if_primitive_lua:N 19158
 - _token_if_primitive_nullfont:N 19158
 - _token_if_primitive_space:w . . . 19158
 - _token_if_primitive_undefined:N 19158
 - _token_tmp:w 868, 19091,
 19100, 19101, 19102, 19103, 19104,
 19105, 19106, 19107, 19110, 19144,
 19145, 19146, 19147, 19149, 19151,
 19152, 19153, 19154, 19155, 19156
 - \toks 464, 19156
 - \toksapp 904
 - \toksdef 465, 3675
 - \tokspre 905
 - \tolerance 466
 - \topmark 467
 - \topmarks 569
 - \topskip 468
 - \tpack 906
 - \tracingassigns 570
 - \tracingcommands 469
 - \tracingfonts 937
 - \tracinggroups 571
 - \tracingifs 572
 - \tracinglostchars 470
 - \tracingmacros 471
 - \tracingnesting 573
 - \tracingonline 472
 - \tracingoutput 473
 - \tracingpages 474
 - \tracingparagraphs 475
 - \tracingrestores 476
 - \tracingscantokens 574
 - \tracingstacklevels 1185
 - \tracingstats 477
 - true 257
 - trunc 253
 - \ttfamily 31576
 - \c_twelve 36423
 - \c_two 36403
 - \c_two_hundred_fifty_five 36437
 - \c_two_hundred_fifty_six 36439
- U
- \u . . . 29389, 31732, 31813, 31814, 31829,
 31830, 31839, 31840, 31853, 31854,
 31855, 31881, 31882, 31907, 31908
 - \uccode 478
 - \Uchar 939
 - \Ucharcat 940
 - \uchyph 479
 - \ucs 1173
 - \Udelcode 941
 - \Udelcodenum 942
 - \Udelimiter 943
 - \Udelimiterover 944
 - \Udelimiterunder 945

| | | | |
|--|------|--|------|
| <code>\Uhexensible</code> | 946 | <code>\Umathopenenclosespacing</code> | 1009 |
| <code>\Umathaccent</code> | 947 | <code>\Umathopeninnerspacing</code> | 1010 |
| <code>\Umathaxis</code> | 948 | <code>\Umathopenopenspacing</code> | 1011 |
| <code>\Umathbinbinspacing</code> | 949 | <code>\Umathopenopspacing</code> | 1012 |
| <code>\Umathbinclosespacing</code> | 950 | <code>\Umathopenordspacing</code> | 1013 |
| <code>\Umathbininnerspacing</code> | 951 | <code>\Umathopenpunctspacing</code> | 1014 |
| <code>\Umathbinopenspacing</code> | 952 | <code>\Umathopenrelspacing</code> | 1015 |
| <code>\Umathbinopspacing</code> | 953 | <code>\Umathoperatorsize</code> | 1016 |
| <code>\Umathbinordspacing</code> | 954 | <code>\Umathopinnerspacing</code> | 1017 |
| <code>\Umathbinpunctspacing</code> | 955 | <code>\Umathopopenspacing</code> | 1018 |
| <code>\Umathbinrelspacing</code> | 956 | <code>\Umathopopspacing</code> | 1019 |
| <code>\Umathchar</code> | 957 | <code>\Umathopordspacing</code> | 1020 |
| <code>\Umathcharclass</code> | 958 | <code>\Umathoppunctspacing</code> | 1021 |
| <code>\Umathchardef</code> | 959 | <code>\Umathoprelspacing</code> | 1022 |
| <code>\Umathcharfam</code> | 960 | <code>\Umathordbinspacing</code> | 1023 |
| <code>\Umathcharnum</code> | 961 | <code>\Umathordclosespacing</code> | 1024 |
| <code>\Umathcharnumdef</code> | 962 | <code>\Umathordinnerspacing</code> | 1025 |
| <code>\Umathcharslot</code> | 963 | <code>\Umathordopenspacing</code> | 1026 |
| <code>\Umathclosebinspacing</code> | 964 | <code>\Umathordopspacing</code> | 1027 |
| <code>\Umathcloseclosespacing</code> | 965 | <code>\Umathordordspacing</code> | 1028 |
| <code>\Umathcloseinnerspacing</code> | 967 | <code>\Umathordpunctspacing</code> | 1029 |
| <code>\Umathcloseopenspacing</code> | 969 | <code>\Umathordrelspacing</code> | 1030 |
| <code>\Umathcloseopspacing</code> | 970 | <code>\Umathoverbarkern</code> | 1031 |
| <code>\Umathcloseordspacing</code> | 971 | <code>\Umathoverbarrule</code> | 1032 |
| <code>\Umathclosepunctspacing</code> | 972 | <code>\Umathoverbarvgap</code> | 1033 |
| <code>\Umathclosereelspacing</code> | 974 | <code>\Umathoverdelimiterbgap</code> | 1034 |
| <code>\Umathcode</code> | 975 | <code>\Umathoverdelimitervgap</code> | 1036 |
| <code>\Umathcodenum</code> | 976 | <code>\Umathpunctbinspacing</code> | 1038 |
| <code>\Umathconnectoroverlapmin</code> | 977 | <code>\Umathpunctclosespacing</code> | 1039 |
| <code>\Umathfractiondelsize</code> | 979 | <code>\Umathpunctinnerspacing</code> | 1041 |
| <code>\Umathfractiondenomdown</code> | 980 | <code>\Umathpunctopenspacing</code> | 1043 |
| <code>\Umathfractiondenomvgap</code> | 982 | <code>\Umathpunctopspacing</code> | 1044 |
| <code>\Umathfractionnumup</code> | 984 | <code>\Umathpunctordspacing</code> | 1045 |
| <code>\Umathfractionnumvgap</code> | 985 | <code>\Umathpunctpunctspacing</code> | 1046 |
| <code>\Umathfractionrule</code> | 986 | <code>\Umathpunctrelspacing</code> | 1048 |
| <code>\Umathinnerbinspacing</code> | 987 | <code>\Umathquad</code> | 1049 |
| <code>\Umathinnerclosespacing</code> | 988 | <code>\Umathradicaldegreeafter</code> | 1050 |
| <code>\Umathinnerinnerspacing</code> | 990 | <code>\Umathradicaldegreebefore</code> | 1052 |
| <code>\Umathinneropenspacing</code> | 992 | <code>\Umathradicaldegreeraise</code> | 1054 |
| <code>\Umathinneropspacing</code> | 993 | <code>\Umathradicalkern</code> | 1056 |
| <code>\Umathinnerordspacing</code> | 994 | <code>\Umathradicalrule</code> | 1057 |
| <code>\Umathinnerpunctspacing</code> | 995 | <code>\Umathradicalvgap</code> | 1058 |
| <code>\Umathinnerrelspacing</code> | 997 | <code>\Umathrelbinspacing</code> | 1059 |
| <code>\Umathlimitabovebgap</code> | 998 | <code>\Umathrelclosespacing</code> | 1060 |
| <code>\Umathlimitabovekern</code> | 999 | <code>\Umathrelinnerspacing</code> | 1061 |
| <code>\Umathlimitabovevgap</code> | 1000 | <code>\Umathreloppspacing</code> | 1062 |
| <code>\Umathlimitbelowbgap</code> | 1001 | <code>\Umathrelopspacing</code> | 1063 |
| <code>\Umathlimitbelowkern</code> | 1002 | <code>\Umathrelordspacing</code> | 1064 |
| <code>\Umathlimitbelowvgap</code> | 1003 | <code>\Umathrelpunctspacing</code> | 1065 |
| <code>\Umathnolimitsubfactor</code> | 1004 | <code>\Umathrelrelspacing</code> | 1066 |
| <code>\Umathnolimitsupfactor</code> | 1005 | <code>\Umathskewedfractionhgap</code> | 1067 |
| <code>\Umathopbinspacing</code> | 1006 | <code>\Umathskewedfractionvgap</code> | 1069 |
| <code>\Umathopclosespacing</code> | 1007 | <code>\Umathspaceafterscript</code> | 1071 |
| <code>\Umathopenbinspacing</code> | 1008 | <code>\Umathstackdenomdown</code> | 1072 |

- `\Umathstacknumup` 1073
- `\Umathstackvgap` 1074
- `\Umathsubshiftdown` 1075
- `\Umathsubshiftdrop` 1076
- `\Umathsubsupshiftdown` 1077
- `\Umathsubsupvgap` 1078
- `\Umathsubtopmax` 1079
- `\Umathsupbottommin` 1080
- `\Umathsupshiftdrop` 1081
- `\Umathsupshiftup` 1082
- `\Umathsupsubbottommax` 1083
- `\Umathunderbarkern` 1084
- `\Umathunderbarrule` 1085
- `\Umathunderbarvgap` 1086
- `\Umathunderdelimitervgap` 1087
- `\Umathunderdelimitervgap` 1089
- undefine commands:
 - `.undefine:` 227, 21268
- `\underline` 480
- `\unexpanded` 575, 2737, 2761
- `\unhbox` 481
- `\unhcopy` 482
- `\uniformdeviate` 938
- `\unkern` 483
- `\unless` 576
- `\Unosubscript` 1091
- `\Unosuperscript` 1092
- `\unpenalty` 484
- `\unskip` 485
- `\unvbox` 486
- `\unvcopy` 487
- `\Uoverdelimiter` 1093
- `\uppercase` 488
- `\upshape` 31582
- `\uptexrevision` 1174
- `\uptexversion` 1175
- `\Uradical` 1094
- `\Uroot` 1095
- usage commands:
 - `.usage:n` 230, 21270
- use commands:
 - `\use:N` 20, 21, 169, 360, 1508, 1656, 1750, 1863, 1865, 1867, 1869, 5029, 5863, 7093, 7264, 8484, 8506, 9256, 9266, 9269, 9463, 9495, 9501, 9508, 9562, 10585, 11086, 11176, 13698, 17143, 17579, 17589, 17694, 17698, 17700, 17702, 17703, 17707, 20052, 20790, 20796, 21079, 29831, 29957, 29977, 30006, 30016, 30115, 30118, 30142, 30144, 30177, 30194, 30217, 30238, 30613, 30621, 30622, 30641, 30656, 30658, 30752, 31598, 33927, 33962, 33965, 33966, 33971, 33973, 33977, 33978, 34171, 34252, 34419, 34545, 34804, 34829, 34885, 35174, 35207, 35386, 35633
 - `\use:n` 23, 25, 102, 195, 354, 404, 410, 420, 515, 551, 552, 609, 671, 797, 944, 1005, 1262, 1509, 1515, 1517, 1520, 1589, 1606, 1632, 1692, 1701, 1712, 1713, 1721, 1967, 2044, 2196, 2211, 2466, 2890, 2939, 3076, 3107, 3193, 4144, 4566, 4763, 4788, 4990, 4993, 5133, 5663, 6090, 6156, 6224, 6676, 6731, 6788, 6848, 7045, 7989, 8048, 8694, 8701, 9393, 9410, 10028, 10218, 10407, 11035, 11520, 12293, 12302, 12497, 12498, 12501, 12653, 12668, 12759, 12793, 12815, 13162, 13237, 13245, 13335, 13356, 13370, 14053, 16144, 16722, 16723, 16724, 16725, 18254, 18255, 18258, 18696, 18974, 19056, 19093, 19112, 19229, 19896, 19897, 19898, 19899, 20246, 21278, 21329, 21371, 21390, 21595, 21629, 21650, 22750, 22758, 22767, 22784, 22792, 22820, 23285, 24834, 28905, 29001, 29192, 29193, 29195, 29879, 29944, 30783, 30804, 31072, 31104, 31270, 31570, 31608, 32107, 33131, 34105, 34241, 34607, 35153, 35268, 35321, 36660
 - `\use:nn` 23, 1520, 2292, 4112, 7938, 8889, 10914, 12124, 12185, 13380, 13854, 18880, 20048, 23315, 23324, 23328, 26754, 28628, 29275, 34933, 35824, 35826, 35831, 35833
 - `\use:nnn` 23, 1520, 2021, 18886
 - `\use:nnnn` 23, 1520
 - `\use_i:nn` ... 24, 353, 358–360, 780, 885, 893, 1131, 1134, 1147, 1151, 1152, 1363–1365, 1460, 1524, 1574, 1650, 1672, 1714, 1813, 1841, 2000, 2748, 2778, 2791, 2836, 3110, 3181, 3503, 3558, 3568, 3578, 3921, 4438, 4950, 4961, 4970, 4973, 4982, 5869, 8696, 12188, 14354, 14359, 14440, 14444, 16133, 16325, 16327, 16709, 16760, 16797, 16895, 19638, 19825, 19883, 19916, 22092, 22094, 22471, 23095, 23285, 24662, 24998, 25293, 25781, 26064, 26583, 26749, 27061, 27071, 27075, 27583, 27788, 28349, 28374, 33943, 34489, 35948, 35950
 - `\use_i:nnn` 24, 737, 1526, 2222, 3215, 4761, 4786, 7166, 13840, 14362, 14868, 16574,

- 17755, 23254, 25250, 26724, 28562
 \backslash use_i:nnnn 24, 343, 566–568, 1526,
 8479, 8481, 8495, 8500, 8516, 8518,
 24832, 25268, 25275, 25468, 28360
 \backslash use_i_delimit_by_q_nil:nw . 25, 1538
 \backslash use_i_delimit_by_q_recursion_-
 stop:nw 25, 1538, 15932, 15948
 \backslash use_i_delimit_by_q_recursion_-
 stop:w 139
 \backslash use_i_delimit_by_q_stop:nw 25, 1538
 \backslash use_i_ii:nnn . 24, 359, 360, 1526,
 1641, 2318, 16162, 16550, 16655, 19842
 \backslash use_ii:nn
 . . 24, 70, 353, 358, 491, 498, 780,
 885, 1131, 1134, 1147, 1151, 1152,
 1164, 1363–1365, 697, 702, 1462,
 1524, 1576, 1674, 1709, 1714, 1815,
 1843, 1998, 2246, 2750, 2793, 2838,
 3923, 4440, 4800, 4952, 4958, 4963,
 4975, 4984, 5144, 5493, 5615, 5792,
 5875, 6175, 7436, 7459, 7533, 8702,
 12137, 12659, 12728, 12805, 12811,
 16135, 19639, 22672, 22695, 23097,
 24461, 24662, 24663, 25295, 26585,
 27067, 27073, 27077, 27585, 27790,
 28220, 28351, 33944, 35954, 35956
 \backslash use_ii:nnn
 24, 360, 1526, 2231, 4798, 5098, 12657
 \backslash use_ii:nnnn . 24, 566, 568, 1526, 8495
 \backslash use_ii_i:nn 24,
 726, 1534, 13859, 13944, 18276, 18363
 \backslash use_iii:nnn 24, 1526, 2240, 2251, 22477
 \backslash use_iii:nnnn 24,
 566, 568, 1526, 8495, 8517, 8519, 8520
 \backslash use_iv:nnnn
 24, 566, 568, 1526, 8495, 8515, 24449
 \backslash use_none:n 25, 439,
 592, 626, 678, 745, 780, 836, 840,
 888, 1002, 1003, 1006, 1325, 698,
 704, 1542, 1640, 1692, 1693, 1712,
 1713, 1969, 2025, 2191, 2638, 2769,
 3148, 3149, 3919, 3968, 4066, 4111,
 4236, 4568, 4838, 4996, 5660, 5955,
 6066, 6800, 8695, 8700, 8984, 9325,
 9528, 9722, 9726, 10558, 10613,
 10669, 10971, 11376, 11379, 12171,
 12248, 12305, 12399, 12639, 12656,
 12673, 12732, 12767, 12771, 12796,
 12973, 12982, 13795, 13818, 13834,
 13846, 13879, 14012, 14044, 14298,
 14308, 14346, 14408, 14572, 14656,
 14761, 14933, 15934, 15949, 16073,
 16089, 16159, 16217, 16559, 17449,
 17455, 17745, 17844, 17888, 17985,
 18086, 18113, 18152, 19278, 19721,
 19738, 20882, 21572, 22192, 22207,
 22466, 22615, 22619, 22623, 22627,
 23912, 24161, 24168, 24185, 24204,
 24227, 24295, 24336, 24461, 24476,
 24497, 24498, 24724, 24725, 25269,
 25272, 26252, 27944, 28229, 31550,
 31605, 34477, 35400, 35916, 35917
 \backslash use_none:nn 25,
 676, 680, 794, 1364, 1542, 1622,
 1630, 3247, 6468, 7179, 8474, 10614,
 10658, 12233, 12389, 12572, 12762,
 13903, 16397, 16581, 17813, 18127,
 18273, 18359, 22531, 22614, 22618,
 22622, 22626, 27939, 31551, 35963
 \backslash use_none:nnn
 25, 690, 1542, 3015, 3030,
 4142, 7709, 7924, 10615, 12719,
 16062, 21019, 21028, 22613, 22617,
 22621, 22625, 23254, 31597, 34689
 \backslash use_none:nnnn
 . 25, 1542, 10616, 20179, 31553, 34996
 \backslash use_none:nnnnn 25,
 356, 628, 986, 1542, 10617, 10627,
 22745, 22779, 22805, 22813, 24858
 \backslash use_none:nnnnnn
 25, 1542, 1757, 10618, 36231
 \backslash use_none:nnnnnnn
 25, 986, 1542, 22747,
 22781, 22807, 22815, 23138, 25309
 \backslash use_none:nnnnnnnn
 25, 360, 1542, 1663, 3096
 \backslash use_none:nnnnnnnnn 25, 1542
 \backslash use_none_delimit_by_q_nil:w 25, 1535
 \backslash use_none_delimit_by_q_recursion_-
 stop:w
 25, 139, 358, 1535, 15926, 15941
 \backslash use_none_delimit_by_q_stop:w . . .
 25, 745, 782, 1535
 \backslash use_none_delimit_by_s_stop:w . . .
 141, 16207
 \backslash useboxresource 931
 \backslash usefont 31553
 \backslash useimageresource 932
 \backslash Uskewed 1096
 \backslash Uskewedwithdelims 1097
 \backslash Ustack 1098
 \backslash Ustartdisplaymath 1099
 \backslash Ustartmath 1100
 \backslash Ustopdisplaymath 1101
 \backslash Ustopmath 1102
 \backslash Usubscript 1103
 \backslash Usuperscript 1104
 \backslash Underdelimiter 1105

| | | | |
|---|--|--|---------------------|
| <code>\Uvextensible</code> | 1106 | <code>\voffset</code> | 498 |
| V | | | |
| <code>\v</code> | 29389, 31737,
31823, 31824, 31825, 31826, 31835,
31836, 31869, 31870, 31877, 31878,
31889, 31890, 31897, 31898, 31901,
31902, 31924, 31925, 31926, 31927,
31928, 31929, 31930, 31931, 31932,
31933, 31934, 31935, 31936, 31937,
31938, 31941, 31942, 31951, 31952 | <code>\vpack</code> | 907 |
| <code>\vadjust</code> | 489 | <code>\vrule</code> | 499 |
| <code>\valign</code> | 490 | <code>\vsize</code> | 500 |
| value commands: | | <code>\vskip</code> | 501 |
| <code>.value_forbidden:n</code> | 227, 21272 | <code>\vsplit</code> | 502 |
| <code>.value_required:n</code> | 227, 21272 | <code>\vss</code> | 503 |
| <code>\vbadness</code> | 491 | <code>\vtop</code> | 504 |
| <code>\vbox</code> | 1327, 492 | W | |
| vbox commands: | | <code>\wd</code> | 505 |
| <code>\vbox:n</code> | 271, 276, 32196 | <code>\widowpenalties</code> | 577 |
| <code>\vbox_gset:Nn</code> | 276, 32210, 32771 | <code>\widowpenalty</code> | 506 |
| <code>\vbox_gset:Nw</code> | 276, 32246, 32846 | <code>\write</code> | 89, 507 |
| <code>\vbox_gset_end:</code> | 276, 32246, 32848 | X | |
| <code>\vbox_gset_split_to_ht:Nnn</code> | 277, 32285 | <code>\xdef</code> | 508 |
| <code>\vbox_gset_to_ht:Nnn</code> | 276, 32234 | xetex commands: | |
| <code>\vbox_gset_to_ht:Nnw</code> | 277, 32267 | <code>\xetex_if_engine:TF</code> | 36585, 36587, 36589 |
| <code>\vbox_gset_top:Nn</code> | 276, 32222 | <code>\xetex_if_engine_p:</code> | 36583 |
| <code>\vbox_set:Nn</code> | 276, 32210, 32765 | <code>\XeTeXcharclass</code> | 715 |
| <code>\vbox_set:Nw</code> | 276, 32246, 32839 | <code>\XeTeXcharglyph</code> | 716 |
| <code>\vbox_set_end:</code> | 276, 277, 32246, 32841 | <code>\XeTeXcountfeatures</code> | 717 |
| <code>\vbox_set_split_to_ht:Nnn</code> | 277, 32285 | <code>\XeTeXcountglyphs</code> | 718 |
| <code>\vbox_set_to_ht:Nnn</code> | 276, 277, 32234 | <code>\XeTeXcountselectors</code> | 719 |
| <code>\vbox_set_to_ht:Nnw</code> | 277, 32267 | <code>\XeTeXcountvariations</code> | 720 |
| <code>\vbox_set_top:Nn</code> | 276, 32222, 32785, 32862 | <code>\XeTeXdashbreakstate</code> | 722 |
| <code>\vbox_to_ht:nn</code> | 276, 32200 | <code>\XeTeXdefaultencoding</code> | 721 |
| <code>\vbox_to_zero:n</code> | 276, 32200 | <code>\XeTeXfeaturecode</code> | 723 |
| <code>\vbox_top:n</code> | 276, 32196 | <code>\XeTeXfeaturename</code> | 724 |
| <code>\vbox_unpack:N</code> | 277, 32281, 32785, 32862 | <code>\XeTeXfindfeaturebyname</code> | 725 |
| <code>\vbox_unpack_clear:N</code> | 36581 | <code>\XeTeXfindselectorbyname</code> | 727 |
| <code>\vbox_unpack_drop:N</code> | 278, 32281, 36582 | <code>\XeTeXfindvariationbyname</code> | 729 |
| <code>\vcenter</code> | 493 | <code>\XeTeXfirstfontchar</code> | 731 |
| vcoffin commands: | | <code>\XeTeXfonttype</code> | 732 |
| <code>\vcoffin_gset:Nnn</code> | 283, 32762 | <code>\XeTeXgenerateactualtext</code> | 733 |
| <code>\vcoffin_gset:Nnw</code> | 283, 32837 | <code>\XeTeXglyph</code> | 735 |
| <code>\vcoffin_gset_end:</code> | 283, 32837 | <code>\XeTeXglyphbounds</code> | 736 |
| <code>\vcoffin_set:Nnn</code> | 283, 32762 | <code>\XeTeXglyphindex</code> | 737 |
| <code>\vcoffin_set:Nnw</code> | 283, 32837 | <code>\XeTeXglyphname</code> | 738 |
| <code>\vcoffin_set_end:</code> | 283, 32837 | <code>\XeTeXinputencoding</code> | 739 |
| <code>\vfi</code> | 1165 | <code>\XeTeXinputnormalization</code> | 740 |
| <code>\vfil</code> | 494 | <code>\XeTeXinterchartokenstate</code> | 742 |
| <code>\vfill</code> | 495 | <code>\XeTeXinterchartoks</code> | 744 |
| <code>\vfilneg</code> | 496 | <code>\XeTeXisdefaultselector</code> | 745 |
| <code>\vfuzz</code> | 497 | <code>\XeTeXisexclusivefeature</code> | 747 |
| | | <code>\XeTeXlastfontchar</code> | 749 |
| | | <code>\XeTeXlinebreaklocale</code> | 751 |
| | | <code>\XeTeXlinebreakpenalty</code> | 752 |
| | | <code>\XeTeXlinebreakskip</code> | 750 |
| | | <code>\XeTeXOTcountfeatures</code> | 753 |
| | | <code>\XeTeXOTcountlanguages</code> | 754 |
| | | <code>\XeTeXOTcountscripts</code> | 755 |

| | | | |
|------------------------------|-----|---------------------------|-----------------|
| \XeTeXOTfeaturetag | 756 | \XeTeXvariationmin | 770 |
| \XeTeXOTlanguagetag | 757 | \XeTeXvariationname | 771 |
| \XeTeXOTscripttag | 758 | \XeTeXversion | 772 |
| \XeTeXpdf file | 759 | \xkanjiskip | 1161 |
| \XeTeXpdfpagecount | 760 | \xleaders | 509 |
| \XeTeXpicfile | 761 | \xspaceskip | 510 |
| \XeTeXrevision | 762 | \xspcode | 1162 |
| \XeTeXselectorname | 763 | | |
| \XeTeXtracingfonts | 764 | Y | |
| \XeTeXupwardsmode | 765 | \ybaselineshift | 1163 |
| \XeTeXuseglyphmetrics | 766 | \year | 511, 1326, 8999 |
| \XeTeXvariation | 767 | \yoko | 1164 |
| \XeTeXvariationdefault | 768 | | |
| \XeTeXvariationmax | 769 | Z | |
| | | \c_zero | 36399 |