

The L^AT_EX3 Sources

The L^AT_EX3 Project*

May 29, 2017

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.

At present, the `expl3` modules are designed to be loaded on top of L^AT_EX 2 ε . In time, a L^AT_EX3 format will be produced based on this code. This allows the code to be used in L^AT_EX 2 ε packages *now* while a stand-alone L^AT_EX3 is developed.

While `expl3` is still experimental, the bundle is now regarded as broadly stable. The syntax conventions and functions provided are now ready for wider use. There may still be changes to some functions, but these will be minor when compared to the scope of `expl3`.

New modules will be added to the distributed version of `expl3` as they reach maturity.

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

Contents

I	Introduction to <code>expl3</code> and this document	1
1	Naming functions and variables	1
1.1	Terminological inexactitude	3
2	Documentation conventions	3
3	Formal language conventions which apply generally	5
4	<code>TeX</code> concepts not supported by <code>LATEX3</code>	5
II	The <code>l3bootstrap</code> package: Bootstrap code	6
1	Using the <code>LATEX3</code> modules	6
1.1	Internal functions and variables	7
III	The <code>l3names</code> package: Namespace for primitives	8
1	Setting up the <code>LATEX3</code> programming language	8
IV	The <code>l3basics</code> package: Basic definitions	9
1	No operation functions	9
2	Grouping material	9
3	Control sequences and functions	10
3.1	Defining functions	10
3.2	Defining new functions using parameter text	11
3.3	Defining new functions using the signature	12
3.4	Copying control sequences	15
3.5	Deleting control sequences	15
3.6	Showing control sequences	15
3.7	Converting to and from control sequences	16
4	Using or removing tokens and arguments	17
4.1	Selecting tokens from delimited arguments	19
5	Predicates and conditionals	19
5.1	Tests on control sequences	20
5.2	Primitive conditionals	21
6	Internal kernel functions	22
V	The <code>l3expn</code> package: Argument expansion	24

1	Defining new variants	24
2	Methods for defining variants	25
3	Introducing the variants	25
4	Manipulating the first argument	26
5	Manipulating two arguments	28
6	Manipulating three arguments	28
7	Unbraced expansion	29
8	Preventing expansion	30
9	Controlled expansion	31
10	Internal functions and variables	32
VI	The <code>I3tl</code> package: Token lists	34
1	Creating and initialising token list variables	34
2	Adding data to token list variables	35
3	Modifying token list variables	36
4	Reassigning token list category codes	36
5	Token list conditionals	37
6	Mapping to token lists	39
7	Using token lists	41
8	Working with the content of token lists	41
9	The first token from a token list	43
10	Using a single item	45
11	Viewing token lists	45
12	Constant token lists	46
13	Scratch token lists	46
14	Internal functions	46
VII	The <code>I3str</code> package: Strings	47

1	Building strings	47
2	Adding data to string variables	48
2.1	String conditionals	48
3	Working with the content of strings	50
4	String manipulation	53
5	Viewing strings	54
6	Constant token lists	55
7	Scratch strings	55
7.1	Internal string functions	55
VIII The l3seq package: Sequences and stacks		57
1	Creating and initialising sequences	57
2	Appending data to sequences	58
3	Recovering items from sequences	58
4	Recovering values from sequences with branching	59
5	Modifying sequences	60
6	Sequence conditionals	61
7	Mapping to sequences	61
8	Using the content of sequences directly	63
9	Sequences as stacks	64
10	Sequences as sets	65
11	Constant and scratch sequences	66
12	Viewing sequences	67
13	Internal sequence functions	67
IX The l3int package: Integers		68
1	Integer expressions	68
2	Creating and initialising integers	69
3	Setting and incrementing integers	70

4	Using integers	70
5	Integer expression conditionals	71
6	Integer expression loops	72
7	Integer step functions	74
8	Formatting integers	74
9	Converting from other formats to integers	76
10	Viewing integers	77
11	Constant integers	78
12	Scratch integers	78
13	Primitive conditionals	79
14	Internal functions	79
X	The <code>I3intarray</code> package: low-level arrays of small integers	81
1	<code>I3intarray</code> documentation	81
	1.1 Internal functions	81
XI	The <code>I3flag</code> package: expandable flags	82
1	Setting up flags	82
2	Expandable flag commands	83
XII	The <code>I3quark</code> package: Quarks	84
1	Introduction to quarks and scan marks	84
	1.1 Quarks	84
2	Defining quarks	84
3	Quark tests	85
4	Recursion	85
5	An example of recursion with quarks	86
6	Internal quark functions	87
7	Scan marks	87

XIII The <code>\3prg</code> package: Control structures	89
1 Defining a set of conditional functions	89
2 The boolean data type	91
3 Boolean expressions	93
4 Logical loops	95
5 Producing multiple copies	96
6 Detecting <code>T_EX</code> 's mode	96
7 Primitive conditionals	96
8 Internal programming functions	97
XIV The <code>\3clist</code> package: Comma separated lists	98
1 Creating and initialising comma lists	98
2 Adding data to comma lists	99
3 Modifying comma lists	100
4 Comma list conditionals	101
5 Mapping to comma lists	101
6 Using the content of comma lists directly	103
7 Comma lists as stacks	104
8 Using a single item	105
9 Viewing comma lists	105
10 Constant and scratch comma lists	106
XV The <code>\3token</code> package: Token manipulation	107
1 Creating character tokens	107
2 Manipulating and interrogating character tokens	108
3 Generic tokens	111
4 Converting tokens	112
5 Token conditionals	112

6	Peeking ahead at the next token	115
7	Decomposing a macro definition	118
8	Description of all possible tokens	119
9	Internal functions	121
	XVI The l3prop package: Property lists	122
1	Creating and initialising property lists	122
2	Adding entries to property lists	123
3	Recovering values from property lists	123
4	Modifying property lists	124
5	Property list conditionals	124
6	Recovering values from property lists with branching	125
7	Mapping to property lists	125
8	Viewing property lists	126
9	Scratch property lists	127
10	Constants	127
11	Internal property list functions	127
	XVII The l3msg package: Messages	128
1	Creating new messages	128
2	Contextual information for messages	129
3	Issuing messages	130
4	Redirecting messages	132
5	Low-level message functions	133
6	Kernel-specific functions	134
7	Expandable errors	136
8	Internal l3msg functions	136
	XVIII The l3file package: File and I/O operations	138

1	File operation functions	138
1.1	Input–output stream management	139
1.2	Reading from files	140
2	Writing to files	142
2.1	Wrapping lines in output	144
2.2	Constant input–output streams	145
2.3	Primitive conditionals	145
2.4	Internal file functions and variables	145
2.5	Internal input–output functions	145
XIX The <code>l3skip</code> package: Dimensions and skips		147
1	Creating and initialising <code>dim</code> variables	147
2	Setting <code>dim</code> variables	148
3	Utilities for dimension calculations	148
4	Dimension expression conditionals	149
5	Dimension expression loops	151
6	Using <code>dim</code> expressions and variables	152
7	Viewing <code>dim</code> variables	154
8	Constant dimensions	154
9	Scratch dimensions	154
10	Creating and initialising <code>skip</code> variables	155
11	Setting <code>skip</code> variables	155
12	Skip expression conditionals	156
13	Using <code>skip</code> expressions and variables	156
14	Viewing <code>skip</code> variables	156
15	Constant skips	157
16	Scratch skips	157
17	Inserting skips into the output	157
18	Creating and initialising <code>muskip</code> variables	158
19	Setting <code>muskip</code> variables	158
20	Using <code>muskip</code> expressions and variables	159

21	Viewing <code>muskip</code> variables	159
22	Constant muskips	160
23	Scratch muskips	160
24	Primitive conditional	160
25	Internal functions	161
XX	The <code>l3keys</code> package: Key–value interfaces	162
1	Creating keys	163
2	Sub-dividing keys	167
3	Choice and multiple choice keys	167
4	Setting keys	170
5	Handling of unknown keys	170
6	Selective key setting	171
7	Utility functions for keys	172
8	Low-level interface for parsing key–val lists	173
XXI	The <code>l3fp</code> package: floating points	175
1	Creating and initialising floating point variables	176
2	Setting floating point variables	176
3	Using floating point numbers	177
4	Floating point conditionals	178
5	Floating point expression loops	180
6	Some useful constants, and scratch variables	181
7	Floating point exceptions	182
8	Viewing floating points	183
9	Floating point expressions	183
9.1	Input of floating point numbers	183
9.2	Precedence of operators	184
9.3	Operations	185

10	Disclaimer and roadmap	191
XXII	The <i>I3sort</i> package: Sorting functions	194
1	Controlling sorting	194
XXIII	The <i>I3tl-analysis</i> package: analysing token lists	195
1	<i>I3tl-analysis</i> documentation	195
XXIV	The <i>I3tl-build</i> package: building token lists	196
1	<i>I3tl-build</i> documentation	196
	1.1 Internal functions	196
XXV	The <i>I3regex</i> package: regular expressions in <i>T_EX</i>	197
1	Regular expressions	197
	1.1 Syntax of regular expressions	197
	1.2 Syntax of the replacement text	201
	1.3 Pre-compiling regular expressions	203
	1.4 Matching	203
	1.5 Submatch extraction	204
	1.6 Replacement	205
	1.7 Bugs, misfeatures, future work, and other possibilities	205
XXVI	The <i>I3box</i> package: Boxes	208
1	Creating and initialising boxes	208
2	Using boxes	209
3	Measuring and setting box dimensions	209
4	Box conditionals	210
5	The last box inserted	210
6	Constant boxes	211
7	Scratch boxes	211
8	Viewing box contents	211
9	Boxes and color	211
10	Horizontal mode boxes	212

11	Vertical mode boxes	213
11.1	Affine transformations	214
12	Primitive box conditionals	217
XXVII The <code>I3coffins</code> package: Coffin code layer		218
1	Creating and initialising coffins	218
2	Setting coffin content and poles	218
3	Joining and using coffins	219
4	Measuring coffins	220
5	Coffin diagnostics	220
5.1	Constants and variables	221
XXVIII The <code>I3color</code> package: Color support		222
1	Color in boxes	222
XXIX The <code>I3sys</code> package: System/runtime functions		223
1	The name of the job	223
2	Date and time	223
3	Engine	223
4	Output format	224
XXX The <code>I3deprecation</code> package: Deprecation errors		225
1	<code>I3deprecation</code> documentation	225
XXXI The <code>I3candidates</code> package: Experimental additions to <code>I3kernel</code>		226
1	Important notice	226
2	Additions to <code>I3box</code>	226
2.1	Viewing part of a box	226
3	Additions to <code>I3clist</code>	227
4	Additions to <code>I3coffins</code>	227

5	Additions to <code>I3file</code>	228
6	Additions to <code>I3int</code>	228
7	Additions to <code>I3msg</code>	228
8	Additions to <code>I3prop</code>	229
9	Additions to <code>I3seq</code>	230
10	Additions to <code>I3skip</code>	230
11	Additions to <code>I3sys</code>	231
12	Additions to <code>I3tl</code>	232
13	Additions to <code>I3tokens</code>	237
 XXXII The <code>I3luatex</code> package:LuaTeX-specific functions		238
1	Breaking out to Lua	238
1.1	T _E X code interfaces	238
1.2	Lua interfaces	239
 XXXIII The <code>I3drivers</code> package: Drivers		240
1	Box clipping	240
2	Box rotation and scaling	240
3	Color support	241
4	Drawing	241
4.1	Path construction	242
4.2	Stroking and filling	242
4.3	Stroke options	243
4.4	Color	244
4.5	Inserting T _E X material	245
4.6	Coordinate system transformations	245
 XXXIV Implementation		245

1	I3bootstrap implementation	245
1.1	Format-specific code	245
1.2	The <code>\pdfstrcmp</code> primitive in X _E T _E X	246
1.3	Loading support Lua code	246
1.4	Engine requirements	247
1.5	Extending allocators	249
1.6	Character data	249
1.7	The L _A T _E X3 code environment	251
2	I3names implementation	252
3	I3basics implementation	273
3.1	Renaming some T _E X primitives (again)	274
3.2	Defining some constants	276
3.3	Defining functions	276
3.4	Selecting tokens	277
3.5	Gobbling tokens from input	278
3.6	Conditional processing and definitions	278
3.7	Dissecting a control sequence	284
3.8	Exist or free	286
3.9	Defining and checking (new) functions	287
3.10	More new definitions	290
3.11	Copying definitions	292
3.12	Undefining functions	292
3.13	Generating parameter text from argument count	292
3.14	Defining functions from a given number of arguments	293
3.15	Using the signature to define functions	294
3.16	Checking control sequence equality	296
3.17	Diagnostic functions	297
3.18	Doing nothing functions	298
3.19	Breaking out of mapping functions	298
4	I3expan implementation	299
4.1	General expansion	299
4.2	Hand-tuned definitions	302
4.3	Definitions with the automated technique	304
4.4	Last-unbraced versions	305
4.5	Preventing expansion	307
4.6	Controlled expansion	307
4.7	Defining function variants	308

5	l3tl implementation	314
5.1	Functions	314
5.2	Constant token lists	316
5.3	Adding to token list variables	316
5.4	Reassigning token list category codes	319
5.5	Modifying token list variables	322
5.6	Token list conditionals	326
5.7	Mapping to token lists	330
5.8	Using token lists	331
5.9	Working with the contents of token lists	332
5.10	Token by token changes	334
5.11	The first token from a token list	336
5.12	Using a single item	341
5.13	Viewing token lists	341
5.14	Scratch token lists	342
5.15	Deprecated functions	342
6	l3str implementation	343
6.1	Creating and setting string variables	343
6.2	String comparisons	344
6.3	Accessing specific characters in a string	347
6.4	Counting characters	351
6.5	The first character in a string	353
6.6	String manipulation	354
6.7	Viewing strings	356
6.8	Unicode data for case changing	356
7	l3seq implementation	359
7.1	Allocation and initialisation	360
7.2	Appending data to either end	363
7.3	Modifying sequences	364
7.4	Sequence conditionals	366
7.5	Recovering data from sequences	367
7.6	Mapping to sequences	371
7.7	Using sequences	373
7.8	Sequence stacks	373
7.9	Viewing sequences	374
7.10	Scratch sequences	375

8	<code>I3int</code> implementation	375
8.1	Integer expressions	376
8.2	Creating and initialising integers	377
8.3	Setting and incrementing integers	379
8.4	Using integers	380
8.5	Integer expression conditionals	380
8.6	Integer expression loops	384
8.7	Integer step functions	385
8.8	Formatting integers	386
8.9	Converting from other formats to integers	392
8.10	Viewing integer	395
8.11	Constant integers	395
8.12	Scratch integers	397
8.13	Deprecated	397
9	<code>I3intarray</code> implementation	397
9.1	Allocating arrays	397
9.2	Array items	398
10	<code>I3flag</code> implementation	399
10.1	Non-expandable flag commands	399
10.2	Expandable flag commands	400
10.3	Option check-declarations	401
11	<code>I3quark</code> implementation	401
11.1	Quarks	402
11.2	Scan marks	405
12	<code>I3prg</code> implementation	405
12.1	Primitive conditionals	405
12.2	Defining a set of conditional functions	406
12.3	The boolean data type	406
12.4	Boolean expressions	409
12.5	Logical loops	415
12.6	Producing multiple copies	416
12.7	Detecting <code>TEx</code> 's mode	417
12.8	Internal programming functions	418
13	<code>I3clist</code> implementation	418
13.1	Allocation and initialisation	419
13.2	Removing spaces around items	421
13.3	Adding data to comma lists	422
13.4	Comma lists as stacks	422
13.5	Modifying comma lists	424
13.6	Comma list conditionals	427
13.7	Mapping to comma lists	428
13.8	Using comma lists	431
13.9	Using a single item	432
13.10	Viewing comma lists	433
13.11	Scratch comma lists	434

14	I3token implementation	434
14.1	Manipulating and interrogating character tokens	434
14.2	Creating character tokens	437
14.3	Generic tokens	441
14.4	Token conditionals	442
14.5	Peeking ahead at the next token	449
14.6	Decomposing a macro definition	454
15	I3prop implementation	455
15.1	Allocation and initialisation	456
15.2	Accessing data in property lists	457
15.3	Property list conditionals	460
15.4	Recovering values from property lists with branching	462
15.5	Mapping to property lists	462
15.6	Viewing property lists	463
16	I3msg implementation	464
16.1	Creating messages	464
16.2	Messages: support functions and text	466
16.3	Showing messages: low level mechanism	467
16.4	Displaying messages	469
16.5	Kernel-specific functions	476
16.6	Expandable errors	482
16.7	Showing variables	483
17	I3file implementation	486
17.1	File operations	486
17.2	Input operations	491
17.2.1	Variables and constants	491
17.2.2	Stream management	492
17.2.3	Reading input	495
17.3	Output operations	497
17.3.1	Variables and constants	497
17.4	Stream management	498
17.4.1	Deferred writing	499
17.4.2	Immediate writing	500
17.4.3	Special characters for writing	500
17.4.4	Hard-wrapping lines to a character count	501
17.5	Messages	509
17.6	Deprecated functions	510

18	I3skip implementation	510
18.1	Length primitives renamed	510
18.2	Creating and initialising <code>dim</code> variables	510
18.3	Setting <code>dim</code> variables	511
18.4	Utilities for dimension calculations	512
18.5	Dimension expression conditionals	513
18.6	Dimension expression loops	514
18.7	Using <code>dim</code> expressions and variables	516
18.8	Viewing <code>dim</code> variables	517
18.9	Constant dimensions	517
18.10	Scratch dimensions	518
18.11	Creating and initialising <code>skip</code> variables	518
18.12	Setting <code>skip</code> variables	519
18.13	Skip expression conditionals	519
18.14	Using <code>skip</code> expressions and variables	520
18.15	Inserting skips into the output	520
18.16	Viewing <code>skip</code> variables	521
18.17	Constant skips	521
18.18	Scratch skips	521
18.19	Creating and initialising <code>muskip</code> variables	521
18.20	Setting <code>muskip</code> variables	522
18.21	Using <code>muskip</code> expressions and variables	523
18.22	Viewing <code>muskip</code> variables	523
18.23	Constant muskips	524
18.24	Scratch muskips	524
19	I3keys Implementation	524
19.1	Low-level interface	524
19.2	Constants and variables	528
19.3	The key defining mechanism	530
19.4	Turning properties into actions	532
19.5	Creating key properties	537
19.6	Setting keys	541
19.7	Utilities	546
19.8	Messages	548
20	I3fp implementation	550
21	I3fp-aux implementation	550
21.1	Internal representation	550
21.2	Using arguments and semicolons	551
21.3	Constants, and structure of floating points	552
21.4	Overflow, underflow, and exact zero	554
21.5	Expanding after a floating point number	554
21.6	Packing digits	555
21.7	Decimate (dividing by a power of 10)	558
21.8	Functions for use within primitive conditional branches	560
21.9	Integer floating points	561
21.10	Small integer floating points	561
21.11	Length of a floating point array	562

21.12x-like expansion expandably	562
21.13Messages	563
22 l3fp-traps Implementation	563
22.1 Flags	564
22.2 Traps	564
22.3 Errors	567
22.4 Messages	568
23 l3fp-round implementation	568
23.1 Rounding tools	569
23.2 The round function	573
24 l3fp-parse implementation	576
24.1 Work plan	576
24.1.1 Storing results	578
24.1.2 Precedence and infix operators	579
24.1.3 Prefix operators, parentheses, and functions	582
24.1.4 Numbers and reading tokens one by one	582
24.2 Main auxiliary functions	584
24.3 Helpers	585
24.4 Parsing one number	586
24.4.1 Numbers: trimming leading zeros	591
24.4.2 Number: small significand	593
24.4.3 Number: large significand	595
24.4.4 Number: beyond 16 digits, rounding	597
24.4.5 Number: finding the exponent	599
24.5 Constants, functions and prefix operators	602
24.5.1 Prefix operators	602
24.5.2 Constants	604
24.5.3 Functions	606
24.6 Main functions	606
24.7 Infix operators	607
24.7.1 Closing parentheses and commas	608
24.7.2 Usual infix operators	610
24.7.3 Juxtaposition	611
24.7.4 Multi-character cases	611
24.7.5 Ternary operator	612
24.7.6 Comparisons	613
24.8 Candidate: defining new l3fp functions	615
24.9 Messages	616
25 l3fp-logic Implementation	617
25.1 Syntax of internal functions	617
25.2 Existence test	618
25.3 Comparison	618
25.4 Floating point expression loops	620
25.5 Extrema	623
25.6 Boolean operations	624
25.7 Ternary operator	625

26	I3fp-basics Implementation	626
26.1	Addition and subtraction	627
26.1.1	Sign, exponent, and special numbers	627
26.1.2	Absolute addition	629
26.1.3	Absolute subtraction	631
26.2	Multiplication	636
26.2.1	Signs, and special numbers	636
26.2.2	Absolute multiplication	637
26.3	Division	639
26.3.1	Signs, and special numbers	639
26.3.2	Work plan	640
26.3.3	Implementing the significand division	643
26.4	Square root	648
26.5	About the sign	655
27	I3fp-extended implementation	656
27.1	Description of fixed point numbers	656
27.2	Helpers for numbers with extended precision	656
27.3	Multiplying a fixed point number by a short one	657
27.4	Dividing a fixed point number by a small integer	658
27.5	Adding and subtracting fixed points	659
27.6	Multiplying fixed points	660
27.7	Combining product and sum of fixed points	661
27.8	Extended-precision floating point numbers	663
27.9	Dividing extended-precision numbers	665
27.10	Inverse square root of extended precision numbers	668
27.11	Converting from fixed point to floating point	670
28	I3fp-expo implementation	672
28.1	Logarithm	673
28.1.1	Work plan	673
28.1.2	Some constants	673
28.1.3	Sign, exponent, and special numbers	673
28.1.4	Absolute ln	674
28.2	Exponential	681
28.2.1	Sign, exponent, and special numbers	681
28.3	Power	685
29	I3fp-trig Implementation	691
29.1	Direct trigonometric functions	692
29.1.1	Filtering special cases	692
29.1.2	Distinguishing small and large arguments	695
29.1.3	Small arguments	696
29.1.4	Argument reduction in degrees	696
29.1.5	Argument reduction in radians	698
29.1.6	Computing the power series	703
29.2	Inverse trigonometric functions	706
29.2.1	Arctangent and arccotangent	707
29.2.2	Arcsine and arccosine	712
29.2.3	Arccosecant and arcsecant	714

30	<i>l3fp-convert</i> implementation	716
30.1	Trimming trailing zeros	716
30.2	Scientific notation	716
30.3	Decimal representation	717
30.4	Token list representation	719
30.5	Formatting	720
30.6	Convert to dimension or integer	720
30.7	Convert from a dimension	720
30.8	Use and eval	721
30.9	Convert an array of floating points to a comma list	722
31	<i>l3fp-random</i> Implementation	722
31.1	Engine support	723
31.2	Random floating point	724
31.3	Random integer	725
32	<i>l3fp-assign</i> implementation	727
32.1	Assigning values	727
32.2	Updating values	728
32.3	Showing values	728
32.4	Some useful constants and scratch variables	729
33	<i>l3sort</i> implementation	729
33.1	Variables	729
33.2	Finding available \toks registers	730
33.3	Protected user commands	732
33.4	Merge sort	735
33.5	Expandable sorting	738
33.6	Messages	742
33.7	Deprecated functions	744
34	<i>l3tl-analysis</i> implementation	744
34.1	Internal functions	744
34.2	Internal format	745
34.3	Variables and helper functions	745
34.4	Plan of attack	747
34.5	Setup	748
34.6	First pass	749
34.7	Second pass	754
34.8	Mapping through the analysis	757
34.9	Showing the results	757
34.10	Messages	760
35	<i>l3tl-build</i> implementation	760
35.1	Variables and helper functions	760
35.2	Building the token list	761

36	I₃regex implementation	762
36.1	Plan of attack	762
36.2	Helpers	764
36.2.1	Constants and variables	765
36.2.2	Testing characters	766
36.2.3	Character property tests	770
36.2.4	Simple character escape	772
36.3	Compiling	777
36.3.1	Variables used when compiling	778
36.3.2	Generic helpers used when compiling	779
36.3.3	Mode	780
36.3.4	Framework	782
36.3.5	Quantifiers	785
36.3.6	Raw characters	787
36.3.7	Character properties	789
36.3.8	Anchoring and simple assertions	790
36.3.9	Character classes	791
36.3.10	Groups and alternations	794
36.3.11	Catcodes and csnames	796
36.3.12	Raw token lists with \u	800
36.3.13	Other	802
36.3.14	Showing regexes	802
36.4	Building	806
36.4.1	Variables used while building	806
36.4.2	Framework	806
36.4.3	Helpers for building an NFA	808
36.4.4	Building classes	809
36.4.5	Building groups	811
36.4.6	Others	816
36.5	Matching	817
36.5.1	Variables used when matching	817
36.5.2	Matching: framework	820
36.5.3	Using states of the NFA	823
36.5.4	Actions when matching	823
36.6	Replacement	826
36.6.1	Variables and helpers used in replacement	826
36.6.2	Query and brace balance	827
36.6.3	Framework	828
36.6.4	Submatches	830
36.6.5	Csnames in replacement	831
36.6.6	Characters in replacement	833
36.6.7	An error	836
36.7	User functions	836
36.7.1	Variables and helpers for user functions	838
36.7.2	Matching	839
36.7.3	Extracting submatches	839
36.7.4	Replacement	843
36.7.5	Storing and showing compiled patterns	845
36.8	Messages	845
36.9	Code for tracing	850

37	I3box implementation	851
37.1	Creating and initialising boxes	851
37.2	Measuring and setting box dimensions	852
37.3	Using boxes	852
37.4	Box conditionals	852
37.5	The last box inserted	853
37.6	Constant boxes	853
37.7	Scratch boxes	853
37.8	Viewing box contents	854
37.9	Horizontal mode boxes	855
37.10	Vertical mode boxes	856
37.11	Affine transformations	858
37.12	Deprecated functions	866
38	I3coffins Implementation	866
38.1	Coffins: data structures and general variables	867
38.2	Basic coffin functions	868
38.3	Measuring coffins	872
38.4	Coffins: handle and pole management	872
38.5	Coffins: calculation of pole intersections	875
38.6	Aligning and typesetting of coffins	878
38.7	Coffin diagnostics	882
38.8	Messages	888
39	I3color Implementation	888
40	I3sys implementation	889
40.1	The name of the job	889
40.2	Time and date	889
40.3	Detecting the engine	889
40.4	Detecting the output	890
41	I3deprecation implementation	891
42	I3candidates Implementation	892
42.1	Additions to I3box	892
42.2	Viewing part of a box	892
42.3	Additions to I3clist	895
42.4	Additions to I3coffins	895
42.5	Rotating coffins	895
42.6	Resizing coffins	900
42.7	Additions to I3file	902
42.8	Additions to I3int	903
42.9	Additions to I3msg	904
42.10	Additions to I3prop	905
42.11	Additions to I3seq	906
42.12	Additions to I3skip	908
42.13	Additions to I3sys	908
42.14	Additions to I3tl	911
42.14.1	Unicode case changing	913

42.14.2 Other additions to <code>\3tl</code>	935
42.15 Additions to <code>\3tokens</code>	937
43 <code>\3luatex</code> implementation	938
43.1 Breaking out to Lua	938
43.2 Messages	939
43.3 Lua functions for internal use	939
43.4 Generic Lua and font support	940
44 <code>\3drivers</code> Implementation	940
44.1 <code>pdfmode</code> driver	941
44.1.1 Basics	941
44.1.2 Box operations	942
44.1.3 Color	943
44.2 Images	944
44.3 <code>dvipdfmx</code> driver	946
44.3.1 Basics	946
44.3.2 Box operations	946
44.3.3 Color	947
44.4 Images	948
44.5 <code>xdvipdfmx</code> driver	949
44.5.1 Color	949
44.6 Images	950
44.7 Drawing commands: <code>pdfmode</code> and <code>(x)dvipdfmx</code>	951
44.8 Drawing	951
44.9 <code>dvips</code> driver	955
44.9.1 Basics	955
44.10 Driver-specific auxiliaries	956
44.10.1 Box operations	956
44.10.2 Color	958
44.11 Images	958
44.12 Drawing	958
44.13 <code>dvisvgm</code> driver	965
44.13.1 Basics	965
44.14 Driver-specific auxiliaries	965
44.14.1 Box operations	966
44.14.2 Color	968
44.15 Drawing	968
Index	976

Part I

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 L^AT_EX3 programming language is found in [expl3.pdf](#).

1 Naming functions and variables

L^AT_EX3 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:

- D The D specifier means *do not use*. All of the T_EX primitives are initially \let to a D name, and some are then given a second name. Only the kernel team should use anything with a D specifier!
- 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 *csname*, and indicates that the argument will be turned into a csname 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 T_EX 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 csname 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 T_EX \edef primitive carries out this type of expansion. Functions which feature an x-type argument are in general *not* expandable, unless specifically noted.

- f** The **f** specifier stands for *full expansion*, and in contrast to **x** stops at the first non-expandable item (reading the argument from left to right) without trying to expand it. For example, when setting a token list variable (a macro used for storage), the sequence

```
\tl_set:Nn \l_mya_tl { A }
\tl_set:Nn \l_myb_tl { B }
\tl_set:Nf \l_mya_tl { \l_mya_tl \l_myb_tl }
```

will leave `\l_mya_tl` with the content `A\l_myb_tl`, as `A` cannot be expanded and so terminates expansion before `\l_myb_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 arbitrary string).

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:

- 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 module¹ name and then a descriptive part. Variables end with a short identifier to show the variable type:

bool Either true or false.

box Box register.

clist Comma separated list.

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

dim “Rigid” lengths.

fp floating-point values;

¹The 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_tmpb_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_tmpa_int` would be very unreadable.

int Integer-valued count register.

prop Property list.

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

skip “Rubber” lengths.

stream An input or output stream (for reading from or writing to, respectively).

t1 Token list variables: placeholder for a token list.

1.1 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 in truth one and the same. 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 `TeXbook` 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.

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:

<code>\ExplSyntaxOn</code> <code>\ExplSyntaxOff</code>	<code>\ExplSyntaxOn ... \ExplSyntaxOff</code>
---	---

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:

`\seq_new:N <sequence>`
`\seq_new:c`

When a number of variants are described, the arguments are usually illustrated only for the base function. Here, `<sequence>` 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 argument (in plain `TeX` terms, inside an `\edef`), as well as within an `f`-type argument. These fully expandable functions are indicated in the documentation by a star:

`\cs_to_str:N *` `\cs_to_str:N <cs>`

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 `<cs>`, shorthand for a `<control sequence>`.

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:

`\seq_map_function:NN *` `\seq_map_function:NN <seq> <function>`

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:

`\sys_if_engine_xetex:TF *` `\sys_if_engine_xetex:TF {<true code>} {<false code>}`

The underlining and italic of `TF` indicates that `\sys_if_engine_xetex:T`, `\sys_if_engine_xetex:F` and `\sys_if_engine_xetex:TF` are all available. Usually, the illustration will use the `TF` variant, and so both `<true code>` and `<false code>` will be shown. The two variant forms `T` and `F` take only `<true code>` and `<false code>`, 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:

`\l_tmpa_tl`

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 `LATeX 2ε` or plain `TeX`. In these cases, the text will include an extra “**TeXhackers note**” section:

`\token_to_str:N *` `\token_to_str:N <token>`

The normal description text.

TeXhackers note: Detail for the experienced `TeX` or `LATeX 2ε` programmer. In this case, it would point out that this function is the `TeX` 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.

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 `<true code>` or the `<false code>` 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.

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_EX2_ε if `\outer` tokens are used in the arguments.

Part II

The **l3bootstrap** package

Bootstrap code

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. These modules will also form the basis of the L^AT_EX3 format, but work in this area is incomplete and not included in this documentation at present.

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 régime in which spaces 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 régime.

```
\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>`. 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_EX3 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}
```

1.1 Internal functions and variables

`\l_kernel_expl_bool`

A boolean which records the current code syntax status: `true` if currently inside a code environment. This variable should only be set by `\ExplSyntaxOn`/`\ExplSyntaxOff`.

Part III

The **I3names** package

Namespace for primitives

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;
- switches to the category code régime for programming;
- provides support settings for building the code as a T_EX format.

This module is entirely dedicated to primitives, 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_ET_EX and LuaT_EX should be consulted for details of the primitives. These are named based on the engine which first introduced them:

\tex_... Introduced by T_EX itself;
\etex_... Introduced by the ε -T_EX extensions;
\pdftex_... Introduced by pdfT_EX;
\xetex_... Introduced by X_ET_EX;
\luatex_... Introduced by LuaT_EX;
\utex_... Introduced by X_ET_EX and LuaT_EX;
\ptex_... Introduced by pT_EX;
\uptex_... Introduced by upT_EX.

Part IV

The **I3basics** 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.

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.

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 <token>`

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 will be 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). The later will be a } if standard category codes apply.

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 by the appropriate arguments absorbed by the function. In the following, *<code>* is therefore used as a shorthand for “replacement text”.

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

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 will be 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 will result 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 will 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 will 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 expansion.

Finally, the functions in Subsections 3.2 and 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 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 2.

3.2 Defining new functions using parameter text

```
\cs_new:Npn \cs_new:cpx
```

`\cs_new:Npn <function> <parameters> {{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. The definition is global and an error will result if the `<function>` is already defined.

```
\cs_new_nopar:Npn \cs_new_nopar:cpx
```

`\cs_new_nopar:Npn <function> <parameters> {{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 definition is global and an error will result if the `<function>` is already defined.

```
\cs_new_protected:Npn \cs_new_protected:cpx
```

`\cs_new_protected:Npn <function> <parameters> {{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. The `<function>` will not expand within an `x`-type argument. The definition is global and an error will result if the `<function>` is already defined.

```
\cs_new_protected_nopar:Npn \cs_new_protected_nopar:cpx
```

`\cs_new_protected_nopar:Npn <function> <parameters> {{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 argument. The definition is global and an error will result if the `<function>` is already defined.

```
\cs_set:Npn \cs_set:cpx
```

`\cs_set:Npn <function> <parameters> {{code}}`

Sets `<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. The assignment of a meaning to the `<function>` is restricted to the current TeX group level.

```
\cs_set_nopar:Npn \cs_set_nopar:cpx
```

`\cs_set_nopar:Npn <function> <parameters> {{code}}`

Sets `<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 assignment of a meaning to the `<function>` is restricted to the current TeX group level.

```
\cs_set_protected:Npn \cs_set_protected:cpx
```

`\cs_set_protected:Npn <function> <parameters> {{code}}`

Sets `<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. The assignment of a meaning to the `<function>` is restricted to the current TeX group level. The `<function>` will not expand within an `x`-type argument.

```
\cs_set_protected_nopar:Npn \cs_set_protected_nopar:Npn <function> <parameters> {<code>}
\cs_set_protected_nopar:cpx
```

Sets *<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 assignment of a meaning to the *<function>* is restricted to the current TeX group level. The *<function>* will not expand within an x-type argument.

```
\cs_gset:Npn \cs_gset:cpx
```

Globally sets *<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. The assignment of a meaning to the *<function>* is *not* restricted to the current TeX group level: the assignment is global.

```
\cs_gset_nopar:Npn \cs_gset_nopar:cpx
```

Globally sets *<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 assignment of a meaning to the *<function>* is *not* restricted to the current TeX group level: the assignment is global.

```
\cs_gset_protected:Npn \cs_gset_protected:cpx
```

Globally sets *<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. The assignment of a meaning to the *<function>* is *not* restricted to the current TeX group level: the assignment is global. The *<function>* will not expand within an x-type argument.

```
\cs_gset_protected_nopar:Npn \cs_gset_protected_nopar:Npn <function> <parameters> {<code>}
```

Globally sets *<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 assignment of a meaning to the *<function>* is *not* restricted to the current TeX group level: the assignment is global. The *<function>* will not expand within an x-type argument.

3.3 Defining new functions using the signature

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

Creates *<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 definition is global and an error will result if the *<function>* is already defined.

```
\cs_new_nopar:Nn
```

```
\cs_new_nopar:(cn|Nx|cx)
```

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

Creates *<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 definition is global and an error will result if the *<function>* is already defined.

```
\cs_new_protected:Nn
```

```
\cs_new_protected:(cn|Nx|cx)
```

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

Creates *<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 argument. The definition is global and an error will result if the *<function>* is already defined.

```
\cs_new_protected_nopar:Nn
```

```
\cs_new_protected_nopar:(cn|Nx|cx)
```

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

Creates *<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 argument. The definition is global and an error will result if the *<function>* is already defined.

```
\cs_set:Nn
```

```
\cs_set:(cn|Nx|cx)
```

```
\cs_set: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 restricted to the current T_EX 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 argument. The assignment of a meaning to the *<function>* is restricted to the current T_EX group level.

<code>\cs_set_protected_nopar:Nn</code>	<code>\cs_set_protected_nopar:Nn <function> {<code>}</code>
<code>\cs_set_protected_nopar:(cn Nx cx)</code>	Sets <i><function></i> to expand to <i><code></i> as replacement text. Within the <i><code></i> , the number of <i><parameters></i> is detected automatically from the function signature. These <i><parameters></i> (#1, #2, etc.) will be replaced by those absorbed by the function. When the <i><function></i> is used the <i><parameters></i> absorbed cannot contain \par tokens. The <i><function></i> will not expand within an x-type argument. The assignment of a meaning to the <i><function></i> is restricted to the current TeX group level.
<code>\cs_gset:Nn</code>	<code>\cs_gset:Nn <function> {<code>}</code>
<code>\cs_gset:(cn Nx cx)</code>	Sets <i><function></i> to expand to <i><code></i> as replacement text. Within the <i><code></i> , the number of <i><parameters></i> is detected automatically from the function signature. These <i><parameters></i> (#1, #2, etc.) will be replaced by those absorbed by the function. The assignment of a meaning to the <i><function></i> is global.
<code>\cs_gset_nopar:Nn</code>	<code>\cs_gset_nopar:Nn <function> {<code>}</code>
<code>\cs_gset_nopar:(cn Nx cx)</code>	Sets <i><function></i> to expand to <i><code></i> as replacement text. Within the <i><code></i> , the number of <i><parameters></i> is detected automatically from the function signature. These <i><parameters></i> (#1, #2, etc.) will be replaced by those absorbed by the function. When the <i><function></i> is used the <i><parameters></i> absorbed cannot contain \par tokens. The assignment of a meaning to the <i><function></i> is global.
<code>\cs_gset_protected:Nn</code>	<code>\cs_gset_protected:Nn <function> {<code>}</code>
<code>\cs_gset_protected:(cn Nx cx)</code>	Sets <i><function></i> to expand to <i><code></i> as replacement text. Within the <i><code></i> , the number of <i><parameters></i> is detected automatically from the function signature. These <i><parameters></i> (#1, #2, etc.) will be replaced by those absorbed by the function. The <i><function></i> will not expand within an x-type argument. The assignment of a meaning to the <i><function></i> 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 <i><function></i> to expand to <i><code></i> as replacement text. Within the <i><code></i> , the number of <i><parameters></i> is detected automatically from the function signature. These <i><parameters></i> (#1, #2, etc.) will be replaced by those absorbed by the function. When the <i><function></i> is used the <i><parameters></i> absorbed cannot contain \par tokens. The <i><function></i> will not expand within an x-type argument. The assignment of a meaning to the <i><function></i> 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>
<small>Updated: 2012-01-14</small>	
Uses the <i><creator></i> function (which should have signature Npn, for example \cs_new:Npn) to define a <i><function></i> which takes <i><number></i> arguments and has <i><code></i> as replacement text. The <i><number></i> of arguments is an integer expression, evaluated as detailed for \int_eval:n.	

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”.

```
\cs_new_eq:NN
\cs_new_eq:(Nc|cN|cc)
```

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.

```
\cs_set_eq:NN
\cs_set_eq:(Nc|cN|cc)
```

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 TeX group level.

```
\cs_gset_eq:NN
\cs_gset_eq:(Nc|cN|cc)
```

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 TeX group level: the assignment is global.

3.5 Deleting control sequences

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

```
\cs_undefine:N
\cs_undefine:c
```

Updated: 2011-09-15

$\langle control\ sequence \rangle$
Sets $\langle control\ sequence \rangle$ to be globally undefined.

3.6 Showing control sequences

```
\cs_meaning:N *
\cs_meaning:c *
```

Updated: 2011-12-22

$\langle control\ sequence \rangle$

This function expands to the *meaning* of the $\langle control\ sequence \rangle$ control sequence. This will show the $\langle replacement\ text \rangle$ for a macro.

TeXhackers note: This is TeX’s `\meaning` primitive. The `c` variant correctly reports undefined arguments.

\cs_show:N
\cs_show:c
Updated: 2017-02-14

\cs_show:N *(control sequence)*

Displays the definition of the *(control sequence)* on the terminal.

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

\cs_log:N
\cs_log:c
New: 2014-08-22
Updated: 2017-02-14

\cs_log:N *(control sequence)*

Writes the definition of the *(control sequence)* in the log file. See also \cs_show:N which displays the result in the terminal.

3.7 Converting to and from control sequences

\use:c *

Converts the given *(control sequence name)* into a single control sequence token. This process requires two expansions. The content for *(control sequence name)* may be literal material or from other expandable functions. The *(control sequence name)* must, when fully expanded, consist of character tokens which are not active: typically, they will be of category code 10 (space), 11 (letter) or 12 (other), or a mixture of these.

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.

\cs_if_exist_use:N *
\cs_if_exist_use:c *
\cs_if_exist_use:NTF *
\cs_if_exist_use:cTF *

New: 2012-11-10

\cs_if_exist_use:N *(control sequence)*
\cs_if_exist_use:NTF *(control sequence)* {*(true code)*} {*(false code)*}

Tests whether the *(control sequence)* is currently defined (whether as a function or another control sequence type), and if it is inserts the *(control sequence)* into the input stream followed by the *(true code)*. Otherwise the *(false code)* is used.

\cs:w *
\cs_end: *

\cs:w *(control sequence name)* \cs_end:

Converts the given *(control sequence name)* into a single control sequence token. This process requires one expansion. The content for *(control sequence name)* may be literal material or from other expandable functions. The *(control sequence name)* must, when fully expanded, consist of character tokens which are not active: typically, they will be 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`.

\cs_to_str:N * `\cs_to_str:N` (*control sequence*)

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

4 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 in absorbing them the outer set will be 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` {{group₁}}
\use:nn * `\use:nn` {{group₁}} {{group₂}}
\use:nnn * `\use:nnn` {{group₁}} {{group₂}} {{group₃}}
\use:nnnn * `\use:nnnn` {{group₁}} {{group₂}} {{group₃}} {{group₄}}

As illustrated, these functions will absorb between one and four arguments, as indicated by the argument specifier. The braces surrounding each argument will be removed leaving the remaining tokens in the input stream. The category code of these tokens will also be 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 } }
```

will result in the input stream containing

```
abc { def }
```

i.e. only the outer braces will be removed.

`\use_i:nn` ★ `\use_i:nn {<arg1>} {<arg2>}`
`\use_ii:nn` ★

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 will also be fixed (if it has not already been by some other absorption). A single expansion is needed for the functions to take effect.

`\use_i:nnn` ★ `\use_i:nnn {<arg1>} {<arg2>} {<arg3>}`
`\use_ii:nnn` ★
`\use_iii:nnn` ★

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 will also be fixed (if it has not already been by some other absorption). A single expansion is needed for the functions to take effect.

`\use_i:nnnn` ★ `\use_i:nnnn {<arg1>} {<arg2>} {<arg3>} {<arg4>}`
`\use_ii:nnnn` ★
`\use_iii:nnnn` ★
`\use_iv:nnnn` ★

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_i:nnnn`, `\use_ii: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 will also be fixed (if it has not already been by some other absorption). A single expansion is needed for the functions to take effect.

`\use_i_ii:nnn` ★ `\use_i_ii:nnn {<arg1>} {<arg2>} {<arg3>}`

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

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

will result in the input stream containing

```
abc { def }
```

i.e. the outer braces will be removed and the third group will be removed.

`\use_none:n` ★ `\use_none:n {<group1>}`
`\use_none:nn` ★
`\use_none:nnn` ★
`\use_none:nnnn` ★
`\use_none:nnnnn` ★
`\use_none:nnnnnn` ★
`\use_none:nnnnnnn` ★
`\use_none:nnnnnnnn` ★

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 `n` arguments may be an unbraced single token (i.e. an `N` argument).

\use:x

Updated: 2011-12-31

\use:x {⟨expandable tokens⟩}

Fully expands the ⟨expandable tokens⟩ and inserts the result into the input stream at the current location. Any hash characters (#) in the argument must be doubled.

4.1 Selecting tokens from delimited arguments

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

\use_none_delimit_by_q_nil:w	★ \use_none_delimit_by_q_nil:w ⟨balanced text⟩ \q_nil
\use_none_delimit_by_q_stop:w	★ \use_none_delimit_by_q_stop:w ⟨balanced text⟩ \q_stop
\use_none_delimit_by_q_recursion_stop:w	★ \use_none_delimit_by_q_recursion_stop:w ⟨balanced text⟩ \q_recursion_stop

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

\use_i_delimit_by_q_nil:nw	★ \use_i_delimit_by_q_nil:nw {⟨inserted tokens⟩} ⟨balanced text⟩
\use_i_delimit_by_q_stop:nw	★ \q_nil
\use_i_delimit_by_q_recursion_stop:nw	★ \use_i_delimit_by_q_stop:nw {⟨inserted tokens⟩} ⟨balanced text⟩ \q_stop \use_i_delimit_by_q_recursion_stop:nw {⟨inserted tokens⟩} ⟨balanced text⟩ \q_recursion_stop

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

5 Predicates and conditionals

LATEX3 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 will turn 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 carry 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 will usually be 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 will 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 will be 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_t1 || \cs_if_free_p:N \g_tmpz_t1
} {\langle true code\rangle} {\langle false code\rangle}
```

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

Primitive conditionals There is a third variety of conditional, which is the original concept used in plain TeX and L^AT_EX 2_E. 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.

<pre>\c_true_bool \c_false_bool</pre>	Constants that represent <code>true</code> and <code>false</code> , respectively. Used to implement predicates.
---------------------------------------	---

5.1 Tests on control sequences

<pre>\cs_if_eq_p:NN * \cs_if_eq_p:NNTF *</pre>	<code>\cs_if_eq_p:NN</code> { <code>\langle cs_1\rangle</code> } { <code>\langle cs_2\rangle</code> } <code>\cs_if_eq_p:NNTF</code> { <code>\langle cs_1\rangle</code> } { <code>\langle cs_2\rangle</code> } {\langle true code\rangle} {\langle false code\rangle}
--	---

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`.

<pre>\cs_if_exist_p:N * \cs_if_exist_p:c * \cs_if_exist:NTF * \cs_if_exist:cTF *</pre>	<code>\cs_if_exist_p:N</code> { <i>control sequence</i> } <code>\cs_if_exist_p:c</code> { <i>control sequence</i> } {\langle true code\rangle} {\langle false code\rangle}
--	---

Tests whether the *(control sequence)* is currently defined (whether as a function or another control sequence type). Any valid definition of *(control sequence)* will evaluate as `true`.

<pre>\cs_if_free_p:N * \cs_if_free_p:c * \cs_if_free:NTF * \cs_if_free:cTF *</pre>	<code>\cs_if_free_p:N</code> { <i>control sequence</i> } <code>\cs_if_free_p:c</code> { <i>control sequence</i> } {\langle true code\rangle} {\langle false code\rangle}
--	---

Tests whether the *(control sequence)* is currently free to be defined. This test will be `false` if the *(control sequence)* currently exists (as defined by `\cs_if_exist:N`).

5.2 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 will often contain 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 will 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: always executes <true code>, while \if_false: always executes <false code>.</code>
<code>\reverse_if:N</code>	★	<code>\reverse_if:N reverses any two-way primitive conditional. \else: and \fi: delimit the branches of the conditional. The function \or: is documented in \3int and used in case switches.</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>
<code>\if_meaning:w</code> executes <code><true code></code> when <code><arg₁></code> and <code><arg₂></code> are the same, otherwise it executes <code><false code></code> . <code><arg₁></code> and <code><arg₂></code> could be functions, variables, tokens; in all cases the <i>unexpanded</i> 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 will 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 <code><cs></code> appears in the hash table or if the control sequence that can be formed from <code><tokens></code> appears in the hash table. The latter function does not turn the control sequence in question into <code>\scan_stop:!</code> 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>	★	

6 Internal kernel functions

__chk_if_exist_cs:N
__chk_if_exist_cs:c

__chk_if_exist_cs:N <cs>

This function checks that <cs> exists according to the criteria for \cs_if_exist_p:N, and if not raises a kernel-level error.

__chk_if_free_cs:N
__chk_if_free_cs:c

__chk_if_free_cs:N <cs>

This function checks that <cs> is free according to the criteria for \cs_if_free_p:N, and if not raises a kernel-level error.

__chk_if_exist_var:N

__chk_if_exist_var:N <var>

This function checks that <var> is defined according to the criteria for \cs_if_free_p:N, and if not raises a kernel-level error. This function is only created if the package option **check-declarations** is active.

__chk_log:x

__chk_log:x {<message text>}

If the **log-functions** option is active, this function writes the <message text> to the log file using \iow_log:x. Otherwise, the <message text> is ignored using \use_none:n.

__chk_suspend_log:
__chk_resume_log:

__chk_suspend_log: ... __chk_log:x ... __chk_resume_log:

Any __chk_log:x command between __chk_suspend_log: and __chk_resume_log: is suppressed. These commands can be nested.

__cs_count_signature:N ★
__cs_count_signature:c ★

__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_split_function>NN ★

__cs_split_function>NN <function> <processor>

Splits the <function> into the <name> (*i.e.* the part before the colon) and the <signature> (*i.e.* after the colon). This information is then placed in the input stream after the <processor> function in three parts: the <name>, the <signature> and a logic token indicating if a colon was found (to differentiate variables from function names). The <name> will not include the escape character, and both the <name> and <signature> are made up of tokens with category code 12 (other). The <processor> should be a function with argument specification :nnN (plus any trailing arguments needed).

__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).

_kernel_register_show:N

Used to show the contents of a TeX register at the terminal, formatted such that internal parts of the mechanism are not visible.

_kernel_register_log:N

_kernel_register_log:N

Part V

The **I3expan** 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 LATEX3 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.

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 from the `\exp_` module. 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` will expand the content of third argument once before any expansion of the first and second arguments. If `\seq_gpush:N` was not defined it could be coded in the following way:

```
\exp_args:NNo \seq_gpush:Nn
  \g_file_name_stack
  \l_tmpa_t1
```

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:N`

```
\cs_new:Npn \seq_gpush:N { \exp_args:NNo \seq_gpush:Nn }
```

Providing variants in this way in style files is uncritical 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.

2 Methods for defining variants

\cs_generate_variant:Nn

Updated: 2015-08-06

```
\cs_generate_variant:Nn <parent control sequence> {<variant argument specifiers>}
```

This function is used to define argument-specifier variants of the *<parent control sequence>* for L^AT_EX3 code-level macros. The *<parent control sequence>* is first separated into the *(base name)* and *(original argument specifier)*. The comma-separated list of *<variant argument specifiers>* is then used to define variants of the *(original argument specifier)* where these are not already defined. For each *<variant>* given, a function is created which will expand its arguments as detailed and pass them to the *<parent control sequence>*. So for example

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

will create a new function `\foo:cN` which will expand its first argument into a control sequence name and pass the result to `\foo:Nn`. Similarly

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

would generate the functions `\foo:NV` and `\foo:cV` in the same way. The `\cs_generate_variant:Nn` function can only be applied if the *<parent control sequence>* is already defined. Only `n` and `N` arguments can be changed to other types. If the *<parent control sequence>* is protected or if the *<variant>* involves `x` arguments, then the *<variant control sequence>* will also be protected. The *<variant>* is created globally, as is any `\exp_args:N<variant>` function needed to carry out the expansion.

3 Introducing the variants

The available internal functions for argument expansion come in two flavours, some of them are faster than others. Therefore (when speed is important) it is usually best to follow the following guidelines when defining new functions that are supposed to come with variant forms:

- Arguments that might need expansion should come first in the list of arguments to make processing faster.
- Arguments that should consist of single tokens should come first.
- Arguments that need full expansion (*i.e.*, are denoted with `x`) should be avoided if possible as they can not be processed expandably, *i.e.*, functions of this type will not work correctly in arguments that are themselves subject to `x` expansion.
- In general, unless in the last position, multi-token arguments `n`, `f`, and `o` will need special processing when more than one argument is being expanded. This special processing is not fast. 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`, with possible trailing `N` or `n`, which are not expanded.

The `V` type returns the value of a register, which can be one of `t1`, `int`, `skip`, `dim`, `toks`, 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 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 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 } }
```

will result in the call \example:n { 3 , \int_eval:n { 3 + 4 } } while using \example:x instead results in \example:n { 3 , 7 } at the cost of being protected. If you use this type of expansion in conditional processing then you should stick to using TF type functions only as it does not try to finish any \if... \fi: itself!

If 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 emphfirst non-expandable token. This means for example that both

```
\tl_set:Nn \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.

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.

<u>\exp_args:N</u> *	<code>\exp_args:N</code> <i>function</i> { <i>tokens</i> } ...	This function absorbs two arguments (the <i>function</i> name and the <i>tokens</i>). The <i>tokens</i> are expanded once, and the result is inserted in braces into the input stream <i>after</i> reinsertion of the <i>function</i> . Thus the <i>function</i> may take more than one argument: all others will be left unchanged.
<u>\exp_args:Nc</u> *	<code>\exp_args:Nc</code> <i>function</i> { <i>tokens</i> }	This function absorbs two arguments (the <i>function</i> name and the <i>tokens</i>). The <i>tokens</i> are expanded until only characters remain, and are then turned into a control sequence. (An internal error will occur if such a conversion is not possible). The result is inserted into the input stream <i>after</i> reinsertion of the <i>function</i> . Thus the <i>function</i> may take more than one argument: all others will be left unchanged.
<u>\exp_args:cc</u> *		The :cc variant constructs the <i>function</i> name in the same manner as described for the <i>tokens</i> .
<u>\exp_args:NV</u> *	<code>\exp_args:NV</code> <i>function</i> < <i>variable</i> >	This function absorbs two arguments (the names of the <i>function</i> and the <i>variable</i>). The content of the <i>variable</i> are recovered and placed inside braces into the input stream <i>after</i> reinsertion of the <i>function</i> . Thus the <i>function</i> may take more than one argument: all others will be left unchanged.
<u>\exp_args:Nv</u> *	<code>\exp_args:Nv</code> <i>function</i> { <i>tokens</i> }	This function absorbs two arguments (the <i>function</i> name and the <i>tokens</i>). The <i>tokens</i> are expanded until only characters remain, and are then turned into a control sequence. (An internal error will occur if such a conversion is not possible). This control sequence should be the name of a <i>variable</i> . The content of the <i>variable</i> are recovered and placed inside braces into the input stream <i>after</i> reinsertion of the <i>function</i> . Thus the <i>function</i> may take more than one argument: all others will be left unchanged.
<u>\exp_args:Nf</u> *	<code>\exp_args:Nf</code> <i>function</i> { <i>tokens</i> }	This function absorbs two arguments (the <i>function</i> name and the <i>tokens</i>). The <i>tokens</i> are fully expanded until the first non-expandable token or space is found, and the result is inserted in braces into the input stream <i>after</i> reinsertion of the <i>function</i> . Thus the <i>function</i> may take more than one argument: all others will be left unchanged.
<u>\exp_args:Nx</u>	<code>\exp_args:Nx</code> <i>function</i> { <i>tokens</i> }	This function absorbs two arguments (the <i>function</i> name and the <i>tokens</i>) and exhaustively expands the <i>tokens</i> second. The result is inserted in braces into the input stream <i>after</i> reinsertion of the <i>function</i> . Thus the <i>function</i> may take more than one argument: all others will be left unchanged.

5 Manipulating two arguments

```
\exp_args:NNo ★ \exp_args:NNc <token1> <token2> {<tokens>}  
\exp_args:NNc ★  
\exp_args:NNv ★  
\exp_args:NNV ★  
\exp_args:NNf ★  
\exp_args:Nco ★  
\exp_args:Ncf ★  
\exp_args:Ncc ★  
\exp_args:NVV ★
```

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:Nno ★ \exp_args:Noo <token> {<tokens1>} {<tokens2>}  
\exp_args:NnV ★  
\exp_args:Nnf ★  
\exp_args:Noo ★  
\exp_args:Nof ★  
\exp_args:Noc ★  
\exp_args:Nff ★  
\exp_args:Nfo ★  
\exp_args:Nnc ★
```

Updated: 2012-01-14

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

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.

6 Manipulating three arguments

```
\exp_args:NNNNo ★ \exp_args:NNNNo <token1> <token2> <token3> {<tokens>}  
\exp_args:NNNV ★  
\exp_args:Nccc ★  
\exp_args:NcNc ★  
\exp_args:NcNo ★  
\exp_args:Ncco ★
```

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 the next item on the input stream, followed by the expansion of the second argument, *etc.*

```
\exp_args:NNoo ★ \exp_args:NNoo <token1> <token2> {<token3>} {<tokens>}  
\exp_args:NNno ★  
\exp_args:Nnno ★  
\exp_args:Nnnnc ★  
\exp_args:Nooo ★
```

These 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 the next item on the input stream, followed by the expansion of the second argument, *etc.* These functions need special (slower) processing.

```
\exp_args:NNNx \exp_args:NNnx \exp_args:NNox \exp_args:Nnrx \exp_args:Nnox \exp_args:Nnox \exp_args:Noox \exp_args:Ncnx \exp_args:Nccx
```

New: 2015-08-12

7 Unbraced expansion

\exp_last_unbraced:NV	*	\exp_last_unbraced:Nno \token \tokens_1 \tokens_2
\exp_last_unbraced:(Nf No Nv)	*	
\exp_last_unbraced:Nco	*	
\exp_last_unbraced:(NcV NNV NNo)	*	
\exp_last_unbraced:Nno	*	
\exp_last_unbraced:(Noo Nfo)	*	
\exp_last_unbraced:NNNV	*	
\exp_last_unbraced:NNNo	*	
\exp_last_unbraced:NnNo	*	

Updated: 2012-02-12

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, and :Nfo variants need special (slower) processing.

TeXhackers 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 `\tokens` and leaves the result in the input stream after reinsertion of `\function`. This function is not expandable.

```
\exp_last_two_unbraced:Noo * \exp_last_two_unbraced:Noo \token \tokens_1 \tokens_2
```

This function absorbs three arguments and expand 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 <token1> <token2>`

Carries out a single expansion of `<token2>` (which may consume arguments) prior to the expansion of `<token1>`. If `<token2>` is a TeX primitive, it will be executed rather than expanded, while if `<token2>` has not expansion (for example, if it is a character) then it will be left unchanged. It is important to notice that `<token1>` may be *any* single token, including group-opening and -closing tokens (`{` or `}` assuming normal TeX category codes). Unless specifically required, expansion should be carried out using an appropriate argument specifier variant or the appropriate `\exp_arg:N` function.

TeXhackers note: This is the TeX primitive `\expandafter` renamed.

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 'expandable' since they themselves will not appear after the expansion has completed.

`\exp_not:N` ★ `\exp_not:N <token>`

Prevents expansion of the `<token>` in a context where it would otherwise be expanded, for example an x-type argument.

TeXhackers note: This is the TeX `\noexpand` primitive.

`\exp_not:c` ★ `\exp_not:c {<tokens>}`

Expands the `<tokens>` until only unexpandable content remains, and then converts this into a control sequence. Further expansion of this control sequence is then inhibited.

`\exp_not:n` ★ `\exp_not:n {<tokens>}`

Prevents expansion of the `<tokens>` in a context where they would otherwise be expanded, for example an x-type argument.

TeXhackers note: This is the ε-Tex `\unexpanded` primitive. Hence its argument *must* be surrounded by braces.

`\exp_not:V` ★ `\exp_not:V <variable>`

Recovers the content of the `<variable>`, then prevents expansion of this material in a context where it would otherwise be expanded, for example an x-type argument.

`\exp_not:v` ★ `\exp_not:v {<tokens>}`

Expands the `<tokens>` until only unexpandable content remains, and then converts this into a control sequence (which should be a `<variable>` name). The content of the `<variable>` is recovered, and further expansion is prevented in a context where it would otherwise be expanded, for example an x-type argument.

`\exp_not:o` ★ `\exp_not:o {<tokens>}`

Expands the *<tokens>* once, then prevents any further expansion in a context where they would otherwise be expanded, for example an **x**-type argument.

`\exp_not:f` ★ `\exp_not:f {<tokens>}`

Expands *<tokens>* fully until the first unexpandable token is found. Expansion then stops, and the result of the expansion (including any tokens which were not expanded) is protected from further expansion.

`\exp_stop_f:` ★ `\foo_bar:f { <tokens> } \exp_stop_f: {<more tokens> }`

Updated: 2011-06-03

This function terminates an **f**-type expansion. Thus if a function `\foo_bar:f` starts an **f**-type expansion and all of *<tokens>* are expandable `\exp_stop_f:` will terminate the expansion of tokens even if *<more tokens>* are also expandable. The function itself is an implicit space token. Inside an **x**-type expansion, it will retain its form, but when typeset it produces the underlying space (\square).

9 Controlled expansion

The `expl3` language makes all efforts to hide the complexity of T_EX 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 T_EX 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. You will find these commands 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!

`\exp:w` ★ `\exp:w {<expandable-tokens>} \exp_end:`

`\exp_end:` ★

New: 2015-08-23

Expands *<expandable-tokens>* until reaching `\exp_end:` at which point expansion stops. 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:` will be misinterpreted later on.²

In typical use cases the `\exp_end:` will be 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.

```
\exp:w          *
\exp_end_continue_f:w *
```

New: 2015-08-23

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

Expands *<expandable-tokens>* until reaching `\exp_end_continue_f:w` at which point expansion continues as an f-type expansion expanding *<further-tokens>* until an unexpandable token is encountered (or the f-type expansion is explicitly terminated by `\exp_stop_f:`). As with all f-type expansions a space ending the expansion will get removed.

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.³

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.

```
\exp:w          *
\exp_end_continue_f:nw *
```

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 a brace group then the braces are removed. If on the other hand it starts with space tokens then these space tokens are removed while searching for the argument. Thus such space tokens will not terminate the f-type expansion.

10 Internal functions and variables

```
\l__exp_internal_t1
```

The `\exp_` module has its private variables to temporarily store results of the argument expansion. This is done to avoid interference with other functions using temporary variables.

²Due to the implementation you might get the character in position 0 in the current font (typically ““) in the output without any error message!

³In this particular case you may get a character into the output as well as an error message.

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

\:::N 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.

\:::c
\:::o
\:::f
\:::x
\:::v
\:::V
\:::

Part VI

The **l3tl** package

Token lists

\TeX works with tokens, and $\text{L}\text{\^A}\text{T}\text{\^E}3$ 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, function 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.

1 Creating and initialising token list variables

`\tl_new:N`
`\tl_new:c`

```
\tl_new:N <tl var>
```

Creates a new `<tl var>` or raises an error if the name is already taken. The declaration is global. The `<tl var>` will initially be empty.

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

```
\tl_const:Nn <tl var> {{token list}}
```

Creates a new constant `<tl var>` or raises an error if the name is already taken. The value of the `<tl var>` will be set globally to the `<token list>`.

`\tl_clear:N`
`\tl_clear:c`
`\tl_gclear:N`
`\tl_gclear:c`

```
\tl_clear:N <tl var>
```

Clears all entries from the `<tl var>`.

```
\tl_clear_new:N  
\tl_clear_new:c  
\tl_gclear_new:N  
\tl_gclear_new:c
```

```
\tl_clear_new:N <tl var>
```

Ensures that the $\langle tl\ var \rangle$ exists globally by applying `\tl_new:N` if necessary, then applies `\tl_(g)clear:N` to leave the $\langle tl\ var \rangle$ empty.

```
\tl_set_eq:NN  
\tl_set_eq:(cN|Nc|cc)  
\tl_gset_eq:NN  
\tl_gset_eq:(cN|Nc|cc)
```

```
\tl_set_eq:NN <tl var1> <tl var2>
```

Sets the content of $\langle tl\ var_1 \rangle$ equal to that of $\langle tl\ var_2 \rangle$.

```
\tl_concat:NNN  
\tl_concat:ccc  
\tl_gconcat:NNN  
\tl_gconcat:ccc
```

New: 2012-05-18

```
\tl_concat:NNN <tl var1> <tl var2> <tl var3>
```

Concatenates the content of $\langle tl\ var_2 \rangle$ and $\langle tl\ var_3 \rangle$ together and saves the result in $\langle tl\ var_1 \rangle$. The $\langle tl\ var_2 \rangle$ will be placed at the left side of the new token list.

```
\tl_if_exist_p:N *  
\tl_if_exist_p:c *  
\tl_if_exist:NTF *  
\tl_if_exist:cTF *
```

New: 2012-03-03

```
\tl_if_exist_p:N <tl var>
```

```
\tl_if_exist:NTF <tl var> {(true code)} {(false code)}
```

Tests whether the $\langle tl\ var \rangle$ is currently defined. This does not check that the $\langle tl\ var \rangle$ really is a token list variable.

```
\tl_set:Nn  
\tl_set:(NV|Nv|No|Nf|Nx|cn|cV|cv|co|cf|cx)  
\tl_gset:Nn  
\tl_gset:(NV|Nv|No|Nf|Nx|cn|cV|cv|co|cf|cx)
```

Sets $\langle tl\ var \rangle$ to contain $\langle tokens \rangle$, removing any previous content from the variable.

```
\tl_put_left:Nn  
\tl_put_left:(NV|No|Nx|cn|cV|co|cx)  
\tl_gput_left:Nn  
\tl_gput_left:(NV|No|Nx|cn|cV|co|cx)
```

```
\tl_put_left:Nn <tl var> {(tokens)}
```

Appends $\langle tokens \rangle$ to the left side of the current content of $\langle tl\ var \rangle$.

```
\tl_put_right:Nn  
\tl_put_right:(NV|No|Nx|cn|cV|co|cx)  
\tl_gput_right:Nn  
\tl_gput_right:(NV|No|Nx|cn|cV|co|cx)
```

```
\tl_put_right:Nn <tl var> {(tokens)}
```

Appends $\langle tokens \rangle$ to the right side of the current content of $\langle tl\ var \rangle$.

3 Modifying token list variables

```
\tl_replace_once:Nnn
\tl_replace_once:cnn
\tl_greplace_once:Nnn
\tl_greplace_once:cnn
```

Updated: 2011-08-11

`\tl_replace_once:Nnn <tl var> {<old tokens>} {<new tokens>}`

Replaces the first (leftmost) occurrence of *<old tokens>* in the *<tl var>* with *<new tokens>*. *<Old 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).

```
\tl_replace_all:Nnn
\tl_replace_all:cnn
\tl_greplace_all:Nnn
\tl_greplace_all:cnn
```

Updated: 2011-08-11

`\tl_replace_all:Nnn <tl var> {<old tokens>} {<new tokens>}`

Replaces all occurrences of *<old tokens>* in the *<tl var>* with *<new tokens>*. *<Old 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 *<old tokens>* may remain after the replacement (see `\tl_remove_all:Nn` for an example).

```
\tl_remove_once:Nn
\tl_remove_once:cn
\tl_gremove_once:Nn
\tl_gremove_once:cn
```

Updated: 2011-08-11

`\tl_remove_once:Nn <tl var> {<tokens>}`

Removes the first (leftmost) occurrence 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).

```
\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 {abcccd} \tl_remove_all:Nn \l_tmpa_tl {bc}`

will result in `\l_tmpa_tl` containing abcd.

4 Reassigning token list category codes

These functions allow the rescanning of tokens: re-apply TeX'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).

```
\tl_set_rescan:Nnn          \tl_set_rescan:Nnn <tl var> {<setup>} {<tokens>}
\tl_set_rescan:(Nno|Nnx|cnn|cno|cnx)
\tl_gset_rescan:Nnn
\tl_gset_rescan:(Nno|Nnx|cnn|cno|cnx)
```

Updated: 2015-08-11

Sets $\langle tl\ var\rangle$ to contain $\langle tokens\rangle$, applying the category code régime specified in the $\langle setup\rangle$ before carrying out the assignment. (Category codes applied to tokens not explicitly covered by the $\langle setup\rangle$ will be those in force at the point of use of $\backslash tl_set_rescan:Nnn$.) This allows the $\langle tl\ var\rangle$ to contain material with category codes other than those that apply when $\langle tokens\rangle$ are absorbed. The $\langle setup\rangle$ is run within a group and may contain any valid input, although only changes in category codes are relevant. See also $\backslash tl_rescan:nn$.

TeXhackers note: The $\langle tokens\rangle$ are first turned into a string (using $\backslash tl_to_str:n$). If the string contains one or more characters with character code $\backslash newlinechar$ (set equal to $\backslash endlinechar$ unless that is equal to 32, before the user $\langle setup\rangle$), 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. Only the case of a single line is supported in LuaTeX because of a bug in this engine.

```
\tl_rescan:nn {<setup>} {<tokens>}
```

Updated: 2015-08-11

Rescans $\langle tokens\rangle$ applying the category code régime specified in the $\langle setup\rangle$, and leaves the resulting tokens in the input stream. (Category codes applied to tokens not explicitly covered by the $\langle setup\rangle$ will be those in force at the point of use of $\backslash tl_rescan:nn$.) The $\langle setup\rangle$ is run within a group and may contain any valid input, although only changes in category codes are relevant. See also $\backslash tl_set_rescan:Nnn$, which is more robust than using $\backslash tl_set:Nn$ in the $\langle tokens\rangle$ argument of $\backslash tl_rescan:nn$.

TeXhackers note: The $\langle tokens\rangle$ are first turned into a string (using $\backslash tl_to_str:n$). If the string contains one or more characters with character code $\backslash newlinechar$ (set equal to $\backslash endlinechar$ unless that is equal to 32, before the user $\langle setup\rangle$), 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. Only the case of a single line is supported in LuaTeX because of a bug in this engine.

5 Token list conditionals

```
\tl_if_blank_p:n   *
\tl_if_blank_p:(V|o) *
\tl_if_blank:nTF   *
\tl_if_blank:(V|o)TF *
```

```
\tl_if_blank_p:n {<token list>}
\tl_if_blank:nTF {<token list>} {<true code>} {<false code>}
```

Tests if the $\langle token\ list\rangle$ consists only of blank spaces (*i.e.* contains no item). The test is **true** if $\langle token\ list\rangle$ is zero or more explicit space characters (explicit tokens with character code 32 and category code 10), and is **false** otherwise.

\tl_if_empty_p:N ★ \tl_if_empty_p:N *tl var*
\tl_if_empty_p:c ★ \tl_if_empty:NTF *tl var* {*true code*} {*false code*}
\tl_if_empty:NTF ★ Tests if the *token list variable* is entirely empty (*i.e.* contains no tokens at all).
\tl_if_empty:cTF ★

\tl_if_empty_p:n ★ \tl_if_empty_p:n {*token list*}
\tl_if_empty_p:(V|o) ★ \tl_if_empty:nTF {*token list*} {*true code*} {*false code*}
\tl_if_empty:nTF ★ Tests if the *token list* is entirely empty (*i.e.* contains no tokens at all).
\tl_if_empty:(V|o)TF ★

New: 2012-05-24

Updated: 2012-06-05

\tl_if_eq_p:NN ★ \tl_if_eq_p:NN *tl var₁* *tl var₂*
\tl_if_eq_p:(Nc|cN|cc) ★ \tl_if_eq:NNTF *tl var₁* *tl var₂* {*true code*} {*false code*}
\tl_if_eq:NNTF ★ \tl_if_eq:(Nc|cN|cc)TF ★ 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**.

\tl_if_eq:nnTF ★ \tl_if_eq:nnTF {*token list₁*} {*token list₂*} {*true code*} {*false code*}

Tests if *token list₁* and *token list₂* contain the same list of tokens, both in respect of character codes and category codes.

\tl_if_in:NnTF ★ \tl_if_in:NnTF *tl var* {*token list*} {*true code*} {*false code*}
\tl_if_in:cNTF ★

Tests if the *token list* is found in the content of the *tl var*. The *token list* 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).

\tl_if_in:nnTF ★ \tl_if_in:nnTF {*token list₁*} {*token list₂*} {*true code*} {*false code*}
\tl_if_in:(Vn|on|no)TF ★

Tests if *token list₂* is found inside *token list₁*. The *token list₂* 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).

\tl_if_single_p:N ★ \tl_if_single_p:N *tl var*
\tl_if_single_p:c ★ \tl_if_single:NTF *tl var* {*true code*} {*false code*}
\tl_if_single:NTF ★ \tl_if_single:cTF ★ Tests if the content of the *tl var* consists of a single item, *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.
Updated: 2011-08-13

```
\tl_if_single_p:n ★          \tl_if_single_p:n {<token list>}
\tl_if_single:nTF ★          \tl_if_single:nTF {<token list>} {<true code>} {<false code>}
```

Updated: 2011-08-13

Tests if the *<token list>* has exactly one item, *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`.

```
\tl_case:Nn ★          \tl_case:NnTF <test token list variable>
\tl_case:cn ★          {
\tl_case:NnTF ★          <token list variable case1> {<code case1>}
\tl_case:cnTF ★          <token list variable case2> {<code case2>}
...                      ...
\tl_case:Nn ★          <token list variable casen> {<code casen>}
}
{<true code>}
{<false 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. 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.

6 Mapping to token lists

```
\tl_map_function:NN ★          \tl_map_function:NN <tl var> <function>
```

`\tl_map_function:cN ★`

Updated: 2012-06-29

Applies *<function>* to every *<item>* in the *<tl var>*. The *<function>* will receive one argument for each iteration. This may be a number of tokens if the *<item>* was stored within braces. Hence the *<function>* should anticipate receiving n-type arguments. See also `\tl_map_function:nN`.

```
\tl_map_function:nN ★          \tl_map_function:nN <token list> <function>
```

Updated: 2012-06-29

Applies *<function>* to every *<item>* in the *<token list>*. The *<function>* will receive one argument for each iteration. This may be a number of tokens if the *<item>* was stored within braces. Hence the *<function>* should anticipate receiving n-type arguments. See also `\tl_map_function:NN`.

```
\tl_map_inline:Nn
\tl_map_inline:cn
```

Updated: 2012-06-29

Applies the *<inline function>* to every *<item>* stored within the *<tl var>*. The *<inline function>* should consist of code which will receive the *<item>* as #1. One in line mapping can be nested inside another. See also `\tl_map_function:NN`.

```
\tl_map_inline:nn <token list> {<inline function>}
```

Updated: 2012-06-29

Applies the *<inline function>* to every *<item>* stored within the *<token list>*. The *<inline function>* should consist of code which will receive the *<item>* as #1. One in line mapping can be nested inside another. See also `\tl_map_function:nN`.

\tl_map_variable:Nn
\tl_map_variable:cNn

Updated: 2012-06-29

\tl_map_variable:NNn <tl var> <variable> {<function>}

Applies the *<function>* to every *<item>* stored within the *<tl var>*. The *<function>* should consist of code which will receive the *<item>* stored in the *<variable>*. One variable mapping can be nested inside another. See also \tl_map_inline:Nn.

\tl_map_variable:nNn

Updated: 2012-06-29

\tl_map_variable:nNn <token list> <variable> {<function>}

Applies the *<function>* to every *<item>* stored within the *<token list>*. The *<function>* should consist of code which will receive the *<item>* stored in the *<variable>*. One variable mapping can be nested inside another. See also \tl_map_inline:nn.

\tl_map_break: ☆

Updated: 2012-06-29

\tl_map_break:

Used to terminate a \tl_map_... function before all entries in the *<token list variable>* have been processed. This will normally take 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 will lead to low level TeX errors.

TeXhackers note: When the mapping is broken, additional tokens may be inserted by the internal macro __prg_break_point:Nn before the *<tokens>* are inserted into the input stream. This will depend on the design of the mapping function.

\tl_map_break:n ☆

Updated: 2012-06-29

\tl_map_break:n {<tokens>}

Used to terminate a \tl_map_... function before all entries in the *<token list variable>* have been processed, inserting the *<tokens>* after the mapping has ended. This will normally take place within a conditional statement, for example

```
\tl_map_inline:Nn \l_my_tl
{
  \str_if_eq:nnT { #1 } { bingo }
  { \tl_map_break:n { <tokens> } }
  % Do something useful
}
```

Use outside of a \tl_map_... scenario will lead to low level TeX errors.

TeXhackers note: When the mapping is broken, additional tokens may be inserted by the internal macro __prg_break_point:Nn before the *<tokens>* are inserted into the input stream. This will depend on the design of the mapping function.

7 Using token lists

\tl_to_str:n ★ \tl_to_str:v ★

\tl_to_str:n {*token list*}
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).

TeXhackers note: 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.

\tl_to_str:N ★ \tl_to_str:c ★

\tl_to_str:N {*tl var*}
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`.

\tl_use:N ★ \tl_use:c ★

\tl_use:N {*tl var*}
Recovering the content of a *tl var* and places it directly in the input stream. An error will be 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.

8 Working with the content of token lists

\tl_count:n ★ \tl_count:(V|o) ★

New: 2012-05-13

\tl_count:n {*tokens*}
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 will ignore any unprotected spaces within *tokens*. See also `\tl_count:N`. This function requires three expansions, giving an *integer denotation*.

\tl_count:N ★
\tl_count:c ★
New: 2012-05-13

\tl_count:N *tl var*

Counts the number of token groups in the *tl var* and leaves this information in the input stream. Unbraced tokens count as one element as do each token group ($\{ \dots \}$). This process will ignore any unprotected spaces within the *tl var*. See also \tl_count:n. This function requires three expansions, giving an *integer denotation*.

\tl_reverse:n ★
\tl_reverse:(V|o) ★
Updated: 2012-01-08

\tl_reverse:n *{token list}*

Reverses the order of the *items* in the *token list*, 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 will preserve 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.

TeXhackers note: The result is returned within \unexpanded, which means that the token list will not expand further when appearing in an x-type argument expansion.

\tl_reverse:N
\tl_reverse:c
\tl_greverse:N
\tl_greverse:c
Updated: 2012-01-08

\tl_reverse:N *tl var*

Reverses the order of the *items* stored in *tl var*, 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 will preserve 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. See also \tl_reverse:n, and, for improved performance, \tl_reverse_items:n.

\tl_reverse_items:n ★
New: 2012-01-08

\tl_reverse_items:n *{token list}*

Reverses the order of the *items* stored in *tl var*, 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 will remove any unprotected space within the *token list*. 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 \tl_reverse:n.

TeXhackers note: The result is returned within \unexpanded, which means that the token list will not expand further when appearing in an x-type argument expansion.

\tl_trim_spaces:n ★
New: 2011-07-09
Updated: 2012-06-25

\tl_trim_spaces:n *{token list}*

Removes any leading and trailing explicit space characters (explicit tokens with character code 32 and category code 10) from the *token list* and leaves the result in the input stream.

TeXhackers note: The result is returned within \unexpanded, which means that the token list will not expand further when appearing in an x-type argument expansion.

\tl_trim_spaces:N
\tl_trim_spaces:c
\tl_gtrim_spaces:N
\tl_gtrim_spaces:c
New: 2011-07-09

\tl_trim_spaces:N *tl var*

Removes any leading and trailing explicit space characters (explicit tokens with character code 32 and category code 10) from the content of the *tl var*. Note that this therefore *resets* the content of the variable.

```
\tl_sort:Nn
\tl_sort:cn
\tl_gsort:Nn
\tl_gsort:cn
```

New: 2017-02-06

```
\tl_sort:Nn <tl var> {<comparison code>}
```

Sorts the items in the *<tl var>* according to the *<comparison code>*, and assigns the result to *<tl var>*. The details of sorting comparison are described in Section 1.

```
\tl_sort:nN *
```

New: 2017-02-06

```
\tl_sort:nN {<token list>} {<conditional>}
```

Sorts the items in the *<token list>*, using the *<conditional>* to compare items, and leaves the result in the input stream. The *<conditional>* should have signature :mnTF, and return **true** if the two items being compared should be left in the same order, and **false** if the items should be swapped. The details of sorting comparison are described in Section 1.

TeXhackers note: The result is returned within `\exp_not:n`, which means that the token list will not expand further when appearing in an `x`-type argument expansion.

9 The first token from a token list

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 }
```

will both leave `a` in the input stream. If the “head” is a brace group, rather than a single token, the braces will be removed, and so

```
\tl_head:n { ~ { ~ ab } c }
```

yields `ab`. A blank *<token list>* (see `\tl_if_blank:nTF`) will result 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 will not expand further when appearing in an `x`-type argument expansion.

```
\tl_head:w ★ \tl_head:w ⟨token list⟩ { } \q_stop
```

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) will result in a low-level TEX 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.

```
\tl_tail:N ★ \tl_tail:n {⟨token list⟩}
```

```
\tl_tail:n ★
```

```
\tl_tail:(v|v|f) ★
```

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 }
```

will both leave `\{bc\}d` in the input stream. A blank *⟨token list⟩* (see `\tl_if_blank:nTF`) will result in `\tl_tail:n` leaving nothing in the input stream.

TEXhackers note: The result is returned within `\exp_not:n`, which means that the token list will not expand further when appearing in an x-type argument expansion.

```
\tl_if_head_eq_catcode_p:nN ★ \tl_if_head_eq_catcode_p:nN {⟨token list⟩} ⟨test token⟩  
\tl_if_head_eq_catcode:nNTF ★ \tl_if_head_eq_catcode:nNTF {⟨token list⟩} ⟨test token⟩  
{⟨true code⟩} {⟨false 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 will always be **false**.

```
\tl_if_head_eq_charcode_p:nN ★ \tl_if_head_eq_charcode_p:nN {⟨token list⟩} ⟨test token⟩  
\tl_if_head_eq_charcode_p:fN ★ \tl_if_head_eq_charcode:nNTF {⟨token list⟩} ⟨test token⟩  
\tl_if_head_eq_charcode:nNTF ★ {⟨true code⟩} {⟨false code⟩}  
\tl_if_head_eq_charcode:fNTF ★
```

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 will always be **false**.

```
\tl_if_head_eq_meaning_p:nN ★ \tl_if_head_eq_meaning_p:nN {⟨token list⟩} ⟨test token⟩  
\tl_if_head_eq_meaning:nNTF ★ \tl_if_head_eq_meaning:nNTF {⟨token list⟩} ⟨test token⟩  
{⟨true code⟩} {⟨false 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 will always be **false**.

```
\tl_if_head_is_group_p:n ★ \tl_if_head_is_group_p:n {\langle token list\rangle}
\tl_if_head_is_group:nTF ★ \tl_if_head_is_group:nTF {\langle token list\rangle} {\langle true code\rangle} {\langle false code\rangle}
```

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.

```
\tl_if_head_is_N_type_p:n ★ \tl_if_head_is_N_type_p:n {\langle token list\rangle}
\tl_if_head_is_N_type:nTF ★ \tl_if_head_is_N_type:nTF {\langle token list\rangle} {\langle true code\rangle} {\langle false code\rangle}
```

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.

```
\tl_if_head_is_space_p:n ★ \tl_if_head_is_space_p:n {\langle token list\rangle}
\tl_if_head_is_space:nTF ★ \tl_if_head_is_space:nTF {\langle token list\rangle} {\langle true code\rangle} {\langle false code\rangle}
```

Updated: 2012-07-08

Tests if the first *<token>* in the *<token list>* is an explicit space character (explicit token with character code 32 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.

10 Using a single item

```
\tl_item:nn ★ \tl_item:nn {\langle token list\rangle} {\langle integer expression\rangle}
\tl_item:Nn ★ \tl_item:Nn {\langle token list\rangle} {\langle integer expression\rangle}
\tl_item:cN ★ \tl_item:cN {\langle token list\rangle} {\langle integer expression\rangle}
```

New: 2014-07-17

Indexing items in the *<token list>* from 1 on the left, this function will evaluate the *<integer expression>* and leave 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.

TeXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the *<item>* will not expand further when appearing in an x-type argument expansion.

11 Viewing token lists

```
\tl_show:N \tl_show:N {\langle tl var\rangle}
\tl_show:c \tl_show:c {\langle tl var\rangle}
```

Updated: 2015-08-01

Displays the content of the *<tl var>* on the terminal.

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

`\tl_show:n`

Updated: 2015-08-07

`\tl_show:n <token list>`

Displays the *<token list>* on the terminal.

TEXhackers note: This is similar to the ε - \TeX primitive `\showtokens`, wrapped to a fixed number of characters per line.

`\tl_log:N`

`\tl_log:c`

New: 2014-08-22

Updated: 2015-08-01

`\tl_log:N <tl var>`

Writes the content of the *<tl var>* in the log file. See also `\tl_show:N` which displays the result in the terminal.

`\tl_log:n`

New: 2014-08-22

Updated: 2015-08-07

`\tl_log:n {<token list>}`

Writes the *<token list>* in the log file. See also `\tl_show:n` which displays the result in the terminal.

12 Constant token lists

`\c_empty_tl`

Constant that is always empty.

`\c_space_tl`

An explicit space character contained in a token list (compare this with `\c_space_token`). For use where an explicit space is required.

13 Scratch token lists

`\l_tmpa_tl`

`\l_tmpb_tl`

Scratch token lists for local assignment. These are never used by the kernel code, and so are safe for use with any $\text{\LaTeX}3$ -defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

`\g_tmpa_tl`

`\g_tmpb_tl`

Scratch token lists for global assignment. These are never used by the kernel code, and so are safe for use with any $\text{\LaTeX}3$ -defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

14 Internal functions

`_tl_trim_spaces:nn`

`_tl_trim_spaces:nn { \q_mark <token list> } {<continuation>}`

This function removes all leading and trailing explicit space characters from the *<token list>*, and expands to the *<continuation>*, followed by a brace group containing `\use_none:n \q_mark <trimmed token list>`. For instance, `\tl_trim_spaces:n` is implemented by taking the *<continuation>* to be `\exp_not:o`, and the o-type expansion removes the `\q_mark`. This function is also used in `\l3clist` and `\l3candidates`.

Part VII

The `\I3str` 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 will simply be 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 will not treat a token list or the corresponding string representation differently.

Note that as string variables are a special case of token list variables the coverage of `\str_...:N` functions is somewhat smaller than `\tl_...:N`.

The functions `\cs_to_str:N`, `\tl_to_str:n`, `\tl_to_str:N` and `\token_to_str:N` (and variants) will generate strings from the appropriate input: these are documented in `\I3basics`, `\I3tl` and `\I3token`, 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.

1 Building strings

`\str_new:N`
`\str_new:c`
New: 2015-09-18

`\str_new:N <str var>`

Creates a new `<str var>` or raises an error if the name is already taken. The declaration is global. The `<str var>` will initially be empty.

`\str_const:Nn`
`\str_const:(Nx|cn|cx)`
New: 2015-09-18

`\str_const:Nn <str var> {<token list>}`

Creates a new constant `<str var>` or raises an error if the name is already taken. The value of the `<str var>` will be set globally to the `<token list>`, converted to a string.

```
\str_clear:N  
\str_clear:c  
\str_gclear:N  
\str_gclear:c
```

New: 2015-09-18

```
\str_clear:N <str var>
```

Clears the content of the *<str var>*.

```
\str_clear_new:N  
\str_clear_new:c
```

New: 2015-09-18

```
\str_clear_new:N <str var>
```

Ensures that the *<str var>* exists globally by applying `\str_new:N` if necessary, then applies `\str_(g)clear:N` to leave the *<str var>* empty.

```
\str_set_eq:NN  
\str_set_eq:(cN|Nc|cc)  
\str_gset_eq:NN  
\str_gset_eq:(cN|Nc|cc)
```

New: 2015-09-18

```
\str_set_eq:NN <str var1> <str var2>
```

Sets the content of *<str var₁>* equal to that of *<str var₂>*.

2 Adding data to string variables

```
\str_set:Nn  
\str_set:(Nx|cn|cx)  
\str_gset:Nn  
\str_gset:(Nx|cn|cx)
```

New: 2015-09-18

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

New: 2015-09-18

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

New: 2015-09-18

```
\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.

2.1 String conditionals

```
\str_if_exist_p:N ★  
\str_if_exist_p:c ★  
\str_if_exist:NTF ★  
\str_if_exist:cTF ★
```

New: 2015-09-18

```
\str_if_exist_p:N <str var>
```

```
\str_if_exist:NTF <str var> {{true code}} {{false code}}
```

Tests whether the *<str var>* is currently defined. This does not check that the *<str var>* really is a string.

```
\str_if_empty_p:N ★ \sr_if_empty_p:N <str var>
\str_if_empty_p:c ★ \str_if_empty:NTF <str var> {{true code}} {{false code}}
\str_if_empty:NTF ★ Tests if the <string variable> is entirely empty (i.e. contains no characters at all).
\str_if_empty:cTF ★
```

New: 2015-09-18

```
\str_if_eq_p:NN ★ \str_if_eq_p:NN <str var1> <str var2>
\str_if_eq_p:(Nc|cN|cc) ★ \str_if_eq:NNTF <str var1> <str var2> {{true code}} {{false code}}
\str_if_eq:NNTF ★ Compares the content of two <str variables> and is logically true if the two contain the
\str_if_eq:(Nc|cN|cc)TF ★ same characters.
```

New: 2015-09-18

```
\str_if_eq_p:nn ★ \str_if_eq_p:nn {{tl1}} {{tl2}}
\str_if_eq_p:(Vn|on|no|nV|VV) ★ \str_if_eq:nnTF {{tl1}} {{tl2}} {{true code}} {{false code}}
\str_if_eq:nnTF ★
\str_if_eq:(Vn|on|no|nV|VV)TF ★
```

Compares the two <token lists> on a character by character basis, and is **true** if the two lists 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**.

```
\str_if_eq_x_p:nn ★ \str_if_eq_x_p:nn {{tl1}} {{tl2}}
\str_if_eq_x:nnTF ★ \str_if_eq_x:nnTF {{tl1}} {{tl2}} {{true code}} {{false code}}
```

New: 2012-06-05

Compares the full expansion of two <token lists> on a character by character basis, and is **true** if the two lists contain the same characters in the same order. Thus for example

```
\str_if_eq_x_p:nn { abc } { \tl_to_str:n { abc } }
```

is logically **true**.

```
\str_case:nn ★ \str_case:nnTF {{test string}}
\str_case:(on|nV|nv) ★ {
\str_case:nnTF ★   {{string case1}} {{code case1}}
\str_case:(on|nV|nv)TF ★   {{string case2}} {{code case2}}
...
{{string casen}} {{code casen}}
}
{{true code}}
{{false code}}
```

Updated: 2015-02-28

This function compares the <test string> in turn with each of the <string cases>. If the two are equal (as described for \str_if_eq:nnTF then the associated <code> is left in the input stream. 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.

\str_case_x:nnTF *

New: 2013-07-24

```
\str_case_x:nnTF {\⟨test string⟩}
{
  {⟨string case1⟩} {⟨code case1⟩}
  {⟨string case2⟩} {⟨code case2⟩}
  ...
  {⟨string casen⟩} {⟨code casen⟩}
}
{⟨true code⟩}
{⟨false code⟩}
```

This function compares the full expansion of the *⟨test string⟩* in turn with the full expansion of the *⟨string cases⟩*. If the two full expansions are equal (as described for \str_if_eq:nnTF then the associated *⟨code⟩* is left in the input stream. 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_x: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.

3 Working with the content of strings

\str_use:N *

\str_use:c *

New: 2015-09-18

\str_use:N {\⟨str var⟩}
\str_use:c
Recovers the content of a *⟨str var⟩* and places it directly in the input stream. An error will be 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.

\str_count:N * \str_count:n {\⟨token list⟩}
\str_count:c *
\str_count:n *
\str_count_ignore_spaces:n *

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:n {\⟨token list⟩}
\str_count_spaces:c *
\str_count_spaces:n *

New: 2015-09-18

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.

```
\str_head:N      * \str_head:n {\<token list>}
\str_head:c      *
\str_head:n      *
\str_head_ignore_spaces:n *
```

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.

```
\str_tail:N      * \str_tail:n {\<token list>}
\str_tail:c      *
\str_tail:n      *
\str_tail_ignore_spaces:n *
```

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` will trim only 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.

```
\str_item:Nn      * \str_item:nn {\<token list>} {\<integer expression>}
\str_item:nn      *
\str_item_ignore_spaces:nn *
```

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 *token list* to a *string*, and leaves in the input stream the characters from the *start index* to the *end index* inclusive. Positive *indices* are counted from the start of the string, 1 being the first character, and negative *indices* are counted from the end of the string, -1 being the last character. If either of *start index* or *end index* is 0, the result is empty. 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 } }
```

will print bcde, cdef, ef, and an empty line to the terminal. The *start index* must always be smaller than or equal to the *end index*: 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.

4 String manipulation

```
\str_lower_case:n ★ \str_lower_case:n {⟨tokens⟩}
\str_lower_case:f ★ \str_upper_case:n {⟨tokens⟩}
```

Converts the input ⟨tokens⟩ 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`.

New: 2015-03-01

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:cpx
    {
        user
        \str_upper_case:f { \tl_head:n {#1} }
        \str_lower_case: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_fold_case:n` for this situation (case folding is distinct from lower casing).
- Case changing text for typesetting: see the `\tl_lower_case:n(n)`, `\tl_upper_case:n(n)` and `\tl_mixed_case: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_fold_case: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 will be case-folded (*i.e.* the ASCII range which coincides with UTF-8). Full UTF-8 support is available with both XeTeX and LuaTeX, subject only to the fact that XeTeX in particular has issues with characters of code above hexadecimal 0xFFFF when interacting with `\tl_to_str:n`.

```
\str_fold_case:n ★  
\str_fold_case:v ★
```

New: 2014-06-19
Updated: 2016-03-07

```
\str_fold_case:n {⟨tokens⟩}
```

Converts the input ⟨tokens⟩ to their string representation, as described for `\tl_to_str:n`, and then folds the case of the resulting ⟨string⟩ 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_fold_case: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_fold_case: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_fold_case: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 will be case-folded (*i.e.* the ASCII range which coincides with UTF-8). Full UTF-8 support is available with both `XeTeX` and `LuaTeX`, subject only to the fact that `XeTeX` in particular has issues with characters of code above hexadecimal 0xFFFF when interacting with `\tl_to_str:n`.

5 Viewing strings

```
\str_show:N  
\str_show:c  
\str_show:n
```

New: 2015-09-18

```
\str_show:N ⟨str var⟩
```

Displays the content of the ⟨str var⟩ on the terminal.

6 Constant token lists

```
\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
```

Constant strings, containing a single character token, with category code 12.

New: 2015-09-19

7 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 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.

```
\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 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.

7.1 Internal string functions

```
\__str_if_eq_x:nn *
```

```
\__str_if_eq_x:nn {\langle t l1 ⟩} {\langle t l2 ⟩}
```

Compares the full expansion of two *<token lists>* on a character by character basis, and is **true** if the two lists contain the same characters in the same order. Leaves 0 in the input stream if the condition is true, and +1 or -1 otherwise.

```
\__str_if_eq_x_return:nn
```

```
\__str_if_eq_x_return:nn {\langle t l1 ⟩} {\langle t l2 ⟩}
```

Compares the full expansion of two *<token lists>* on a character by character basis, and is **true** if the two lists contain the same characters in the same order. Either **\prg_return_true:** or **\prg_return_false:** is then left in the input stream. This is a version of **\str_if_eq_x:nnTF** coded for speed.

```
\__str_to_other:n *
```

```
\__str_to_other:n {\langle token list ⟩}
```

Converts the *<token list>* to a *<other string>*, 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.

`__str_to_other_fast:n` ★

`__str_to_other_fast:n {<token list>}`

Same behaviour `__str_to_other:n` but only restricted-expandable. It takes a time linear in the character count of the string. It is used for `\iow_wrap:nnnN`.

`__str_count:n` ★

`__str_count:n {<other string>}`

This function expects an argument that is entirely made of characters with category “other”, as produced by `__str_to_other:n`. It leaves in the input stream the number of character tokens in the `<other string>`, faster than the analogous `\str_count:n` function.

`__str_range:nnn` ★

`__str_range:nnn {<other string>} {<start index>} {<end index>}`

Identical to `\str_range:nnn` except that the first argument is expected to be entirely made of characters with category “other”, as produced by `__str_to_other:n`, and the result is also an `<other string>`.

Part VIII

The **l3seq** package

Sequences and stacks

LATEX3 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 LATEX3. This is achieved using a number of dedicated stack functions.

1 Creating and initialising sequences

`\seq_new:N`
`\seq_new:c`

Creates a new *(sequence)* or raises an error if the name is already taken. The declaration is global. The *(sequence)* will initially contain no items.

`\seq_clear:N`
`\seq_clear:c`
`\seq_gclear:N`
`\seq_gclear:c`

Clears all items from the *(sequence)*.

`\seq_clear_new:N`
`\seq_clear_new:c`
`\seq_gclear_new:N`
`\seq_gclear_new:c`

Ensures that the *(sequence)* exists globally by applying `\seq_new:N` if necessary, then applies `\seq_(g)clear:N` to leave the *(sequence)* empty.

`\seq_set_eq:NN`
`\seq_set_eq:(cN|Nc|cc)`
`\seq_gset_eq:NN`
`\seq_gset_eq:(cN|Nc|cc)`

Sets the content of *(sequence₁)* equal to that of *(sequence₂)*.

<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.

```
\seq_set_split:Nnn
\seq_set_split:NnV
\seq_gset_split:Nnn
\seq_gset_split:NnV
```

New: 2011-08-15
Updated: 2012-07-02

```
\seq_set_split:Nnn <sequence> {(delimiter)} {{token list}}
```

Splits the *<sequence>* 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 `I3clist` 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>*.

```
\seq_concat:NNN
\seq_concat:ccc
\seq_gconcat:NNN
\seq_gconcat:ccc
```

```
\seq_concat:NNN <sequence1> <sequence2> <sequence3>
```

Concatenates the content of *<sequence2>* and *<sequence3>* together and saves the result in *<sequence1>*. The items in *<sequence2>* will be placed at the left side of the new sequence.

```
\seq_if_exist_p:N *
\seq_if_exist_p:c *
\seq_if_exist:NTF *
\seq_if_exist:cTF *
```

New: 2012-03-03

```
\seq_if_exist_p:N <sequence>
\seq_if_exist:NTF <sequence> {{true code}} {{false code}}
```

Tests whether the *<sequence>* is currently defined. This does not check that the *<sequence>* really is a sequence variable.

```
\seq_put_left:Nn
\seq_put_left:(NV|Nv|No|Nx|cn|cV|cv|co|cx)
\seq_gput_left:Nn
\seq_gput_left:(NV|Nv|No|Nx|cn|cV|cv|co|cx)
```

```
\seq_put_left:Nn <sequence> {{item}}
```

Appends the *<item>* to the left of the *<sequence>*.

```
\seq_put_right:Nn
\seq_put_right:(NV|Nv|No|Nx|cn|cV|cv|co|cx)
\seq_gput_right:Nn
\seq_gput_right:(NV|Nv|No|Nx|cn|cV|cv|co|cx)
```

```
\seq_put_right:Nn <sequence> {{item}}
```

Appends the *<item>* to the right of the *<sequence>*.

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 *<token list variable>* used with `\tl_set:Nn` and *never* `\tl_gset:Nn`.

```
\seq_get_left:NN
\seq_get_left:cN
```

Updated: 2012-05-14

```
\seq_get_left:NN <sequence> <token list variable>
```

Stores the left-most item from a *<sequence>* in the *<token list variable>* without removing it from the *<sequence>*. The *<token list variable>* is assigned locally. If *<sequence>* is empty the *<token list variable>* will contain the special marker `\q_no_value`.

```
\seq_get_right:NN  
\seq_get_right:cN
```

Updated: 2012-05-19

```
\seq_get_right:NN <sequence> <token list variable>
```

Stores the right-most item from a *<sequence>* in the *<token list variable>* without removing it from the *<sequence>*. The *<token list variable>* is assigned locally. If *<sequence>* is empty the *<token list variable>* will contain the special marker `\q_no_value`.

```
\seq_pop_left:NN  
\seq_pop_left:cN
```

Updated: 2012-05-14

```
\seq_pop_left:NN <sequence> <token list variable>
```

Pops the left-most item from a *<sequence>* into the *<token list variable>*, i.e. removes the item from the sequence and stores it in the *<token list variable>*. Both of the variables are assigned locally. If *<sequence>* is empty the *<token list variable>* will contain the special marker `\q_no_value`.

```
\seq_gpop_left:NN  
\seq_gpop_left:cN
```

Updated: 2012-05-14

```
\seq_gpop_left:NN <sequence> <token list variable>
```

Pops the left-most item from a *<sequence>* into the *<token list variable>*, i.e. removes the item from the sequence and stores it in the *<token list variable>*. The *<sequence>* is modified globally, while the assignment of the *<token list variable>* is local. If *<sequence>* is empty the *<token list variable>* will contain the special marker `\q_no_value`.

```
\seq_pop_right:NN  
\seq_pop_right:cN
```

Updated: 2012-05-19

```
\seq_pop_right:NN <sequence> <token list variable>
```

Pops the right-most item from a *<sequence>* into the *<token list variable>*, i.e. removes the item from the sequence and stores it in the *<token list variable>*. Both of the variables are assigned locally. If *<sequence>* is empty the *<token list variable>* will contain the special marker `\q_no_value`.

```
\seq_gpop_right:NN  
\seq_gpop_right:cN
```

Updated: 2012-05-19

```
\seq_gpop_right:NN <sequence> <token list variable>
```

Pops the right-most item from a *<sequence>* into the *<token list variable>*, i.e. removes the item from the sequence and stores it in the *<token list variable>*. The *<sequence>* is modified globally, while the assignment of the *<token list variable>* is local. If *<sequence>* is empty the *<token list variable>* will contain the special marker `\q_no_value`.

```
\seq_item:Nn ★  
\seq_item:cN ★
```

New: 2014-07-17

```
\seq_item:Nn <sequence> {\<integer expression>}
```

Indexing items in the *<sequence>* from 1 at the top (left), this function will evaluate the *<integer expression>* and leave the appropriate item from the sequence in the input stream. If the *<integer expression>* is negative, indexing occurs from the bottom (right) of the sequence. When the *<integer expression>* is larger than the number of items in the *<sequence>* (as calculated by `\seq_count:N`) then the function will expand to nothing.

TEXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the *<item>* will not expand further when appearing in an x-type argument expansion.

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.

```
\seq_get_left:NNTF
\seq_get_left:cNTF
```

New: 2012-05-14
Updated: 2012-05-19

```
\seq_get_left:NNTF <sequence> <token list variable> {{true code}} {{false code}}
```

If the *<sequence>* is empty, leaves the *<false code>* in the input stream. The value of the *<token list variable>* is not defined in this case and should not be relied upon. If the *<sequence>* is non-empty, stores the left-most item from a *<sequence>* in the *<token list variable>* without removing it from a *<sequence>*. The *<token list variable>* is assigned locally.

```
\seq_get_right:NNTF
\seq_get_right:cNTF
```

New: 2012-05-19

```
\seq_get_right:NNTF <sequence> <token list variable> {{true code}} {{false code}}
```

If the *<sequence>* is empty, leaves the *<false code>* in the input stream. The value of the *<token list variable>* is not defined in this case and should not be relied upon. If the *<sequence>* is non-empty, stores the right-most item from a *<sequence>* in the *<token list variable>* without removing it from a *<sequence>*. The *<token list variable>* is assigned locally.

```
\seq_pop_left:NNTF
\seq_pop_left:cNTF
```

New: 2012-05-14
Updated: 2012-05-19

```
\seq_pop_left:NNTF <sequence> <token list variable> {{true code}} {{false code}}
```

If the *<sequence>* is empty, leaves the *<false code>* in the input stream. The value of the *<token list variable>* is not defined in this case and should not be relied upon. If the *<sequence>* is non-empty, pops the left-most item from a *<sequence>* in the *<token list variable>*, i.e. removes the item from a *<sequence>*. Both the *<sequence>* and the *<token list variable>* are assigned locally.

```
\seq_gpop_left:NNTF
\seq_gpop_left:cNTF
```

New: 2012-05-14
Updated: 2012-05-19

```
\seq_gpop_left:NNTF <sequence> <token list variable> {{true code}} {{false code}}
```

If the *<sequence>* is empty, leaves the *<false code>* in the input stream. The value of the *<token list variable>* is not defined in this case and should not be relied upon. If the *<sequence>* is non-empty, pops the left-most item from a *<sequence>* in the *<token list variable>*, i.e. removes the item from a *<sequence>*. The *<sequence>* is modified globally, while the *<token list variable>* is assigned locally.

```
\seq_pop_right:NNTF
\seq_pop_right:cNTF
```

New: 2012-05-19

```
\seq_pop_right:NNTF <sequence> <token list variable> {{true code}} {{false code}}
```

If the *<sequence>* is empty, leaves the *<false code>* in the input stream. The value of the *<token list variable>* is not defined in this case and should not be relied upon. If the *<sequence>* is non-empty, pops the right-most item from a *<sequence>* in the *<token list variable>*, i.e. removes the item from a *<sequence>*. Both the *<sequence>* and the *<token list variable>* are assigned locally.

```
\seq_gpop_right:NNTF
\seq_gpop_right:cNTF
```

New: 2012-05-19

```
\seq_gpop_right:NNTF <sequence> <token list variable> {{true code}} {{false code}}
```

If the *<sequence>* is empty, leaves the *<false code>* in the input stream. The value of the *<token list variable>* is not defined in this case and should not be relied upon. If the *<sequence>* is non-empty, pops the right-most item from a *<sequence>* in the *<token list variable>*, i.e. removes the item from a *<sequence>*. The *<sequence>* is modified globally, while the *<token list variable>* is assigned locally.

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 <sequence>
```

Removes duplicate items from the *<sequence>*, leaving the left most copy of each item in the *<sequence>*. The *<item>* comparison takes place on a token basis, as for `\tl_if_eq:nnTF`.

TeXhackers note: This function iterates through every item in the *<sequence>* and does a comparison with the *<items>* 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 <sequence> {<item>}
```

Removes every occurrence of *<item>* from the *<sequence>*. The *<item>* 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
```

New: 2014-07-18

```
\seq_reverse:N <sequence>
```

Reverses the order of the items stored in the *<sequence>*.

```
\seq_sort:Nn  
\seq_sort:cn  
\seq_gsort:Nn  
\seq_gsort:cn
```

New: 2017-02-06

```
\seq_sort:Nn <sequence> {{comparison code}}
```

Sorts the items in the *<sequence>* according to the *<comparison code>*, and assigns the result to *<sequence>*. The details of sorting comparison are described in Section 1.

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 <sequence>
```

```
\seq_if_empty:NTF <sequence> {{true code}} {{false code}}
```

Tests if the *<sequence>* is empty (containing no items).

```
\seq_if_in:NnTF  
\seq_if_in:(NV|Nv|No|Nx|cn|cV|cv|co|cx)TF
```

```
\seq_if_in:NnTF <sequence> {{<item>}} {{true code}} {{false code}}
```

Tests if the *<item>* is present in the *<sequence>*.

7 Mapping to sequences

```
\seq_map_function:NN ★  
\seq_map_function:cN ★
```

Updated: 2012-06-29

```
\seq_map_function:NN <sequence> <function>
```

Applies *<function>* to every *<item>* stored in the *<sequence>*. The *<function>* will receive one argument for each iteration. The *<items>* are returned from left to right. The function `\seq_map_inline:Nn` is faster than `\seq_map_function:NN` for sequences with more than about 10 items. One mapping may be nested inside another.

`\seq_map_inline:Nn`
`\seq_map_inline:cn`

Updated: 2012-06-29

`\seq_map_inline:Nn <sequence> {<inline function>}`

Applies *<inline function>* to every *<item>* stored within the *<sequence>*. The *<inline function>* should consist of code which will receive the *<item>* as #1. One in line mapping can be nested inside another. The *<items>* are returned from left to right.

`\seq_map_variable>NNn`
`\seq_map_variable:(Ncn|cNn|ccn)`

Updated: 2012-06-29

`\seq_map_variable>NNn <sequence> {tl var.} {function using tl var.}`

Stores each entry in the *<sequence>* in turn in the *{tl var.}* and applies the *{function using tl var.}* The *{function}* will usually consist of code making use of the *{tl var.}*, but this is not enforced. One variable mapping can be nested inside another. The *<items>* are returned from left to right.

`\seq_map_break: ☆`

Updated: 2012-06-29

`\seq_map_break:`

Used to terminate a `\seq_map_...` function before all entries in the *<sequence>* have been processed. This will normally take 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 will lead to low level T_EX errors.

TEXhackers note: When the mapping is broken, additional tokens may be inserted by the internal macro `__prg_break_point:Nn` before further items are taken from the input stream. This will depend on the design of the mapping function.

\seq_map_break:n ★Updated: 2012-06-29

\seq_map_break:n {<tokens>}

Used to terminate a `\seq_map_...` function before all entries in the `<sequence>` have been processed, inserting the `<tokens>` after the mapping has ended. This will normally take place within a conditional statement, for example

```
\seq_map_inline:Nn \l_my_seq
{
  \str_if_eq:nnTF { #1 } { bingo }
    { \seq_map_break:n { <tokens> } }
  {
    % Do something useful
  }
}
```

Use outside of a `\seq_map_...` scenario will lead to low level T_EX errors.

TeXhackers note: When the mapping is broken, additional tokens may be inserted by the internal macro `__prg_break_point:Nn` before the `<tokens>` are inserted into the input stream. This will depend on the design of the mapping function.

\seq_count:N ★**\seq_count:c** ★New: 2012-07-13

\seq_count:N <sequence>

Leaves the number of items in the `<sequence>` in the input stream as an `<integer denotation>`. The total number of items in a `<sequence>` will include those which are empty and duplicates, *i.e.* every item in a `<sequence>` is unique.

8 Using the content of sequences directly

\seq_use:Nnnn ★**\seq_use:cnnn** ★New: 2013-05-26

\seq_use:Nnnn <seq var> {<separator between two>}

{<separator between more than two>} {<separator between final two>}

Places the contents of the `<seq var>` in the input stream, with the appropriate `<separator>` between the items. Namely, if the sequence 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 sequence has exactly two items, then they are placed in the input stream separated by the `<separator between two>`. If the sequence has a single item, it is placed in the input stream, and an empty sequence produces no output. An error will be 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~ }
```

will insert “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.

TeXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the `<items>` will not expand further when appearing in an x-type argument expansion.

\seq_use:Nn ★
\seq_use:cN ★
New: 2013-05-26

\seq_use:Nn *seq var* {*separator*}

Places the contents of the *seq var* in the input stream, with the *separator* between the items. If the sequence has a single item, it is placed in the input stream with no *separator*, and an empty sequence produces no output. An error will be 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~ }
```

will insert “a and b and c and de and f” in the input stream.

TEXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the *items* will not expand further when appearing in an x-type argument expansion.

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 *sequence* <token list variable>

Reads the top item from a *sequence* into the *<token list variable>* without removing it from the *sequence*. The *<token list variable>* is assigned locally. If *sequence* is empty the *<token list variable>* will contain the special marker `\q_no_value`.

\seq_pop:NN
\seq_pop:cN
Updated: 2012-05-14

\seq_pop:NN *sequence* <token list variable>

Pops the top item from a *sequence* into the *<token list variable>*. Both of the variables are assigned locally. If *sequence* is empty the *<token list variable>* will contain the special marker `\q_no_value`.

\seq_gpop:NN
\seq_gpop:cN
Updated: 2012-05-14

\seq_gpop:NN *sequence* <token list variable>

Pops the top item from a *sequence* into the *<token list variable>*. The *sequence* is modified globally, while the *<token list variable>* is assigned locally. If *sequence* is empty the *<token list variable>* will contain the special marker `\q_no_value`.

\seq_get:NNTF
\seq_get:cNTF
New: 2012-05-14
Updated: 2012-05-19

\seq_get:NNTF *sequence* <token list variable> {*true code*} {*false code*}

If the *sequence* is empty, leaves the *<false code>* in the input stream. The value of the *<token list variable>* is not defined in this case and should not be relied upon. If the *sequence* is non-empty, stores the top item from a *sequence* in the *<token list variable>* without removing it from the *sequence*. The *<token list variable>* is assigned locally.

\seq_pop:NNTF
\seq_pop:cNTF

New: 2012-05-14
Updated: 2012-05-19

\seq_pop:NNTF *sequence* {*token list variable*} {*true code*} {*false code*}

If the *sequence* is empty, leaves the *false code* in the input stream. The value of the *token list variable* is not defined in this case and should not be relied upon. If the *sequence* is non-empty, pops the top item from the *sequence* in the *token list variable*, i.e. removes the item from the *sequence*. Both the *sequence* and the *token list variable* are assigned locally.

\seq_gpop:NNTF
\seq_gpop:cNTF

New: 2012-05-14
Updated: 2012-05-19

\seq_gpop:NNTF *sequence* {*token list variable*} {*true code*} {*false code*}

If the *sequence* is empty, leaves the *false code* in the input stream. The value of the *token list variable* is not defined in this case and should not be relied upon. If the *sequence* is non-empty, pops the top item from the *sequence* in the *token list variable*, i.e. removes the item from the *sequence*. The *sequence* is modified globally, while the *token list variable* 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 *sequence* {*item*}

Adds the {*item*} to the top of the *sequence*.

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 *sequence variable* only has distinct items, use \seq_remove_duplicates:N *sequence variable*. This function is relatively slow, and to avoid performance issues one should only use it when necessary.

Some operations on a set *seq var* are straightforward. For instance, \seq_count:N *seq var* expands to the number of items, while \seq_if_in:NnTF *seq var* {*item*} tests if the *item* is in the set.

Adding an *item* to a set *seq var* can be done by appending it to the *seq var* if it is not already in the *seq var*:

```
\seq_if_in:NnF seq var {item}  
{ \seq_put_right:Nn seq var {item} }
```

Removing an *item* from a set *seq var* can be done using \seq_remove_all:Nn,

```
\seq_remove_all:Nn seq var {item}
```

The intersection of two sets *seq var₁* and *seq var₂* can be stored into *seq var₃* by collecting items of *seq var₁* which are in *seq var₂*.

```

\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 $\langle \text{seq var}_3 \rangle$ is different from the other two sequence variables. To cover all cases, items should first be collected in a sequence $\backslash \text{l_}\langle \text{pkg} \rangle\text{_internal_seq}$, then $\langle \text{seq var}_3 \rangle$ should be set equal to this internal sequence. The same remark applies to other set functions.

The union of two sets $\langle \text{seq var}_1 \rangle$ and $\langle \text{seq var}_2 \rangle$ can be stored into $\langle \text{seq var}_3 \rangle$ through

```

\seq_concat:NNN <seq var3> <seq var1> <seq var2>
\seq_remove_duplicates:N <seq var3>

```

or by adding items to (a copy of) $\langle \text{seq var}_1 \rangle$ 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 $\langle \text{seq var}_2 \rangle$ is short compared to $\langle \text{seq var}_1 \rangle$.

The difference of two sets $\langle \text{seq var}_1 \rangle$ and $\langle \text{seq var}_2 \rangle$ can be stored into $\langle \text{seq var}_3 \rangle$ by removing items of the $\langle \text{seq var}_2 \rangle$ from (a copy of) the $\langle \text{seq var}_1 \rangle$ 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 $\langle \text{seq var}_1 \rangle$ and $\langle \text{seq var}_2 \rangle$ can be stored into $\langle \text{seq var}_3 \rangle$ by computing the difference between $\langle \text{seq var}_1 \rangle$ and $\langle \text{seq var}_2 \rangle$ and storing the result as $\backslash \text{l_}\langle \text{pkg} \rangle\text{_internal_seq}$, then the difference between $\langle \text{seq var}_2 \rangle$ and $\langle \text{seq var}_1 \rangle$, and finally concatenating the two differences to get the symmetric differences.

```

\seq_set_eq:NN \backslash \text{l\_}\langle \text{pkg} \rangle\text{\_internal\_seq} <seq var1>
\seq_map_inline:Nn <seq var2>
{ \seq_remove_all:Nn \backslash \text{l\_}\langle \text{pkg} \rangle\text{\_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> \backslash \text{l\_}\langle \text{pkg} \rangle\text{\_internal\_seq}

```

11 Constant and scratch sequences

\c_empty_seq Constant that is always empty.

New: 2012-07-02

`\l_tmpa_seq`
`\l_tmpb_seq`
New: 2012-04-26

Scratch sequences 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.

`\g_tmpa_seq`
`\g_tmpb_seq`
New: 2012-04-26

Scratch sequences 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.

12 Viewing sequences

`\seq_show:N`
`\seq_show:c`
Updated: 2015-08-01

`\seq_show:N <sequence>`
Displays the entries in the *<sequence>* in the terminal.

`\seq_log:N`
`\seq_log:c`
New: 2014-08-12
Updated: 2015-08-01

`\seq_log:N <sequence>`
Writes the entries in the *<sequence>* in the log file.

13 Internal sequence functions

`\s_seq`

This scan mark (equal to `\scan_stop:`) marks the beginning of a sequence variable.

`_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 will be raised. The definition should always be set globally.

`_seq_push_item_def:n`
`_seq_push_item_def:x`

`_seq_push_item_def:n {<code>}`

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`.

Part IX

The **I3int** 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`”).

1 Integer expressions

`\int_eval:n` ★ `\int_eval:n {<integer expression>}`

Evaluates the *<integer expression>*, expanding any integer and token list variables within the *<expression>* to their content (without requiring `\int_use:N`/`\tl_use:N`) and applying the standard mathematical rules. For example both

```
\int_eval: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_eval:n { \l_my_tl + \l_my_int * 3 - ( 3 + 4 * 5 ) }
```

both evaluate to -6 . The `{<integer expression>}` may contain the operators `+`, `-`, `*` and `/`, along with parenthesis `(` and `)`. Any functions within the expressions should expand to an *<integer denotation>*: a sequence of a sign and digits matching the regex `\-?[0-9]+`. After expansion `\int_eval:n` yields an *<integer denotation>* which is left in the input stream.

TEXhackers note: Exactly two expansions are needed to evaluate `\int_eval:n`. The result is *not* an *<internal integer>*, and therefore requires suitable termination if used in a TeX-style integer assignment.

`\int_abs:n` ★ `\int_abs:n {<integer expression>}`

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.

`\int_div_round:nn` ★ `\int_div_round:nn {<intexpr1>} {<intexpr2>}`

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.

`\int_div_truncate:nn` *

Updated: 2012-02-09

`\int_div_truncate:nn {⟨intexpr₁⟩} {⟨intexpr₂⟩}`

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.

`\int_max:nn` *

`\int_min:nn` *

Updated: 2012-09-26

`\int_max:nn {⟨intexpr₁⟩} {⟨intexpr₂⟩}`
`\int_min:nn {⟨intexpr₁⟩} {⟨intexpr₂⟩}`

Evaluates the *⟨integer expressions⟩* as described for `\int_eval:n` and leaves either the larger or smaller value in the input stream as an *⟨integer denotation⟩* after two expansions.

`\int_mod:nn` *

Updated: 2012-09-26

`\int_mod:nn {⟨intexpr₁⟩} {⟨intexpr₂⟩}`

Evaluates the two *⟨integer expressions⟩* as described earlier, then calculates the integer remainder of dividing the first expression by the second. This is obtained by subtracting `\int_div_truncate:nn {⟨intexpr₁⟩} {⟨intexpr₂⟩}` times *⟨intexpr₂⟩* from *⟨intexpr₁⟩*. Thus, the result has the same sign as *⟨intexpr₁⟩* and its absolute value is strictly less than that of *⟨intexpr₂⟩*. The result is left in the input stream as an *⟨integer denotation⟩* after two expansions.

2 Creating and initialising integers

`\int_new:N`

`\int_new:c`

`\int_new:N ⟨integer⟩`

Creates a new *⟨integer⟩* or raises an error if the name is already taken. The declaration is global. The *⟨integer⟩* will initially be equal to 0.

`\int_const:Nn`

`\int_const:cn`

Updated: 2011-10-22

`\int_const:Nn ⟨integer⟩ {⟨integer expression⟩}`

Creates a new constant *⟨integer⟩* or raises an error if the name is already taken. The value of the *⟨integer⟩* will be set globally to the *⟨integer expression⟩*.

`\int_zero:N`

`\int_zero:c`

`\int_gzero:N`

`\int_gzero:c`

`\int_zero:N ⟨integer⟩`

Sets *⟨integer⟩* to 0.

`\int_zero_new:N`

`\int_zero_new:c`

`\int_gzero_new:N`

`\int_gzero_new:c`

New: 2011-12-13

`\int_zero_new:N ⟨integer⟩`

Ensures that the *⟨integer⟩* exists globally by applying `\int_new:N` if necessary, then applies `\int_(g)zero:N` to leave the *⟨integer⟩* set to zero.

`\int_set_eq:NN`

`\int_set_eq:(cN|Nc|cc)`

`\int_gset_eq:NN`

`\int_gset_eq:(cN|Nc|cc)`

`\int_set_eq:NN ⟨integer₁⟩ ⟨integer₂⟩`

Sets the content of *⟨integer₁⟩* equal to that of *⟨integer₂⟩*.

```
\int_if_exist_p:N ★ \int_if_exist_p:N <int>
\int_if_exist_p:c ★ \int_if_exist:NTF <int> {<true code>} {<false code>}
\int_if_exist:NTF ★ Tests whether the <int> is currently defined. This does not check that the <int> really is
\int_if_exist:cTF ★ an integer variable.
```

New: 2012-03-03

3 Setting and incrementing integers

```
\int_add:Nn \int_add:Nn <integer> {<integer expression>}
\int_add:cn Adds the result of the <integer expression> to the current content of the <integer>.
```

\int_gadd:Nn
\int_gadd:cn

Updated: 2011-10-22

```
\int_decr:N \int_decr:N <integer>
\int_decr:c Decreases the value stored in <integer> by 1.
```

\int_gdecr:N
\int_gdecr:c

```
\int_incr:N \int_incr:N <integer>
\int_incr:c Increases the value stored in <integer> by 1.
```

\int_gincr:N
\int_gincr:c

```
\int_set:Nn \int_set:Nn <integer> {<integer expression>}
\int_set:cn Sets <integer> to the value of <integer expression>, which must evaluate to an integer (as
\int_gset:Nn described for \int_eval:n).
```

\int_gset:cn

Updated: 2011-10-22

```
\int_sub:Nn \int_sub:Nn <integer> {<integer expression>}
\int_sub:cn Subtracts the result of the <integer expression> from the current content of the <integer>.
```

\int_gsub:Nn
\int_gsub:cn

Updated: 2011-10-22

4 Using integers

```
\int_use:N ★ \int_use:N <integer>
\int_use:c ★ Recovers the content of an <integer> and places it directly in the input stream. An
Updated: 2011-10-22 error will be 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.

5 Integer expression conditionals

```
\int_compare_p:nNn *
\int_compare:nNnTF *
```

```
\int_compare_p:nNn {\langle intexpr_1\rangle} {\langle relation\rangle} {\langle intexpr_2\rangle}
\int_compare:nNnTF
  {\langle intexpr_1\rangle} {\langle relation\rangle} {\langle intexpr_2\rangle}
  {\langle true code\rangle} {\langle false code\rangle}
```

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	<

```
\int_compare_p:n *
\int_compare:nTF *
```

Updated: 2013-01-13

```
\int_compare_p:n
{
  \intexpr_1 {\relation_1}
  ...
  \intexpr_N {\relation_N}
  \intexpr_{N+1}
}

\int_compare:nTF
{
  \intexpr_1 {\relation_1}
  ...
  \intexpr_N {\relation_N}
  \intexpr_{N+1}
}
{\true code} {\false code}
```

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	!=

```
\int_case:nn   ★ \int_case:nnTF {\langle test integer expression\rangle}
\int_case:nnTF ★
{
  {\langle intexpr case1\rangle} {\langle code case1\rangle}
  {\langle intexpr case2\rangle} {\langle code case2\rangle}
  ...
  {\langle intexpr casen\rangle} {\langle code casen\rangle}
}
{\true code}
{\false code}
```

New: 2013-07-24

This function evaluates the $\langle test\ integer\ expression\rangle$ and compares this in turn to each of the $\langle integer\ expression\ cases\rangle$. If the two are equal then the associated $\langle code\rangle$ is left in the input stream. 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 function $\backslash int_case:nn$, 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! }
```

will leave “Medium” in the input stream.

```
\int_if_even_p:n ★ \int_if_odd_p:n {\langle integer expression\rangle}
\int_if_even:nTF ★ \int_if_odd:nTF {\langle integer expression\rangle}
\int_if_odd_p:n ★   {\langle true code\rangle} {\langle false code\rangle}
\int_if_odd:nTF ★
```

This function first evaluates the $\langle integer\ expression\rangle$ as described for $\backslash int_eval:n$. It then evaluates if this is odd or even, as appropriate.

6 Integer expression loops

```
\int_do_until:nNnn ★ \int_do_until:nNnn {\langle intexpr1\rangle} {\langle relation\rangle} {\langle intexpr2\rangle} {\langle code\rangle}
```

Places the $\langle code\rangle$ in the input stream for TeX to process, and then evaluates the relationship between the two $\langle integer\ expressions\rangle$ as described for $\backslash int_compare:nNnTF$. If the test is **false** then the $\langle code\rangle$ will be inserted into the input stream again and a loop will occur until the $\langle relation\rangle$ is **true**.

```
\int_do_while:nNnn ★ \int_do_while:nNnn {\langle intexpr1\rangle} {\langle relation\rangle} {\langle intexpr2\rangle} {\langle code\rangle}
```

Places the $\langle code\rangle$ in the input stream for TeX to process, and then evaluates the relationship between the two $\langle integer\ expressions\rangle$ as described for $\backslash int_compare:nNnTF$. If the test is **true** then the $\langle code\rangle$ will be inserted into the input stream again and a loop will occur until the $\langle relation\rangle$ is **false**.

<code>\int_until_do:nNnn</code>	<code>\int_until_do:nNnn {\langle intexpr_1 \rangle} {\langle relation \rangle} {\langle intexpr_2 \rangle} {\langle code \rangle}</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 <code>false</code> . After the <i><code></i> has been processed by TeX the test will be repeated, and a loop will occur until the test is <code>true</code> .
<code>\int_while_do:nNnn</code>	<code>\int_while_do:nNnn {\langle intexpr_1 \rangle} {\langle relation \rangle} {\langle intexpr_2 \rangle} {\langle code \rangle}</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 <code>true</code> . After the <i><code></i> has been processed by TeX the test will be repeated, and a loop will occur until the test is <code>false</code> .
<code>\int_do_until:nn</code>	<code>\int_do_until:nn {\langle integer relation \rangle} {\langle code \rangle}</code> Places the <i><code></i> in the input stream for TeX to process, and then evaluates the <i><integer relation></i> as described for <code>\int_compare:nTF</code> . If the test is <code>false</code> then the <i><code></i> will be inserted into the input stream again and a loop will occur until the <i><relation></i> is <code>true</code> .
<code>\int_do_while:nn</code>	<code>\int_do_while:nn {\langle integer relation \rangle} {\langle code \rangle}</code> Places the <i><code></i> in the input stream for TeX to process, and then evaluates the <i><integer relation></i> as described for <code>\int_compare:nTF</code> . If the test is <code>true</code> then the <i><code></i> will be inserted into the input stream again and a loop will occur until the <i><relation></i> is <code>false</code> .
<code>\int_until_do:nn</code>	<code>\int_until_do:nn {\langle integer relation \rangle} {\langle code \rangle}</code> 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 <code>false</code> . After the <i><code></i> has been processed by TeX the test will be repeated, and a loop will occur until the test is <code>true</code> .
<code>\int_while_do:nn</code>	<code>\int_while_do:nn {\langle integer relation \rangle} {\langle code \rangle}</code> 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 <code>true</code> . After the <i><code></i> has been processed by TeX the test will be repeated, and a loop will occur until the test is <code>false</code> .

7 Integer step functions

\int_step_function:nnN 

New: 2012-06-04
Updated: 2014-05-30

\int_step_function:nnN {<initial value>} {<step>} {<final value>} {<function>}

This function first evaluates the *<initial value>*, *<step>* and *<final value>*, all of which should be integer expressions. The *<function>* is then placed in front of each *<value>* from the *<initial value>* to the *<final value>* in turn (using *<step>* between each *<value>*). The *<step>* must be non-zero. If the *<step>* is positive, the loop stops when the *<value>* becomes larger than the *<final value>*. If the *<step>* is negative, the loop stops when the *<value>* becomes smaller than the *<final value>*. The *<function>* should absorb one numerical argument. For example

```
\cs_set:Npn \my_func:n #1 { [I~saw~#1] \quad }
\int_step_function:nnN { 1 } { 1 } { 5 } \my_func:n
```

would print

```
[I saw 1] [I saw 2] [I saw 3] [I saw 4] [I saw 5]
```

\int_step_inline:nnn

New: 2012-06-04
Updated: 2014-05-30

\int_step_inline:nnn {<initial value>} {<step>} {<final value>} {<code>}

This function first evaluates the *<initial value>*, *<step>* and *<final value>*, all of which should be integer expressions. Then for each *<value>* from the *<initial value>* to the *<final value>* in turn (using *<step>* between each *<value>*), the *<code>* is inserted into the input stream with *#1* replaced by the current *<value>*. Thus the *<code>* should define a function of one argument (*#1*).

\int_step_variable:nnNn

New: 2012-06-04
Updated: 2014-05-30

\int_step_variable:nnNn

{<initial value>} {<step>} {<final value>} {tl var} {<code>}

This function first evaluates the *<initial value>*, *<step>* and *<final value>*, all of which should be integer expressions. Then for each *<value>* from the *<initial value>* to the *<final value>* in turn (using *<step>* between each *<value>*), the *<code>* is inserted into the input stream, with the *{tl var}* defined as the current *<value>*. Thus the *<code>* should make use of the *{tl var}*.

8 Formatting integers

Integers can be placed into the output stream with formatting. These conversions apply to any integer expressions.

\int_to_arabic:n *

Updated: 2011-10-22

\int_to_arabic:n {<integer expression>}

Places the value of the *<integer expression>* in the input stream as digits, with category code 12 (other).

\int_to_alpha:n ★
\int_to_Alph:n ★

Updated: 2011-09-17

`\int_to_alpha:n {<integer expression>}`

Evaluates the *<integer expression>* 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_alpha:n { 1 }`

places `a` in the input stream,

`\int_to_alpha:n { 26 }`

is represented as `z` and

`\int_to_alpha: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_alpha: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).

\int_to_symbols:nnn ★
Updated: 2011-09-17

`\int_to_symbols:nnn
{<integer expression>} {<total symbols>}
(value to symbol mapping)`

This is the low-level function for conversion of an *<integer expression>* into a symbolic form (which will often be letters). The *<total symbols>* available should be given as an integer expression. Values are actually converted to symbols according to the *(value to symbol mapping)*. This should be given as *<total symbols>* pairs of entries, a number and the appropriate symbol. Thus the `\int_to_alpha:n` function is defined as

```
\cs_new:Npn \int_to_alpha:n #1
{
    \int_to_symbols:nnn {#1} { 26 }
    {
        { 1 } { a }
        { 2 } { b }
        ...
        { 26 } { z }
    }
}
```

\int_to_bin:n ★

New: 2014-02-11

`\int_to_bin:n {<integer expression>}`

Calculates the value of the *<integer expression>* and places the binary representation of the result in the input stream.

`\int_to_hex:n` *

`\int_to_Hex:n` *

New: 2014-02-11

`\int_to_hex:n {<integer expression>}`

Calculates the value of the *<integer expression>* 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 `\int_to_hex:n` and upper case ones for `\int_to_Hex:n`. The resulting tokens are digits with category code 12 (other) and letters with category code 11 (letter).

`\int_to_oct:n` *

New: 2014-02-11

`\int_to_oct:n {<integer expression>}`

Calculates the value of the *<integer expression>* 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).

`\int_to_base:nn` *

`\int_to_Base:nn` *

Updated: 2014-02-11

`\int_to_base:nn {<integer expression>} {<base>}`

Calculates the value of the *<integer expression>* and converts it into the appropriate representation in the *<base>*; the latter 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 `\int_to_base:n` and upper case ones for `\int_to_Base:n`. The maximum *<base>* value is 36. The resulting tokens are digits with category code 12 (other) and letters with category code 11 (letter).

TeXhackers note: This is a generic version of `\int_to_bin:n`, etc.

`\int_to_roman:n` *

`\int_to_Roman:n` *

Updated: 2011-10-22

`\int_to_roman:n {<integer expression>}`

Places the value of the *<integer expression>* in the input stream as Roman numerals, either lower case (`\int_to_roman:n`) or upper case (`\int_to_Roman:n`). The Roman numerals are letters with category code 11 (letter).

9 Converting from other formats to integers

`\int_from_alpha:n` *

Updated: 2014-08-25

`\int_from_alpha:n {<letters>}`

Converts the *<letters>* into the integer (base 10) representation and leaves this in the input stream. The *<letters>* 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 `\int_to_alpha:n` and `\int_to_Alpha:n`.

`\int_from_bin:n` *

New: 2014-02-11

Updated: 2014-08-25

`\int_from_bin:n {<binary number>}`

Converts the *<binary number>* into the integer (base 10) representation and leaves this in the input stream. The *<binary number>* 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 `\int_to_bin:n`.

\int_from_hex:n *

New: 2014-02-11
Updated: 2014-08-25

`\int_from_hex:n {<hexadecimal number>}`

Converts the *<hexadecimal number>* into the integer (base 10) representation and leaves this in the input stream. Digits greater than 9 may be represented in the *<hexadecimal number>* by upper or lower case letters. The *<hexadecimal number>* 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 `\int_to_hex:n` and `\int_to_Hex:n`.

\int_from_oct:n *

New: 2014-02-11
Updated: 2014-08-25

`\int_from_oct:n {<octal number>}`

Converts the *<octal number>* into the integer (base 10) representation and leaves this in the input stream. The *<octal number>* 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 `\int_to_oct:n`.

\int_from_roman:n *

Updated: 2014-08-25

`\int_from_roman:n {<roman numeral>}`

Converts the *<roman numeral>* into the integer (base 10) representation and leaves this in the input stream. The *<roman numeral>* is first converted to a string, with no expansion. The *<roman numeral>* may be in upper or lower case; if the numeral contains characters besides mdclxvi or MDCLXVI then the resulting value will be -1. This is the inverse function of `\int_to_roman:n` and `\int_to_Roman:n`.

\int_from_base:nn *

Updated: 2014-08-25

`\int_from_base:nn {<number>} {<base>}`

Converts the *<number>* expressed in *<base>* into the appropriate value in base 10. The *<number>* is first converted to a string, with no expansion. The *<number>* should consist of digits and letters (either lower or upper case), plus optionally a leading sign. The maximum *<base>* value is 36. This is the inverse function of `\int_to_base:nn` and `\int_to_Base:nn`.

10 Viewing integers

\int_show:N**\int_show:c**

`\int_show:N <integer>`

Displays the value of the *<integer>* on the terminal.

\int_show:n

New: 2011-11-22
Updated: 2015-08-07

`\int_show:n {<integer expression>}`

Displays the result of evaluating the *<integer expression>* on the terminal.

\int_log:N**\int_log:c**

New: 2014-08-22

Updated: 2015-08-03

`\int_log:N <integer>`

Writes the value of the *<integer>* in the log file.

\int_log:n

New: 2014-08-22

Updated: 2015-08-07

`\int_log:n {<integer expression>}`

Writes the result of evaluating the *<integer expression>* in the log file.

11 Constant integers

```
\c_zero
\c_one
\c_two
\c_three
\c_four
\c_five
\c_six
\c_seven
\c_eight
\c_nine
\c_ten
\c_eleven
\c_twelve
\c_thirteen
\c_fourteen
\c_fifteen
\c_sixteen
\c_thirty_two
\c_one_hundred
\c_two_hundred_fifty_five
\c_two_hundred_fifty_six
\c_one_thousand
\c_ten_thousand
```

Integer values used with primitive tests and assignments: self-terminating nature makes these more convenient and faster than literal numbers.

`\c_max_int` The maximum value that can be stored as an integer.

`\c_max_register_int` Maximum number of registers.

`\c_max_char_int` Maximum character code completely supported by the engine.

12 Scratch integers

```
\l_tmpa_int
\l_tmpb_int
```

Scratch integer 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.

```
\g_tmpa_int
\g_tmpb_int
```

Scratch integer 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.

13 Primitive conditionals

\if_int_compare:w ★ \if_int_compare:w ⟨integer₁⟩ ⟨relation⟩ ⟨integer₂⟩
 ⟨true code⟩
\else:
 ⟨false code⟩
\fi:
Compare two integers using ⟨relation⟩, which must be one of =, < or > with category code 12. The \else: branch is optional.

TeXhackers note: These are both names for the TeX primitive \ifnum.

\if_case:w ★ \if_case:w ⟨integer⟩ ⟨case₀⟩
\or: ★ \or: ⟨case₁⟩
 \or: ...
 \else: ⟨default⟩
\fi:
Selects a case to execute based on the value of the ⟨integer⟩. The first case ((case₀)) is executed if ⟨integer⟩ is 0, the second ((case₁)) if the ⟨integer⟩ is 1, etc. The ⟨integer⟩ may be a literal, a constant or an integer expression (e.g. using \int_eval:n).

TeXhackers note: These are the TeX primitives \ifcase and \or.

\if_int_odd:w ★ \if_int_odd:w ⟨tokens⟩ ⟨optional space⟩
 ⟨true code⟩
\else:
 ⟨true code⟩
\fi:
Expands ⟨tokens⟩ until a non-numeric token or a space is found, and tests whether the resulting ⟨integer⟩ is odd. If so, ⟨true code⟩ is executed. The \else: branch is optional.

TeXhackers note: This is the TeX primitive \ifodd.

14 Internal functions

_int_to_roman:w ★ _int_to_roman:w ⟨integer⟩ ⟨space⟩ or ⟨non-expandable token⟩
Converts ⟨integer⟩ to its lower case Roman representation. Expansion ends when a space or non-expandable token is found. Note that this function produces a string of letters with category code 12 and that protected functions are expanded by this process. Negative ⟨integer⟩ values result in no output, although the function does not terminate expansion until a suitable endpoint is found in the same way as for positive numbers.

TeXhackers note: This is the TeX primitive \romannumeral renamed.

__int_value:w * __int_value:w *integer*
 __int_value:w *tokens* *optional space*

Expands *tokens* until an *integer* is formed. One space may be gobbled in the process.

TeXhackers note: This is the T_EX primitive `\number`.

__int_eval:w * __int_eval:w *intexpr* __int_eval_end:
__int_eval_end: *

Evaluates *integer expression* as described for `\int_eval:n`. The evaluation stops when an unexpandable token which is not a valid part of an integer is read or when `__int_eval_end:` is reached. The latter is gobbled by the scanner mechanism: `__int_eval_end:` itself is unexpandable but used correctly the entire construct is expandable.

TeXhackers note: This is the ε-T_EX primitive `\numexpr`.

__prg_compare_error: __prg_compare_error:
__prg_compare_error:Nw __prg_compare_error:Nw *token*

These are used within `\int_compare:nTF`, `\dim_compare:nTF` and so on to recover correctly if the n-type argument does not contain a properly-formed relation.

Part X

The **I3intarray** package: low-level arrays of small integers

1 I3intarray documentation

This module provides no user function: at present it is meant for kernel use only.

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 `\seq` 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 `I3intarray` 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 TeXLive settings).

1.1 Internal functions

`_intarray_new:Nn`

`_intarray_new:Nn <intarray var> {<size>}`

Evaluates the integer expression `<size>` and allocates an `<integer array variable>` with that number of (zero) entries.

`_intarray_count:N *`

`_intarray_count:N <intarray var>`

Expands to the number of entries in the `<integer array variable>`. Contrarily to `\seq_count:N` this is performed in constant time.

`_intarray_gset:Nnn` `_intarray_gset_fast:Nnn`

`_intarray_gset:Nnn <intarray var> {<position>} {<value>}`
`_intarray_gset_fast:Nnn <intarray var> {<position>} {<value>}`

Stores the result of evaluating the integer expression `<value>` into the `<integer array variable>` at the (integer expression) `<position>`. While `_intarray_gset:Nnn` checks that the `<position>` is between 1 and the `_intarray_count:N` and that the `<value>`'s absolute value is at most $2^{30} - 1$, the “fast” function performs no such bound check. Assignments are always global.

`_intarray_item:Nn *` `_intarray_item_fast:Nn *`

`_intarray_item:Nn <intarray var> {<position>}`
`_intarray_item_fast:Nn <intarray var> {<position>}`

Expands to the integer entry stored at the (integer expression) `<position>` in the `<integer array variable>`. While `_intarray_item:Nn` checks that the `<position>` is between 1 and the `_intarray_count:N`, the “fast” function performs no such bound check.

Part XI

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.

1 Setting up flags

`\flag_new:n` `\flag_new:n {<flag name>}`

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* will initially have zero height.

`\flag_clear:n` `\flag_clear:n {<flag name>}`

The *flag*’s height is set to zero. The assignment is local.

`\flag_clear_new:n` `\flag_clear_new:n {<flag name>}`

Ensures that the *flag* exists globally by applying `\flag_new:n` if necessary, then applies `\flag_clear:n`, setting the height to zero locally.

`\flag_show:n` `\flag_show:n {<flag name>}`

Displays the *flag*’s height in the terminal.

`\flag_log:n` `\flag_log:n {<flag name>}`

Writes the *flag*’s height to the log file.

2 Expandable flag commands

`\flag_if_exist_p:n *` `\flag_if_exist:n {<flag name>}`

`\flag_if_exist:nTF *`
This function returns `true` if the `<flag name>` references a flag that has been defined previously, and `false` otherwise.

`\flag_if_raised_p:n *` `\flag_if_raised:n {<flag name>}`

`\flag_if_raised:nTF *`
This function returns `true` if the `<flag>` has non-zero height, and `false` if the `<flag>` has zero height.

`\flag_height:n *` `\flag_height:n {<flag name>}`

Expands to the height of the `<flag>` as an integer denotation.

`\flag_raise:n *` `\flag_raise:n {<flag name>}`

The `<flag>`'s height is increased by 1 locally.

Part XII

The **I3quark** package

Quarks

1 Introduction to quarks and scan marks

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_`. Scan marks are for internal use by the kernel: they are not intended for more general use.

1.1 Quarks

Quarks are control sequences 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, with the most command use case as 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. An example of the quark testing functions and their use in recursion can be seen in the implementation of `\clist_map_function:NN`.

2 Defining quarks

`\quark_new:N` `\quark_new:N <quark>`

Creates a new `<quark>` which expands only to `<quark>`. The `<quark>` will be defined globally, and an error message will be raised if the name was already taken.

<u>\q_stop</u>	Used as a marker for delimited arguments, such as \cs_set:Npn \tmp:w #1#2 \q_stop {#1}
<u>\q_mark</u>	Used as a marker for delimited arguments when \q_stop is already in use.
<u>\q_nil</u>	Quark to mark a null value in structured variables or functions. Used as an end delimiter when this may itself may need to be tested (in contrast to \q_stop, which is only ever used as a delimiter).
<u>\q_no_value</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 \prop_get:NnN if there is no data to return.

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.

\quark_if_nil_p:N ★
\quark_if_nil:NTF ★

Tests if the *<token>* is equal to \q_nil.

\quark_if_nil_p:n ★
\quark_if_nil_p:(o|V) ★
\quark_if_nil:nTF ★
\quark_if_nil:(o|V)TF ★

Tests if the *<token list>* contains only \q_nil (distinct from *<token list>* being empty or containing \q_nil plus one or more other tokens).

\quark_if_no_value_p:N ★
\quark_if_no_value_p:c ★
\quark_if_no_value:NTF ★
\quark_if_no_value:cTF ★

Tests if the *<token>* is equal to \q_no_value.

\quark_if_no_value_p:n ★
\quark_if_no_value:nTF ★

Tests if the *<token list>* contains only \q_no_value (distinct from *<token list>* being empty or containing \q_no_value plus one or more other tokens).

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 5.

\q_recursion_tail

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.

\q_recursion_stop

This quark is added *after* the data structure. Its purpose is to make it possible to terminate the recursion at any point easily.

\quark_if_recursion_tail_stop:N \quark_if_recursion_tail_stop:N <token>

Tests if *<token>* 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.

\quark_if_recursion_tail_stop:n \quark_if_recursion_tail_stop:n {<token list>}
\quark_if_recursion_tail_stop:o

Updated: 2011-09-06

Tests if the *<token list>* 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.

\quark_if_recursion_tail_stop_do:Nn \quark_if_recursion_tail_stop_do:Nn <token> {{<insertion>}}

Tests if *<token>* 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. The *<insertion>* code is then added to the input stream after the recursion has ended.

\quark_if_recursion_tail_stop_do:nn \quark_if_recursion_tail_stop_do:nn {<token list>} {{<insertion>}}
\quark_if_recursion_tail_stop_do:on

Updated: 2011-09-06

Tests if the *<token list>* 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. The *<insertion>* code is then added to the input stream after the recursion has ended.

5 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 will do 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 will overwrite the definition of `__my_map dbl_fn:nn`.

6 Internal quark functions

<code>__quark_if_recursion_tail_break:NN</code>	<code>__quark_if_recursion_tail_break:nN {<token list>}</code>
<code>__quark_if_recursion_tail_break:nN</code>	<code>\<type>_map_break:</code>

Tests if *<token list>* contains only `\q_recursion_tail`, and if so terminates the recursion using `\<type>_map_break:`. The recursion end should be marked by `\prg_break_point:Nn \<type>_map_break:`.

7 Scan marks

Scan marks are control sequences set equal to `\scan_stop:`, hence will never expand in an expansion context and will be (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 T_EX 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).

The scan marks system is only for internal use by the kernel team in a small number of very specific places. These functions should not be used more generally.

__scan_new:N *__scan_new:N <scan mark>*

Creates a new *<scan mark>* which is set equal to `\scan_stop:`. The *<scan mark>* will be defined globally, and an error message will be raised if the name was already taken by another scan mark.

\s_stop Used at the end of a set of instructions, as a marker that can be jumped to using `__use_none_delimit_by_s_stop:w`.

__use_none_delimit_by_s_stop:w `__use_none_delimit_by_s_stop:w <tokens> \s_stop`

Removes the *<tokens>* and `\s_stop` from the input stream. This leads to a low-level TeX error if `\s_stop` is absent.

Part XIII

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.

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

1 Defining a set of conditional functions

```
\prg_new_conditional:Npnn
\prg_set_conditional:Npnn
\prg_new_conditional:Nnn
\prg_set_conditional:Nnn
```

Updated: 2012-02-06

```
\prg_new_conditional:Npnn \<name>:<arg spec> <parameters> {\<conditions>} {\<code>}
\prg_new_conditional:Nnn \<name>:<arg spec> {\<conditions>} {\<code>}
```

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 will 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`.

```
\prg_new_protected_conditional:Npnn
\prg_set_protected_conditional:Npnn
\prg_new_protected_conditional:Nnn
\prg_set_protected_conditional:Nnn
```

Updated: 2012-02-06

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

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 will check for existing definitions and perform assignments globally (*cf.* `\cs_new:Npn`) whereas the `set` version will 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:

- $\langle name \rangle_p : \langle arg\ spec \rangle$ — 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.
- $\langle name \rangle : \langle arg\ spec \rangle T$ — a function with one more argument than the original $\langle arg\ spec \rangle$ demands. The $\langle true\ branch \rangle$ code in this additional argument will be left on the input stream only if the test is **true**.
- $\langle name \rangle : \langle arg\ spec \rangle F$ — a function with one more argument than the original $\langle arg\ spec \rangle$ demands. The $\langle false\ branch \rangle$ code in this additional argument will be left on the input stream only if the test is **false**.
- $\langle name \rangle : \langle arg\ spec \rangle TF$ — a function with two more arguments than the original $\langle arg\ spec \rangle$ demands. The $\langle true\ branch \rangle$ code in the first additional argument will be left on the input stream if the test is **true**, while the $\langle false\ branch \rangle$ code in the second argument will be left on the input stream if the test is **false**.

The $\langle code \rangle$ of the test may use $\langle parameters \rangle$ as specified by the second argument to $\text{\prg_set_conditional:Npnn}$: this should match the $\langle argument\ specification \rangle$ 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 $\langle code \rangle$, the functions \prg_return_true: and $\text{\prg_return_false:}$ 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_t1 #1
    \prg_return_true:
  \else:
    \if_meaning:w \l_tmpa_t1 #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 $\langle conditions \rangle$ 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.

$\text{\prg_new_eq_conditional:NNn}$ $\text{\prg_new_eq_conditional:NNn} \langle name_1 \rangle : \langle arg\ spec_1 \rangle \langle name_2 \rangle : \langle arg\ spec_2 \rangle$
 $\text{\prg_set_eq_conditional:NNn}$ $\{ \langle conditions \rangle \}$

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

```
\prg_return_true: *
\prg_return_false: *
```

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:Npn`, *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.

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 will generally mean being constructed from predicate functions, possibly nested).

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

```
\bool_new:N \bool_new:c
```

Creates a new `\langle boolean\rangle` or raises an error if the name is already taken. The declaration is global. The `\langle boolean\rangle` will initially be `false`.

```
\bool_set_false:N \bool_set_false:c
\bool_gset_false:N \bool_gset_false:c
```

`\bool_set_false:N \langle boolean\rangle`

Sets `\langle boolean\rangle` logically `false`.

```
\bool_set_true:N \bool_set_true:c
\bool_gset_true:N \bool_gset_true:c
```

`\bool_set_true:N \langle boolean\rangle`

Sets `\langle boolean\rangle` logically `true`.

<code>\bool_set_eq:NN</code>	<code>\bool_set_eq:NN <boolean₁> <boolean₂></code>
<code>\bool_set_eq:(cN Nc cc)</code>	Sets <code><boolean₁></code> to the current value of <code><boolean₂></code> .
<code>\bool_gset_eq:NN</code>	
<code>\bool_gset_eq:(cN Nc cc)</code>	
<hr/>	
<code>\bool_set:Nn</code>	<code>\bool_set:Nn <boolean> {<boolexpr>}</code>
<code>\bool_set:cn</code>	Evaluates the <code><boolean expression></code> as described for <code>\bool_if:nTF</code> , and sets the <code><boolean></code> variable to the logical truth of this evaluation.
<code>\bool_gset:Nn</code>	
<code>\bool_gset:cn</code>	
<hr/>	
Updated: 2012-07-08	
<hr/>	
<code>\bool_if_p:N *</code>	<code>\bool_if_p:N <boolean></code>
<code>\bool_if_p:c *</code>	<code>\bool_if:NTF <boolean> {<true code>} {<false code>}</code>
<code>\bool_if:NTF *</code>	Tests the current truth of <code><boolean></code> , and continues expansion based on this result.
<code>\bool_if:cTF *</code>	
<hr/>	
<code>\bool_show:N</code>	<code>\bool_show:N <boolean></code>
<code>\bool_show:c</code>	Displays the logical truth of the <code><boolean></code> on the terminal.
New: 2012-02-09	
Updated: 2015-08-01	
<hr/>	
<code>\bool_show:n</code>	<code>\bool_show:n {<boolean expression>}</code>
New: 2012-02-09	Displays the logical truth of the <code><boolean expression></code> on the terminal.
Updated: 2015-08-07	
<hr/>	
<code>\bool_log:N</code>	<code>\bool_log:N <boolean></code>
<code>\bool_log:c</code>	Writes the logical truth of the <code><boolean></code> in the log file.
New: 2014-08-22	
Updated: 2015-08-03	
<hr/>	
<code>\bool_log:n</code>	<code>\bool_log:n {<boolean expression>}</code>
New: 2014-08-22	Writes the logical truth of the <code><boolean expression></code> in the log file.
Updated: 2015-08-07	
<hr/>	
<code>\bool_if_exist_p:N *</code>	<code>\bool_if_exist_p:N <boolean></code>
<code>\bool_if_exist_p:c *</code>	<code>\bool_if_exist:NTF <boolean> {<true code>} {<false code>}</code>
<code>\bool_if_exist:NTF *</code>	Tests whether the <code><boolean></code> is currently defined. This does not check that the <code><boolean></code> really is a boolean variable.
<code>\bool_if_exist:cTF *</code>	
New: 2012-03-03	
<hr/>	
<code>\l_tmpa_bool</code>	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.
<code>\l_tmpb_bool</code>	
<hr/>	
<code>\g_tmpa_bool</code>	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.
<code>\g_tmpb_bool</code>	

3 Boolean expressions

As we have a boolean datatype and predicate functions returning boolean $\langle \text{true} \rangle$ or $\langle \text{false} \rangle$ values, it seems only fitting that we also provide a parser for $\langle \text{boolean expressions} \rangle$.

A boolean expression is an expression which given input in the form of predicate functions and boolean variables, return boolean $\langle \text{true} \rangle$ or $\langle \text{false} \rangle$. 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.

At present, the infix operators `&&` and `||` perform lazy evaluation, but this will change in the near future. Contrarily to some other programming languages, the operators `&&` and `||` will 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.

TEXhackers note: The eager evaluation of boolean expressions is unfortunately necessary. Indeed, a lazy parser can get confused if `&&` and `||` appear as (unbraced) arguments of some predicates.

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`.

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

Updated: 2012-07-08

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.

```
\bool_lazy_all_p:n ★ \bool_lazy_all_p:n {<boolexpr1>} {<boolexpr2>} ... {<boolexprN>} }
\bool_lazy_all:nTF ★ \bool_lazy_all:nTF {<boolexpr1>} {<boolexpr2>} ... {<boolexprN>} } {<true code>}
{<false code>}
```

New: 2015-11-15

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` will be evaluated. See also `\bool_lazy_and:nnTF` when there are only two *<boolean expressions>*.

```
\bool_lazy_and_p:nn ★ \bool_lazy_and_p:nn {<boolexpr1>} {<boolexpr2>}
\bool_lazy_and:nnTF ★ \bool_lazy_and:nnTF {<boolexpr1>} {<boolexpr2>} {<true code>} {<false code>}
```

New: 2015-11-15

Implements the “And” operation between two boolean expressions, hence is `true` if both are `true`. Contrarily to the infix operator `&&`, the *<boolexpr2>* will only be 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>*.

```
\bool_lazy_any_p:n ★ \bool_lazy_any_p:n {<boolexpr1>} {<boolexpr2>} ... {<boolexprN>} }
\bool_lazy_any:nTF ★ \bool_lazy_any:nTF {<boolexpr1>} {<boolexpr2>} ... {<boolexprN>} } {<true code>}
{<false code>}
```

New: 2015-11-15

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` will be evaluated. See also `\bool_lazy_or:nnTF` when there are only two *<boolean expressions>*.

```
\bool_lazy_or_p:nn ★ \bool_lazy_or_p:nn {<boolexpr1>} {<boolexpr2>}
\bool_lazy_or:nnTF ★ \bool_lazy_or:nnTF {<boolexpr1>} {<boolexpr2>} {<true code>} {<false code>}
```

New: 2015-11-15

Implements the “Or” operation between two boolean expressions, hence is `true` if either one is `true`. Contrarily to the infix operator `||`, the *<boolexpr2>* will only be 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>*.

```
\bool_not_p:n ★ \bool_not_p:n {<boolean expression>}
```

Updated: 2012-07-08

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

```
\bool_xor_p:nn ★ \bool_xor_p:nn {<boolexpr1>} {<boolexpr2>}
```

Updated: 2012-07-08

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

4 Logical loops

Loops using either boolean expressions or stored boolean values.

\bool_do_until:Nn ☆
\bool_do_until:cn ☆

\bool_do_until:Nn <boolean> {<code>}

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>* will be inserted into the input stream again and the process will loop until the *<boolean>* is **true**.

\bool_do_while:Nn ☆
\bool_do_while:cn ☆

\bool_do_while:Nn <boolean> {<code>}

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

\bool_until_do:Nn ☆
\bool_until_do:cn ☆

\bool_until_do:Nn <boolean> {<code>}

This function firsts checks the logical value of the *<boolean>*. If it is **false** the *<code>* is placed in the input stream and expanded. After the completion of the *<code>* the truth of the *<boolean>* is re-evaluated. The process will then loop until the *<boolean>* is **true**.

\bool_while_do:Nn ☆
\bool_while_do:cn ☆

\bool_while_do:Nn <boolean> {<code>}

This function firsts checks the logical value of the *<boolean>*. If it is **true** the *<code>* is placed in the input stream and expanded. After the completion of the *<code>* the truth of the *<boolean>* is re-evaluated. The process will then loop until the *<boolean>* is **false**.

\bool_do_until:nn ☆
Updated: 2012-07-08

\bool_do_until:nn {<boolean expression>} {<code>}

Places the *<code>* in the input stream for TeX to process, and then checks the logical value of the *<boolean expression>* as described for \bool_if:nTF. If it is **false** then the *<code>* will be inserted into the input stream again and the process will loop until the *<boolean expression>* evaluates to **true**.

\bool_do_while:nn ☆
Updated: 2012-07-08

\bool_do_while:nn {<boolean expression>} {<code>}

Places the *<code>* in the input stream for TeX to process, and then checks the logical value of the *<boolean expression>* as described for \bool_if:nTF. If it is **true** then the *<code>* will be inserted into the input stream again and the process will loop until the *<boolean expression>* evaluates to **false**.

\bool_until_do:nn ☆
Updated: 2012-07-08

\bool_until_do:nn {<boolean expression>} {<code>}

This function firsts checks the logical value of the *<boolean expression>* (as described for \bool_if:nTF). If it is **false** the *<code>* is placed in the input stream and expanded. After the completion of the *<code>* the truth of the *<boolean expression>* is re-evaluated. The process will then loop until the *<boolean expression>* is **true**.

\bool_while_do:nn ☆
Updated: 2012-07-08

\bool_while_do:nn {<boolean expression>} {<code>}

This function firsts checks the logical value of the *<boolean expression>* (as described for \bool_if:nTF). If it is **true** the *<code>* is placed in the input stream and expanded. After the completion of the *<code>* the truth of the *<boolean expression>* is re-evaluated. The process will then loop until the *<boolean expression>* is **false**.

5 Producing multiple copies

\prg_replicate:nn *

Updated: 2011-07-04

\prg_replicate:nn {⟨integer expression⟩} {⟨tokens⟩}

Evaluates the ⟨integer expression⟩ (which should be zero or positive) and creates the resulting number of copies of the ⟨tokens⟩. The function is both expandable and safe for nesting. It yields its result after two expansion steps.

6 Detecting T_EX’s mode

\mode_if_horizontal_p: *

\mode_if_horizontal_p:

\mode_if_horizontal:TF {⟨true code⟩} {⟨false code⟩}

Detects if T_EX is currently in horizontal mode.

\mode_if_inner_p: *

\mode_if_inner_p:

\mode_if_inner:TF {⟨true code⟩} {⟨false code⟩}

Detects if T_EX is currently in inner mode.

\mode_if_math_p: *

\mode_if_math:TF {⟨true code⟩} {⟨false code⟩}

Detects if T_EX is currently in maths mode.

Updated: 2011-09-05

\mode_if_vertical_p: *

\mode_if_vertical_p:

\mode_if_vertical:TF {⟨true code⟩} {⟨false code⟩}

Detects if T_EX is currently in vertical mode.

7 Primitive conditionals

\if_predicate:w *

\if_predicate:w ⟨predicate⟩ ⟨true code⟩ \else: ⟨false code⟩ \fi:

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.)

\if_bool:N *

\if_bool:N ⟨boolean⟩ ⟨true code⟩ \else: ⟨false code⟩ \fi:

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

8 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 TEX 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 `\halign`. This is necessary to allow grabbing of tokens for testing purposes, as TEX uses group level to determine the effect of alignment tokens. Without the special grouping, the use of a function such as `\peek_after:Nw` will result in a forbidden comparison of the internal `\endtemplate` token, yielding a fatal error. Each `\group_align_safe_begin:` must be matched by a `\group_align_safe_end:`, although this does not have to occur within the same function.

```
\__prg_break_point:Nn *
```

```
\__prg_break_point:Nn \<type>_map_break: <tokens>
```

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. After the loop ends, the `<tokens>` are 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_i:nn`.

```
\__prg_map_break:Nn *
```

```
\__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 `<user code>` after the `<ending code>` for the loop. The function breaks loops, inserting their `<ending code>`, until reaching a loop with the same `<type>` as its first argument. This `\<type>_map_break:` argument is simply used as a recognizable marker for the `<type>`.

```
\g__prg_map_int
```

This integer is used by non-expandable mapping functions to track the level of nesting in force. The functions `__prg_map_1:w`, `__prg_map_2:w`, *etc.*, labelled by `\g__prg_map_int` hold functions to be mapped over various list datatypes in inline and variable mappings.

```
\__prg_break_point: *
```

This copy of `\prg_do_nothing:` is used to mark the end of a fast short-term recursions: the function `__prg_break:n` uses this to break out of the loop.

```
\__prg_break: *
\__prg_break:n *
```

```
\__prg_break:n {<tokens>} ... \__prg_break_point:
```

Breaks a recursion which has no `<ending code>` and which is not a user-breakable mapping (see for instance `\prop_get:Nn`), and inserts `<tokens>` in the input stream.

Part XIV

The `\clist` package

Comma separated lists

Comma lists contain ordered data where items can be added to the left or right end of the list. The resulting ordered list can then be mapped over using `\clist_map_function:N`. Several items can be added at once, and spaces are removed from both sides of each item on input. Hence,

```
\clist_new:N \l_my_clist
\clist_put_left:Nn \l_my_clist { ~ a ~ , ~ {b} ~ }
\clist_put_right:Nn \l_my_clist { ~ { c ~ } , d }
```

results in `\l_my_clist` containing `a,{b},{c~},d`. Comma lists cannot contain empty items, thus

```
\clist_clear_new:N \l_my_clist
\clist_put_right:Nn \l_my_clist { , ~ , , }
\clist_if_empty:NTF \l_my_clist { true } { false }
```

will leave `true` in the input stream. To include an item which contains a comma, or starts or ends with a space, surround it with braces. The sequence data type should be preferred to comma lists if items are to contain `{`, `}`, or `#` (assuming the usual T_EX category codes apply).

1 Creating and initialising comma lists

```
\clist_new:N <comma list>
```

```
\clist_new:c
```

Creates a new `<comma list>` or raises an error if the name is already taken. The declaration is global. The `<comma list>` will initially contain no items.

```
\clist_const:Nn
```

```
\clist_const:(Nx|cn|cx)
```

New: 2014-07-05

```
\clist_const:Nn <clist var> {{<comma list>}}
```

Creates a new constant `<clist var>` or raises an error if the name is already taken. The value of the `<clist var>` will be set globally to the `<comma list>`.

```
\clist_clear:N
```

```
\clist_clear:c
```

```
\clist_gclear:N
```

```
\clist_gclear:c
```

```
\clist_clear:N <comma list>
```

Clears all items from the `<comma list>`.

```
\clist_clear_new:N
```

```
\clist_clear_new:c
```

```
\clist_gclear_new:N
```

```
\clist_gclear_new:c
```

```
\clist_clear_new:N <comma list>
```

Ensures that the `<comma list>` exists globally by applying `\clist_new:N` if necessary, then applies `\clist_(g)clear:N` to leave the list empty.

```
\clist_set_eq:NN          \clist_set_eq:NN <comma list1> <comma list2>
\clist_set_eq:(cN|Nc|cc)   Sets the content of <comma list1> equal to that of <comma list2>.
\clist_gset_eq:NN
\clist_gset_eq:(cN|Nc|cc)
```

```
\clist_set_from_seq:NN      \clist_set_from_seq:NN <comma list> <sequence>
\clist_set_from_seq:(cN|Nc|cc)
\clist_gset_from_seq:NN
\clist_gset_from_seq:(cN|Nc|cc)
```

New: 2014-07-17

Converts the data in the <sequence> into a <comma list>: the original <sequence> is unchanged. Items which contain either spaces or commas are surrounded by braces.

```
\clist_concat:NNN          \clist_concat:NNN <comma list1> <comma list2> <comma list3>
\clist_concat:ccc
\clist_gconcat:NNN
\clist_gconcat:ccc
```

Concatenates the content of <comma list₂> and <comma list₃> together and saves the result in <comma list₁>. The items in <comma list₂> will be placed at the left side of the new comma list.

```
\clist_if_exist_p:N *       \clist_if_exist_p:N <comma list>
\clist_if_exist_p:c *
\clist_if_exist:NTF *
\clist_if_exist:cTF *
```

Tests whether the <comma list> is currently defined. This does not check that the <comma list> really is a comma list.

New: 2012-03-03

2 Adding data to comma lists

```
\clist_set:Nn              \clist_set:Nn <comma list> {\<item1>, ..., <itemn>}
\clist_set:(NV|No|Nx|cn|cV|co|cx)
\clist_gset:Nn
\clist_gset:(NV|No|Nx|cn|cV|co|cx)
```

New: 2011-09-06

Sets <comma list> to contain the <items>, removing any previous content from the variable. Spaces are removed from both sides of each item.

```
\clist_put_left:Nn          \clist_put_left:Nn <comma list> {\<item1>, ..., <itemn>}
\clist_put_left:(NV|No|Nx|cn|cV|co|cx)
\clist_gput_left:Nn
\clist_gput_left:(NV|No|Nx|cn|cV|co|cx)
```

Updated: 2011-09-05

Appends the <items> to the left of the <comma list>. Spaces are removed from both sides of each item.

```
\clist_put_right:Nn
\clist_put_right:(NV|No|Nx|cn|cV|co|cx)
\clist_gput_right:Nn
\clist_gput_right:(NV|No|Nx|cn|cV|co|cx)
```

Updated: 2011-09-05

```
\clist_put_right:Nn <comma list> {\<item1>, ..., <itemn>}
```

Appends the *items* to the right of the *comma list*. Spaces are removed from both sides of each item.

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 *comma list*, leaving the left most copy of each item in the *comma list*. The *item* comparison takes place on a token basis, as for `\tl_if_eq:nn(TF)`.

TeXhackers note: This function iterates through every item in the *comma list* and does a comparison with the *items* already checked. It is therefore relatively slow with large comma lists. Furthermore, it will not work if any of the items in the *comma list* contains {}, {}, or # (assuming the usual TeX category codes apply).

```
\clist_remove_all:Nn
\clist_remove_all:cn
\clist_gremove_all:Nn
\clist_gremove_all:cn
```

Updated: 2011-09-06

```
\clist_remove_all:Nn <comma list> {\<item>}
```

Removes every occurrence of *item* from the *comma list*. The *item* comparison takes place on a token basis, as for `\tl_if_eq:nn(TF)`.

TeXhackers note: The *item* may not contain {}, {}, or # (assuming the usual TeX category codes apply).

```
\clist_reverse:N
\clist_reverse:c
\clist_greverse:N
\clist_greverse:c
```

New: 2014-07-18

```
\clist_reverse:N <comma list>
```

Reverses the order of items stored in the *comma list*.

```
\clist_reverse:n
```

New: 2014-07-18

```
\clist_reverse:n {\<comma list>}
```

Leaves the items in the *comma list* in the input stream in reverse order. Braces and spaces are preserved by this process.

TeXhackers note: The result is returned within `\unexpanded`, which means that the comma list will not expand further when appearing in an x-type argument expansion.

```
\clist_sort:Nn
\clist_sort:cn
\clist_gsort:Nn
\clist_gsort:cn
```

New: 2017-02-06

```
\clist_sort:Nn <clist var> {<comparison code>}
```

Sorts the items in the *<clist var>* according to the *<comparison code>*, and assigns the result to *<clist var>*. The details of sorting comparison are described in Section 1.

```
\clist_if_empty_p:N *
\clist_if_empty_p:c *
\clist_if_empty:NTF *
\clist_if_empty:cTF *
```

```
\clist_if_empty_p:N <comma list>
\clist_if_empty:NTF <comma list> {<true code>} {<false code>}
```

Tests if the *<comma list>* is empty (containing no items).

```
\clist_if_empty_p:n *
\clist_if_empty:nTF *
```

```
\clist_if_empty_p:n {<comma list>}
\clist_if_empty:nTF {<comma list>} {<true code>} {<false code>}
```

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.

```
\clist_if_in:NnTF
\clist_if_in:(NV|No|cn|cV|co)TF
\clist_if_in:nnTF
\clist_if_in:(nV|no)TF
```

```
\clist_if_in:NnTF <comma list> {<item>} {<true code>} {<false code>}
```

Updated: 2011-09-06

Tests if the *<item>* is present in the *<comma list>*. In the case of an *n*-type *<comma list>*, spaces are stripped from each item, but braces are not removed. Hence,

```
\clist_if_in:nnTF { a , {b}~ , {b} , c } { b } {true} {false}
```

yields *false*.

TeXhackers note: The *<item>* may not contain *{*, *}*, or *#* (assuming the usual TeX category codes apply), and should not contain *,* nor start or end with a space.

5 Mapping to comma lists

The functions described in this section apply a specified function to each item of a comma list.

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 your 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 <comma list> <function>
```

Applies *<function>* to every *<item>* stored in the *<comma list>*. The *<function>* will receive one argument for each iteration. The *<items>* are returned from left to right. The function `\clist_map_inline:Nn` is in general more efficient than `\clist_map_function:NN`. One mapping may be nested inside another.

```
\clist_map_inline:Nn
\clist_map_inline:cn
\clist_map_inline:nn
```

Updated: 2012-06-29

```
\clist_map_inline:Nn <comma list> {<inline function>}
```

Applies *<inline function>* to every *<item>* stored within the *<comma list>*. The *<inline function>* should consist of code which will receive the *<item>* as #1. One in line mapping can be nested inside another. The *<items>* 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 <comma list> <tl var.> {<function using tl var.>}
```

Stores each entry in the *<comma list>* in turn in the *<tl var.>* and applies the *<function using tl var.>* The *<function>* will usually consist of code making use of the *<tl var.>*, but this is not enforced. One variable mapping can be nested inside another. The *<items>* are returned from left to right.

```
\clist_map_break: ☆
```

Updated: 2012-06-29

```
\clist_map_break:
```

Used to terminate a `\clist_map_...` function before all entries in the *<comma list>* have been processed. This will normally take 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 will lead to low level T_EX errors.

T_EXhackers note: When the mapping is broken, additional tokens may be inserted by the internal macro `__prg_break_point:Nn` before further items are taken from the input stream. This will depend on the design of the mapping function.

\clist_map_break:n ★

Updated: 2012-06-29

\clist_map_break:n {⟨tokens⟩}

Used to terminate a \clist_map_... function before all entries in the ⟨comma list⟩ have been processed, inserting the ⟨tokens⟩ after the mapping has ended. This will normally take place within a conditional statement, for example

```
\clist_map_inline:Nn \l_my_clist
{
  \str_if_eq:nnTF { #1 } { bingo }
    { \clist_map_break:n { <tokens> } }
  {
    % Do something useful
  }
}
```

Use outside of a \clist_map_... scenario will lead to low level T_EX errors.

T_EXhackers note: When the mapping is broken, additional tokens may be inserted by the internal macro __prg_break_point:Nn before the ⟨tokens⟩ are inserted into the input stream. This will depend 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⟩ will include those which are duplicates, i.e. every item in a ⟨comma list⟩ is unique.

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 will be 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~ }
```

will insert “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⟩ will not expand further when appearing in an x-type argument expansion.

\clist_use:Nn ★
\clist_use:cN ★
New: 2013-05-26

\clist_use:Nn *clist var* {*separator*}

Places the contents of the *clist var* in the input stream, with the *separator* 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 will be 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~ }
```

will insert “a and b and c and de and f” in the input stream.

TEXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the *items* will not expand further when appearing in an x-type argument expansion.

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:cN
Updated: 2012-05-14

\clist_get:NN *comma list* *token list variable*

Stores the left-most item from a *comma list* in the *token list variable* without removing it from the *comma list*. The *token list variable* is assigned locally. If the *comma list* is empty the *token list variable* will contain the marker value `\q_no_value`.

\clist_get:NNTF
\clist_get:cNTF
New: 2012-05-14

\clist_get:NNTF *comma list* *token list variable* {*true code*} {*false code*}

If the *comma list* is empty, leaves the *false code* in the input stream. The value of the *token list variable* is not defined in this case and should not be relied upon. If the *comma list* is non-empty, stores the top item from the *comma list* in the *token list variable* without removing it from the *comma list*. The *token list variable* is assigned locally.

\clist_pop:NN
\clist_pop:cN
Updated: 2011-09-06

\clist_pop:NN *comma list* *token list variable*

Pops the left-most item from a *comma list* into the *token list variable*, i.e. removes the item from the comma list and stores it in the *token list variable*. Both of the variables are assigned locally.

\clist_gpop:NN
\clist_gpop:cN

\clist_gpop:NN *comma list* *token list variable*

Pops the left-most item from a *comma list* into the *token list variable*, i.e. removes the item from the comma list and stores it in the *token list variable*. The *comma list* is modified globally, while the assignment of the *token list variable* is local.

```
\clist_pop:NNTF
\clist_pop:cNTF
```

New: 2012-05-14

```
\clist_pop:NNTF <comma list> <token list variable> {<true code>} {<false code>}
```

If the *<comma list>* is empty, leaves the *<false code>* in the input stream. The value of the *<token list variable>* is not defined in this case and should not be relied upon. If the *<comma list>* is non-empty, pops the top item from the *<comma list>* in the *<token list variable>*, *i.e.* removes the item from the *<comma list>*. Both the *<comma list>* and the *<token list variable>* are assigned locally.

```
\clist_gpop:NNTF
\clist_gpop:cNTF
```

New: 2012-05-14

```
\clist_gpop:NNTF <comma list> <token list variable> {<true code>} {<false code>}
```

If the *<comma list>* is empty, leaves the *<false code>* in the input stream. The value of the *<token list variable>* is not defined in this case and should not be relied upon. If the *<comma list>* is non-empty, pops the top item from the *<comma list>* in the *<token list variable>*, *i.e.* removes the item from the *<comma list>*. The *<comma list>* is modified globally, while the *<token list variable>* 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 <comma list> {<items>}
```

Adds the *{<items>}* to the top of the *<comma list>*. Spaces are removed from both sides of each item.

8 Using a single item

```
\clist_item:Nn *
\clist_item:cn *
\clist_item:nn *
```

New: 2014-07-17

```
\clist_item:Nn <comma list> {<integer expression>}
```

Indexing items in the *<comma list>* from 1 at the top (left), this function will evaluate the *<integer expression>* and leave the appropriate item from the comma list in the input stream. If the *<integer expression>* is negative, indexing occurs from the bottom (right) of the comma list. When the *<integer expression>* is larger than the number of items in the *<comma list>* (as calculated by `\clist_count:N`) then the function will expand to nothing.

TEXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the *<item>* will not expand further when appearing in an x-type argument expansion.

9 Viewing comma lists

```
\clist_show:N
\clist_show:c
```

Updated: 2015-08-03

```
\clist_show:N <comma list>
```

Displays the entries in the *<comma list>* in the terminal.

```
\clist_show:n
```

Updated: 2013-08-03

```
\clist_show:n {<tokens>}
```

Displays the entries in the comma list in the terminal.

`\clist_log:N` `\clist_log:N {comma list}`
`\clist_log:c`

New: 2014-08-22 Writes the entries in the *comma list* in the log file. See also `\clist_show:N` which displays the result in the terminal.
Updated: 2015-08-03

`\clist_log:n` `\clist_log:n {\{tokens\}}`
`\clist_log:n`

New: 2014-08-22 Writes the entries in the comma list in the log file. See also `\clist_show:n` which displays the result in the terminal.

10 Constant and scratch comma lists

`\c_empty_clist` Constant that is always empty.
`\c_empty_clist`

New: 2012-07-02

`\l_tmpa_clist` Scratch comma lists 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.
`\l_tmpb_clist`

New: 2011-09-06

`\g_tmpa_clist` Scratch comma lists 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.
`\g_tmpb_clist`

New: 2011-09-06

Part XV

The **I3token** 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\texEX, 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 will have two primary function categories: `\token_` for anything that deals with tokens and `\peek_` for looking ahead in the token stream.

Most functions we will 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\texEX, 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\texEX distinguishes them when searching for a delimited argument. Namely, the example function `\show_until_if:w` defined below will take 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 8.

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  
\char_set_active_eq:nc  
\char_gset_active_eq:nN  
\char_gset_active_eq:nc
```

New: 2015-11-12

`\char_set_active_eq:NN <char> <function>`

Sets the behaviour of the `<char>` in situations where it is active (category code 13) to be equivalent to that of the `<function>`. The category code of the `<char>` is *unchanged* by this process. The `<function>` may itself be an active character.

`\char_set_active_eq:nN {<integer expression>} <function>`

Sets the behaviour of the `<char>` which has character code as given by the `<integer expression>` in situations where it is active (category code 13) to be equivalent to that of the `<function>`. The category code of the `<char>` is *unchanged* by this process. The `<function>` may itself be an active character.

```
\char_generate:nn *
```

New: 2015-09-09

```
\char_generate:nn {<charcode>} {<catcode>}
```

Generates a character token of the given $\langle \text{charcode} \rangle$ and $\langle \text{catcode} \rangle$ (both of which may be integer expressions). The $\langle \text{catcode} \rangle$ 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)

and other values will raise an error.

The $\langle \text{charcode} \rangle$ may be any one valid for the engine in use. Note however that for X_ET_EX releases prior to 0.999992 only the 8-bit range (0 to 255) is accepted due to engine limitations.

2 Manipulating and interrogating character tokens

```
\char_set_catcode_escape:N          \char_set_catcode_letter:N <character>
\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
```

Updated: 2015-11-11

Sets the category code of the $\langle \text{character} \rangle$ to that indicated in the function name. Depending on the current category code of the $\langle \text{token} \rangle$ the escape token may also be needed:

```
\char_set_catcode_other:N \%
```

The assignment is local.

```
\char_set_catcode_escape:n          \char_set_catcode_letter:n {\langle integer expression\rangle}
\char_set_catcode_group_begin:n      \char_set_catcode_group_end: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_active:n          \char_set_catcode_comment:n
\char_set_catcode_comment:n         \char_set_catcode_invalid:n
```

Updated: 2015-11-11

Sets the category code of the *<character>* which has character code as given by the *<integer expression>*. This version can be used to set up characters which cannot otherwise be given (*cf.* the N-type variants). The assignment is local.

```
\char_set_catcode:nn
```

Updated: 2015-11-11

These functions set the category code of the *<character>* which has character code as given by the *<integer expression>*. The first *<integer expression>* 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.

```
\char_value_catcode:n *
```

```
\char_value_catcode:n {\langle integer expression\rangle}
```

Expands to the current category code of the *<character>* with character code given by the *<integer expression>*.

```
\char_show_value_catcode:n
```

```
\char_show_value_catcode:n {\langle integer expression\rangle}
```

Displays the current category code of the *<character>* with character code given by the *<integer expression>* on the terminal.

```
\char_set_lccode:nn
```

Updated: 2015-08-06

```
\char_set_lccode:nn {\langle intexpr1\rangle} {\langle intexpr2\rangle}
```

Sets up the behaviour of the *<character>* when found inside `\tl_to_lowercase:n`, such that *<character₁>* will be converted into *<character₂>*. The two *<characters>* may be specified using an *<integer expression>* for the character code concerned. This may include the TeX ‘*<character>*’ 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.

\char_value_lccode:n *

```
\char_value_lccode:n {<integer expression>}
```

Expands to the current lower case code of the *character* with character code given by the *integer expression*.

\char_show_value_lccode:n

```
\char_show_value_lccode:n {<integer expression>}
```

Displays the current lower case code of the *character* with character code given by the *integer expression* on the terminal.

\char_set_uccode:nn

Updated: 2015-08-06

```
\char_set_uccode:nn {<intexpr1>} {<intexpr2>}
```

Sets up the behaviour of the *character* when found inside `\tl_to_uppercase:n`, such that *character₁* will be converted into *character₂*. The two *characters* may be specified using an *integer expression* for the character code concerned. This may include the TeX ‘*character*’ method for converting a single character into its character code:

```
\char_set_uccode:nn { '\a } { '\A } % Standard behaviour  
\char_set_uccode:nn { '\A } { '\A - 32 }  
\char_set_uccode:nn { 60 } { 50 }
```

The setting applies within the current TeX group.

\char_value_uccode:n *

```
\char_value_uccode:n {<integer expression>}
```

Expands to the current upper case code of the *character* with character code given by the *integer expression*.

\char_show_value_uccode:n

```
\char_show_value_uccode:n {<integer expression>}
```

Displays the current upper case code of the *character* with character code given by the *integer expression* on the terminal.

\char_set_mathcode:nn

Updated: 2015-08-06

```
\char_set_mathcode:nn {<intexpr1>} {<intexpr2>}
```

This function sets up the math code of *character*. The *character* is specified as an *integer expression* which will be used as the character code of the relevant character. The setting applies within the current TeX group.

\char_value_mathcode:n *

```
\char_value_mathcode:n {<integer expression>}
```

Expands to the current math code of the *character* with character code given by the *integer expression*.

\char_show_value_mathcode:n

```
\char_show_value_mathcode:n {<integer expression>}
```

Displays the current math code of the *character* with character code given by the *integer expression* on the terminal.

\char_set_sfcode:nn

Updated: 2015-08-06

```
\char_set_sfcode:nn {<intexpr1>} {<intexpr2>}
```

This function sets up the space factor for the *character*. The *character* is specified as an *integer expression* which will be used as the character code of the relevant character. The setting applies within the current TeX group.

\char_value_sfcode:n *

`\char_value_sfcode:n {<integer expression>}`

Expands to the current space factor for the *<character>* with character code given by the *<integer expression>*.

\char_show_value_sfcode:n

`\char_show_value_sfcode:n {<integer expression>}`

Displays the current space factor for the *<character>* with character code given by the *<integer expression>* on the terminal.

\l_char_active_seq

New: 2012-01-23
Updated: 2015-11-11

Used to track which tokens may require special handling at the document level as they are (or have been at some point) of category *<active>* (catcode 13). Each entry in the sequence consists of a single escaped token, for example `\~`. Active tokens should be added to the sequence when they are defined for general document use.

\l_char_special_seq

New: 2012-01-23
Updated: 2015-11-11

Used to track which tokens will require special handling when working with verbatim-like material at the document level as they are not of categories *<letter>* (catcode 11) or *<other>* (catcode 12). Each entry in the sequence consists of a single escaped token, for example `\\"` for the backslash or `\{` for an opening brace. Escaped tokens should be added to the sequence when they are defined for general document use.

3 Generic tokens

\token_new:Nn

`\token_new:Nn <token1> {<token2>}`

Defines *<token₁>* to globally be a snapshot of *<token₂>*. This will be an implicit representation of *<token₂>*.

\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

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.

\c_catcode_letter_token

\c_catcode_other_token

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.

\c_catcode_active_t1

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.

4 Converting tokens

```
\token_to_meaning:N ★ \token_to_meaning:N <token>
```

`\token_to_meaning:c ★`
Inserts the current meaning of the `<token>` into the input stream as a series of characters of category code 12 (other). This will be the primitive TeX description of the `<token>`, thus for example both functions defined by `\cs_set_nopar:Npn` and token list variables defined using `\tl_new:N` will be described as **macros**.

TeXhackers note: This is the TeX primitive `\meaning`.

```
\token_to_str:N ★ \token_to_str:N <token>
```

`\token_to_str:c ★`
Converts the given `<token>` into a series of characters with category code 12 (other). The current escape character will be the first character in the sequence, although this will also have category code 12 (the escape character is part of the `<token>`). This function requires only a single expansion.

TeXhackers note: `\token_to_str:N` is the TeX primitive `\string` renamed.

5 Token conditionals

```
\token_if_group_begin_p:N ★ \token_if_group_begin_p:N <token>
```

```
\token_if_group_begin:NTF ★ \token_if_group_begin:NTF <token> {<true code>} {<false code>}
```

Tests if `<token>` 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.

```
\token_if_group_end_p:N ★ \token_if_group_end_p:N <token>
```

```
\token_if_group_end:NTF ★ \token_if_group_end:NTF <token> {<true code>} {<false code>}
```

Tests if `<token>` 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.

```
\token_if_math_toggle_p:N ★ \token_if_math_toggle_p:N <token>
```

```
\token_if_math_toggle:NTF ★ \token_if_math_toggle:NTF <token> {<true code>} {<false code>}
```

Tests if `<token>` has the category code of a math shift token (\$) when normal TeX category codes are in force).

```
\token_if_alignment_p:N ★ \token_if_alignment_p:N <token>
```

```
\token_if_alignment:NTF ★ \token_if_alignment:NTF <token> {<true code>} {<false code>}
```

Tests if `<token>` has the category code of an alignment token (& when normal TeX 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_alignment:NTF <token> {\{true code\}} {\{false code\}}</code>
Tests if <i><token></i> has the category code of a macro parameter token (# when normal TeX 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 <i><token></i> has the category code of a superscript token (^ when normal TeX 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 <i><token></i> has the category code of a subscript token (_) when normal TeX 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 <i><token></i> 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 <i><token></i> 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 <i><token></i> 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 <i><token></i> has the category code of an active character.	
<code>\token_if_eq_catcode_p:NN *</code>	<code>\token_if_eq_catcode_p:NN <token₁> <token₂></code>
<code>\token_if_eq_catcode:NNTF *</code>	<code>\token_if_eq_catcode:NNTF <token₁> <token₂> {\{true code\}} {\{false code\}}</code>
Tests if the two <i><tokens></i> have the same category code.	
<code>\token_if_eq_charcode_p:NN *</code>	<code>\token_if_eq_charcode_p:NN <token₁> <token₂></code>
<code>\token_if_eq_charcode:NNTF *</code>	<code>\token_if_eq_charcode:NNTF <token₁> <token₂> {\{true code\}} {\{false code\}}</code>
Tests if the two <i><tokens></i> have the same character code.	
<code>\token_if_eq_meaning_p:NN *</code>	<code>\token_if_eq_meaning_p:NN <token₁> <token₂></code>
<code>\token_if_eq_meaning:NNTF *</code>	<code>\token_if_eq_meaning:NNTF <token₁> <token₂> {\{true code\}} {\{false code\}}</code>
Tests if the two <i><tokens></i> have the same meaning when expanded.	

```
\token_if_macro_p:N ★ \token_if_macro_p:N ⟨token⟩
\token_if_macro:NTF ★ \token_if_macro:NTF ⟨token⟩ {⟨true code⟩} {⟨false code⟩}
```

Updated: 2011-05-23

Tests if the ⟨token⟩ is a TeX macro.

```
\token_if_cs_p:N ★ \token_if_cs_p:N ⟨token⟩
\token_if_cs:NTF ★ \token_if_cs:NTF ⟨token⟩ {⟨true code⟩} {⟨false code⟩}
```

Tests if the ⟨token⟩ is a control sequence.

```
\token_if_expandable_p:N ★ \token_if_expandable_p:N ⟨token⟩
\token_if_expandable:NTF ★ \token_if_expandable:NTF ⟨token⟩ {⟨true code⟩} {⟨false code⟩}
```

Tests if the ⟨token⟩ is expandable. This test returns ⟨false⟩ for an undefined token.

```
\token_if_long_macro_p:N ★ \token_if_long_macro_p:N ⟨token⟩
\token_if_long_macro:NTF ★ \token_if_long_macro:NTF ⟨token⟩ {⟨true code⟩} {⟨false code⟩}
```

Updated: 2012-01-20

Tests if the ⟨token⟩ is a long macro.

```
\token_if_protected_macro_p:N ★ \token_if_protected_macro_p:N ⟨token⟩
\token_if_protected_macro:NTF ★ \token_if_protected_macro:NTF ⟨token⟩ {⟨true code⟩} {⟨false code⟩}
```

Updated: 2012-01-20

Tests if the ⟨token⟩ is a protected macro: a macro which is both protected and long will return logical **false**.

```
\token_if_protected_long_macro_p:N ★ \token_if_protected_long_macro_p:N ⟨token⟩
\token_if_protected_long_macro:NTF ★ \token_if_protected_long_macro:NTF ⟨token⟩ {⟨true code⟩} {⟨false code⟩}
```

Tests if the ⟨token⟩ is a protected long macro.

```
\token_if_chardef_p:N ★ \token_if_chardef_p:N ⟨token⟩
\token_if_chardef:NTF ★ \token_if_chardef:NTF ⟨token⟩ {⟨true code⟩} {⟨false code⟩}
```

Updated: 2012-01-20

Tests if the ⟨token⟩ is defined to be a chardef.

TeXhackers note: Booleans, boxes and small integer constants are implemented as chardefs.

```
\token_if_mathchardef_p:N ★ \token_if_mathchardef_p:N ⟨token⟩
\token_if_mathchardef:NTF ★ \token_if_mathchardef:NTF ⟨token⟩ {⟨true code⟩} {⟨false code⟩}
```

Updated: 2012-01-20

Tests if the ⟨token⟩ is defined to be a mathchardef.

```
\token_if_dim_register_p:N ★ \token_if_dim_register_p:N ⟨token⟩
\token_if_dim_register:NTF ★ \token_if_dim_register:NTF ⟨token⟩ {⟨true code⟩} {⟨false code⟩}
```

Updated: 2012-01-20

Tests if the ⟨token⟩ is defined to be a dimension register.

```
\token_if_int_register_p:N ★ \token_if_int_register_p:N ⟨token⟩
\token_if_int_register:NTF ★ \token_if_int_register:NTF ⟨token⟩ {⟨true code⟩} {⟨false 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.

```
\token_if_muskip_register_p:N ★ \token_if_muskip_register_p:N ⟨token⟩
\token_if_muskip_register:NTF ★ \token_if_muskip_register:NTF ⟨token⟩ {⟨true code⟩} {⟨false code⟩}
```

New: 2012-02-15

Tests if the ⟨token⟩ is defined to be a muskip register.

```
\token_if_skip_register_p:N ★ \token_if_skip_register_p:N ⟨token⟩
\token_if_skip_register:NTF ★ \token_if_skip_register:NTF ⟨token⟩ {⟨true code⟩} {⟨false code⟩}
```

Updated: 2012-01-20

Tests if the ⟨token⟩ is defined to be a skip register.

```
\token_if_toks_register_p:N ★ \token_if_toks_register_p:N ⟨token⟩
\token_if_toks_register:NTF ★ \token_if_toks_register:NTF ⟨token⟩ {⟨true code⟩} {⟨false code⟩}
```

Updated: 2012-01-20

Tests if the ⟨token⟩ is defined to be a toks register (not used by L^AT_EX3).

```
\token_if_primitive_p:N ★ \token_if_primitive_p:N ⟨token⟩
\token_if_primitive:NTF ★ \token_if_primitive:NTF ⟨token⟩ {⟨true code⟩} {⟨false code⟩}
```

Updated: 2011-05-23

Tests if the ⟨token⟩ is an engine primitive.

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.

```
\peek_after:Nw \peek_after:Nw ⟨function⟩ ⟨token⟩
```

Locally sets the test variable `\l_peek_token` equal to ⟨token⟩ (as an implicit token, *not* as a token list), and then expands the ⟨function⟩. The ⟨token⟩ will remain in the input stream as the next item after the ⟨function⟩. The ⟨token⟩ here may be `\l`, `{` or `}` (assuming normal TeX category codes), *i.e.* it is not necessarily the next argument which would be grabbed by a normal function.

\peek_gafter:Nw

```
\peek_gafter:Nw <function> <token>
```

Globally sets the test variable `\g_peek_token` equal to `<token>` (as an implicit token, *not* as a token list), and then expands the `<function>`. The `<token>` will remain in the input stream as the next item after the `<function>`. The `<token>` here may be `\`, `{` or `}` (assuming normal TeX category codes), *i.e.* it is not necessarily the next argument which would be grabbed by a normal function.

\l_peek_token

Token set by `\peek_after:Nw` and available for testing as described above.

\g_peek_token

Token set by `\peek_gafter:Nw` and available for testing as described above.

\peek_catcode:NTF

Updated: 2012-12-20

```
\peek_catcode:NTF <test token> {<true code>} {<false code>}
```

Tests if the next `<token>` in the input stream has the same category code as the `<test token>` (as defined by the test `\token_if_eq_catcode:NNTF`). Spaces are respected by the test and the `<token>` will be left in the input stream after the `<true code>` or `<false code>` (as appropriate to the result of the test).

\peek_catcode_ignore_spaces:NTF

Updated: 2012-12-20

```
\peek_catcode_ignore_spaces:NTF <test token> {<true code>} {<false code>}
```

Tests if the next non-space `<token>` in the input stream has the same category code as the `<test token>` (as defined by the test `\token_if_eq_catcode:NNTF`). Explicit and implicit space tokens (with character code 32 and category code 10) are ignored and removed by the test and the `<token>` will be left in the input stream after the `<true code>` or `<false code>` (as appropriate to the result of the test).

\peek_catcode_remove:NTF

Updated: 2012-12-20

```
\peek_catcode_remove:NTF <test token> {<true code>} {<false code>}
```

Tests if the next `<token>` in the input stream has the same category code as the `<test token>` (as defined by the test `\token_if_eq_catcode:NNTF`). Spaces are respected by the test and the `<token>` will be removed from the input stream if the test is true. The function will then place either the `<true code>` or `<false code>` in the input stream (as appropriate to the result of the test).

\peek_catcode_remove_ignore_spaces:NTF

Updated: 2012-12-20

```
\peek_catcode_remove_ignore_spaces:NTF <test token> {<true code>} {<false code>}
```

Tests if the next non-space `<token>` in the input stream has the same category code as the `<test token>` (as defined by the test `\token_if_eq_catcode:NNTF`). Explicit and implicit space tokens (with character code 32 and category code 10) are ignored and removed by the test and the `<token>` will be removed from the input stream if the test is true. The function will then place either the `<true code>` or `<false code>` in the input stream (as appropriate to the result of the test).

\peek_charcode:NTF

Updated: 2012-12-20

```
\peek_charcode:NTF <test token> {<true code>} {<false code>}
```

Tests if the next `<token>` in the input stream has the same character code as the `<test token>` (as defined by the test `\token_if_eq_charcode:NNTF`). Spaces are respected by the test and the `<token>` will be left in the input stream after the `<true code>` or `<false code>` (as appropriate to the result of the test).

\peek_charcode_ignore_spaces:NTFUpdated: 2012-12-20

```
\peek_charcode_ignore_spaces:NTF <test token> {{true code}} {{false code}}
```

Tests if the next non-space *<token>* in the input stream has the same character code as the *<test token>* (as defined by the test \token_if_eq_charcode:NNTF). Explicit and implicit space tokens (with character code 32 and category code 10) are ignored and removed by the test and the *<token>* will be left in the input stream after the *<true code>* or *<false code>* (as appropriate to the result of the test).

\peek_charcode_remove:NTFUpdated: 2012-12-20

```
\peek_charcode_remove:NTF <test token> {{true code}} {{false code}}
```

Tests if the next *<token>* in the input stream has the same character code as the *<test token>* (as defined by the test \token_if_eq_charcode:NNTF). Spaces are respected by the test and the *<token>* will be removed from the input stream if the test is true. The function will then place either the *<true code>* or *<false code>* in the input stream (as appropriate to the result of the test).

\peek_charcode_remove_ignore_spaces:NTFUpdated: 2012-12-20

```
\peek_charcode_remove_ignore_spaces:NTF <test token> {{true code}} {{false code}}
```

Tests if the next non-space *<token>* in the input stream has the same character code as the *<test token>* (as defined by the test \token_if_eq_charcode:NNTF). Explicit and implicit space tokens (with character code 32 and category code 10) are ignored and removed by the test and the *<token>* will be removed from the input stream if the test is true. The function will then place either the *<true code>* or *<false code>* in the input stream (as appropriate to the result of the test).

\peek_meaning:NTFUpdated: 2011-07-02

```
\peek_meaning:NTF <test token> {{true code}} {{false code}}
```

Tests if the next *<token>* in the input stream has the same meaning as the *<test token>* (as defined by the test \token_if_eq_meaning:NNTF). Spaces are respected by the test and the *<token>* will be left in the input stream after the *<true code>* or *<false code>* (as appropriate to the result of the test).

\peek_meaning_ignore_spaces:NTFUpdated: 2012-12-05

```
\peek_meaning_ignore_spaces:NTF <test token> {{true code}} {{false code}}
```

Tests if the next non-space *<token>* in the input stream has the same meaning as the *<test token>* (as defined by the test \token_if_eq_meaning:NNTF). Explicit and implicit space tokens (with character code 32 and category code 10) are ignored and removed by the test and the *<token>* will be left in the input stream after the *<true code>* or *<false code>* (as appropriate to the result of the test).

\peek_meaning_remove:NTFUpdated: 2011-07-02

```
\peek_meaning_remove:NTF <test token> {{true code}} {{false code}}
```

Tests if the next *<token>* in the input stream has the same meaning as the *<test token>* (as defined by the test \token_if_eq_meaning:NNTF). Spaces are respected by the test and the *<token>* will be removed from the input stream if the test is true. The function will then place either the *<true code>* or *<false code>* in the input stream (as appropriate to the result of the test).

\peek_meaning_remove_ignore_spaces:NTF \peek_meaning_remove_ignore_spaces:NTF {test token} {<true code>} {<false code>}

Tests if the next non-space *<token>* in the input stream has the same meaning as the *<test token>* (as defined by the test \token_if_eq_meaning:NNTF). Explicit and implicit space tokens (with character code 32 and category code 10) are ignored and removed by the test and the *<token>* will be removed from the input stream if the test is true. The function will then place either the *<true code>* or *<false code>* in the input stream (as appropriate to the result of the test).

7 Decomposing a macro definition

These 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 later case, all three functions leave `\scan_stop`: in the input stream.

\token_get_arg_spec:N \star \token_get_arg_spec:N *⟨token⟩*

If the $\langle token \rangle$ is a macro, this function will leave the primitive T_EX argument specification in input stream as a string of tokens of category code 12 (with spaces having category code 10). Thus for example for a token `\next` defined by

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

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

TeXhackers note: If the arg spec. contains the string `->`, then the `spec` function will produce incorrect results.

```
\token_get_replacement_spec:N ★ \token_get_replacement_spec:N <token>
```

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

```
\cs_set:Npn \next { x ~ y }
```

will leave `x#1` `y#2` in the input stream. If the `<token>` is not a macro then `\scan_stop`: will be left in the input stream.

TEXhackers note: If the arg spec. contains the string `->`, then the `spec` function will produce incorrect results.

```
\token_get_prefix_spec:N *
```

```
\token_get_prefix_spec:N <token>
```

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

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

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

8 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).⁴

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 $\langle token \rangle$ is expandable) results in an internal token, displayed (temporarily) as `\notexpanded: <token>`, whose shape coincides with the $\langle token \rangle$ and whose meaning differs from `\relax`.
- An `\outer endtemplate:` (expanding to another internal token, `end of alignment template`) can be encountered when peeking ahead at the next token.
- Tricky programming might access a frozen `\endwrite`.
- Some frozen tokens can only be accessed in interactive sessions: `\cr`, `\right`, `\endgroup`, `\fi`, `\inaccessible`.

⁴In LuaTeX, there is also the case of “bytes”, which behave as character tokens of category code 12 (other) and character code between 1114112 and 1114366. They are used to output individual bytes to files, rather than UTF-8.

The meaning of a (non-active) character token is fixed by its category code (and character code) and cannot be changed. We will 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 \LaTeX3 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 will call such tokens *implicit* character tokens. The meaning is otherwise in the following list:

- a macro, used in \LaTeX3 for most functions and some variables (`t1`, `fp`, `seq`, ...),
- a primitive such as `\def` or `\topmark`, used in \LaTeX3 for some functions,
- a register such as `\count123`, used in \LaTeX3 for the implementation of some variables (`int`, `dim`, ...),
- a constant integer such as `\char"56` or `\mathchar"121`,
- a font selection command,
- undefined.

Macros be `\protected` or not, `\long` or not (the opposite of what \LaTeX3 calls `nopar`), and `\outer` or not (unused in \LaTeX3). Their \meaning takes the form

$\langle properties \rangle \text{ macro} : \langle parameters \rangle \rightarrow \langle replacement \rangle$

where $\langle properties \rangle$ is among `\protected\long\outer`, $\langle parameters \rangle$ describes parameters that the macro expects, such as `#1#2#3`, and $\langle replacement \rangle$ describes how the parameters are manipulated, such as `#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.

9 Internal functions

`_char_generate:nn *`

New: 2016-03-25

`_char_generate:nn {\<charcode>} {\<catcode>}`

This function is identical in operation to the public `\char_generate:nn` but omits various sanity tests. In particular, this means it is used in certain places where engine variations need to be accounted for by the kernel. The *<catcode>* must give an explicit integer after a single expansion.

Part XVI

The **\3prop** package

Property lists

LATEX3 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 $\langle balanced\ text \rangle$. 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 will overwrite 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 `keys` module.

1 Creating and initialising property lists

`\prop_new:N` `\prop_new:c`

Creates a new $\langle property\ list \rangle$ or raises an error if the name is already taken. The declaration is global. The $\langle property\ list \rangle$ will initially contain no entries.

`\prop_clear:N` `\prop_clear:c`

`\prop_gclear:N` `\prop_gclear:c`

Clears all entries from the $\langle property\ list \rangle$.

`\prop_clear_new:N` `\prop_clear_new:c`

`\prop_gclear_new:N` `\prop_gclear_new:c`

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` $\langle property\ list_1 \rangle$ $\langle property\ list_2 \rangle$

Sets the content of $\langle property\ list_1 \rangle$ equal to that of $\langle property\ list_2 \rangle$.

2 Adding entries to property lists

```
\prop_put:Nnn
\prop_put:(NnV|Nno|Nnx|NVn|NVV|Non|Noo|cnn|cnV|cno|cnx|cVn|cVV|con|coo)
\prop_gput:Nnn
\prop_gput:(NnV|Nno|Nnx|NVn|NVV|Non|Noo|cnn|cnV|cno|cnx|cVn|cVV|con|coo)
```

Updated: 2012-07-09

```
\prop_put:Nnn <property list>
{<key>} {<value>}
```

Adds an entry to the *<property list>* which may be accessed using the *<key>* and which has *<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. If the *<key>* is already present in the *<property list>*, the existing entry is overwritten by the new *<value>*.

```
\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 *<key>* is present in the *<property list>* then no action is taken. If the *<key>* is not present in the *<property list>* then a new entry is added. 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.

3 Recovering values from property lists

```
\prop_get:NnN
\prop_get:(NVN|NoN|cnN|cVN|coN)
```

Updated: 2011-08-28

```
\prop_get:NnN <property list> {<key>} <tl var>
```

Recovers the *<value>* stored with *<key>* from the *<property list>*, and places this in the *<token list variable>*. If the *<key>* is not found in the *<property list>* then the *<token list variable>* will contain the special marker `\q_no_value`. The *<token list variable>* 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 *<value>* stored with *<key>* from the *<property list>*, and places this in the *<token list variable>*. If the *<key>* is not found in the *<property list>* then the *<token list variable>* will contain the special marker `\q_no_value`. The *<key>* and *<value>* are then deleted from the property list. Both assignments are local. See also `\prop_pop:NnNTF`.

```
\prop_gpop:NnN
\prop_gpop:(NoN|cnN|coN)
```

Updated: 2011-08-18

```
\prop_gpop:NnN <property list> {<key>} <tl var>
```

Recovers the *<value>* stored with *<key>* from the *<property list>*, and places this in the *<token list variable>*. If the *<key>* is not found in the *<property list>* then the *<token list variable>* will contain the special marker `\q_no_value`. The *<key>* and *<value>* are then deleted from the property list. The *<property list>* is modified globally, while the assignment of the *<token list variable>* is local. See also `\prop_gpop:NnNTF`.

```
\prop_item:Nn ★  
\prop_item:cn ★  
New: 2014-07-17
```

```
\prop_item:Nn <property list> {<key>}
```

Expands to the *<value>* corresponding to the *<key>* in the *<property list>*. If the *<key>* is missing, this has an empty expansion.

TeXhackers 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 *<value>* will not expand further when appearing in an `x`-type argument expansion.

4 Modifying property lists

```
\prop_remove:Nn  
\prop_remove:(NV|cn|cV)  
\prop_gremove:Nn  
\prop_gremove:(NV|cn|cV)  
New: 2012-05-12
```

```
\prop_remove:Nn <property list> {<key>}
```

Removes the entry listed under *<key>* from the *<property list>*. If the *<key>* is not found in the *<property list>* no change occurs, *i.e.* there is no need to test for the existence of a key before deleting it.

5 Property list conditionals

```
\prop_if_exist_p:N ★  
\prop_if_exist_p:c ★  
\prop_if_exist:NTF ★  
\prop_if_exist:cTF ★  
New: 2012-03-03
```

```
\prop_if_exist_p:N <property list>  
\prop_if_exist:NTF <property list> {<true code>} {<false code>}
```

Tests whether the *<property list>* is currently defined. This does not check that the *<property list>* really is a property list variable.

```
\prop_if_empty_p:N ★  
\prop_if_empty_p:c ★  
\prop_if_empty:NTF ★  
\prop_if_empty:cTF ★
```

```
\prop_if_empty_p:N <property list>  
\prop_if_empty:NTF <property list> {<true code>} {<false code>}
```

Tests if the *<property list>* is empty (containing no entries).

```
\prop_if_in_p:Nn ★  
\prop_if_in_p:(NV|No|cn|cV|co) ★  
\prop_if_in:NnTF ★  
\prop_if_in:(NV|No|cn|cV|co)TF ★
```

Updated: 2011-09-15

```
\prop_if_in:NnTF <property list> {<key>} {<true code>} {<false code>}
```

Tests if the *<key>* is present in the *<property list>*, making the comparison using the method described by `\str_if_eq:nnTF`.

TeXhackers note: This function iterates through every key–value pair in the *<property list>* and is therefore slower than using the non-expandable `\prop_get:NnNTF`.

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.

\prop_get:NnNTF
\prop_get:(NVN|NoN|cnN|cVN|coN)TF

Updated: 2012-05-19

\prop_get:NnNTF *property list* {<key>} {<token list variable>} {<true code>} {<false code>}

If the *<key>* is not present in the *<property list>*, leaves the *<false code>* in the input stream. The value of the *<token list variable>* is not defined in this case and should not be relied upon. If the *<key>* is present in the *<property list>*, stores the corresponding *<value>* in the *<token list variable>* without removing it from the *<property list>*, then leaves the *<true code>* in the input stream. The *<token list variable>* is assigned locally.

\prop_pop:NnNTF
\prop_pop:cnNTF

New: 2011-08-18
Updated: 2012-05-19

\prop_pop:NnNTF *property list* {<key>} {<token list variable>} {<true code>} {<false code>}

If the *<key>* is not present in the *<property list>*, leaves the *<false code>* in the input stream. The value of the *<token list variable>* is not defined in this case and should not be relied upon. If the *<key>* is present in the *<property list>*, pops the corresponding *<value>* in the *<token list variable>*, *i.e.* removes the item from the *<property list>*. Both the *<property list>* and the *<token list variable>* are assigned locally.

\prop_gpop:NnNTF
\prop_gpop:cnNTF

New: 2011-08-18
Updated: 2012-05-19

\prop_gpop:NnNTF *property list* {<key>} {<token list variable>} {<true code>} {<false code>}

If the *<key>* is not present in the *<property list>*, leaves the *<false code>* in the input stream. The value of the *<token list variable>* is not defined in this case and should not be relied upon. If the *<key>* is present in the *<property list>*, pops the corresponding *<value>* in the *<token list variable>*, *i.e.* removes the item from the *<property list>*. The *<property list>* is modified globally, while the *<token list variable>* is assigned locally.

7 Mapping to property lists

\prop_map_function:NN ☆
\prop_map_function:cN ☆

Updated: 2013-01-08

\prop_map_function:NN *property list* *function*

Applies *<function>* to every *<entry>* stored in the *<property list>*. The *<function>* will receive two argument for each iteration: the *<key>* and associated *<value>*. The order in which *<entries>* are returned is not defined and should not be relied upon.

\prop_map_inline:Nn
\prop_map_inline:cN

Updated: 2013-01-08

\prop_map_inline:Nn *property list* {<inline function>}

Applies *<inline function>* to every *<entry>* stored within the *<property list>*. The *<inline function>* should consist of code which will receive the *<key>* as #1 and the *<value>* as #2. The order in which *<entries>* are returned is not defined and should not be relied upon.

\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 will normally take 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 will lead to low level T_EX errors.

\prop_map_break:n ☆Updated: 2012-06-29

\prop_map_break:n {<tokens>}

Used to terminate a \prop_map_... function before all entries in the *property list* have been processed, inserting the *<tokens>* after the mapping has ended. This will normally take place within a conditional statement, for example

```
\prop_map_inline:Nn \l_my_prop
{
    \str_if_eq:nnTF { #1 } { bingo }
        { \prop_map_break:n { <tokens> } }
    {
        % Do something useful
    }
}
```

Use outside of a \prop_map_... scenario will lead to low level T_EX errors.

8 Viewing property lists

\prop_show:N**\prop_show:c**Updated: 2015-08-01

\prop_show:N <property list>

Displays the entries in the *property list* in the terminal.

\prop_log:N**\prop_log:c**

New: 2014-08-12

Updated: 2015-08-01

\prop_log:N <property list>

Writes the entries in the *property list* in the log file.

9 Scratch property lists

\l_tmpa_prop

\l_tmpb_prop

New: 2012-06-23

Scratch property lists 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.

\g_tmpa_prop

\g_tmpb_prop

New: 2012-06-23

Scratch property lists 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.

10 Constants

\c_empty_prop

A permanently-empty property list used for internal comparisons.

11 Internal property list functions

\s_prop

The internal token used at the beginning of property lists. This is also used after each *key* (see __prop_pair:wn).

__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 will be raised. The definition should always be set globally.

\l__prop_internal_tl

Token list used to store new key–value pairs to be inserted by functions of the \prop_put:Nnn family.

__prop_split:NnTF

Updated: 2013-01-08

__prop_split:NnTF <property list> {<key>} {<true code>} {<false code>}

Splits the *property list* at the *key*, giving three token lists: the *extract* of *property list* before the *key*, the *value* associated with the *key* and the *extract* of the *property list* after the *value*. Both *extracts* retain the internal structure of a property list, and the concatenation of the two *extracts* is a property list. If the *key* is present in the *property list* then the *true code* is left in the input stream, with #1, #2, and #3 replaced by the first *extract*, the *value*, and the second extract. If the *key* is not present in the *property list* then the *false code* is left in the input stream, with no trailing material. Both *true code* and *false code* 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 *true code* for the three extracts from the property list. The *key* comparison takes place as described for \str_if_eq:nn.

Part XVII

The `I3msg` 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 `I3msg` module provides a consistent method for doing this (as opposed to writing directly to the terminal or log).

The system used by `I3msg` 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.

1 Creating new messages

All messages have to be created before they can be used. The text of messages will automatically be wrapped to the length available in the console. As a result, formatting is only needed where it will help 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 L^AT_EX while still being filterable at a more granular level. Thus for example

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

will allow only those messages from the `submodule` to be filtered out.

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

Updated: 2011-08-16

```
\msg_new:nnnn {\langle module \rangle} {\langle message \rangle} {\langle text \rangle} {\langle more text \rangle}
```

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

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

```
\msg_set:nnnn {\langle module \rangle} {\langle message \rangle} {\langle text \rangle} {\langle more text \rangle}
```

Sets up the text for a `\langle message \rangle` for a given `\langle module \rangle`. The message will be defined to first give `\langle text \rangle` and then `\langle more text \rangle` if the user requests it. If no `\langle more text \rangle` is available then a standard text is given instead. Within `\langle text \rangle` and `\langle more text \rangle` 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_p:nn {*module*} {*message*}
\msg_if_exist:nnTF ★ \msg_if_exist:nnTF {*module*} {*message*} {*true code*} {*false code*}

New: 2012-03-03

2 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 proceeded by the text `on line`.

\msg_line_number: ★ \msg_line_number:

Prints the current line number when a message is given.

\msg_fatal_text:n ★ \msg_fatal_text:n {*module*}

Produces the standard text

Fatal *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.

\msg_critical_text:n ★ \msg_critical_text:n {*module*}

Produces the standard text

Critical *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.

\msg_error_text:n ★ \msg_error_text:n {*module*}

Produces the standard text

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.

\msg_warning_text:n ★ \msg_warning_text:n {*module*}

Produces the standard text

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.

```
\msg_info_text:n *
```

```
\msg_info_text:n {\<module>}
```

Produces the standard text:

```
<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.

```
\msg_see_documentation_text:n *
```

```
\msg_see_documentation_text:n {\<module>}
```

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.

3 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 will be ignored, or empty arguments added (of course the sense of the message may be impaired). The four arguments will be converted to strings before being added to the message text: the x-type variants should be used to expand material.

```
\msg_fatal:nnnnnn  
\msg_fatal:nnxxxx  
\msg_fatal:nnnnn  
\msg_fatal:nnxxx  
\msg_fatal:nnnn  
\msg_fatal:nnxx  
\msg_fatal:nnn  
\msg_fatal:nnx  
\msg_fatal:nn
```

Updated: 2012-08-11

```
\msg_fatal:nnnnnn {\<module>} {\<message>} {\<arg one>} {\<arg two>} {\<arg three>}  
{\<arg four>}
```

Issues *<module>* error *<message>*, passing *<arg one>* to *<arg four>* to the text-creating functions. After issuing a fatal error the TeX run will halt.

```
\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 *<module>* error *<message>*, passing *<arg one>* to *<arg four>* to the text-creating functions. After issuing a critical error, TeX will stop reading the current input file. This may halt the TeX run (if the current file is the main file) or may abort reading a sub-file.

TeXhackers note: The TeX `\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:nnnnnnn {\module} {\message} {\arg one} {\arg two} {\arg three}
{\arg four}
```

Issues *<module>* error *<message>*, passing *<arg one>* to *<arg four>* to the text-creating functions. The error will interrupt processing and issue the text at the terminal. After user input, the run will continue.

```
\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 *<module>* warning *<message>*, passing *<arg one>* to *<arg four>* to the text-creating functions. The warning text will be added to the log file and the terminal, but the T_EX run will not be interrupted.

```
\msg_info:nnnnnn
\msg_info:nnxxxx
\msg_info:nnnnn
\msg_info:nnxxx
\msg_info:nnnn
\msg_info:nnxx
\msg_info:nnn
\msg_info:nnx
\msg_info:nn
```

Updated: 2012-08-11

```
\msg_info:nnnnnnn {\module} {\message} {\arg one} {\arg two} {\arg three} {\arg
four}
```

Issues *<module>* information *<message>*, passing *<arg one>* to *<arg four>* to the text-creating functions. The information text will be added to the log file.

```
\msg_log:nnnnnn
\msg_log:nnxxxx
\msg_log:nnnnn
\msg_log:nnxxx
\msg_log:nnnn
\msg_log:nnxx
\msg_log:nnn
\msg_log:nnx
\msg_log:nn
```

Updated: 2012-08-11

```
\msg_log:nnnnnnn {\module} {\message} {\arg one} {\arg two} {\arg three} {\arg
four}
```

Issues *<module>* information *<message>*, passing *<arg one>* to *<arg four>* to the text-creating functions. The information text will be added to the log file: the output is briefer than \msg_info:nnnnnn.

```
\msg_none:nnnnnn
\msg_none:nnxxxx
\msg_none:nnnnn
\msg_none:nnxxx
\msg_none:nnnn
\msg_none:nnxx
\msg_none:nnn
\msg_none:nnx
\msg_none:nn
```

Updated: 2012-08-11

```
\msg_none:nnnnnn {<module>} {<message>} {<arg one>} {<arg two>} {<arg three>} {<arg four>}
```

Does nothing: used as a message class to prevent any output at all (see the discussion of message redirection).

4 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 will raise 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 will raise errors 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 $\langle class\ one\rangle$ so that they are processed using the code for those of $\langle class\ two\rangle$.

\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 }
```

5 Low-level message functions

The lower-level message functions should usually be accessed from the higher-level system. However, there are occasions where direct access to these functions is desirable.

\msg_interrupt:nnn

New: 2012-06-28

\msg_interrupt:nnn {<first line>} {<text>} {<extra text>}

Interrupts the TeX run, issuing a formatted message comprising *<first line>* and *<text>* laid out in the format

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!
! <first line>
!
! <text>
!
.....
```

where the *<text>* will be wrapped to fit within the current line length. The user may then request more information, at which stage the *<extra text>* will be shown in the terminal in the format

```
| , , , , , , , , , , , , , , , , , , ,
| <extra text>
| .....
```

where the *<extra text>* will be wrapped within the current line length. Wrapping of both *<text>* and *<more text>* takes place using `\iow_wrap:nnnN`; the documentation for the latter should be consulted for full details.

`\msg_log:n` `\msg_log:n {<text>}`

New: 2012-06-28

Writes to the log file with the `<text>` laid out in the format

```
.....  
. <text>  
.....
```

where the `<text>` will be wrapped to fit within the current line length. Wrapping takes place using `\iow_wrap:nnnN`; the documentation for the latter should be consulted for full details.

`\msg_term:n` `\msg_term:n {<text>}`

New: 2012-06-28

Writes to the terminal and log file with the `<text>` laid out in the format

```
*****  
* <text>  
*****
```

where the `<text>` will be wrapped to fit within the current line length. Wrapping takes place using `\iow_wrap:nnnN`; the documentation for the latter should be consulted for full details.

6 Kernel-specific functions

Messages from L^AT_EX3 itself are handled by the general message system, but have their own functions. This allows some text to be pre-defined, and also ensures that serious errors can be handled properly.

`_msg_kernel_new:nnnn` `_msg_kernel_new:nnn` `_msg_kernel_new:n` `_msg_kernel_new` `_msg_kernel_new:nnnn {<module>} {<message>} {<text>} {<more text>}`

Updated: 2011-08-16

Creates a kernel `<message>` for a given `<module>`. The message will be 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 and expanded at the time the message is used. An error will be raised if the `<message>` already exists.

`_msg_kernel_set:nnnn` `_msg_kernel_set:nnn` `_msg_kernel_set:n` `_msg_kernel_set` `_msg_kernel_set:nnnn {<module>} {<message>} {<text>} {<more text>}`

Sets up the text for a kernel `<message>` for a given `<module>`. The message will be 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 and expanded at the time the message is used.

```
\_\_msg_kernel_fatal:nnnnnn  
\_\_msg_kernel_fatal:nnxxxx  
\_\_msg_kernel_fatal:nnnnn  
\_\_msg_kernel_fatal:nnxxx  
\_\_msg_kernel_fatal:nnnn  
\_\_msg_kernel_fatal:nnxx  
\_\_msg_kernel_fatal:nnn  
\_\_msg_kernel_fatal:nnx  
\_\_msg_kernel_fatal:nn
```

Updated: 2012-08-11

```
\_\_msg_kernel_fatal:nnnnnnn {\<module>} {\<message>} {\<arg one>} {\<arg two>} {\<arg three>} {\<arg four>}
```

Issues kernel *<module>* error *<message>*, passing *<arg one>* to *<arg four>* to the text-creating functions. After issuing a fatal error the TeX run will halt. Cannot be redirected.

```
\_\_msg_kernel_error:nnnnnn  
\_\_msg_kernel_error:nnxxxx  
\_\_msg_kernel_error:nnnnn  
\_\_msg_kernel_error:nnxxx  
\_\_msg_kernel_error:nnnn  
\_\_msg_kernel_error:nnxx  
\_\_msg_kernel_error:nnn  
\_\_msg_kernel_error:nnx  
\_\_msg_kernel_error:nn
```

Updated: 2012-08-11

```
\_\_msg_kernel_error:nnnnnnn {\<module>} {\<message>} {\<arg one>} {\<arg two>} {\<arg three>} {\<arg four>}
```

Issues kernel *<module>* error *<message>*, passing *<arg one>* to *<arg four>* to the text-creating functions. The error will stop processing and issue the text at the terminal. After user input, the run will continue. Cannot be redirected.

```
\_\_msg_kernel_warning:nnnnnn  
\_\_msg_kernel_warning:nnxxxx  
\_\_msg_kernel_warning:nnnnn  
\_\_msg_kernel_warning:nnxxx  
\_\_msg_kernel_warning:nnnn  
\_\_msg_kernel_warning:nnxx  
\_\_msg_kernel_warning:nnn  
\_\_msg_kernel_warning:nnx  
\_\_msg_kernel_warning:nn
```

Updated: 2012-08-11

```
\_\_msg_kernel_warning:nnnnnnn {\<module>} {\<message>} {\<arg one>} {\<arg two>} {\<arg three>} {\<arg four>}
```

Issues kernel *<module>* warning *<message>*, passing *<arg one>* to *<arg four>* to the text-creating functions. The warning text will be added to the log file, but the TeX run will not be interrupted.

```
\_\_msg_kernel_info:nnnnnn  
\_\_msg_kernel_info:nnxxxx  
\_\_msg_kernel_info:nnnnn  
\_\_msg_kernel_info:nnxxx  
\_\_msg_kernel_info:nnnn  
\_\_msg_kernel_info:nnxx  
\_\_msg_kernel_info:nnn  
\_\_msg_kernel_info:nnx  
\_\_msg_kernel_info:nn
```

Updated: 2012-08-11

```
\_\_msg_kernel_info:nnnnnnn {\<module>} {\<message>} {\<arg one>} {\<arg two>} {\<arg three>} {\<arg four>}
```

Issues kernel *<module>* information *<message>*, passing *<arg one>* to *<arg four>* to the text-creating functions. The information text will be added to the log file.

7 Expandable errors

In a few places, the L^AT_EX3 kernel needs to produce errors in an expansion only context. This 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. However, the interface is similar, with the important caveat that the message text and arguments are not expanded, and messages should be very short.

```
\__msg_kernel_expandable_error:nnnnnn ★ \__msg_kernel_expandable_error:nnnnnn {\langle module\rangle} {\langle message\rangle}
\__msg_kernel_expandable_error:nnnnn ★ {\langle arg one\rangle} {\langle arg two\rangle} {\langle arg three\rangle} {\langle arg four\rangle}
\__msg_kernel_expandable_error:nnnn ★
\__msg_kernel_expandable_error:nnn ★
\__msg_kernel_expandable_error:nnn ★
\__msg_kernel_expandable_error:nn ★
```

New: 2011-11-23

Issues an error, passing *⟨arg one⟩* to *⟨arg four⟩* to the text-creating functions. The resulting string must be shorter than a line, otherwise it will be cropped.

```
\__msg_expandable_error:n ★ \__msg_expandable_error:n {\langle error message\rangle}
```

New: 2011-08-11
Updated: 2011-08-13

Issues an “Undefined error” message from T_EX itself, and prints the *⟨error message⟩*. The *⟨error message⟩* must be short: it is cropped at the end of one line.

TEXhackers note: This function expands to an empty token list after two steps. Tokens inserted in response to T_EX’s prompt are read with the current category code setting, and inserted just after the place where the error message was issued.

8 Internal l3msg functions

The following functions are used in several kernel modules.

```
\__msg_log_next: ★ \__msg_log_next: {\langle show-command\rangle}
```

New: 2015-08-05

Causes the next *⟨show-command⟩* to send its output to the log file instead of the terminal. This allows for instance `\cs_log:N` to be defined as `__msg_log_next: \cs_show:N`. The effect of this command lasts until the next use of `__msg_show_wrap:Nn` or `__msg_show_wrap:n` or `__msg_show_variable>NNNnn`, in other words until the next time the ε-T_EX primitive `\showtokens` would have been used for showing to the terminal or until the next `variable-not-defined` error.

```
\__msg_show_pre:nnnnnn ★ \__msg_show_pre:nnnnnn {\langle module\rangle} {\langle message\rangle} {\langle arg one\rangle} {\langle arg two\rangle}
\__msg_show_pre:(nxxxx|nnnnnV) {\langle arg three\rangle} {\langle arg four\rangle}
```

New: 2015-08-05

Prints the *⟨message⟩* from *⟨module⟩* in the terminal (or log file if `__msg_log_next:` was issued) without formatting. Used in messages which print complex variable contents completely.

`__msg_show_variable:NNNnn`
New: 2015-08-04

`__msg_show_variable:NNNnn <variable> <if-exist> <if-empty> {<msg>} {{<formatted content>}}`

If the `<variable>` does not exist according to `<if-exist>` (typically `\cs_if_exist:NTF`) then throw an error and do nothing more. Otherwise, if `<msg>` is not empty, display the message `LaTeX/kernel/show-<msg>` with `\token_to_str:N <variable>` as a first argument, and a second argument that is ? or empty depending on the result of `<if-empty>` (typically `\tl_if_empty:NTF`) on the `<variable>`. Then display the `<formatted content>` by giving it as an argument to `__msg_show_wrap:n`.

`__msg_show_wrap:Nn`
New: 2015-08-03
Updated: 2015-08-07

`__msg_show_wrap:Nn <function> {{<expression>}}`

Shows or logs the `<expression>` (turned into a string), an equal sign, and the result of applying the `<function>` to the `{<expression>}`. For instance, if the `<function>` is `\int_eval:n` and the `<expression>` is `1+2` then this will log `> 1+2=3`. The case where the `<function>` is `\tl_to_str:n` is special: then the string representation of the `<expression>` is only logged once.

`__msg_show_wrap:n`
New: 2015-08-03

`__msg_show_wrap:n {{<formatted text>}}`

Shows or logs the `<formatted text>`. After expansion, unless it is empty, the `<formatted text>` must contain `>`, and the part of `<formatted text>` before the first `>` is removed. Failure to do so causes low-level T_EX errors.

`__msg_show_item:n`
`__msg_show_item:nn`
`__msg_show_item_unbraced:nn`
Updated: 2012-09-09

`__msg_show_item:n <item>`
`__msg_show_item:nn <item-key> <item-value>`

Auxiliary functions used within the last argument of `__msg_show_variable:NNNnn` or `__msg_show_wrap:n` to format variable items correctly for display. The `__msg_show_item:n` version is used for simple lists, the `__msg_show_item:nn` and `__msg_show_item_unbraced:nn` versions for key–value like data structures.

`\c__msg_coding_error_text_t1`

The text

`This is a coding error.`

used by kernel functions when erroneous programming input is encountered.

Part XVIII

The **I3file** 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` will attempt to locate them 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. Any active characters (as declared in `\l_char_active_seq`) will *not* be expanded, allowing the direct use of these in file names. File names will be quoted using " tokens if they contain spaces: as a result, " tokens are *not* permitted in file names.

1 File operation functions

`\g_file_current_name_tl`

Contains the name of the current `LATEX` file. This variable should not be modified: it is intended for information only. It will be equal to `\c_sys_jobname_str` at the start of a `LATEX` run and will be modified each time a file is read using `\file_input:n`.

`\file_if_exist:nTF`

Updated: 2012-02-10

```
\file_if_exist:nTF {\<file name>} {\<true code>} {\<false code>}
```

Searches for *<file name>* using the current `TeX` search path and the additional paths controlled by `\file_path_include:n`.

`\file_add_path:nN`

Updated: 2012-02-10

```
\file_add_path:nN {\<file name>} {\<tl var>}
```

Searches for *<file name>* in the path as detailed for `\file_if_exist:nTF`, and if found sets the *<tl var>* the fully-qualified name of the file, *i.e.* the path and file name. If the file is not found then the *<tl var>* will contain the marker `\q_no_value`.

`\file_input:n`

Updated: 2012-02-17

```
\file_input:n {\<file name>}
```

Searches for *<file name>* in the path as detailed for `\file_if_exist:nTF`, and if found reads in the file as additional `LATEX` source. All files read are recorded for information and the file name stack is updated by this function. An error will be raised if the file is not found.

`\file_path_include:n`

Updated: 2012-07-04

```
\file_path_include:n {\<path>}
```

Adds *<path>* to the list of those used to search when reading files. The assignment is local. The *<path>* is processed in the same way as a *<file name>*, *i.e.*, with `x`-type expansion except active characters.

`\file_path_remove:n`

Updated: 2012-07-04

`\file_path_remove:n {<path>}`

Removes $\langle path \rangle$ from the list of those used to search when reading files. The assignment is local. The $\langle path \rangle$ is processed in the same way as a $\langle file\ name \rangle$, *i.e.*, with x-type expansion except active characters.

`\file_list:`

`\file_list:`

This function will list all files loaded using `\file_input:n` in the log file.

1.1 Input–output stream management

As TeX is limited to 16 input streams and 16 output streams, direct use of the streams by the programmer is not supported in L^AT_EX3. 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.

`\ior_new:N`

`\ior_new:c`

`\iow_new:N`

`\iow_new:c`

New: 2011-09-26

Updated: 2011-12-27

`\ior_new:N <stream>`

`\iow_new:N <stream>`

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 `\..._open:Nn` 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 `\c_term`....

`\ior_open:Nn`

`\ior_open:cn`

Updated: 2012-02-10

`\ior_open:Nn <stream> {<file\ name>}`

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 `\ior_close:N` instruction is given or the TeX run ends.

`\ior_open:NnTF`

`\ior_open:cnTF`

New: 2013-01-12

`\ior_open:NnTF <stream> {<file\ name>} {<true\ code>} {<false\ code>}`

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 `\ior_close:N` instruction is given or the TeX 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.

`\iow_open:Nn`

`\iow_open:cn`

Updated: 2012-02-09

`\iow_open:Nn <stream> {<file\ name>}`

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 `\iow_close:N` instruction is given or the TeX run ends. Opening a file for writing will clear any existing content in the file (*i.e.* writing is *not* additive).

```
\ior_close:N  
\ior_close:c  
\iow_close:N  
\iow_close:c
```

Updated: 2012-07-31

```
\ior_close:N <stream>  
\iow_close:N <stream>
```

Closes the *<stream>*. Streams should always be closed when they are finished with as this ensures that they remain available to other programmers.

```
\ior_list_streams:  
\iow_list_streams:
```

Updated: 2015-08-01

```
\ior_list_streams:  
\iow_list_streams:
```

Displays a list of the file names associated with each open stream: intended for tracking down problems.

1.2 Reading from files

```
\ior_get:NN
```

New: 2012-06-24

```
\ior_get:NN <stream> <token list variable>
```

Function that reads one or more lines (until an equal number of left and right braces are found) from the input *<stream>* and stores the result locally in the *<token list>* variable. If the *<stream>* is not open, input is requested from the terminal. The material read from the *<stream>* will be 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 % will have the line ending converted to a space, so for example input

```
a b c
```

will result 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.

TeXhackers 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.

\ior_str_get:NN

New: 2016-12-04

\ior_str_get:NN *stream* {*token list variable*}

Function that reads one line from the input *stream* and stores the result locally in the *token list* variable. If the *stream* is not open, input is requested from the terminal. The material is read from the *stream* 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 will always only read one line and any blank lines in the input will result in the *token list variable* being empty. Unlike \ior_get:NN, line ends do not receive any special treatment. Thus input

a b c

will result in a token list a b c with the letters a, b, and c having category code 12.

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.

\ior_map_inline:Nn

New: 2012-02-11

\ior_map_inline:Nn *stream* {*inline function*}

Applies the *inline function* to each set of *lines* 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 *inline function* should consist of code which will receive the *line* as #1.

\ior_str_map_inline:Nn

New: 2012-02-11

\ior_str_map_inline:Nn {*stream*} {*inline function*}

Applies the *inline function* to every *line* in the *stream*. The material is read from the *stream* as a series of tokens with category code 12 (other), with the exception of space characters which are given category code 10 (space). The *inline function* should consist of code which will receive the *line* 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.

\ior_map_break:

New: 2012-06-29

\ior_map_break:

Used to terminate a \ior_map_... function before all lines from the *stream* have been processed. This will normally take 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 will lead to low level TeX errors.

TeXhackers note: When the mapping is broken, additional tokens may be inserted by the internal macro __prg_break_point:Nn before further items are taken from the input stream. This will depend on the design of the mapping function.

\ior_map_break:nNew: 2012-06-29

\ior_map_break:n {*tokens*}

Used to terminate a \ior_map_... function before all lines in the *stream* have been processed, inserting the *tokens* after the mapping has ended. This will normally take place within a conditional statement, for example

```
\ior_map_inline:Nn \l_my_iор
{
    \str_if_eq:nnTF { #1 } { bingo }
        { \ior_map_break:n { <tokens> } }
    {
        % Do something useful
    }
}
```

Use outside of a \ior_map_... scenario will lead to low level TEX errors.

TeXhackers note: When the mapping is broken, additional tokens may be inserted by the internal macro __prg_break_point:Nn before the *tokens* are inserted into the input stream. This will depend on the design of the mapping function.

\ior_if_eof_p:N *
\ior_if_eof:NTF *Updated: 2012-02-10

\ior_if_eof_p:N *stream*
\ior_if_eof:NTF *stream* {*true code*} {*false code*}

Tests if the end of a *stream* has been reached during a reading operation. The test will also return a **true** value if the *stream* is not open.

2 Writing to files

\iow_now:Nn**\iow_now:(Nx|cn|cx)**Updated: 2012-06-05

\iow_now:Nn *stream* {*tokens*}

This functions writes *tokens* to the specified *stream* immediately (*i.e.* the write operation is called on expansion of \iow_now:Nn).

\iow_log:n**\iow_log:x**\iow_log:n {*tokens*}

This function writes the given *tokens* to the log (transcript) file immediately: it is a dedicated version of \iow_now:Nn.

\iow_term:n**\iow_term:x**\iow_term:n {*tokens*}

This function writes the given *tokens* to the terminal file immediately: it is a dedicated version of \iow_now:Nn.

\iow_shipout:Nn
\iow_shipout:(Nx|cn|cx)

\iow_shipout:Nn *stream* {*tokens*}

This functions writes *tokens* to the specified *stream* when the current page is finalised (*i.e.* at shipout). The x-type variants expand the *tokens* at the point where the function is used but *not* when the resulting tokens are written to the *stream* (*cf.* \iow_shipout_x:Nn).

TeXhackers 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 will not be recognized in the argument of \iow_shipout:Nn. This may lead to the insertion of additionnal unwanted line-breaks.

\iow_shipout_x:Nn
\iow_shipout_x:(Nx|cn|cx)

Updated: 2012-09-08

\iow_shipout_x:Nn *stream* {*tokens*}

This functions writes *tokens* to the specified *stream* when the current page is finalised (*i.e.* at shipout). The *tokens* 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).

TeXhackers note: This is a wrapper around the TeX 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 will not be recognized in the argument of \iow_shipout:Nn. This may lead to the insertion of additionnal unwanted line-breaks.

\iow_char:N *

\iow_char:N \iow_char:N

Inserts *char* 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 *tokens* 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 L^AT_EX, the character inserted by \iow_newline: will not be recognized by TeX, which may lead to the insertion of additionnal unwanted line-breaks. This issue only affects \iow_shipout:Nn, \iow_shipout_x:Nn and direct uses of primitive operations.

2.1 Wrapping lines in output

`\iow_wrap:nnnN {⟨text⟩} {⟨run-on text⟩} {⟨set up⟩} ⟨function⟩`

New: 2012-06-28
Updated: 2015-08-05

This function will wrap the *⟨text⟩* to a fixed number of characters per line. At the start of each line which is wrapped, the *⟨run-on text⟩* will be inserted. The line character count targeted will be the value of `\l_iow_line_count_int` minus the number of characters in the *⟨run-on text⟩* 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 *⟨text⟩* and *⟨run-on text⟩* are exhaustively expanded by the function, with the following substitutions:

- `\` 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_indent:n` may be used to indent a part of the *⟨text⟩* (not the *⟨run-on text⟩*).

Additional functions may be added to the wrapping by using the *⟨set up⟩*, which is executed before the wrapping takes place: this may include overriding the substitutions listed.

Any expandable material in the *⟨text⟩* 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 *⟨function⟩*, which will typically be a wrapper around a write operation. The output of `\iow_wrap:nnnN` (*i.e.* the argument passed to the *⟨function⟩*) will consist of characters of category “other” (category code 12), with the exception of spaces which will have category “space” (category code 10). This means that the output will *not* expand further when written to a file.

TeXhackers note: Internally, `\iow_wrap:nnnN` carries out an `x`-type expansion on the *⟨text⟩* 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 *⟨text⟩*.

`\iow_indent:n`

New: 2011-09-21

`\iow_indent:n {⟨text⟩}`

In the first argument of `\iow_wrap:nnnN` (for instance in messages), indents *⟨text⟩* by four spaces. This function will not cause a line break, and only affects lines which start within the scope of the *⟨text⟩*. In case the indented *⟨text⟩* 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 TeX system in use: the standard value is 78, which is typically correct for unmodified TeXlive and MiKTeX systems.

`\c_catcode_other_space_tl`

New: 2011-09-05

Token list containing one character with category code 12, (“other”), and character code 32 (space).

2.2 Constant input–output streams

\c_term_ior

Constant input stream for reading from the terminal. Reading from this stream using `\ior_get:NN` or similar will result in a prompt from TeX of the form

`<t1>=`

\c_log_iow \c_term_iow

Constant output streams for writing to the log and to the terminal (plus the log), respectively.

2.3 Primitive conditionals

```
\if_eof:w ★ \if_eof:w ⟨stream⟩
    ⟨true code⟩
\else:
    ⟨false code⟩
\fi:
```

Tests if the `⟨stream⟩` returns “end of file”, which is true for non-existent files. The `\else:` branch is optional.

TeXhackers note: This is the TeX primitive `\ifeof`.

2.4 Internal file functions and variables

\g_file_internal_ior

Used to test for the existence of files when opening.

\l_file_internal_name_tl

Used to return the full name of a file for internal use. This is set by `\file_if_exist:nTF` and `_file_if_exist:nT`, and the value may then be used to load a file directly provided no further operations intervene.

_file_name_sanitize:nn

New: 2012-02-09

```
\_file_name_sanitize:nn {⟨name⟩} {⟨tokens⟩}
```

Exhaustively-expands the `⟨name⟩` with the exception of any category `⟨active⟩` (catcode 13) tokens, which are not expanded. The list of `⟨active⟩` tokens is taken from `\l_char_active_seq`. The `⟨sanitized name⟩` is then inserted (in braces) after the `⟨tokens⟩`, which should further process the file name. If any spaces are found in the name after expansion, an error is raised.

2.5 Internal input–output functions

_ior_open:Nn

_ior_open:No

New: 2012-01-23

```
\_ior_open:Nn ⟨stream⟩ {⟨file name⟩}
```

This function has identical syntax to the public version. However, it does not take precautions against active characters in the `⟨file name⟩`, and it does not attempt to add a `⟨path⟩` to the `⟨file name⟩`: it is therefore intended to be used by higher-level functions which have already fully expanded the `⟨file name⟩` and which need to perform multiple open or close operations. See for example the implementation of `\file_add_path:nN`,

__iow_with:Nnn

New: 2014-08-23

`__iow_with:Nnn <integer> {<value>} {<code>}`

If the *<integer>* is equal to the *<value>* then this function simply runs the *<code>*. Otherwise it saves the current value of the *<integer>*, sets it to the *<value>*, runs the *<code>*, and restores the *<integer>* to its former value. This is used to ensure that the `\newlinechar` is 10 when writing to a stream, which lets `\iow_newline:` work, and that `\errorcontextlines` is -1 when displaying a message.

Part XIX

The **I3skip** package

Dimensions and skips

LATEX3 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 `mu`). There are common features in the creation and setting of length variables, but for clarity the functions are grouped by variable type.

1 Creating and initialising `dim` variables

`\dim_new:N` `\dim_new:c`

Creates a new `<dimension>` or raises an error if the name is already taken. The declaration is global. The `<dimension>` will initially be equal to 0 pt.

`\dim_const:Nn` `\dim_const:cn`

New: 2012-03-05

`\dim_const:Nn` `\dim_const:cn`

Creates a new constant `<dimension>` or raises an error if the name is already taken. The value of the `<dimension>` will be set globally to the `<dimension expression>`.

`\dim_zero:N` `\dim_zero:c`
`\dim_gzero:N` `\dim_gzero:c`

`\dim_zero:N` `\dim_zero:c`

Sets `<dimension>` to 0 pt.

`\dim_zero_new:N` `\dim_zero_new:c`
`\dim_gzero_new:N` `\dim_gzero_new:c`

New: 2012-01-07

`\dim_zero_new:N` `\dim_zero_new:c`

Ensures that the `<dimension>` exists globally by applying `\dim_new:N` if necessary, then applies `\dim_(g)zero:N` to leave the `<dimension>` set to zero.

`\dim_if_exist_p:N` * `\dim_if_exist_p:c` *

`\dim_if_exist:NTF` * `\dim_if_exist:cTF` *

Tests whether the `<dimension>` is currently defined. This does not check that the `<dimension>` really is a dimension variable.

New: 2012-03-03

2 Setting `dim` variables

`\dim_add:Nn` `\dim_add:Nn <dimension> {<dimension expression>}`
`\dim_add:cn` Adds the result of the `<dimension expression>` to the current content of the `<dimension>`.
`\dim_gadd:Nn`
`\dim_gadd:cn`

Updated: 2011-10-22

`\dim_set:Nn` `\dim_set:Nn <dimension> {<dimension expression>}`
`\dim_set:cn` Sets `<dimension>` to the value of `<dimension expression>`, which must evaluate to a length with units.
`\dim_gset:Nn`
`\dim_gset:cn`

Updated: 2011-10-22

`\dim_set_eq:NN`
`\dim_set_eq:(cN|Nc|cc)`
`\dim_gset_eq:NN`
`\dim_gset_eq:(cN|Nc|cc)`

`\dim_set_eq:NN <dimension1> <dimension2>`
Sets the content of `<dimension1>` equal to that of `<dimension2>`.

`\dim_sub:Nn` `\dim_sub:Nn <dimension> {<dimension expression>}`
`\dim_sub:cn` Subtracts the result of the `<dimension expression>` from the current content of the `<dimension>`.
`\dim_gsub:Nn`
`\dim_gsub:cn`

Updated: 2011-10-22

3 Utilities for dimension calculations

`\dim_abs:n` ★ `\dim_abs:n {<dimexpr>}`
Updated: 2012-09-26

Converts the `<dimexpr>` to its absolute value, leaving the result in the input stream as a `<dimension denotation>`.

`\dim_max:nn` ★ `\dim_max:nn {<dimexpr1>} {<dimexpr2>}`
`\dim_min:nn` ★ `\dim_min:nn {<dimexpr1>} {<dimexpr2>}`

New: 2012-09-09
Updated: 2012-09-26

Evaluates the two `<dimension expressions>` and leaves either the maximum or minimum value in the input stream as appropriate, as a `<dimension denotation>`.

\dim_ratio:nn ★

Updated: 2011-10-22

\dim_ratio:nn {*dimexpr₁*} {*dimexpr₂*}

Parses the two *dimension expressions* and converts the ratio of the two to a form suitable for use inside a *dimension expression*. 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 ration 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
```

will display 327680/655360 on the terminal.

4 Dimension expression conditionals

\dim_compare_p:nNn ★
\dim_compare:nNnTF ★

\dim_compare_p:nNn {*dimexpr₁*} {*relation*} {*dimexpr₂*}
\dim_compare:nNnTF
{*dimexpr₁*} {*relation*} {*dimexpr₂*}
{*true code*} {*false code*}

This function first evaluates each of the *dimension expressions* as described for \dim_eval:n. The two results are then compared using the *relation*:

Equal	=
Greater than	>
Less than	<

```
\dim_compare_p:n ★ \dim_compare:nTF ★
\dim_compare_p:n
{
  ⟨dimexpr_1⟩ ⟨relation_1⟩
  ...
  ⟨dimexpr_N⟩ ⟨relation_N⟩
  ⟨dimexpr_{N+1}⟩
}
\dim_compare:nTF
{
  ⟨dimexpr_1⟩ ⟨relation_1⟩
  ...
  ⟨dimexpr_N⟩ ⟨relation_N⟩
  ⟨dimexpr_{N+1}⟩
}
{⟨true code⟩} {⟨false code⟩}
```

Updated: 2013-01-13

This function evaluates the $\langle dimension\ expressions\rangle$ as described for `\dim_eval:n` and compares consecutive result using the corresponding $\langle relation\rangle$, namely it compares $\langle dimexpr_1\rangle$ and $\langle dimexpr_2\rangle$ using the $\langle relation_1\rangle$, then $\langle dimexpr_2\rangle$ and $\langle dimexpr_3\rangle$ using the $\langle relation_2\rangle$, until finally comparing $\langle dimexpr_N\rangle$ and $\langle dimexpr_{N+1}\rangle$ using the $\langle relation_N\rangle$. The test yields `true` if all comparisons are `true`. Each $\langle dimension\ expression\rangle$ is evaluated only once, and the evaluation is lazy, in the sense that if one comparison is `false`, then no other $\langle dimension\ expression\rangle$ is evaluated and no other comparison is performed. The $\langle relations\rangle$ can be any of the following:

Equal	= or ==
Greater than or equal to	\geq
Greater than	$>$
Less than or equal to	\leq
Less than	$<$
Not equal	\neq

```
\dim_case:nn  ★ \dim_case:nnTF ★
\dim_case:nnTF ★
New: 2013-07-24
```

```

{<dimension expression>} {<test dimension expression>}
{
  {<dimexpr case1>} {<code case1>}
  {<dimexpr case2>} {<code case2>}
  ...
  {<dimexpr casen>} {<code casen>}
}
{<true code>}
{<false 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. 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! }
```

will leave “Medium” in the input stream.

5 Dimension expression loops

```
\dim_do_until:nNnn ★ \dim_do_until:nNnn {<dimexpr1>} {<relation>} {<dimexpr2>} {<code>}
```

Places the *<code>* in the input stream for TeX 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>* will be inserted into the input stream again and a loop will occur until the *<relation>* is `true`.

```
\dim_do_while:nNnn ★ \dim_do_while:nNnn {<dimexpr1>} {<relation>} {<dimexpr2>} {<code>}
```

Places the *<code>* in the input stream for TeX 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>* will be inserted into the input stream again and a loop will occur until the *<relation>* is `false`.

```
\dim_until_do:nNnn ★ \dim_until_do:nNnn {<dimexpr1>} {<relation>} {<dimexpr2>} {<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 TeX the test will be repeated, and a loop will occur until the test is `true`.

<code>\dim_while_do:nNnn</code>	<code>\dim_while_do:nNnn {\langle dimexpr_1\rangle} {\langle relation\rangle} {\langle dimexpr_2\rangle} {\langle code\rangle}</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 <code>true</code> . After the <i>code</i> has been processed by TeX the test will be repeated, and a loop will occur until the test is <code>false</code> .
<code>\dim_do_until:nn</code>	<code>\dim_do_until:nn {\langle dimension relation\rangle} {\langle code\rangle}</code>
Updated: 2013-01-13	Places the <i>code</i> in the input stream for TeX to process, and then evaluates the <i>dimension relation</i> as described for <code>\dim_compare:nTF</code> . If the test is <code>false</code> then the <i>code</i> will be inserted into the input stream again and a loop will occur until the <i>relation</i> is <code>true</code> .
<code>\dim_do_while:nn</code>	<code>\dim_do_while:nn {\langle dimension relation\rangle} {\langle code\rangle}</code>
Updated: 2013-01-13	Places the <i>code</i> in the input stream for TeX to process, and then evaluates the <i>dimension relation</i> as described for <code>\dim_compare:nTF</code> . If the test is <code>true</code> then the <i>code</i> will be inserted into the input stream again and a loop will occur until the <i>relation</i> is <code>false</code> .
<code>\dim_until_do:nn</code>	<code>\dim_until_do:nn {\langle dimension relation\rangle} {\langle code\rangle}</code>
Updated: 2013-01-13	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 <code>false</code> . After the <i>code</i> has been processed by TeX the test will be repeated, and a loop will occur until the test is <code>true</code> .
<code>\dim_while_do:nn</code>	<code>\dim_while_do:nn {\langle dimension relation\rangle} {\langle code\rangle}</code>
Updated: 2013-01-13	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 <code>true</code> . After the <i>code</i> has been processed by TeX the test will be repeated, and a loop will occur until the test is <code>false</code> .

6 Using dim expressions and variables

<code>\dim_eval:n</code>	<code>\dim_eval:n {\langle dimension expression\rangle}</code>
Updated: 2011-10-22	Evaluates the <i>dimension expression</i> , expanding any dimensions and token list variables within the <i>expression</i> to their content (without requiring <code>\dim_use:N/\t1_use:N</code>) and applying the standard mathematical rules. The result of the calculation is left in the input stream as a <i>dimension denotation</i> after two expansions. This will be expressed in points (pt), and will require suitable termination if used in a TeX-style assignment as it is <i>not</i> an <i>internal dimension</i> .
<code>\dim_use:N</code>	<code>\dim_use:N {\langle dimension\rangle}</code>
<code>\dim_use:c</code>	Recovering the content of a <i>dimension</i> and places it directly in the input stream. An error will be raised if the variable does not exist or if it is invalid. Can be omitted in places where a <i>dimension</i> is required (such as in the argument of <code>\dim_eval:n</code>).

TeXhackers note: `\dim_use:N` is the TeX primitive `\the:` this is one of several L^AT_EX3 names for this primitive.

\dim_to_decimal:n *New: 2014-07-15

\dim_to_decimal:n {*dimexpr*}

Evaluates the *dimension expression*, 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.

\dim_to_decimal_in_bp:n *New: 2014-07-15

\dim_to_decimal_in_bp:n {*dimexpr*}

Evaluates the *dimension expression*, and leaves the result, expressed in big points (bp) 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_in_bp:n { 1pt }

leaves 0.99628 in the input stream, *i.e.* the magnitude of one (TeX) point when converted to big points.

\dim_to_decimal_in_sp:n *New: 2015-05-18

\dim_to_decimal_in_sp:n {*dimexpr*}

Evaluates the *dimension expression*, and leaves the result, expressed in scaled points (sp) in the input stream, with *no units*. The result will necessarily be an integer.

\dim_to_decimal_in_unit:nn * \dim_to_decimal_in_unit:nn {*dimexpr*₁} {*dimexpr*₂}New: 2014-07-15

Evaluates the *dimension expressions*, and leaves the value of *dimexpr*₁, expressed in a unit given by *dimexpr*₂, in the input stream. The result is a decimal number, 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_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-Tex primitives, which is required internally by \dim_to_decimal_in_unit:nn.

\dim_to_fp:n *

New: 2012-05-08

`\dim_to_fp:n {<dimexpr>}`

Expands to an internal floating point number equal to the value of the $\langle dimexpr \rangle$ in pt. Since dimension expressions are evaluated much faster than their floating point equivalent, `\dim_to_fp:n` can be used to speed up parts of a computation where a low precision is acceptable.

7 Viewing dim variables

\dim_show:N

\dim_show:c

`\dim_show:N <dimension>`

Displays the value of the $\langle dimension \rangle$ on the terminal.

\dim_show:n

New: 2011-11-22

Updated: 2015-08-07

`\dim_show:n {<dimension expression>}`

Displays the result of evaluating the $\langle dimension expression \rangle$ on the terminal.

\dim_log:N

\dim_log:c

New: 2014-08-22

Updated: 2015-08-03

`\dim_log:N <dimension>`

Writes the value of the $\langle dimension \rangle$ in the log file.

\dim_log:n

New: 2014-08-22

Updated: 2015-08-07

`\dim_log:n {<dimension expression>}`

Writes the result of evaluating the $\langle dimension expression \rangle$ in the log file.

8 Constant dimensions

\c_max_dim

The maximum value that can be stored as a dimension. This can also be used as a component of a skip.

\c_zero_dim

A zero length as a dimension. This can also be used as a component of a skip.

9 Scratch dimensions

\l_tmpa_dim

\l_tmpb_dim

Scratch dimension 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.

\g_tmpa_dim

\g_tmpb_dim

Scratch dimension 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.

10 Creating and initialising skip variables

\skip_new:N
\skip_new:c

\skip_new:N *skip*
Creates a new *skip* or raises an error if the name is already taken. The declaration is global. The *skip* will initially be equal to 0 pt.

\skip_const:Nn
\skip_const:cn

New: 2012-03-05

\skip_const:Nn *skip* {*skip expression*}

Creates a new constant *skip* or raises an error if the name is already taken. The value of the *skip* will be set globally to the *skip expression*.

\skip_zero:N
\skip_zero:c
\skip_gzero:N
\skip_gzero:c

\skip_zero:N *skip*

Sets *skip* to 0 pt.

\skip_zero_new:N
\skip_zero_new:c
\skip_gzero_new:N
\skip_gzero_new:c

New: 2012-01-07

\skip_zero_new:N *skip*

Ensures that the *skip* exists globally by applying \skip_new:N if necessary, then applies \skip_(g)zero:N to leave the *skip* set to zero.

\skip_if_exist_p:N ★
\skip_if_exist_p:c ★
\skip_if_exist:NTF ★
\skip_if_exist:cTF ★

New: 2012-03-03

\skip_if_exist_p:N *skip*
\skip_if_exist:NTF *skip* {*true code*} {*false code*}

Tests whether the *skip* is currently defined. This does not check that the *skip* really is a skip variable.

\skip_add:Nn
\skip_add:cn
\skip_gadd:Nn
\skip_gadd:cn

Updated: 2011-10-22

\skip_add:Nn *skip* {*skip expression*}

Adds the result of the *skip expression* to the current content of the *skip*.

\skip_set:Nn
\skip_set:cn
\skip_gset:Nn
\skip_gset:cn

Updated: 2011-10-22

\skip_set:Nn *skip* {*skip expression*}

Sets *skip* to the value of *skip expression*, which must evaluate to a length with units and may include a rubber component (for example 1 cm plus 0.5 cm).

\skip_set_eq:NN
\skip_set_eq:(cN|Nc|cc)
\skip_gset_eq:NN
\skip_gset_eq:(cN|Nc|cc)

\skip_set_eq:NN *skip*₁ *skip*₂

Sets the content of *skip*₁ equal to that of *skip*₂.

```
\skip_sub:Nn
\skip_sub:cn
\skip_gsub:Nn
\skip_gsub:cn
```

Updated: 2011-10-22

`\skip_sub:Nn <skip> {<skip expression>}`

Subtracts the result of the *<skip expression>* from the current content of the *<skip>*.

```
\skip_if_eq_p:nn *
\skip_if_eq:nnTF *
```

`\skip_if_eq_p:nn {{skipexpr_1}} {{skipexpr_2}}
\dim_compare:nTF
 {{skipexpr_1}} {{skipexpr_2}}
 {{true code}} {{false code}}`

This function first evaluates each of the *<skip expressions>* 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.

```
\skip_if_finite_p:n *
\skip_if_finite:nTF *
```

New: 2012-03-05

`\skip_if_finite_p:n {{skipexpr}}
\skip_if_finite:nTF {{skipexpr}} {{true code}} {{false code}}`

Evaluates the *<skip expression>* as described for `\skip_eval:n`, and then tests if all of its components are finite.

13 Using skip expressions and variables

```
\skip_eval:n *
```

Updated: 2011-10-22

`\skip_eval:n {<skip expression>}`

Evaluates the *<skip expression>*, expanding any skips and token list variables within the *<expression>* 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 *<glue denotation>* after two expansions. This will be expressed in points (`pt`), and will require suitable termination if used in a TeX-style assignment as it is *not* an *<internal glue>*.

```
\skip_use:N *
\skip_use:c *
```

`\skip_use:N <skip>`

Recovering the content of a *<skip>* and places it directly in the input stream. An error will be raised if the variable does not exist or if it is invalid. Can be omitted in places where a *<dimension>* is required (such as in the argument of `\skip_eval:n`).

TEXhackers note: `\skip_use:N` is the TeX primitive `\the:` this is one of several L^AT_EX3 names for this primitive.

14 Viewing skip variables

```
\skip_show:N
\skip_show:c
```

`\skip_show:N <skip>`

Displays the value of the *<skip>* on the terminal.

`\skip_show:n` `\skip_show:n {<skip expression>}`
New: 2011-11-22
Updated: 2015-08-07

Displays the result of evaluating the *<skip expression>* on the terminal.

`\skip_log:N` `\skip_log:N {skip}`
`\skip_log:c`
New: 2014-08-22
Updated: 2015-08-03

Writes the value of the *(skip)* in the log file.

`\skip_log:n` `\skip_log:n {<skip expression>}`
New: 2014-08-22
Updated: 2015-08-07

Writes the result of evaluating the *<skip expression>* in the log file.

15 Constant skips

`\c_max_skip`
Updated: 2012-11-02

The maximum value that can be stored as a skip (equal to `\c_max_dim` in length), with no stretch nor shrink component.

`\c_zero_skip`
Updated: 2012-11-01

A zero length as a skip, with no stretch nor shrink component.

16 Scratch skips

`\l_tmpa_skip` `\l_tmpb_skip`

Scratch skip 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.

`\g_tmpa_skip` `\g_tmpb_skip`

Scratch skip 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.

17 Inserting skips into the output

`\skip_horizontal:N` `\skip_horizontal:N {skip}`
`\skip_horizontal:c` `\skip_horizontal:n {<skipexpr>}`
`\skip_horizontal:n`
Updated: 2011-10-22

Inserts a horizontal *(skip)* into the current list.

TeXhackers note: `\skip_horizontal:N` is the TeX primitive `\hskip` renamed.

```
\skip_vertical:N  
\skip_vertical:c  
\skip_vertical:n
```

Updated: 2011-10-22

```
\skip_vertical:N <skip>  
\skip_vertical:n {<skipexpr>}
```

Inserts a vertical *<skip>* into the current list.

TeXhackers note: `\skip_vertical:N` is the TeX primitive `\vskip` renamed.

18 Creating and initialising **muskip** variables

```
\muskip_new:N  
\muskip_new:c
```

```
\muskip_new:N <muskip>
```

Creates a new *<muskip>* or raises an error if the name is already taken. The declaration is global. The *<muskip>* will initially be equal to 0mu.

```
\muskip_const:Nn  
\muskip_const:cn
```

New: 2012-03-05

```
\muskip_const:Nn <muskip> {<muskip expression>}
```

Creates a new constant *<muskip>* or raises an error if the name is already taken. The value of the *<muskip>* will be set globally to the *<muskip expression>*.

```
\muskip_zero:N  
\muskip_zero:c  
\muskip_gzero:N  
\muskip_gzero:c
```

```
\skip_zero:N <muskip>
```

Sets *<muskip>* to 0mu.

```
\muskip_zero_new:N  
\muskip_zero_new:c  
\muskip_gzero_new:N  
\muskip_gzero_new:c
```

New: 2012-01-07

```
\muskip_zero_new:N <muskip>
```

Ensures that the *<muskip>* exists globally by applying `\muskip_new:N` if necessary, then applies `\muskip_(g)zero:N` to leave the *<muskip>* set to zero.

```
\muskip_if_exist_p:N ★  
\muskip_if_exist_p:c ★  
\muskip_if_exist:NTF ★  
\muskip_if_exist:cTF ★
```

New: 2012-03-03

```
\muskip_if_exist_p:N <muskip>
```

```
\muskip_if_exist:NTF <muskip> {<true code>} {<false code>}
```

Tests whether the *<muskip>* is currently defined. This does not check that the *<muskip>* really is a muskip variable.

19 Setting **muskip** variables

```
\muskip_add:Nn  
\muskip_add:cn  
\muskip_gadd:Nn  
\muskip_gadd:cn
```

Updated: 2011-10-22

```
\muskip_add:Nn <muskip> {<muskip expression>}
```

Adds the result of the *<muskip expression>* to the current content of the *<muskip>*.

```
\muskip_set:Nn
\muskip_set:cn
\muskip_gset:Nn
\muskip_gset:cn
```

Updated: 2011-10-22

```
\muskip_set:Nn <muskip> {<muskip expression>}
```

Sets *<muskip>* to the value of *<muskip expression>*, which must evaluate to a math length with units and may include a rubber component (for example 1 mu plus 0.5 mu).

```
\muskip_set_eq:NN
\muskip_set_eq:(cN|Nc|cc)
\muskip_gset_eq:NN
\muskip_gset_eq:(cN|Nc|cc)
```

```
\muskip_sub:Nn
\muskip_sub:cn
\muskip_gsub:Nn
\muskip_gsub:cn
```

Updated: 2011-10-22

```
\muskip_sub:Nn <muskip> {<muskip expression>}
```

Subtracts the result of the *<muskip expression>* from the current content of the *<skip>*.

20 Using muskip expressions and variables

```
\muskip_eval:n *
```

Updated: 2011-10-22

```
\muskip_eval:n {<muskip expression>}
```

Evaluates the *<muskip expression>*, expanding any skips and token list variables within the *<expression>* to their content (without requiring `\muskip_use:N/\t1_use:N`) and applying the standard mathematical rules. The result of the calculation is left in the input stream as a *<muglue denotation>* after two expansions. This will be expressed in mu, and will require suitable termination if used in a TeX-style assignment as it is *not* an *<internal muglue>*.

```
\muskip_use:N *
\muskip_use:c *
```

```
\muskip_use:N <muskip>
```

Recovers the content of a *<skip>* and places it directly in the input stream. An error will be raised if the variable does not exist or if it is invalid. Can be omitted in places where a *<dimension>* is required (such as in the argument of `\muskip_eval:n`).

TeXhackers note: `\muskip_use:N` is the TeX primitive `\the:` this is one of several L^AT_EX3 names for this primitive.

21 Viewing muskip variables

```
\muskip_show:N
\muskip_show:c
```

Updated: 2015-08-03

```
\muskip_show:N <muskip>
```

Displays the value of the *<muskip>* on the terminal.

```
\muskip_show:n
```

New: 2011-11-22

Updated: 2015-08-07

```
\muskip_show:n {<muskip expression>}
```

Displays the result of evaluating the *<muskip expression>* on the terminal.

<u>\muskip_log:N</u>	\muskip_log:N <i><muskip></i>
<u>\muskip_log:c</u>	Writes the value of the <i><muskip></i> in the log file.
New: 2014-08-22	
Updated: 2015-08-03	
<u>\muskip_log:n</u>	\muskip_log:n {<muskip expression>}
New: 2014-08-22	Writes the result of evaluating the <i><muskip expression></i> in the log file.
Updated: 2015-08-07	

22 Constant muskips

<u>\c_max_muskip</u>	The maximum value that can be stored as a muskip, with no stretch nor shrink component.
<u>\c_zero_muskip</u>	A zero length as a muskip, with no stretch nor shrink component.

23 Scratch muskips

<u>\l_tmpa_muskip</u>	Scratch muskip for local assignment. These are never used by the kernel code, and so are safe for use with any LATEX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
<u>\g_tmpa_muskip</u>	Scratch muskip for global assignment. These are never used by the kernel code, and so are safe for use with any LATEX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

24 Primitive conditional

<u>\if_dim:w</u>	\if_dim:w <i>(dimen1)</i> <i>(relation)</i> <i>(dimen2)</i> <i>(true code)</i> \else: <i>(false)</i> \fi:
	Compare two dimensions. The <i>(relation)</i> is one of <, = or > with category code 12.

TeXhackers note: This is the TeX primitive \ifdim.

25 Internal functions

<code>__dim_eval:w</code> *	<code>__dim_eval:w <dimexpr> __dim_eval_end:</code>
<code>__dim_eval_end:</code> *	Evaluates <i><dimension expression></i> as described for <code>\dim_eval:n</code> . The evaluation stops when an unexpandable token which is not a valid part of a dimension is read or when <code>__dim_eval_end:</code> is reached. The latter is gobbled by the scanner mechanism: <code>__dim_eval_end:</code> itself is unexpandable but used correctly the entire construct is expandable.

TExhackers note: This is the ε -TEx primitive `\dimexpr`.

Part XX

The **I3keys** 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  
}
```

At a document level, `\keys_set:nn` will be 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`; spaces are *ignored* in key names. As will be discussed in section 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_t1 { key }
\keys_define:nn { mymodule }
{
    \l_mymodule_tmp_t1 .code:n = code
}
```

will create a key called `\l_mymodule_tmp_t1`, and not one called `key`.

1 Creating keys

`\keys_define:nn`

Updated: 2015-11-07

`\keys_define:nn {<module>} {<keyval list>}`

Parses the `<keyval list>` and defines the keys listed there for `<module>`. The `<module>` name should be a text value, but there are no restrictions on the nature of the text. In practice the `<module>` should be chosen to be unique to the module in question (unless deliberately adding keys to an existing module).

The `<keyval list>` 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 `<key>`, which when used may be supplied with a `<value>`. 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., will override one another. Some other properties are mutually exclusive, notably `.value_required:n` and `.value_forbidden:n`, and so will 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 will define the key within the current TeX scope.

`.bool_set:N`
`.bool_set:c`
`.bool_gset:N`
`.bool_gset:c`

Updated: 2013-07-08

`<key> .bool_set:N = <boolean>`

Defines `<key>` to set `<boolean>` to `<value>` (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

`<key> .bool_set_inverse:N = <boolean>`

Defines `<key>` to set `<boolean>` to the logical inverse of `<value>` (which must be either `true` or `false`). If the `<boolean>` does not exist, it will be created globally at the point that the key is set up.

`.choice:`

`<key> .choice:`

Sets `<key>` to act as a choice key. Each valid choice for `<key>` must then be created, as discussed in section 3.

`.choices:nn`
`.choices:(Vn|on|xn)`

New: 2011-08-21

Updated: 2013-07-10

`<key> .choices:nn = {<choices>} {<code>}`

Sets `<key>` to act as a choice key, and defines a series `<choices>` which are implemented using the `<code>`. Inside `<code>`, `\l_keys_choice_t1` will be the name of the choice made, and `\l_keys_choice_int` will be the position of the choice in the list of `<choices>` (indexed from 1). Choices are discussed in detail in section 3.

`.clist_set:N`
`.clist_set:c`
`.clist_gset:N`
`.clist_gset:c`

New: 2011-09-11

`<key> .clist_set:N = <comma list variable>`

Defines `<key>` to set `<comma list variable>` to `<value>`. Spaces around commas and empty items will be stripped. If the variable does not exist, it will be created globally at the point that the key is set up.

`.code:n`

Updated: 2013-07-10

`<key> .code:n = {<code>}`

Stores the `<code>` for execution when `<key>` is used. The `<code>` can include one parameter (#1), which will be the `<value>` given for the `<key>`. The x-type variant will expand `<code>` at the point where the `<key>` is created.

.default:n
.default:(v|o|x)
Updated: 2013-07-09

`<key> .default:n = {<default>}`

Creates a `<default>` value for `<key>`, which is used if no value is given. This will be used if only the key name is given, but not if a blank `<value>` is given:

```
\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 '
}
```

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 will not trigger an error.

.dim_set:N
.dim_set:c
.dim_gset:N
.dim_gset:c

`<key> .dim_set:N = <dimension>`

Defines `<key>` to set `<dimension>` to `<value>` (which must a dimension expression). If the variable does not exist, it will be created globally at the point that the key is set up.

.fp_set:N
.fp_set:c
.fp_gset:N
.fp_gset:c

`<key> .fp_set:N = <floating point>`

Defines `<key>` to set `<floating point>` to `<value>` (which must a floating point expression). If the variable does not exist, it will be created globally at the point that the key is set up.

.groups:n
New: 2013-07-14

`<key> .groups:n = {<groups>}`

Defines `<key>` as belonging to the `<groups>` declared. Groups provide a “secondary axis” for selectively setting keys, and are described in Section 6.

.inherit:n
New: 2016-11-22

`<key> .inherit:n = {<parents>}`

Specifies that the `<key>` path should inherit the keys listed as `<parents>`. For example, with setting

```
\keys_define:n { foo } { test .code:n = \tl_show:n {#1} }
\keys_define:n { } { bar .inherit:n = foo }
```

setting

```
\keys_set:n { bar } { test = a }
```

will be equivalent to

```
\keys_set:n { foo } { test = a }
```

.initial:n
.initial:(V|o|x)

Updated: 2013-07-09

`<key> .initial:n = {<value>}`

Initialises the `<key>` with the `<value>`, equivalent to

`\keys_set:nn {<module>} { <key> = <value> }`

.int_set:N
.int_set:c
.int_gset:N
.int_gset:c

`<key> .int_set:N = <integer>`

Defines `<key>` to set `<integer>` to `<value>` (which must be an integer expression). If the variable does not exist, it will be created globally at the point that the key is set up.

.meta:n

Updated: 2013-07-10

`<key> .meta:n = {<keyval list>}`

Makes `<key>` a meta-key, which will set `<keyval list>` in one go. If `<key>` is given with a value at the time the key is used, then the value will be passed through to the subsidiary `<keys>` for processing (as #1).

.meta:nn

New: 2013-07-10

`<key> .meta:nn = {<path>} {<keyval list>}`

Makes `<key>` a meta-key, which will set `<keyval list>` in one go using the `<path>` in place of the current one. If `<key>` is given with a value at the time the key is used, then the value will be passed through to the subsidiary `<keys>` for processing (as #1).

.multichoice:

New: 2011-08-21

`<key> .multichoice:`

Sets `<key>` to act as a multiple choice key. Each valid choice for `<key>` must then be created, as discussed in section 3.

.multichoice:nn
.multichoice:(Vn|on|xn)

New: 2011-08-21

Updated: 2013-07-10

`<key> .multichoice:nn {<choices>} {<code>}`

Sets `<key>` to act as a multiple choice key, and defines a series `<choices>` which are implemented using the `<code>`. Inside `<code>`, `\l_keys_choice_t1` will be the name of the choice made, and `\l_keys_choice_int` will be the position of the choice in the list of `<choices>` (indexed from 1). Choices are discussed in detail in section 3.

.skip_set:N
.skip_set:c
.skip_gset:N
.skip_gset:c

`<key> .skip_set:N = <skip>`

Defines `<key>` to set `<skip>` to `<value>` (which must be a skip expression). If the variable does not exist, it will be created globally at the point that the key is set up.

.tl_set:N
.tl_set:c
.tl_gset:N
.tl_gset:c

`<key> .tl_set:N = <token list variable>`

Defines `<key>` to set `<token list variable>` to `<value>`. If the variable does not exist, it will be created globally at the point that the key is set up.

.tl_set_x:N
.tl_set_x:c
.tl_gset_x:N
.tl_gset_x:c

`<key> .tl_set_x:N = <token list variable>`

Defines `<key>` to set `<token list variable>` to `<value>`, which will be subjected to an `x`-type expansion (*i.e.* using `\tl_set:Nx`). If the variable does not exist, it will be created globally at the point that the key is set up.

.undefine:	<code><key> .undefine:</code>
New: 2015-07-14	Removes the definition of the <code><key></code> within the current scope.
.value_forbidden:n	<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 <code>false</code> will cancel the restriction.
.value_required:n	<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 <code>false</code> will cancel the restriction.

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 { module / 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 `module/subgroup/key`.

As will be illustrated in the next section, this subdivision is particularly relevant to making multiple choices.

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 5. 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:nnnn { 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`.

4 Setting keys

```
\keys_set:nn
\keys_set:(nV|nv|no)
```

Updated: 2015-11-07

```
\l_keys_key_tl
\l_keys_path_tl
\l_keys_value_tl
```

Updated: 2015-07-14

```
\keys_set:nn {⟨module⟩} {⟨keyval list⟩}
```

Parses the *⟨keyval list⟩*, and sets those keys which are defined for *⟨module⟩*. The behaviour on finding an unknown key can be set by defining a special `unknown` key: this will be illustrated later.

For each key processed, information of the full *path* of the key, the *name* of the key and the *value* of the key is available within three token list variables. These may be used within the code of the key.

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_tl`, and will have been processed by `\tl_to_str:n`.

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_tl`, and will have been processed by `\tl_to_str:n`.

5 Handling of unknown keys

If a key has not previously been defined (is unknown), `\keys_set:nn` will look 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_tl'~to~'#1'.
}
```

```
\keys_set_known:nnN           \keys_set_known:nnN {<module>} {<keyval list>} <t1>
\keys_set_known:(nVN|nvN|noN)
\keys_set_known:nn
\keys_set_known:(nV|nv|no)
```

New: 2011-08-23

Updated: 2017-05-27

In some cases, the desired behavior is to simply ignore unknown keys, collecting up information on these for later processing. The `\keys_set_known:nnN` 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. The key-value pairs for each *unknown* key name will be stored in the `<t1>` in a comma-separated form (*i.e.* an edited version of the `<keyval list>`). The `\keys_set_known:nn` version skips this stage.

Use of `\keys_set_known:nnN` can be nested, with the correct residual `<keyval list>` returned at each stage.

6 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` will attempt 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 ,
}
```

will assign `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.

```
\keys_set_filter:nnnN          \keys_set_filter:nnnN {<module>} {<groups>} {<keyval list>} <t1>
\keys_set_filter:(nnVN|nnvN|nnoN)
\keys_set_filter:nnn
\keys_set_filter:(nnV|nnv|nno)
```

New: 2013-07-14
Updated: 2017-05-27

Activates key filtering in an “opt-out” sense: keys assigned to any of the *<groups>* specified will be ignored. The *<groups>* are given as a comma-separated list. Unknown keys are not assigned to any group and will thus always be set. The key–value pairs for each key which is filtered out will be stored in the *<t1>* 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.

```
\keys_set_groups:nnn          \keys_set_groups:nnn {<module>} {<groups>} {<keyval list>}
\keys_set_groups:(nnV|nnv|nno)
```

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 will be set. The *<groups>* are given as a comma-separated list. Unknown keys are not assigned to any group and will thus never be set.

7 Utility functions for keys

```
\keys_if_exist_p:nn *          \keys_if_exist_p:nn {<module>} {<key>}
\keys_if_exist:nnTF *
```

Updated: 2015-11-07

Tests if the *<key>* exists for *<module>*, *i.e.* if any code has been defined for *<key>*.

```
\keys_if_choice_exist_p:nnn *  \keys_if_choice_exist_p:nnn {<module>} {<key>} {<choice>}
\keys_if_choice_exist:nnnTF *
```

New: 2011-08-21

Updated: 2015-11-07

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.

```
\keys_show:nn
```

Updated: 2015-08-09

```
\keys_show:nn {<module>} {<key>}
```

Displays in the terminal the information associated to the *<key>* for a *<module>*, including the function which is used to actually implement it.

```
\keys_log:nn
```

New: 2014-08-22

Updated: 2015-08-09

```
\keys_log:nn {<module>} {<key>}
```

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.

8 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 $\langle\text{key-value list}\rangle$ into $\langle\text{keys}\rangle$ and associated $\langle\text{values}\rangle$. 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 will have exactly one set removed (after space trimming), thus

```
key = {value here},
```

and

```
key = value here,
```

are treated identically.

\keyval_parse:NNn

Updated: 2011-09-08

\keyval_parse:NNn <function₁> <function₂> {<key-value list>}

Parses the <key-value list> into a series of <keys> and associated <values>, or keys alone (if no <value> was given). <function₁> should take one argument, while <function₂> should absorb two arguments. After \keyval_parse:NNn has parsed the <key-value list>, <function₁> will be used to process keys given with no value and <function₂> will be used to process keys given with a value. The order of the <keys> in the <key-value list> will be preserved. Thus

```
\keyval_parse:NNn \function:n \function:nn
{ key1 = value1 , key2 = value2, key3 = , key4 }
```

will be 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 <key> and <value>, then one *outer* set of braces is removed from the <key> and <value> as part of the processing.

Part XXI

The **I3fp** 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: $x < y$, $x \leq y$, $x >? y$, $x != y$ 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 .
- Trigonometry: $\sin x$, $\cos x$, $\tan x$, $\cot x$, $\sec x$, $\csc x$ expecting their arguments in radians, and $\text{sind } x$, $\text{cosd } x$, $\text{tand } x$, $\text{cotd } x$, $\text{seed } x$, $\text{csed } x$ 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 $\text{asind } x$, $\text{acosd } x$, $\text{atand } x$, $\text{cotd } x$, $\text{asecd } x$, $\text{acsed } x$ giving a result in degrees.

(not yet) Hyperbolic functions and their inverse functions: $\sinh x$, $\cosh x$, $\tanh x$, $\coth x$, $\sech x$, $\text{csch } x$, and $\text{asinh } x$, $\text{acosh } x$, $\text{atanh } x$, $\text{acoth } x$, $\text{asech } x$, $\text{acsch } x$.

- Extrema: $\max(x, y, \dots)$, $\min(x, y, \dots)$, $\text{abs}(x)$.
- Rounding functions ($n = 0$ by default, $t = \text{NaN}$ by default): $\text{trunc}(x, n)$ rounds towards zero, $\text{floor}(x, n)$ rounds towards $-\infty$, $\text{ceil}(x, n)$ rounds towards $+\infty$, $\text{round}(x, n, t)$ 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: $\text{rand}()$, $\text{randint}(m, n)$ in pdfTeX and LuaTeX engines.
- Constants: pi , deg (one degree in radians).
- Dimensions, automatically expressed in points, e.g., pc is 12.
- Automatic conversion (no need for $\backslash\langle\text{type}\rangle\text{use:N}$) of integer, dimension, and skip variables to floating points, expressing dimensions in points and ignoring the stretch and shrink components of skips.

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. See section 9.1 for a description of what a floating point is, section 9.2 for details about how an expression is parsed, and section 9.3 to know what the various operations do. Some operations may raise exceptions (error messages), described in section 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} $.
```

But in all fairness, this 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{\calcnum}{m}
{ \num{ \fp_to_scientific:n {#1} } }
\ExplSyntaxOff
\begin{document}
\calcnum{2 pi * sin(2.3^5)}
\end{document}
```

1 Creating and initialising floating point variables

`\fp_new:N` `\fp_new:c`

Updated: 2012-05-08

`\fp_new:N <fp var>`

Creates a new `<fp var>` or raises an error if the name is already taken. The declaration is global. The `<fp var>` will initially be `+0`.

`\fp_const:Nn`

`\fp_const:cn`

Updated: 2012-05-08

`\fp_const:Nn <fp var> {<floating point expression>}`

Creates a new constant `<fp var>` or raises an error if the name is already taken. The `<fp var>` will be set globally equal to the result of evaluating the `<floating point expression>`.

`\fp_zero:N`

`\fp_zero:c`

`\fp_gzero:N`

`\fp_gzero:c`

Updated: 2012-05-08

`\fp_zero:N <fp var>`

Sets the `<fp var>` to `+0`.

`\fp_zero_new:N`

`\fp_zero_new:c`

`\fp_gzero_new:N`

`\fp_gzero_new:c`

Updated: 2012-05-08

`\fp_zero_new:N <fp var>`

Ensures that the `<fp var>` exists globally by applying `\fp_new:N` if necessary, then applies `\fp_(g)zero:N` to leave the `<fp var>` set to `+0`.

2 Setting floating point variables

`\fp_set:Nn`

`\fp_set:cn`

`\fp_gset:Nn`

`\fp_gset:cn`

Updated: 2012-05-08

`\fp_set:Nn <fp var> {<floating point expression>}`

Sets `<fp var>` equal to the result of computing the `<floating point expression>`.

```
\fp_set_eq:NN
\fp_set_eq:(cN|Nc|cc)
\fp_gset_eq:NN
\fp_gset_eq:(cN|Nc|cc)
```

Updated: 2012-05-08

```
\fp_set_eq:NN <fp var1> <fp var2>
```

Sets the floating point variable $\langle fp \ var_1 \rangle$ equal to the current value of $\langle fp \ var_2 \rangle$.

```
\fp_add:Nn
\fp_add:cn
\fp_gadd:Nn
\fp_gadd:cn
```

Updated: 2012-05-08

```
\fp_add:Nn <fp var> {<floating point expression>}
```

Adds the result of computing the $\langle floating \ point \ expression \rangle$ to the $\langle fp \ var \rangle$.

```
\fp_sub:Nn
\fp_sub:cn
\fp_gsub:Nn
\fp_gsub:cn
```

Updated: 2012-05-08

```
\fp_sub:Nn <fp var> {<floating point expression>}
```

Subtracts the result of computing the $\langle floating \ point \ expression \rangle$ from the $\langle fp \ var \rangle$.

3 Using floating point numbers

```
\fp_eval:n *
```

New: 2012-05-08

Updated: 2012-07-08

```
\fp_eval:n {<floating point expression>}
```

Evaluates the $\langle floating \ point \ expression \rangle$ 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. This function is identical to `\fp_to_decimal:n`.

```
\fp_to_decimal:N *
\fp_to_decimal:c *
\fp_to_decimal:n *
```

New: 2012-05-08

Updated: 2012-07-08

```
\fp_to_decimal:N <fp var>
\fp_to_decimal:n {<floating point expression>}
```

Evaluates the $\langle floating \ point \ expression \rangle$ 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.

```
\fp_to_dim:N *
\fp_to_dim:c *
\fp_to_dim:n *
```

Updated: 2016-03-22

```
\fp_to_dim:N <fp var>
\fp_to_dim:n {<floating point expression>}
```

Evaluates the $\langle floating \ point \ expression \rangle$ 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. The values $\pm\infty$ and NaN trigger an “invalid operation” exception.

\fp_to_int:N ★ \fp_to_int:c ★ \fp_to_int:n ★
 Updated: 2012-07-08

\fp_to_int:N *(fp var)*
 \fp_to_int:n {*floating point expression*}
 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. The values $\pm\infty$ and `NaN` trigger an “invalid operation” exception.

\fp_to_scientific:N ★ \fp_to_scientific:c ★ \fp_to_scientific:n ★
 New: 2012-05-08
 Updated: 2016-03-22

\fp_to_scientific:N *(fp var)*
 \fp_to_scientific:n {*floating point expression*}
 Evaluates the *floating point expression* and expresses the result in scientific notation:
 $\langle optional \text{--} \rangle \langle digit \rangle . \langle 15 \text{ digits} \rangle \mathbf{e} \langle optional \text{ sign} \rangle \langle exponent \rangle$

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).

\fp_to_t1:N ★ \fp_to_t1:c ★ \fp_to_t1:n ★
 Updated: 2016-03-22

\fp_to_t1:N *(fp var)*
 \fp_to_t1:n {*floating point expression*}
 Evaluates the *floating point expression* 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 \fp_to_scientific:n). 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 \fp_to_decimal:n). Negative numbers start with `-`. The special values ± 0 , $\pm\infty$ and `NaN` are rendered as `0`, `-0`, `inf`, `-inf`, and `nan` respectively. Normal category codes apply and thus `inf` or `nan`, if produced, will be made up of letters.

\fp_use:N ★ \fp_use:c ★
 Updated: 2012-07-08

\fp_use:N *(fp var)*
 Inserts the value of the *fp var* 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 `NaN` trigger an “invalid operation” exception. This function is identical to \fp_to_decimal:N.

4 Floating point conditionals

\fp_if_exist_p:N ★ \fp_if_exist_p:c ★ \fp_if_exist:NTF ★ \fp_if_exist:cTF ★
 Updated: 2012-05-08

\fp_if_exist_p:N *(fp var)*
 \fp_if_exist:NTF *(fp var)* {*true code*} {*false code*}
 Tests whether the *fp var* is currently defined. This does not check that the *fp var* really is a floating point variable.

```
\fp_compare_p:nNn ★
\fp_compare:nNnTF ★
```

Updated: 2012-05-08

```
\fp_compare_p:nNn {⟨fpexpr₁⟩} ⟨relation⟩ {⟨fpexpr₂⟩}
\fp_compare:nNnTF {⟨fpexpr₁⟩} ⟨relation⟩ {⟨fpexpr₂⟩} {⟨true code⟩} {⟨false code⟩}
```

Compares the $\langle fpexpr_1 \rangle$ and the $\langle fpexpr_2 \rangle$, and returns **true** if the $\langle relation \rangle$ is obeyed. Two floating point numbers x and y may obey four mutually exclusive relations: $x \langle y, x=y, x \rangle y$, or x and y are not ordered. The latter case occurs exactly when one or both operands is **NaN**, and this relation is denoted by the symbol $?$. 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
```

```
\fp_compare_p:n ★
\fp_compare:nTF ★
```

Updated: 2012-12-14

```
\fp_compare_p:n
{
  ⟨fpexpr₁⟩ ⟨relation₁⟩
  ...
  ⟨fpexprₙ⟩ ⟨relationₙ⟩
  ⟨fpexpr_{n+1}⟩
}
\fp_compare:nTF
{
  ⟨fpexpr₁⟩ ⟨relation₁⟩
  ...
  ⟨fpexprₙ⟩ ⟨relationₙ⟩
  ⟨fpexpr_{n+1}⟩
}
{⟨true code⟩} {⟨false code⟩}
```

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 $\langle intexpr_1 \rangle$ and $\langle intexpr_2 \rangle$ using the $\langle relation_1 \rangle$, then $\langle intexpr_2 \rangle$ and $\langle intexpr_3 \rangle$ using the $\langle relation_2 \rangle$, until finally comparing $\langle intexpr_N \rangle$ and $\langle intexpr_{N+1} \rangle$ 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 point numbers x and y may obey four mutually exclusive relations: $x \langle y, x=y, x \rangle y$, or x and y are not ordered. The latter case occurs exactly when one or both operands is **NaN**, and this relation is denoted by the symbol $?$. 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 floatin 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 $\leq\geq$ (comparable).

5 Floating point expression loops

\fp_do_until:nNnn ☆

New: 2012-08-16

\fp_do_until:nNnn {\(fexpr_1)} \relation {\(fexpr_2)} {\(code)}

Places the `(code)` in the input stream for TeX to process, and then evaluates the relationship between the two *floating point expressions* as described for \fp_compare:nNnTF. If the test is `false` then the `(code)` will be inserted into the input stream again and a loop will occur until the `(relation)` is `true`.

\fp_do_while:nNnn ☆

New: 2012-08-16

\fp_do_while:nNnn {\(fexpr_1)} \relation {\(fexpr_2)} {\(code)}

Places the `(code)` in the input stream for TeX to process, and then evaluates the relationship between the two *floating point expressions* as described for \fp_compare:nNnTF. If the test is `true` then the `(code)` will be inserted into the input stream again and a loop will occur until the `(relation)` is `false`.

\fp_until_do:nNnn ☆

New: 2012-08-16

\fp_until_do:nNnn {\(fexpr_1)} \relation {\(fexpr_2)} {\(code)}

Evaluates the relationship between the two *floating point expressions* as described for \fp_compare:nNnTF, and then places the `(code)` in the input stream if the `(relation)` is `false`. After the `(code)` has been processed by TeX the test will be repeated, and a loop will occur until the test is `true`.

\fp_while_do:nNnn ☆

New: 2012-08-16

\fp_while_do:nNnn {\(fexpr_1)} \relation {\(fexpr_2)} {\(code)}

Evaluates the relationship between the two *floating point expressions* as described for \fp_compare:nNnTF, and then places the `(code)` in the input stream if the `(relation)` is `true`. After the `(code)` has been processed by TeX the test will be repeated, and a loop will occur until the test is `false`.

\fp_do_until:nn ☆

New: 2012-08-16

\fp_do_until:nn {\(fexpr_1)} \relation {\(fexpr_2)} {\(code)}

Places the `(code)` in the input stream for TeX to process, and then evaluates the relationship between the two *floating point expressions* as described for \fp_compare:nTF. If the test is `false` then the `(code)` will be inserted into the input stream again and a loop will occur until the `(relation)` is `true`.

\fp_do_while:nn ☆

New: 2012-08-16

\fp_do_while:nn {\(fexpr_1)} \relation {\(fexpr_2)} {\(code)}

Places the `(code)` in the input stream for TeX to process, and then evaluates the relationship between the two *floating point expressions* as described for \fp_compare:nTF. If the test is `true` then the `(code)` will be inserted into the input stream again and a loop will occur until the `(relation)` is `false`.

\fp_until_do:nn ☆

New: 2012-08-16

\fp_until_do:nn {\(fexpr_1)} \relation {\(fexpr_2)} {\(code)}

Evaluates the relationship between the two *floating point expressions* as described for \fp_compare:nTF, and then places the `(code)` in the input stream if the `(relation)` is `false`. After the `(code)` has been processed by TeX the test will be repeated, and a loop will occur until the test is `true`.

\fp_while_do:nn ☆

New: 2012-08-16

\fp_while_do:nn { ⟨fexpr1⟩ ⟨relation⟩ ⟨fexpr2⟩ } {⟨code⟩}

Evaluates the relationship between the two *floating point expressions* as described for \fp_compare:nTF, and then places the ⟨code⟩ in the input stream if the ⟨relation⟩ is true. After the ⟨code⟩ has been processed by TeX the test will be repeated, and a loop will occur 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 {⟨initial value⟩} {⟨step⟩} {⟨final value⟩} {⟨function⟩}

This function first evaluates the ⟨initial value⟩, ⟨step⟩ and ⟨final value⟩, all of which should be floating point expressions. The ⟨function⟩ is then placed in front of each ⟨value⟩ from the ⟨initial value⟩ to the ⟨final value⟩ in turn (using ⟨step⟩ between each ⟨value⟩). The ⟨step⟩ must be non-zero. If the ⟨step⟩ is positive, the loop stops when the ⟨value⟩ becomes larger than the ⟨final value⟩. If the ⟨step⟩ is negative, the loop stops when the ⟨value⟩ becomes smaller than the ⟨final value⟩. The ⟨function⟩ 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]
```

TeXhackers note: Due to rounding, it may happen that adding the ⟨step⟩ to the ⟨value⟩ does not change the ⟨value⟩; 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 {⟨initial value⟩} {⟨step⟩} {⟨final value⟩} {⟨code⟩}

This function first evaluates the ⟨initial value⟩, ⟨step⟩ and ⟨final value⟩, all of which should be floating point expressions. Then for each ⟨value⟩ from the ⟨initial value⟩ to the ⟨final value⟩ in turn (using ⟨step⟩ between each ⟨value⟩), the ⟨code⟩ is inserted into the input stream with #1 replaced by the current ⟨value⟩. Thus the ⟨code⟩ should define a function of one argument (#1).

\fp_step_variable:nnnNn

New: 2017-04-12

\fp_step_variable:nnnNn
{⟨initial value⟩} {⟨step⟩} {⟨final value⟩} {⟨tl var⟩} {⟨code⟩}

This function first evaluates the ⟨initial value⟩, ⟨step⟩ and ⟨final value⟩, all of which should be floating point expressions. Then for each ⟨value⟩ from the ⟨initial value⟩ to the ⟨final value⟩ in turn (using ⟨step⟩ between each ⟨value⟩), the ⟨code⟩ is inserted into the input stream, with the ⟨tl var⟩ defined as the current ⟨value⟩. Thus the ⟨code⟩ should make use of the ⟨tl var⟩.

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

One as an fp: useful for comparisons in some places.

New: 2012-05-08

**\c_inf_fp
\c_minus_inf_fp**

Infinity, with either sign. These can be input directly in a floating point expression as `inf` and `-inf`.

New: 2012-05-08

\c_e_fp

Updated: 2012-05-08

The value of the base of the natural logarithm, $e = \exp(1)$.

\c_pi_fp

Updated: 2013-11-17

The value of π . This can be input directly in a floating point expression as `pi`.

\c_one_degree_fp

New: 2012-05-08

Updated: 2013-11-17

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 `deg`.

\l_tmpa_fp**\l_tmpb_fp**

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.

\g_tmpa_fp**\g_tmpb_fp**

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.

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 `\f3flag`.

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.

`\fp_trap:nn`

New: 2012-07-19
Updated: 2017-02-13

```
\fp_trap:nn {\exception} {\trap type}
```

All occurrences of the `\exception` (`overflow`, `underflow`, `invalid_operation` or `division_by_zero`) within the current group are treated as `\trap type`, which can be

- `none`: the `\exception` will be entirely ignored, and leave no trace;
- `flag`: the `\exception` will turn the corresponding flag on when it occurs;
- `error`: additionally, the `\exception` 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.

8 Viewing floating points

`\fp_show:N`

New: 2012-05-08
Updated: 2015-08-07

```
\fp_show:N {fp var}  
\fp_show:n {\floating point expression}
```

Evaluates the `\floating point expression` and displays the result in the terminal.

`\fp_log:N`

New: 2014-08-22
Updated: 2015-08-07

```
\fp_log:N {fp var}  
\fp_log:n {\floating point expression}
```

Evaluates the `\floating point expression` and writes the result in the log file.

9 Floating point expressions

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`, 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`.

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 will be ignored.
- Any unrecognizable string triggers an error, and produces a `NaN`.

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 `+`, `-`, `!`.
- Binary `*`, `/`, and implicit multiplication by juxtaposition (`2pi`, `3(4+5)`, etc.).
- Binary `+` and `-`.
- Comparisons `>=`, `!=`, `<?`, etc.
- Logical `and`, denoted by `&&`.
- Logical `or`, denoted by `||`.
- Ternary operator `?:` (right associative).

The precedence of operations can be overridden using parentheses. In particular, those precedences imply that

$$\begin{aligned}\sin2\pi &= \sin(2\pi) = 0, \\ 2^{2\max(3,4)} &= 2^{2\max(3,4)} = 256.\end{aligned}$$

Functions are called on the value of their argument, contrarily to TeX macros.

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`.

`?:` `\fp_eval:n { <operand1 > ? <operand2 > : <operand3 > }`

The ternary operator `?:` results in `<operand2` if `<operand1` is true, and `<operand3` if it is false (equal to ± 0). All three `<operands` are evaluated in all cases. 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 `<operand1` is true (non-zero), use that value, otherwise the value of `<operand2`. Both `<operands` are evaluated in all cases.

&& `\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.

```
<           \fp_eval:n
=             {
>               <operand1> <relation1>
?               ...
    <operandN> <relationN>
    <operandN+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 $\langle operand_i \rangle \langle relation_j \rangle \langle operand_{i+1} \rangle$ are true, and `+0` otherwise. All $\langle operands \rangle$ are evaluated 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.

***** `\fp_eval:n { <operand1> * <operand2> }`
/ `\fp_eval:n { <operand1> / <operand2> }`
- Computes the product or the ratio of its two $\langle operands \rangle$. 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.

+ `\fp_eval:n { + <operand> }`
- `\fp_eval:n { - <operand> }`
! `\fp_eval:n { ! <operand> }`
- The unary `+` does nothing, the unary `-` changes the sign of the $\langle operand \rangle$, and `! <operand>` evaluates to `1` if $\langle operand \rangle$ is false 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/q with p integer and q odd; the result is `+0` if `abs(<operand1>) ^ <operand2>` 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.

abs `\fp_eval:n { abs(<fpexpr>) }`

Computes the absolute value of the $\langle fpexpr \rangle$. This function does not raise any exception beyond those raised when computing its operand $\langle fpexpr \rangle$. 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.

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.

max `\fp_eval:n { max(<fpexpr1> , <fpexpr2> , ...) }`
min `\fp_eval:n { min(<fpexpr1> , <fpexpr2> , ...) }`

Evaluates each $\langle fpexpr \rangle$ and computes the largest (smallest) of those. If any of the $\langle fpexpr \rangle$ is a NaN, the result is NaN. Those operations do not raise exceptions.

round `\fp_eval:n { round(<fpexpr>) }`
trunc `\fp_eval:n { round(<fpexpr1> , <fpexpr2>) }`
ceil `\fp_eval:n { round(<fpexpr1> , <fpexpr2> , <fpexpr3>) }`
floor `\fp_eval:n { round(<fpexpr1> , <fpexpr2> , <fpexpr3>) }`

New: 2013-12-14
Updated: 2015-08-08

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 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.e., $\langle fpexpr_1 \rangle$ is rounded to an integer. The rounding direction depends on the function.

- **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 **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**, or the deprecated **round-**, yields the largest multiple of 10^{-n} smaller or equal to x (“round towards negative infinity”);
- **ceil**, or the deprecated **round+**, yields the smallest multiple of 10^{-n} greater or equal to x (“round towards positive infinity”);
- **trunc**, or the deprecated **round0**, 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 $\langle fpexpr_2 \rangle < -9984$).

sign `\fp_eval:n { sign(<fpexpr>) }`

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 NaN for NaN. This operation does not raise exceptions.

```
sin          \fp_eval:n { sin( <fpexpr> ) }
cos          \fp_eval:n { cos( <fpexpr> ) }
tan          \fp_eval:n { tan( <fpexpr> ) }
cot          \fp_eval:n { cot( <fpexpr> ) }
csc          \fp_eval:n { csc( <fpexpr> ) }
sec          \fp_eval:n { sec( <fpexpr> ) }
```

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, $\sin(8\pi)$ is not quite zero, while its analogue $\text{sind}(8 \times 180)$ 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.

```
sind         \fp_eval:n { sind( <fpexpr> ) }
cosd         \fp_eval:n { cosd( <fpexpr> ) }
tand         \fp_eval:n { tand( <fpexpr> ) }
cotd         \fp_eval:n { cotd( <fpexpr> ) }
cscd         \fp_eval:n { cscd( <fpexpr> ) }
secd         \fp_eval:n { secd( <fpexpr> ) }
```

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, $\sin(8\pi)$ is not quite zero, while its analogue $\text{sind}(8 \times 180)$ 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.

```
asin         \fp_eval:n { asin( <fpexpr> ) }
acos         \fp_eval:n { acos( <fpexpr> ) }
acsc         \fp_eval:n { acsc( <fpexpr> ) }
asec         \fp_eval:n { asec( <fpexpr> ) }
```

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.

```
asind        \fp_eval:n { asind( <fpexpr> ) }
acosd        \fp_eval:n { acosd( <fpexpr> ) }
acscd        \fp_eval:n { acscd( <fpexpr> ) }
asecd        \fp_eval:n { aecd( <fpexpr> ) }
```

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.

```
atan          \fp_eval:n { atan( <fpexpr> ) }
acot          \fp_eval:n { atan( <fpexpr1> , <fpexpr2> ) }


---

New: 2013-11-02 \fp_eval:n { acot( <fpexpr> ) }
\fp_eval:n { acot( <fpexpr1> , <fpexpr2> ) }
```

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. Only the “underflow” exception can occur.

```
atand         \fp_eval:n { atand( <fpexpr> ) }
acotd         \fp_eval:n { atand( <fpexpr1> , <fpexpr2> ) }


---

New: 2013-11-02 \fp_eval:n { acotd( <fpexpr> ) }
\fp_eval:n { acotd( <fpexpr1> , <fpexpr2> ) }
```

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. Only the “underflow” exception can occur.

```
sqrt          \fp_eval:n { sqrt( <fpexpr> ) }
```

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; no other exception can occur. Special values yield $\sqrt{-0} = -0$, $\sqrt{+0} = +0$, $\sqrt{+\infty} = +\infty$ and $\sqrt{\text{NaN}} = \text{NaN}$.

randNew: 2016-12-05

\fp_eval:n { rand() }

Produces a pseudo-random floating-point number (multiple of 10^{-16}) between 0 included and 1 excluded. Available in pdfTeX and LuaTeX engines only.

TeXhackers note: This is based on pseudo-random numbers provided by the engine's primitive `\pdfuniformdeviate` in pdfTeX and `\uniformdeviate` in LuaTeX. The underlying code in pdfTeX and LuaTeX is based on Metapost, which follows an additive scheme recommended in Section 3.6 of "The Art of Computer Programming, Volume 2".

While we are more careful than `\uniformdeviate` 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.

The random seed can be queried using `\pdfrandomseed` and set using `\pdfsetrandomseed` (in LuaTeX `\randomseed` and `\setrandomseed`). 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 $[0, 2^{28} - 1]$.

randintNew: 2016-12-05

\fp_eval:n { randint(<fpexpr>) }
\fp_eval:n { randint(<fpexpr_1> , <fpexpr_2>) }

Produces a pseudo-random integer between 1 and $\langle \text{fpexpr} \rangle$ or between $\langle \text{fpexpr}_1 \rangle$ and $\langle \text{fpexpr}_2 \rangle$ 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.

inf

The special values $+\infty$, $-\infty$, and `NaN` are represented as `inf`, `-inf` and `nan` (see `\c_-inf_fp`, `\c_minus_inf_fp` and `\c_nan_fp`).

pi

The value of π (see `\c_pi_fp`).

deg

The value of 1° in radians (see `\c_one_degree_fp`).

<u>em</u>	Those units of measurement are equal to their values in <code>pt</code> , namely
<u>ex</u>	
<u>in</u>	$1\text{in} = 72.27\text{pt}$
<u>pt</u>	$1\text{pt} = 1\text{pt}$
<u>pc</u>	$1\text{pc} = 12\text{pt}$
<u>cm</u>	$1\text{cm} = \frac{1}{2.54}\text{in} = 28.45275590551181\text{pt}$
<u>mm</u>	$1\text{mm} = \frac{1}{25.4}\text{in} = 2.845275590551181\text{pt}$
<u>dd</u>	$1\text{dd} = 0.376065\text{mm} = 1.07000856496063\text{pt}$
<u>cc</u>	$1\text{cc} = 12\text{dd} = 12.84010277952756\text{pt}$
<u>nd</u>	$1\text{nd} = 0.375\text{mm} = 1.066978346456693\text{pt}$
<u>nc</u>	$1\text{nc} = 12\text{nd} = 12.80374015748031\text{pt}$
<u>bp</u>	$1\text{bp} = \frac{1}{72}\text{in} = 1.00375\text{pt}$
<u>sp</u>	$1\text{sp} = 2^{-16}\text{pt} = 1.52587890625e - 5\text{pt}.$

The values of the (font-dependent) units `em` and `ex` are gathered from `TEX` when the surrounding floating point expression is evaluated.

<u>true</u>	Other names for 1 and +0.
<u>false</u>	

\fp_abs:n ★

New: 2012-05-14
Updated: 2012-07-08

`\fp_abs:n {<floating point expression>}`

Evaluates the `<floating point expression>` as described for `\fp_eval:n` and leaves the absolute value of the result in the input stream. This function does not raise any exception beyond those raised when evaluating its argument. Within floating point expressions, `abs()` can be used.

\fp_max:nn ★

New: 2012-09-26

`\fp_max:nn {<fp expression 1>} {<fp expression 2>}`

Evaluates the `<floating point expressions>` as described for `\fp_eval:n` and leaves the resulting larger (`max`) or smaller (`min`) value in the input stream. This function does not raise any exception beyond those raised when evaluating its argument. Within floating point expressions, `max()` and `min()` can be used.

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.

- Decide what exponent range to consider.
- Support signalling `nan`.

- Modulo and remainder, and rounding functions `quantize`, `quantize0`, `quantize+`, `quantize-`, `quantize=`, `round=`. Should the modulo also be provided as (catcode 12) %?
- `\fp_format:nn {⟨fexpr⟩} {⟨format⟩}`, but what should `⟨format⟩` 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 `\fp_if_nan:nTF`, and an `isnan` function?
- Support keyword arguments?

Pgfmath also provides box-measurements (depth, height, width), but boxes are not possible expandably.

Bugs.

- 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 {0pt}` 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 TeX “number too large” error.
- Subnormals are not implemented.

Possible optimizations/improvements.

- Document that `l3trial/l3fp-types` introduces tools for adding new types.
- In subsection 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 $c = 2000/(\lfloor 200x \rfloor + 1) \in [10, 95]$ 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 TeX fp packages, the international standards,...
- Also take into account the “inexact” exception?
- Support multi-character prefix operators (*e.g.*, `@/` or whatever)?

Part XXII

The **l3sort** package

Sorting functions

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: }
}
```

will result 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 will yield a trivial sort: the final order is identical to the original order. Conversely, using a *(comparison code)* consisting only of `\sort_return_swapped:` will reverse the list (in a fairly inefficient way).

TeXhackers 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.

Part XXIII

The **\3tl-analysis** package: analysing token lists

1 \3tl-analysis documentation

This module mostly provides internal functions for use in the `\3regex` module. However, it provides as a side-effect a user debugging function, very similar to the `\ShowTokens` macro from the `ted` package.

```
\tl_show_analysis:N  
\tl_show_analysis:n
```

New: 2017-05-26

`\tl_show_analysis:n {\langle token list\rangle}`

Displays to the terminal 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.

Part XXIV

The **\3tl-build** package: building token lists

1 **\3tl-build** documentation

This module provides no user function: it is meant for kernel use only.

There are two main ways of building token lists from individual tokens. Either in one go within an x-expanding assignment, or by repeatedly using `\t1_put_right:Nn`. The first method takes a linear time, but only allows expandable operations. The second method takes a time quadratic in the length of the token list, but allows expandable and non-expandable operations.

The goal of this module is to provide functions to build a token list piece by piece in linear time, while allowing non-expandable operations. This is achieved by abusing `\toks`: adding some tokens to the token list is done by storing them in a free token register (time $O(1)$ for each such operation). Those token registers are only put together at the end, within an x-expanding assignment, which takes a linear time.⁵ Of course, all this must be done in a group: we can't go and clobber the values of legitimate `\toks` used by L^AT_EX 2 _{ε} .

Since none of the current applications need the ability to insert material on the left of the token list, I have not implemented that. This could be done for instance by using odd-numbered `\toks` for the left part, and even-numbered `\toks` for the right part.

1.1 Internal functions

`_t1_build:Nw`
`_t1_gbuild:Nw`
`_t1_build_x:Nw`
`_t1_gbuild_x:Nw`

`_t1_build:Nw <tl var> ...`
`_t1_build_one:n {{tokens1}} ...`
`_t1_build_one:n {{tokens2}} ...`
...
`_t1_build_end:`

Defines the `<tl var>` to contain the contents of `<tokens1>` followed by `<tokens2>`, etc. This is built in such a way to be more efficient than repeatedly using `\t1_put_right:Nn`. The code in “...” does not need to be expandable. The commands `_t1_build:Nw` and `_t1_build_end:` start and end a group. The assignment to the `<tl var>` occurs just after the end of that group, using `\t1_set:Nn`, `\t1_gset:Nn`, `\t1_set:Nx`, or `\t1_gset:Nx`.

`_t1_build_one:n`
`_t1_build_one:(o|x)`

`_t1_build_one:n {{tokens}}`

This function may only be used within the scope of a `_t1_build:Nw` function. It adds the `<tokens>` on the right of the current token list.

`_t1_build_end:`

Ends the scope started by `_t1_build:Nw`, and performs the relevant assignment.

⁵If we run out of token registers, then the currently filled-up `\toks` are put together in a temporary token list, and cleared, and we ultimately use `\t1_put_right:Nx` to put those chunks together. Hence the true asymptotic is quadratic, with a very small constant.

Part XXV

The **l3regex** package: regular expressions in **T_EX**

1 Regular expressions

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 **T_EX** 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 add a comma at the end of each word:

```
\regex_replace_all:nnN { \w+ } { \0 , } \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).

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

```
\regex_const:Nn \c_foo_regex { \c{begin} \cB. (\c[^BE].*) \cE. }
```

stores in `\c_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.

1.1 Syntax of 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 will 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).

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.*, alpha-numerics and underscore, equivalent to `[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 will 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).

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.

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.

Alternation and capturing groups.

A|B|C Either one of A, B, or C.

(...) Capturing group.

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

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

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 csname matches the *(regex)*, anchored at the beginning and end, so that \c{begin} matches exactly \begin, and nothing else.

\c{X} Applies to the next object, which can be a character, character property, class, or group, and forces this object to only match tokens with category X (any of CBEMTPUDSLOA). For instance, \c{L[A-Z\d]} matches uppercase letters and digits of category code letter, \c{C}. matches any control sequence, and \c{O(abc)} matches abc where each character has category other.

\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, `\c0\d \c[LO] [A-F]` matches what TeX 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\c0*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{\langle tl var name\rangle}` matches the exact contents of the token list `\langle tl var\rangle`. Within a `\c{\dots}` control sequence matching, the `\u` escape sequence only expands its argument once, in effect performing `\tl_to_str:v`. Quantifiers are not supported directly: use a group.

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{\dots}` nor `\u{\dots}` are affected by the `i` option.

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.

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:nnNTF`.

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.

1.2 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:

- $\backslash 0$ is the whole match;
- $\backslash 1, \backslash 2, \dots, \backslash 9$ or $\backslash g\{\langle number \rangle\}$ are the submatches (empty if there are fewer than $\langle number \rangle$ capturing groups);
- $\backslash _$ inserts a space (spaces are ignored when not escaped);
- $\backslash a, \backslash e, \backslash f, \backslash n, \backslash r, \backslash t, \backslash x{hhh}, \backslash x\{hhh\}$ correspond to single characters as in regular expressions;
- $\backslash c\{\langle cs name \rangle\}$ inserts a control sequence;
- $\backslash c\langle category \rangle\langle character \rangle$ (see below);
- $\backslash u\{\langle tl var name \rangle\}$ inserts the contents of the $\langle tl var \rangle$ (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 T_EX, for instance use $\backslash \#$). Non-alphanumeric characters can always be safely escaped with a backslash.

For instance,

```
\tl_set:Nn \l_my_t1 { Hello,\~world! }
\regex_replace_all:nnN { ([er]?\l|o) . } { (\0--\1) } \l_my_t1
```

results in $\backslash l_my_t1$ holding $H(e\~l\~o)(o,--o) w(o\~r\~o)(d--d)!$

Submatches always keep the same category codes as in the original token list. The characters inserted by the replacement have category code 12 (other) by default, with the exception of space characters. Spaces inserted through $\backslash _$ have category code 10, while spaces inserted through $\backslash x{20}$ or $\backslash x\{20\}$ have category code 12. The escape sequence $\backslash c$ allows to insert characters with arbitrary category codes, as well as control sequences.

- $\backslash cX(\dots)$ 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). This can be nested, for instance $\backslash cL(Hello\cS\ world)!$
- $\backslash c\{\langle text \rangle\}$ Produces the control sequence with cname $\langle text \rangle$. The $\langle text \rangle$ may contain references to the submatches $\backslash 0, \backslash 1$, and so on, as in the example for $\backslash u$ below.

The escape sequence $\backslash u\{\langle tl var name \rangle\}$ allows to insert the contents of the token list with name $\langle tl var name \rangle$ directly into the replacement, giving an easier control of category codes. Within $\backslash c\{\dots\}$ and $\backslash u\{\dots\}$ constructions, the $\backslash u$ and $\backslash c$ escape sequences perform $\backslash tl_to_str:v$, namely extract the value of the control sequence and turn it into a string.

Matches can be used within the arguments of $\backslash c$ and $\backslash u$. For instance,

```
\tl_set:Nn \l_my_one_t1 { first }
\tl_set:Nn \l_my_two_t1 { \emph{second} }
\tl_set:Nn \l_my_t1 { one , two , one , one }
\regex_replace_all:nnN { [^,]+ } { \u{\l_my_\0_t1} } \l_my_t1
```

results in $\backslash l_my_t1$ holding $first,\emph{second},first,first$.

1.3 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.

\regex_new:N

New: 2017-05-26

\regex_new:N (*regex var*)

Creates a new *⟨regex var⟩* or raises an error if the name is already taken. The declaration is global. The *⟨regex var⟩* will initially be such that it never matches.

\regex_set:Nn

\regex_gset:Nn

\regex_const:Nn

New: 2017-05-26

\regex_set:Nn (*regex var*) {*(regular expression)*}

Stores a compiled version of the *⟨regular expression⟩* in the *⟨regex var⟩*. For instance, this function can be used as

```
\regex_new:N \l_my_regex
\regex_set:Nn \l_my_regex { my\ (simple\b)? reg(ex|ular\b) expression }
```

The assignment is local for `\regex_set:Nn` and global for `\regex_gset:Nn`. Use `\regex_const:Nn` for compiled expressions which will never change.

\regex_show:n

\regex_show:N

New: 2017-05-26

\regex_show:n {*(regex)*}

Shows how l3regex interprets the *⟨regex⟩*. For instance, `\regex_show:n {\A X|Y}` shows

```
+--branch
  anchor at start (\A)
  char code 88
+--branch
  char code 89
```

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

1.4 Matching

All regular expression functions are available in both :n and :N variants. The former require a “standard” regular expression, while the latter require a compiled expression as generated by `\regex_(g)set:Nn`.

\regex_match:nnTF

\regex_match:NnTF

New: 2017-05-26

\regex_match:nnTF {*(regular expression)*} {*(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]* } { abecdex } { 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 $\langle \text{int var} \rangle$ within the current TeX group level equal to the number of times $\langle \text{regular expression} \rangle$ appears in $\langle \text{token list} \rangle$. The search starts by finding the left-most longest match, respecting greedy and ungreedy 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) } { abbababbbb } \l_foo_int
```

results in $\backslash l_foo_int$ taking the value 5.

1.5 Submatch extraction

```
\regex_extract_once:nnNTF
\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 $\langle \text{regular expression} \rangle$ in the $\langle \text{token list} \rangle$. If it exists, the match is stored as the zeroeth item of the $\langle \text{seq var} \rangle$, and further items are the contents of capturing groups, in the order of their opening parenthesis. The $\langle \text{seq var} \rangle$ is assigned locally. If there is no match, the $\langle \text{seq var} \rangle$ is cleared. The testing versions insert the $\langle \text{true code} \rangle$ into the input stream if a match was found, and the $\langle \text{false code} \rangle$ 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 $\backslash A$ and at the end with $\backslash Z$) will match the whole token list. The first capturing group, $(La)?$, matches La , and the second capturing group, $(!*)$, matches $!!!$. Thus, $\backslash l_foo_seq$ will contain the items $\{LaTeX!!!\}$, $\{La\}$, and $\{!!!\}$, and the **true** branch is left in the input stream.

```
\regex_extract_all:nnNTF
\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 $\langle \text{regular expression} \rangle$ in the $\langle \text{token list} \rangle$, and stores all the sub-match information in a single sequence (concatenating the results of multiple $\backslash \text{regex_extract_once:nnN}$ calls). The $\langle \text{seq var} \rangle$ is assigned locally. If there is no match, the $\langle \text{seq var} \rangle$ is cleared. The testing versions insert the $\langle \text{true code} \rangle$ into the input stream if a match was found, and the $\langle \text{false code} \rangle$ otherwise. For instance, assume that you type

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

Then the regular expression will match twice, and the resulting sequence contains the two items $\{Hello\}$ and $\{world\}$, and the **true** branch is left in the input stream.

```
\regex_split:nnNTF
```

```
\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.

1.6 Replacement

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

```
\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 *<token list>* 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>*.

1.7 Bugs, misfeatures, future work, and other possibilities

The following need to be done now.

- Change user function names!
 - Clean up the use of messages.
 - Rewrite the documentation in a more ordered way, perhaps add a BNF?
- Additional error-checking to come.
- Currently, *a{\x34}* is recognized as *a{4}*.
 - 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`.
 - Enforce that `\cC` can only be followed by a match-all dot.
 - 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 `.s` 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 `current_state` and `current_submatches`.
 - If possible, when a state is reused by the same thread, kill other subthreads.
 - Use an array rather than `\l_regex_balance_t1` to build `_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.

- `\ddd`, matching the character with octal code `ddd`;
- Callout with `(?C...)`;
- 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.

- `\cx`, similar to TeX’s own `\^{\x}`;
- Comments: TeX already has its own system for comments.
- `\Q... \E` escaping: this would require to read the argument verbatim, which is not in the scope of this module.
- 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.
- Recursion: this is a non-regular feature.
- Back-references: non-regular feature, this requires backtracking, which is prohibitively slow.
- Backtracking control verbs: intrinsically tied to backtracking.
- `\C` single byte in UTF-8 mode: XeTeX and LuaTeX serve us characters directly, and splitting those into bytes is tricky, encoding dependent, and most likely not useful anyways.

Part XXVI

The **I3box** package

Boxes

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_`.

1 Creating and initialising boxes

`\box_new:N <box>`

`\box_new:c`

Creates a new `<box>` or raises an error if the name is already taken. The declaration is global. The `<box>` will initially be void.

`\box_clear:N <box>`

`\box_clear:c`

`\box_gclear:N`

`\box_gclear:c`

Clears the content of the `<box>` by setting the box equal to `\c_void_box`.

`\box_clear_new:N <box>`

`\box_clear_new:c`

`\box_gclear_new:N`

`\box_gclear_new:c`

Ensures that the `<box>` exists globally by applying `\box_new:N` if necessary, then applies `\box_(g)clear:N` to leave the `<box>` empty.

`\box_set_eq:NN <box1> <box2>`

`\box_set_eq:(cN|Nc|cc)`

`\box_gset_eq:NN`

`\box_gset_eq:(cN|Nc|cc)`

Sets the content of `<box1>` equal to that of `<box2>`.

`\box_set_eq_clear:NN <box1> <box2>`

`\box_set_eq_clear:(cN|Nc|cc)`

Sets the content of `<box1>` within the current TeX group equal to that of `<box2>`, then clears `<box2>` globally.

`\box_gset_eq_clear:NN <box1> <box2>`

`\box_gset_eq_clear:(cN|Nc|cc)`

Sets the content of `<box1>` equal to that of `<box2>`, then clears `<box2>`. These assignments are global.

`\box_if_exist_p:N *`

`\box_if_exist_p:c *`

`\box_if_exist:NTF *`

`\box_if_exist:cTF *`

`\box_if_exist_p:N <box>`

`\box_if_exist:NTF <box> {<true code>} {<false code>}`

Tests whether the `<box>` is currently defined. This does not check that the `<box>` really is a box.

2 Using boxes

\box_use:N \box_use:N *box*
\box_use:c

Inserts the current content of the *box* onto the current list for typesetting.

TeXhackers note: This is the TeX primitive \copy.

\box_use_clear:N \box_use_clear:N *box*

\box_use_clear:c

Inserts the current content of the *box* onto the current list for typesetting, then globally clears the content of the *box*.

TeXhackers note: This is the TeX primitive \box.

\box_move_right:nn \box_move_right:nn {*dimexpr*} {*box function*}
\box_move_left:nn

This function operates in vertical mode, and inserts the material specified by the *box function* such that its reference point is displaced horizontally by the given *dimexpr* from the reference point for typesetting, to the right or left as appropriate. The *box function* should be a box operation such as \box_use:N \<box> or a “raw” box specification such as \vbox:n { xyz }.

\box_move_up:nn \box_move_up:nn {*dimexpr*} {*box function*}
\box_move_down:nn

This function operates in horizontal mode, and inserts the material specified by the *box function* such that its reference point is displaced vertical by the given *dimexpr* from the reference point for typesetting, up or down as appropriate. The *box function* should be a box operation such as \box_use:N \<box> or a “raw” box specification such as \vbox:n { xyz }.

3 Measuring and setting box dimensions

\box_dp:N \box_dp:N *box*

\box_dp:c

Calculates the depth (below the baseline) of the *box* in a form suitable for use in a *dimension expression*.

TeXhackers note: This is the TeX primitive \dp.

\box_ht:N \box_ht:N *box*

\box_ht:c

Calculates the height (above the baseline) of the *box* in a form suitable for use in a *dimension expression*.

TeXhackers note: This is the TeX primitive \ht.

`\box_wd:N` `\box_wd:c`

Calculates the width of the `<box>` in a form suitable for use in a `{dimension expression}`.

TEXhackers note: This is the TeX primitive `\wd`.

`\box_set_dp:Nn` `\box_set_dp:cn`

Updated: 2011-10-22

`\box_set_dp:Nn <box> {<dimension expression>}`

Set the depth (below the baseline) of the `<box>` to the value of the `{<dimension expression>}`. This is a global assignment.

`\box_set_ht:Nn` `\box_set_ht:cn`

Updated: 2011-10-22

`\box_set_ht:Nn <box> {<dimension expression>}`

Set the height (above the baseline) of the `<box>` to the value of the `{<dimension expression>}`. This is a global assignment.

`\box_set_wd:Nn` `\box_set_wd:cn`

Updated: 2011-10-22

`\box_set_wd:Nn <box> {<dimension expression>}`

Set the width of the `<box>` to the value of the `{<dimension expression>}`. This is a global assignment.

4 Box conditionals

`\box_if_empty_p:N *` `\box_if_empty_p:c *`
`\box_if_empty_p:cTF *` `\box_if_empty:cTF *`

`\box_if_empty:NTF <box> {<true code>} {<false code>}`

Tests if `<box>` is a empty (equal to `\c_empty_box`).

`\box_if_horizontal_p:N *` `\box_if_horizontal_p:c *`
`\box_if_horizontal_p:cTF *` `\box_if_horizontal:cTF *`

`\box_if_horizontal:N <box>`

`\box_if_horizontal:NTF <box> {<true code>} {<false code>}`

Tests if `<box>` is a horizontal box.

`\box_if_vertical_p:N *` `\box_if_vertical_p:c *`
`\box_if_vertical_p:cTF *` `\box_if_vertical:cTF *`

`\box_if_vertical:N <box>`

`\box_if_vertical:NTF <box> {<true code>} {<false code>}`

Tests if `<box>` is a vertical box.

5 The last box inserted

`\box_set_to_last:N` `\box_set_to_last:c`
`\box_gset_to_last:N` `\box_gset_to_last:c`

`\box_set_to_last:N <box>`

Sets the `<box>` 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 `<box>` will always be void as it is not possible to recover the last added item.

6 Constant boxes

\c_empty_box

This is a permanently empty box, which is neither set as horizontal nor vertical.

Updated: 2012-11-04

7 Scratch boxes

\l_tmpa_box
\tmpb_box

Updated: 2012-11-04

Scratch boxes 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.

\g_tmpa_box
\tmpb_box

Scratch boxes 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 Viewing box contents

\box_show:N
\box_show:c

Updated: 2012-05-11

\box_show:N <box>

Shows full details of the content of the <box> in the terminal.

\box_show:Nnn
\box_show:cnn

New: 2012-05-11

\box_show:Nnn <box> <intexpr₁> <intexpr₂>

Display the contents of <box> in the terminal, showing the first <intexpr₁> items of the box, and descending into <intexpr₂> group levels.

\box_log:N
\box_log:c

New: 2012-05-11

\box_log:N <box>

Writes full details of the content of the <box> to the log.

\box_log:Nnn
\box_log:cnn

New: 2012-05-11

\box_log:Nnn <box> <intexpr₁> <intexpr₂>

Writes the contents of <box> to the log, showing the first <intexpr₁> items of the box, and descending into <intexpr₂> group levels.

9 Boxes and color

All L^AT_EX3 boxes are “color safe”: a color set inside the box will not apply after the end of the box has occurred.

10 Horizontal mode boxes

\hbox:n

Updated: 2017-04-05

\hbox:n {*contents*}

Typesets the *contents* into a horizontal box of natural width and then includes this box in the current list for typesetting.

\hbox_to_wd:nn

Updated: 2017-04-05

\hbox_to_wd:nn {*dimexpr*} {*contents*}

Typesets the *contents* into a horizontal box of width *dimexpr* and then includes this box in the current list for typesetting.

\hbox_to_zero:n

Updated: 2017-04-05

\hbox_to_zero:n {*contents*}

Typesets the *contents* into a horizontal box of zero width and then includes this box in the current list for typesetting.

\hbox_set:Nn

\hbox_set:cn

\hbox_gset:Nn

\hbox_gset:cn

Updated: 2017-04-05

\hbox_set:Nn {*box*} {*contents*}

Typesets the *contents* at natural width and then stores the result inside the *box*.

\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 *contents* to the width given by the *dimexpr* and then stores the result inside the *box*.

\hbox_overlap_right:n

Updated: 2017-04-05

\hbox_overlap_right:n {*contents*}

Typesets the *contents* into a horizontal box of zero width such that material will protrude to the right of the insertion point.

\hbox_overlap_left:n

Updated: 2017-04-05

\hbox_overlap_left:n {*contents*}

Typesets the *contents* into a horizontal box of zero width such that material will protrude 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 *contents* at natural width and then stores the result inside the *box*. In contrast to \hbox_set:Nn this function does not absorb the argument when finding the *content*, and so can be used in circumstances where the *content* may not be a simple argument.

`\hbox_unpack:N`
`\hbox_unpack:c`

`\hbox_unpack:N <box>`

Unpacks the content of the horizontal `<box>`, retaining any stretching or shrinking applied when the `<box>` was set.

TeXhackers note: This is the TeX primitive `\unhcopy`.

`\hbox_unpack_clear:N`
`\hbox_unpack_clear:c`

`\hbox_unpack_clear:N <box>`

Unpacks the content of the horizontal `<box>`, retaining any stretching or shrinking applied when the `<box>` was set. The `<box>` is then cleared globally.

TeXhackers note: This is the TeX primitive `\unhbox`.

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 will have 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 will typically be non-zero.

`\vbox:n`

Updated: 2017-04-05

`\vbox:n {<contents>}`

Typesets the `<contents>` into a vertical box of natural height and includes this box in the current list for typesetting.

`\vbox_top:n`

Updated: 2017-04-05

`\vbox_top:n {<contents>}`

Typesets the `<contents>` into a vertical box of natural height and includes this box in the current list for typesetting. The baseline of the box will be equal to that of the *first* item added to the box.

`\vbox_to_ht:nn`

Updated: 2017-04-05

`\vbox_to_ht:nn {<dimexpr>} {<contents>}`

Typesets the `<contents>` into a vertical box of height `<dimexpr>` and then includes this box in the current list for typesetting.

`\vbox_to_zero:n`

Updated: 2017-04-05

`\vbox_to_zero:n {<contents>}`

Typesets the `<contents>` into a vertical box of zero height and then includes this box in the current list for typesetting.

`\vbox_set:Nn`

`\vbox_set:cn`

`\vbox_gset:Nn`

`\vbox_gset:cn`

Updated: 2017-04-05

`\vbox_set:Nn <box> {<contents>}`

Typesets the `<contents>` at natural height and then stores the result inside the `<box>`.

```
\vbox_set_top:Nn
\vbox_set_top:cN
\vbox_gset_top:Nn
\vbox_gset_top:cN
```

Updated: 2017-04-05

`\vbox_set_top:Nn <box> {\<contents>}`

Typesets the *<contents>* at natural height and then stores the result inside the *<box>*. The baseline of the box will be equal to that of the *first* item added to the box.

```
\vbox_set_to_ht:Nnn
\vbox_set_to_ht:cnn
\vbox_gset_to_ht:Nnn
\vbox_gset_to_ht:cnn
```

Updated: 2017-04-05

`\vbox_set_to_ht:Nnn <box> {\<dimexpr>} {\<contents>}`

Typesets the *<contents>* to the height given by the *<dimexpr>* and then stores the result inside the *<box>*.

```
\vbox_set:Nw
\vbox_set:cw
\vbox_set_end:
\vbox_gset:Nw
\vbox_gset:cw
\vbox_gset_end:
```

Updated: 2017-04-058

`\vbox_set:Nw <box> {\<contents>} \vbox_set_end:`

Typesets the *<contents>* at natural height and then stores the result inside the *<box>*. In contrast to `\vbox_set:Nn` this function does not absorb the argument when finding the *<content>*, and so can be used in circumstances where the *<content>* may not be a simple argument.

```
\vbox_set_split_to_ht:NNn

```

Updated: 2011-10-22

`\vbox_set_split_to_ht:NNn <box1> <box2> {\<dimexpr>}`

Sets *<box₁>* to contain material to the height given by the *<dimexpr>* by removing content from the top of *<box₂>* (which must be a vertical box).

TeXhackers note: This is the TeX primitive `\vsplit`.

```
\vbox_unpack:N
\vbox_unpack:c
```

`\vbox_unpack:N <box>`

Unpacks the content of the vertical *<box>*, retaining any stretching or shrinking applied when the *<box>* was set.

TeXhackers note: This is the TeX primitive `\unvcopy`.

```
\vbox_unpack_clear:N
\vbox_unpack_clear:c
```

`\vbox_unpack:N <box>`

Unpacks the content of the vertical *<box>*, retaining any stretching or shrinking applied when the *<box>* was set. The *<box>* is then cleared globally.

TeXhackers note: This is the TeX primitive `\unvbox`.

11.1 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.

```
\box_autosize_to_wd_and_ht:Nnn \box_autosize_to_wd_and_ht:Nnn <box> {\<x-size>} {\<y-size>}
```

New: 2017-04-04

Resizes the $\langle box \rangle$ to fit within the given $\langle x\text{-size} \rangle$ (horizontally) and $\langle y\text{-size} \rangle$ (vertically); both of the sizes are dimension expressions. The $\langle y\text{-size} \rangle$ is the height only: it does not include any depth. The updated $\langle box \rangle$ will be an `hbox`, irrespective of the nature of the $\langle box \rangle$ before the resizing is applied. The final size of the $\langle box \rangle$ will be the smaller of $\{\langle x\text{-size} \rangle\}$ and $\{\langle y\text{-size} \rangle\}$, *i.e.* the result will fit within the dimensions specified. Negative sizes will cause the material in the $\langle box \rangle$ to be reversed in direction, but the reference point of the $\langle box \rangle$ will be unchanged. Thus a negative $\langle y\text{-size} \rangle$ will result in the $\langle box \rangle$ having a depth dependent on the height of the original and *vice versa*. The resizing applies within the current TeX group level.

```
\box_autosize_to_wd_and_ht_plus_dp:Nnn \box_autosize_to_wd_and_ht_plus_dp:Nnn <box> {\<x-size>} {\<y-size>}
```

New: 2017-04-04

Resizes the $\langle box \rangle$ to fit within the given $\langle x\text{-size} \rangle$ (horizontally) and $\langle y\text{-size} \rangle$ (vertically); both of the sizes are dimension expressions. The $\langle y\text{-size} \rangle$ is the total vertical size (height plus depth). The updated $\langle box \rangle$ will be an `hbox`, irrespective of the nature of the $\langle box \rangle$ before the resizing is applied. The final size of the $\langle box \rangle$ will be the smaller of $\{\langle x\text{-size} \rangle\}$ and $\{\langle y\text{-size} \rangle\}$, *i.e.* the result will fit within the dimensions specified. Negative sizes will cause the material in the $\langle box \rangle$ to be reversed in direction, but the reference point of the $\langle box \rangle$ will be unchanged. Thus a negative $\langle y\text{-size} \rangle$ will result in the $\langle box \rangle$ having a depth dependent on the height of the original and *vice versa*. The resizing applies within the current TeX group level.

```
\box_resize_to_ht:Nn \box_resize_to_ht:cN
```

Resizes the $\langle box \rangle$ to $\langle y\text{-size} \rangle$ (vertically), scaling the horizontal size by the same amount; $\langle y\text{-size} \rangle$ is a dimension expression. The $\langle y\text{-size} \rangle$ is the height only: it does not include any depth. The updated $\langle box \rangle$ will be an `hbox`, irrespective of the nature of the $\langle box \rangle$ before the resizing is applied. A negative $\langle y\text{-size} \rangle$ will cause the material in the $\langle box \rangle$ to be reversed in direction, but the reference point of the $\langle box \rangle$ will be unchanged. Thus a negative $\langle y\text{-size} \rangle$ will result in the $\langle box \rangle$ having a depth dependent on the height of the original and *vice versa*. The resizing applies within the current TeX group level.

```
\box_resize_to_ht_plus_dp:Nn \box_resize_to_ht_plus_dp:Nn <box> {\<y-size>}
```

Resizes the $\langle box \rangle$ to $\langle y\text{-size} \rangle$ (vertically), scaling the horizontal size by the same amount; $\langle y\text{-size} \rangle$ is a dimension expression. The $\langle y\text{-size} \rangle$ is the total vertical size (height plus depth). The updated $\langle box \rangle$ will be an `hbox`, irrespective of the nature of the $\langle box \rangle$ before the resizing is applied. A negative $\langle y\text{-size} \rangle$ will cause the material in the $\langle box \rangle$ to be reversed in direction, but the reference point of the $\langle box \rangle$ will be unchanged. Thus a negative $\langle y\text{-size} \rangle$ will result in the $\langle box \rangle$ having a depth dependent on the height of the original and *vice versa*. The resizing applies within the current TeX group level.

`\box_resize_to_wd:Nn`
`\box_resize_to_wd:cN`

`\box_resize_to_wd:Nn <box> {<x-size>}`

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 $\langle box \rangle$ will be an `hbox`, irrespective of the nature of the $\langle box \rangle$ before the resizing is applied. A negative $\langle x-size \rangle$ will cause the material in the $\langle box \rangle$ to be reversed in direction, but the reference point of the $\langle box \rangle$ will be unchanged. Thus a negative $\langle x-size \rangle$ will result in the $\langle box \rangle$ having a depth dependent on the height of the original and *vice versa*. The resizing applies within the current TeX group level.

`\box_resize_to_wd_and_ht:Nnn` `\box_resize_to_wd_and_ht:Nnn <box> {<x-size>} {<y-size>}`
`\box_resize_to_wd_and_ht:cnn`

New: 2014-07-03

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 $\langle box \rangle$ will be an `hbox`, irrespective of the nature of the $\langle box \rangle$ before the resizing is applied. Negative sizes will cause the material in the $\langle box \rangle$ to be reversed in direction, but the reference point of the $\langle box \rangle$ will be unchanged. Thus a negative $\langle y-size \rangle$ will result in the $\langle box \rangle$ having a depth dependent on the height of the original and *vice versa*. The resizing applies within the current TeX group level.

`\box_resize_to_wd_and_ht_plus_dp:Nnn`
`\box_resize_to_wd_and_ht_plus_dp:cnn`

New: 2017-04-06

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 $\langle box \rangle$ will be an `hbox`, irrespective of the nature of the $\langle box \rangle$ before the resizing is applied. Negative sizes will cause the material in the $\langle box \rangle$ to be reversed in direction, but the reference point of the $\langle box \rangle$ will be unchanged. Thus a negative $\langle y-size \rangle$ will result in the $\langle box \rangle$ having a depth dependent on the height of the original and *vice versa*. The resizing applies within the current TeX group level.

`\box_rotate:Nn`
`\box_rotate:cN`

`\box_rotate:Nn <box> {<angle>}`

Rotates the $\langle box \rangle$ by $\langle angle \rangle$ (in degrees) anti-clockwise about its reference point. The reference point of the updated box will be moved horizontally such that it is at the left side of the smallest rectangle enclosing the rotated material. The updated $\langle box \rangle$ will be an `hbox`, irrespective of the nature of the $\langle box \rangle$ before the rotation is applied. The rotation applies within the current TeX group level.

`\box_scale:Nnn`
`\box_scale:cnn`

`\box_scale:Nnn <box> {<x-scale>} {<y-scale>}`

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 $\langle box \rangle$ will be an `hbox`, irrespective of the nature of the $\langle box \rangle$ before the scaling is applied. Negative scalings will cause the material in the $\langle box \rangle$ to be reversed in direction, but the reference point of the $\langle box \rangle$ will be unchanged. Thus a negative $\langle y-scale \rangle$ will result in the $\langle box \rangle$ having a depth dependent on the height of the original and *vice versa*. The resizing applies within the current TeX group level.

12 Primitive box conditionals

\if_hbox:N \star \if_hbox:N $\langle box \rangle$
 ⟨true code⟩
 \else:
 ⟨false code⟩
 \fi:
Tests is $\langle box \rangle$ is a horizontal box.

TeXhackers note: This is the TeX primitive \ifhbox.

\if_vbox:N \star \if_vbox:N $\langle box \rangle$
 ⟨true code⟩
 \else:
 ⟨false code⟩
 \fi:
Tests is $\langle box \rangle$ is a vertical box.

TeXhackers note: This is the TeX primitive \ifvbox.

\if_box_empty:N \star \if_box_empty:N $\langle box \rangle$
 ⟨true code⟩
 \else:
 ⟨false code⟩
 \fi:
Tests is $\langle box \rangle$ is an empty (void) box.

TeXhackers note: This is the TeX primitive \ifvoid.

Part XXVII

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).

1 Creating and initialising coffins

`\coffin_new:N` `\coffin_new:c`
New: 2011-08-17

`\coffin_new:N` `\coffin_new:c`
Creates a new `\coffin` or raises an error if the name is already taken. The declaration is global. The `\coffin` will initially be empty.

`\coffin_clear:N` `\coffin_clear:c`
New: 2011-08-17

`\coffin_clear:N` `\coffin_clear:c`
Clears the content of the `\coffin` within the current T_EX group level.

`\coffin_set_eq:NN`
`\coffin_set_eq:(Nc|cN|cc)`
New: 2011-08-17

`\coffin_set_eq:NN` `\coffin_set_eq:(Nc|cN|cc)`
Sets both the content and poles of `\coffin1` equal to those of `\coffin2` within the current T_EX group level.

`\coffin_if_exist_p:N` *
`\coffin_if_exist_p:c` *
`\coffin_if_exist:NTF` *
`\coffin_if_exist:cTF` *
New: 2012-06-20

`\coffin_if_exist_p:N` `\coffin_if_exist_p:c` *
`\coffin_if_exist:NTF` `\coffin_if_exist:cTF` *
Tests whether the `\coffin` is currently defined.

`\hcoffin_set:Nn` `\hcoffin_set:cn`
New: 2011-08-17
Updated: 2011-09-03

`\hcoffin_set:Nn` `\hcoffin_set:cn`
Typesets the `\material` in horizontal mode, storing the result in the `\coffin`. The standard poles for the `\coffin` are then set up based on the size of the typeset material.

`\hcoffin_set:Nw` `\hcoffin_set:cw`
`\hcoffin_set_end:`
New: 2011-09-10

`\hcoffin_set:Nw` `\hcoffin_set:cw`
`\hcoffin_set_end:`
Typesets the `\material` in horizontal mode, storing the result in the `\coffin`. The standard poles for the `\coffin` 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
```

New: 2011-08-17
Updated: 2012-05-22

```
\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:
```

New: 2011-09-10
Updated: 2012-05-22

```
\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:Nnn <coffin>
\coffin_set_horizontal_pole:cnn {\<pole>} {\<offset>}
```

New: 2012-07-20

Sets the $\langle pole \rangle$ to run horizontally through the $\langle coffin \rangle$. The $\langle pole \rangle$ will be located at the $\langle offset \rangle$ from the bottom edge of the bounding box of the $\langle coffin \rangle$. The $\langle offset \rangle$ should be given as a dimension expression.

```
\coffin_set_vertical_pole:Nnn \coffin_set_vertical_pole:Nnn <coffin> {\<pole>} {\<offset>}
\coffin_set_vertical_pole:cnn
```

New: 2012-07-20

Sets the $\langle pole \rangle$ to run vertically through the $\langle coffin \rangle$. The $\langle pole \rangle$ will be located 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.

3 Joining and using coffins

```
\coffin_attach:NnnNnnnn
```

```
\coffin_attach:(cnnNnnnn|Nnncnnnn|cnncnnnn)
```

```
\coffin_attach:NnnNnnnn
```

```
<coffin1> {\<coffin1-pole1>} {\<coffin1-pole2>}
<coffin2> {\<coffin2-pole1>} {\<coffin2-pole2>}
{\<x-offset>} {\<y-offset>}
```

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 $\langle coffin_1\text{-}pole_1 \rangle$ and $\langle coffin_1\text{-}pole_2 \rangle$, and $\langle handle_2 \rangle$, the point of intersection of $\langle coffin_2\text{-}pole_1 \rangle$ and $\langle coffin_2\text{-}pole_2 \rangle$. $\langle coffin_2 \rangle$ 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 $\langle x\text{-}offset \rangle$ and $\langle y\text{-}offset \rangle$. The two offsets should be given as dimension expressions.

```
\coffin_join:NnnNnnn
\coffin_join:(cnnNnnnn|Nnncnnnn|cnncnnnn)
```

```
\coffin_join:NnnNnnn
  <coffin1> {(coffin1-pole1)} {(coffin1-pole2)}
  <coffin2> {(coffin2-pole1)} {(coffin2-pole2)}
  {<x-offset>} {<y-offset>}
```

This function joins $\langle \text{coffin}_2 \rangle$ to $\langle \text{coffin}_1 \rangle$ such that the bounding box of $\langle \text{coffin}_1 \rangle$ may expand. The new bounding box will cover the area containing the bounding boxes of the two original coffins. The alignment is carried out by first calculating $\langle \text{handle}_1 \rangle$, the point of intersection of $\langle \text{coffin}_1\text{-pole}_1 \rangle$ and $\langle \text{coffin}_1\text{-pole}_2 \rangle$, and $\langle \text{handle}_2 \rangle$, the point of intersection of $\langle \text{coffin}_2\text{-pole}_1 \rangle$ and $\langle \text{coffin}_2\text{-pole}_2 \rangle$. $\langle \text{coffin}_2 \rangle$ is then attached to $\langle \text{coffin}_1 \rangle$ such that the relationship between $\langle \text{handle}_1 \rangle$ and $\langle \text{handle}_2 \rangle$ is described by the $\langle x\text{-offset} \rangle$ and $\langle y\text{-offset} \rangle$. The two offsets should be given as dimension expressions.

```
\coffin_typeset:Nnnnn
\coffin_typeset:cnnnn
```

Updated: 2012-07-20

```
\coffin_typeset:Nnnnn <coffin> {<pole1>} {<pole2>}
  {<x-offset>} {<y-offset>}
```

Typesetting is carried out by first calculating $\langle \text{handle} \rangle$, the point of intersection of $\langle \text{pole}_1 \rangle$ and $\langle \text{pole}_2 \rangle$. The coffin is then typeset in horizontal mode such that the relationship between the current reference point in the document and the $\langle \text{handle} \rangle$ is described by the $\langle x\text{-offset} \rangle$ and $\langle y\text{-offset} \rangle$. 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.

4 Measuring coffins

```
\coffin_dp:N
\coffin_dp:c
```

```
\coffin_dp:N <coffin>
```

Calculates the depth (below the baseline) of the $\langle \text{coffin} \rangle$ in a form suitable for use in a $\langle \text{dimension expression} \rangle$.

```
\coffin_ht:N
\coffin_ht:c
```

```
\coffin_ht:N <coffin>
```

Calculates the height (above the baseline) of the $\langle \text{coffin} \rangle$ in a form suitable for use in a $\langle \text{dimension expression} \rangle$.

```
\coffin_wd:N
\coffin_wd:c
```

```
\coffin_wd:N <coffin>
```

Calculates the width of the $\langle \text{coffin} \rangle$ in a form suitable for use in a $\langle \text{dimension expression} \rangle$.

5 Coffin diagnostics

```
\coffin_display_handles:Nn
\coffin_display_handles:cn
```

Updated: 2011-09-02

```
\coffin_display_handles:Nn <coffin> {<color>}
```

This function first calculates the intersections between all of the $\langle \text{poles} \rangle$ of the $\langle \text{coffin} \rangle$ to give a set of $\langle \text{handles} \rangle$. It then prints the $\langle \text{coffin} \rangle$ at the current location in the source, with the position of the $\langle \text{handles} \rangle$ marked on the coffin. The $\langle \text{handles} \rangle$ will be labelled as part of this process: the locations of the $\langle \text{handles} \rangle$ and the labels are both printed in the $\langle \text{color} \rangle$ specified.

```
\coffin_mark_handle:Nnnn
\coffin_mark_handle:cnnn
```

Updated: 2011-09-02

```
\coffin_mark_handle:Nnnn <coffin> {\<pole1>} {\<pole2>} {\<color>}
```

This function first calculates the *<handle>* for the *<coffin>* as defined by the intersection of *<pole1>* and *<pole2>*. It then marks the position of the *<handle>* on the *<coffin>*. The *<handle>* will be labelled as part of this process: the location of the *<handle>* and the label are both printed in the *<color>* specified.

```
\coffin_show_structure:N
\coffin_show_structure:c
```

Updated: 2015-08-01

```
\coffin_show_structure:N <coffin>
```

This function shows the structural information about the *<coffin>* 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.

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.

```
\coffin_log_structure:N
\coffin_log_structure:c
```

New: 2014-08-22
Updated: 2015-08-01

```
\coffin_log_structure:N <coffin>
```

This function writes the structural information about the *<coffin>* in the log file. See also *\coffin_show_structure:N* which displays the result in the terminal.

5.1 Constants and variables

```
\c_empty_coffin
```

A permanently empty coffin.

```
\l_tmpa_coffin
\l_tmpb_coffin
```

New: 2012-06-19

Scratch coffins 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.

Part XXVIII

The **l3color** package

Color support

This module provides support for color in L^AT_EX3. At present, the material here is mainly intended to support a small number of low-level requirements in other l3kernel modules.

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
```

Creates a color group: one used to “trap” color settings.

```
\color_ensure_current:  
New: 2011-09-03
```

Ensures that material inside a box will use 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.

Part XXIX

The `\I3sys` package

System/runtime functions

1 The name of the job

`\c_sys_jobname_str`

New: 2015-09-19

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

TeXhackers note: This copies the contents of the primitive `\jobname`. It is a constant that is set by TeX and should not be overwritten by the package.

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.

TeXhackers 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.

3 Engine

`\sys_if_engine_luatex_p: *` `\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.

New: 2015-09-07

`\c_sys_engine_str`

New: 2015-09-19

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

4 Output format

```
\sys_if_output_dvi_p: *          \sys_if_output_dvi:TF {<true code>} {<false code>}  
\sys_if_output_dvi:TF *          Conditionals which give the current output mode the TEX run is operating in. This will  
\sys_if_output_pdf_p: *          always be one of two outcomes, DVI mode or PDF mode. The two sets of conditionals  
\sys_if_output_pdf:TF *          are thus complementary and are both provided to allow the programmer to emphasise  
New: 2015-09-19
```

```
\c_sys_output_str               The current output mode given as a lower case string; will be one of dvi or pdf.  
New: 2015-09-19
```

Part XXX

The **I3deprecation** package

Deprecation errors

1 I3deprecation documentation

A few commands have had to be deprecated over the years. This module defines deprecated and deleted commands to produce an error.

`\deprecation_error`: Defines commands that will soon become deprecated to produce errors.

(End definition for `\deprecation_error`:. This function is documented on page ??.)

Part XXXI

The **I3candidates** package

Experimental additions to **I3kernel**

1 Important notice

This module provides a space in which functions can be added to **I3kernel** (`expl3`) while still being experimental.

As such, the functions here may not remain in their current form, or indeed at all, in **I3kernel in the future.**

In contrast to the material in **I3experimental**, 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!

2 Additions to **I3box**

2.1 Viewing part of a box

`\box_clip:N <box>`

`\box_clip:c`

Clips the `<box>` in the output so that only material inside the bounding box is displayed in the output. The updated `<box>` will be an hbox, irrespective of the nature of the `<box>` before the clipping is applied. The clipping applies within the current **T_EX** group level.

These functions require the **L_AT_EX3 native drivers: they will not work with the **L_AT_EX 2_E** **graphics** drivers!**

TeXhackers 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_trim:Nnnnn  
\box_trim:cnnnn
```

```
\box_trim:Nnnnn <box> {\<left>} {\<bottom>} {\<right>} {\<top>}
```

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 output of the bounding box will still be displayed in the output unless $\backslash box_clip:N$ is subsequently applied. The updated $\langle box \rangle$ will be an hbox, irrespective of the nature of the $\langle box \rangle$ before the trim operation is applied. The adjustment applies within the current TeX group level. The behavior of the operation where the trims requested is greater than the size of the box is undefined.

```
\box_viewport:Nnnnn  
\box_viewport:cnnnn
```

```
\box_viewport:Nnnnn <box> {\<llx>} {\<lly>} {\<urx>} {\<ury>}
```

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 output of the bounding box will still be displayed in the output unless $\backslash box_clip:N$ is subsequently applied. The updated $\langle box \rangle$ will be an hbox, irrespective of the nature of the $\langle box \rangle$ before the viewport operation is applied. The adjustment applies within the current TeX group level.

3 Additions to **I3clist**

```
\clist_rand_item:N *  
\clist_rand_item:c *  
\clist_rand_item:n *
```

New: 2016-12-06

```
\clist_rand_item:N <clist var>  
\clist_rand_item:n {\<comma list>}
```

Selects a pseudo-random item of the $\langle comma list \rangle$. If the $\langle comma list \rangle$ has no item, the result is empty. This is only available in pdfTeX and LuaTeX.

TeXhackers note: The result is returned within the $\backslash unexpanded$ primitive ($\exp_not:n$), which means that the $\langle item \rangle$ will not expand further when appearing in an x-type argument expansion.

4 Additions to **I3coffins**

```
\coffin_resize:Nnn  
\coffin_resize:cnn
```

```
\coffin_resize:Nnn <coffin> {\<width>} {\<total-height>}
```

Resized the $\langle coffin \rangle$ to $\langle width \rangle$ and $\langle total-height \rangle$, both of which should be given as dimension expressions.

```
\coffin_rotate:Nn  
\coffin_rotate:cn
```

```
\coffin_rotate:Nn <coffin> {\<angle>}
```

Rotates the $\langle coffin \rangle$ by the given $\langle angle \rangle$ (given in degrees counter-clockwise). This process will rotate both the coffin content and poles. Multiple rotations will not result in the bounding box of the coffin growing unnecessarily.

```
\coffin_scale:Nnn  
\coffin_scale:cnn
```

```
\coffin_scale:Nnn <coffin> {\<x-scale>} {\<y-scale>}
```

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.

5 Additions to `l3file`

`\file_if_exist_input:nTF`

New: 2014-07-02

```
\file_if_exist_input:n {\file name}
\file_if_exist_input:nTF {\file name} {\true code} {\false code}
```

Searches for *file name* using the current TeX search path and the additional paths controlled by `\file_path_include:n`. If found, inserts the *true code* then reads in the file as additional L^AT_EX source as described for `\file_input:n`. Note that `\file_if_exist_input:n` does not raise an error if the file is not found, in contrast to `\file_input:n`.

`\ior_log_streams:`
`\iow_log_streams:`

New: 2014-08-22

```
\ior_log_streams:
\iow_log_streams:
```

Writes in the log file a list of the file names associated with each open stream: intended for tracking down problems.

6 Additions to `l3int`

`\int_rand:nn *`

New: 2016-12-06

```
\int_rand:nn {\integer expr_1} {\integer expr_2}
```

Evaluates the two *integer expressions* and produces a pseudo-random number between the two (with bounds included). This is only available in pdfTeX and LuaTeX.

7 Additions to `l3msg`

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, the message text and arguments are not expanded, 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. Another particularity of expandable messages is that they cannot be redirected or turned off by the user.

```
\msg_expandable_error:nnnnn ★ \msg_expandable_error:nnnnnn {⟨module⟩} {⟨message⟩} {⟨arg one⟩} {⟨arg two⟩} {⟨arg three⟩} {⟨arg four⟩}
\msg_expandable_error:nnffff ★
\msg_expandable_error:nnnnn ★
\msg_expandable_error:nnfff ★
\msg_expandable_error:nnnn ★
\msg_expandable_error:nnff ★
\msg_expandable_error:nnn ★
\msg_expandable_error:nnf ★
\msg_expandable_error:nn ★
```

New: 2015-08-06

Issues an “Undefined error” message from TeX 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.

8 Additions to `\prop`

```
\prop_count:N ★ \prop_count:N ⟨property list⟩
\prop_count:c ★
```

Leaves the number of key–value pairs in the ⟨property list⟩ in the input stream as an ⟨integer denotation⟩.

```
\prop_map_tokens:Nn ★ \prop_map_tokens:Nn ⟨property list⟩ {⟨code⟩}
\prop_map_tokens:cn ★
```

Analogue of `\prop_map_function:NN` which maps several tokens instead of a single function. The ⟨code⟩ receives each key–value pair in the ⟨property list⟩ as two trailing brace groups. For instance,

```
\prop_map_tokens:Nn \l_my_prop { \str_if_eq:nnT { mykey } }
```

will expand to the value corresponding to `mykey`: for each pair in `\l_my_prop` the function `\str_if_eq:nnT` receives `mykey`, the ⟨key⟩ and the ⟨value⟩ as its three arguments. For that specific task, `\prop_item:Nn` is faster.

```
\prop_rand_key_value:N ★ \prop_rand_key_value:N ⟨prop var⟩
\prop_rand_key_value:c ★
```

New: 2016-12-06

Selects a pseudo-random key–value pair in the ⟨property list⟩ and returns {⟨key⟩}{⟨value⟩}. If the ⟨property list⟩ is empty the result is empty. This is only available in pdfTeX and LuaTeX.

TeXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the ⟨value⟩ will not expand further when appearing in an x-type argument expansion.

9 Additions to `I3seq`

`\seq_mapthread_function:NNN` ★ `\seq_mapthread_function:NNN <seq1> <seq2> <function>`
`\seq_mapthread_function:(NcN|cNN|ccN)` ★

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>` will receive two n-type arguments for each iteration. The mapping will terminate when the end of either sequence is reached (*i.e.* whichever sequence has fewer items determines how many iterations occur).

`\seq_set_filter:NNn`
`\seq_gset_filter:NNn`

Evaluates the `<inline boolexpr>` for every `<item>` stored within the `<sequence2>`. The `<inline boolexpr>` will receive 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 will lead to low-level TeX errors.

`\seq_set_map:NNn`
`\seq_gset_map:NNn`

New: 2011-12-22

Applies `<inline function>` to every `<item>` stored within the `<sequence2>`. The `<inline function>` should consist of code which will receive the `<item>` as #1. The sequence resulting from x-expanding `<inline function>` applied to each `<item>` is assigned to `<sequence1>`. As such, the code in `<inline function>` should be expandable.

TeXhackers note: Contrarily to other mapping functions, `\seq_map_break:` cannot be used in this function, and will lead to low-level TeX errors.

`\seq_rand_item:N` ★ `\seq_rand_item:N <seq var>`

New: 2016-12-06

Selects a pseudo-random item of the `<sequence>`. If the `<sequence>` is empty the result is empty. This is only available in pdfTeX and LuaTeX.

TeXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the `<item>` will not expand further when appearing in an x-type argument expansion.

10 Additions to `I3skip`

`\skip_split_finite_else_action:nnNN` `\skip_split_finite_else_action:nnNN {<skipexpr>} {<action>}`
`<dimen1> <dimen2>`

Checks if the `<skipexpr>` contains finite glue. If it does then it assigns `<dimen1>` the stretch component and `<dimen2>` the shrink component. If it contains infinite glue set `<dimen1>` and `<dimen2>` to 0pt and place #2 into the input stream: this is usually an error or warning message of some sort.

11 Additions to l3sys

<code>\sys_if_rand_exist_p: *</code>	<code>\sys_if_rand_exist_p:</code>
<code>\sys_if_rand_exist:TF *</code>	<code>\sys_if_rand_exist:TF {<true code>} {<false code>}</code>
New: 2017-05-27	Tests if the engine has a pseudo-random number generator. Currently this is the case in pdfTeX and LuaTeX.
<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>	<code>\sys_gset_rand_seed:n {<integer>}</code>
New: 2017-05-27	Sets the seed for the engine's pseudo-random number generator to the <i><integer expression></i> . The assignment is global. This random seed affects all <code>\..._rand</code> functions (such as <code>\int_rand:nn</code> or <code>\clist_rand_item:n</code>) as well as other packages relying on the engine's random number generator. Currently only the absolute value of the seed is used. In engines without random number support this produces an error.
<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	<ul style="list-style-type: none">0 Shell escape is disabled1 Unrestricted shell escape is enabled2 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 will return 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 <i>unrestricted</i> 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 <i>restricted</i> shell escape is enabled. This will return 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 <code>\sys_if_shell:</code> .

```
\sys_shell_now:n {\tokens}
\sys_shell_now:x
```

New: 2017-05-27

Execute *tokens* through shell escape immediately.

```
\sys_shell_shipout:n {\tokens}
\sys_shell_shipout:x
```

New: 2017-05-27

Execute *tokens* through shell escape at shipout.

12 Additions to l3tl

```
\tl_if_single_token_p:n *
\tl_if_single_token:nTF *
```

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.

```
\tl_reverse_tokens:n *
```

This function, which works directly on TeX tokens, reverses the order of the *tokens*: the first will be the last and the last will become first. Spaces are preserved. The reversal also operates within brace groups, but the braces themselves are not exchanged, as this would lead to an unbalanced token list. For instance, `\tl_reverse_tokens:n {a~{b()}}` leaves `{()}~{a}` in the input stream. This function requires two steps of expansion.

TeXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the token list will not expand further when appearing in an `x`-type argument expansion.

```
\tl_count_tokens:n *
```

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. This function requires three expansions, giving an *integer denotation*.

```
\tl_lower_case:n *
\tl_upper_case:n *
\tl_mixed_case:n *
\tl_lower_case:nn *
\tl_upper_case:nn *
\tl_mixed_case:nn *
```

New: 2014-06-30
Updated: 2016-01-12

`\tl_upper_case:n {\tokens}`
`\tl_upper_case:nn {\language} {\tokens}`

These functions are intended to be applied to input which may be regarded broadly as “text”. They traverse the *tokens* and change the case of characters as discussed below. The character code of the characters replaced may be arbitrary: the replacement characters will have standard document-level category codes (11 for letters, 12 for letter-like characters which can also be case-changed). Begin-group and end-group characters in the *tokens* are normalized and become `{` and `}`, respectively.

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_lower_case:n`, `\str_upper_case:n` and `\str_fold_case:n`.

The functions perform expansion on the input in most cases. In particular, input in the form of token lists or expandable functions will be expanded *unless* it falls within

one of the special handling classes described below. This expansion approach means that in general the result of case changing will match the “natural” outcome expected from a “functional” approach to case modification. For example

```
\tl_set:Nn \l_tmpa_tl { hello }
\tl_upper_case:n { \l_tmpa_tl \c_space_tl world }
```

will produce

```
HELLO WORLD
```

The expansion approach taken means that in package mode any L^AT_EX 2 _{ε} “robust” commands which may appear in the input should be converted to engine-protected versions using for example the `\robustify` command from the `etoolbox` package.

\l_t1_case_change_math_t1 Case changing will not take place within math mode material so for example

```
\tl_upper_case:n { Some~text~$y = mx + c$~with~{Braces} }
```

will become

```
SOME TEXT $y = mx + c$ WITH {BRACES}
```

Material inside math mode is left entirely unchanged: in particular, no expansion is undertaken.

Detection of math mode is controlled by the list of tokens in `\l_t1_case_change_math_t1`, which should be in open–close pairs. In package mode the standard settings is

```
$ $ \(\ )
```

Note that while expansion occurs when searching the text it does not apply to math mode material (which should be unaffected by case changing). As such, whilst the opening token for math mode may be “hidden” inside a command/macro, the closing one cannot be as this is being searched for in math mode. Typically, in the types of “text” the case changing functions are intended to apply to this should not be an issue.

\l_t1_case_change_exclude_t1

Case changing can be prevented by using any command on the list `\l_t1_case_change_exclude_t1`. Each entry should be a function to be followed by one argument: the latter will be preserved as-is with no expansion. Thus for example following

```
\tl_put_right:Nn \l_t1_case_change_exclude_t1 { \NoChangeCase }
```

the input

```
\tl_upper_case:n
{ Some~text~$y = mx + c$~with~\NoChangeCase {Protection} }
```

will result in

```
SOME TEXT $y = mx + c$ WITH \NoChangeCase {Protection}
```

Notice that the case changing mapping preserves the inclusion of the escape functions: it is left to other code to provide suitable definitions (typically equivalent to `\use:n`). In particular, the result of case changing is returned protected by `\exp_not:n`.

When used with L^AT_EX 2_& the commands `\cite`, `\ensuremath`, `\label` and `\ref` are automatically included in the list for exclusion from case changing.

\l_t1_case_change_accents_t1

This list specifies accent commands which should be left unexpanded in the output. This allows for example

```
\tl_upper_case:n { \" { a } }
```

to yield

```
\" { A }
```

irrespective of the expandability of `\\"`.

The standard contents of this variable is `\\", \', \., \^, \~, \c, \H, \k, \r, \t, \u` and `\v`.

“Mixed” case conversion may be regarded informally as converting the first character of the *(tokens)* to upper case and the rest to lower case. However, the process is more complex than this as there are some situations where a single lower case character maps to a special form, for example `ij` in Dutch which becomes `IJ`. As such, `\tl_mixed_case:n(n)` implement a more sophisticated mapping which accounts for this and for modifying accents on the first letter. Spaces at the start of the *(tokens)* are ignored when finding the first “letter” for conversion.

```
\tl_mixed_case:n { hello-WORLD }  % => "Hello world"
\tl_mixed_case:n { ~hello-WORLD } % => " Hello world"
\tl_mixed_case:n { {hello}~WORLD } % => "{Hello} world"
```

When finding the first “letter” for this process, any content in math mode or covered by `\l_t1_case_change_exclude_t1` is ignored.

(Note that the Unicode Consortium describe this as “title case”, but that in English title case applies on a word-by-word basis. The “mixed” case implemented here is a lower level concept needed for both “title” and “sentence” casing of text.)

\l_t1_mixed_case_ignore_t1

The list of characters to ignore when searching for the first “letter” in mixed-casing is determined by `\l_t1_mixed_change_ignore_t1`. This has the standard setting

`([{ ‘ -`

where comparisons are made on a character basis.

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 X_ET_X or L_AT_EX 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 font encoding. Thus for example Äđ can be case-changed using pdfT_EX. For pT_EX only the ASCII range is covered as the engine treats input outside of this range as east Asian.

Context-sensitive mappings are enabled: language-dependent cases are discussed below. Context detection will expand input but treats any unexpandable control sequences as “failures” to match a context.

Language-sensitive conversions are enabled using the `\l_t1_set_from_file` command, 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 lower casing I-dot and introduced when upper casing i-dotless.
- German (`de-alt`). An alternative mapping for German in which the lower case *Eszett* maps to a *großes Eszett*.
- Lithuanian (`lt`). The lower case letters i and j should retain a dot above when the accents grave, acute or tilde are present. This is implemented for lower casing of the relevant upper case letters both when input as single Unicode codepoints and when using combining accents. The combining dot is removed when upper casing 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 mixed cased input produces `IJ` rather than `Ij`. The output retains two separate letters, thus this transformation *is* available using pdfT_EX.

Creating additional context-sensitive mappings requires knowledge of the underlying mapping implementation used here. The team are happy to add these to the kernel where they are well-documented (*e.g.* in Unicode Consortium or relevant government publications).

**\t1_set_from_file:Nnn
\t1_set_from_file:cnn
\t1_gset_from_file:Nnn
\t1_gset_from_file:cnn**

`\t1_set_from_file:Nnn <t1> {<setup>} {<filename>}`

Defines `<t1>` to the contents of `<filename>`. Category codes may need to be set appropriately via the `<setup>` argument.

New: 2014-06-25

```
\tl_set_from_file_x:Nnn
\tl_set_from_file_x:cnn
\tl_gset_from_file_x:Nnn
\tl_gset_from_file_x:cnn
```

New: 2014-06-25

```
\tl_rand_item:N *
\tl_rand_item:c *
\tl_rand_item:n *
```

New: 2016-12-06

```
\tl_set_from_file_x:Nnn <tl> {<setup>} {<filename>}
```

Defines *<tl>* to the contents of *<filename>*, expanding the contents of the file as it is read. Category codes and other definitions may need to be set appropriately via the *<setup>* argument.

```
\tl_rand_item:N <tl var>
\tl_rand_item:n {<token list>}
```

Selects a pseudo-random item of the *<token list>*. If the *<token list>* is blank, the result is empty. This is only available in pdfTeX and LuaTeX.

TeXhackers note: The result is returned within the \unexpanded primitive (`\exp_not:n`), which means that the *<item>* will not expand further when appearing in an x-type argument expansion.

```
\tl_range:Nnn *
\tl_range:cnn *
\tl_range:nnn *
```

New: 2017-02-17

```
\tl_range:Nnn <tl var> {<start index>} {<end index>}
\tl_range:nnn {<token list>} {<start index>} {<end index>}
```

Leaves in the input stream the items from the *<start index>* to the *<end index>* inclusive. Positive *<indices>* are counted from the start of the *<token list>*, 1 being the first item, and negative *<indices>* are counted from the end of the token list, -1 being the last item. If either of *<start index>* or *<end index>* is 0, the result is empty. For instance,

```
\iow_term:x { \tl_range:nnn { abcdef } { 2 } { 5 } }
\iow_term:x { \tl_range:nnn { abcdef } { -4 } { -1 } }
\iow_term:x { \tl_range:nnn { abcdef } { -2 } { -1 } }
\iow_term:x { \tl_range:nnn { abcdef } { 0 } { -1 } }
```

will print `bcd`, `cde`, `ef`, and an empty line to the terminal. The *<start index>* must always be smaller than or equal to the *<end index>*: if this is not the case then no output is generated. Thus

```
\iow_term:x { \tl_range:nnn { abcdef } { 5 } { 2 } }
\iow_term:x { \tl_range:nnn { abcdef } { -1 } { -4 } }
```

both yield empty token lists.

TeXhackers note: The result is returned within the \unexpanded primitive (`\exp_not:n`), which means that the *<item>* will not expand further when appearing in an x-type argument expansion.

13 Additions to `I3tokens`

`\peek_N_type:TF` `\peek_N_type:TF {<true code>} {<false code>}`

Updated: 2012-12-20

Tests if the next $\langle token \rangle$ in the input stream can be safely grabbed as an N-type argument. The test will be $\langle false \rangle$ if the next $\langle token \rangle$ 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 $\langle true \rangle$ in all other cases. Note that a $\langle true \rangle$ result ensures that the next $\langle token \rangle$ is a valid N-type argument. However, if the next $\langle token \rangle$ is for instance `\c_space_token`, the test will take the $\langle false \rangle$ branch, even though the next $\langle token \rangle$ is in fact a valid N-type argument. The $\langle token \rangle$ will be left in the input stream after the $\langle true code \rangle$ or $\langle false code \rangle$ (as appropriate to the result of the test).

Part XXXII

The **l3luatex** package

LuaTeX-specific functions

1 Breaking out to Lua

The LuaTeX 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 pdfTeX or XeTeX these will raise an error: use `\sys_if_engine_luatex:T` to avoid this. Details of coding the LuaTeX engine are detailed in the LuaTeX manual.

1.1 TeX code interfaces

`\lua_now_x:n` ★ `\lua_now:n` ★
New: 2015-06-29

`\lua_now:n {<token list>}`

The `<token list>` is first tokenized by TeX, which will include converting line ends to spaces in the usual TeX manner and which respects currently-applicable TeX category codes. The resulting `<Lua input>` is passed to the Lua interpreter for processing. Each `\lua_now:n` block is treated by Lua as a separate chunk. The Lua interpreter will execute the `<Lua input>` immediately, and in an expandable manner.

In the case of the `\lua_now_x:n` version the input is fully expanded by TeX in an x-type manner *but* the function remains fully expandable.

TeXhackers note: `\lua_now_x:n` is a macro wrapper around `\directlua`: when LuaTeX is in use two expansions will be required to yield the result of the Lua code.

`\lua_shipout_x:n` `\lua_shipout:n`
New: 2015-06-30

`\lua_shipout:n {<token list>}`

The `<token list>` is first tokenized by TeX, which will include converting line ends to spaces in the usual TeX manner and which respects currently-applicable TeX category codes. The resulting `<Lua input>` 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 `<Lua input>` during the page-building routine: no TeX expansion of the `<Lua input>` will occur at this stage.

In the case of the `\lua_shipout_x:n` version the input is fully expanded by TeX in an x-type manner during the shipout operation.

TeXhackers note: At a TeX level, the `<Lua input>` is stored as a “whatsit”.

```
\lua_escape_x:n ★  
\lua_escape:n ★
```

New: 2015-06-29

```
\lua_escape:n {<token list>}
```

Converts the *<token list>* 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 `\n` and `\r`, respectively.

In the case of the `\lua_escape_x:n` version the input is fully expanded by TeX in an `x`-type manner *but* the function remains fully expandable.

TeXhackers note: `\lua_escape_x:n` is a macro wrapper around `\luaescapestring`: when LuaTeX is in use two expansions will be required to yield the result of the Lua code.

1.2 Lua interfaces

As well as interfaces for TeX, there are a small number of Lua functions provided here. Currently these are intended for internal use only.

```
l3kernel.strcmp
```

```
\l3kernel.strcmp(<str one>, <str two>)
```

Compares the two strings and returns 0 to TeX if the two are identical.

```
l3kernel.charcat
```

```
\l3kernel.charcat(<charcode>, <catcode>)
```

Constructs a character of *<charcode>* and *<catcode>* and returns the result to TeX.

Part XXXIII

The **I3drivers** package

Drivers

\TeX relies on drivers in order to carry out a number of tasks, such as using color, including graphics and setting up hyper-links. The nature of the code required depends on the exact driver in use. Currently, $\text{\LaTeX}3$ is aware of the following drivers:

- **pdfmode**: The “driver” for direct PDF output by *both* pdf\TeX and Lua\TeX (no separate driver is used in this case: the engine deals with PDF creation itself).
- **dvips**: The dvips program, which works in conjugation with pdf\TeX or Lua\TeX in DVI mode.
- **dvipdfmx**: The dvipdfmx program, which works in conjugation with pdf\TeX or Lua\TeX in DVI mode.
- **dvisvgm**: The dvisvgm program, which works in conjugation with pdf\TeX or Lua\TeX when run in DVI mode as well as with (u)p \TeX and X \TeX .
- **xdvipdfmx**: The driver used by X \TeX .

The code here is all very low-level, and should not in general be used outside of the kernel. It is also important to note that many of the functions here are closely tied to the immediate level “up”, and they must be used in the correct contexts.

1 Box clipping

`__driver_box_use_clip:N`
New: 2011-11-11

`__driver_box_use_clip:N <box>`

Inserts the content of the $\langle box \rangle$ at the current insertion point such that any material outside of the bounding box will not be displayed by the driver. The material in the $\langle box \rangle$ is still placed in the output stream: the clipping takes place at a driver level.

This function should only be used within a surrounding horizontal box construct.

2 Box rotation and scaling

`__driver_box_use_rotate:Nn` `__driver_box_use_rotate:Nn <box> {<angle>}`
New: 2016-05-12

Inserts the content of the $\langle box \rangle$ at the current insertion point rotated by the $\langle angle \rangle$ (expressed in degrees). The material is inserted with no apparent height or width, and is rotated such the the \TeX reference point of the box is the center of rotation and remains the reference point after rotation. It is the responsibility of the code using this function to adjust the apparent size of the box to be correct at the \TeX side.

This function should only be used within a surrounding horizontal box construct.

```
\__driver_box_use_scale:Nnn \__driver_box_use_scale:Nnn <box> {<x-scale>} {<y-scale>}
```

New: 2016-05-12

Inserts the content of the *<box>* at the current insertion point scale by the *<x-scale>* and *<y-scale>*. The material is inserted with no apparent height or width. It is the responsibility of the code using this function to adjust the apparent size of the box to be correct at the TeX side.

This function should only be used within a surrounding horizontal box construct.

3 Color support

```
\__driver_color_ensure_current: \__driver_color_ensure_current:
```

New: 2011-09-03

Updated: 2012-05-18

Ensures that the color used to typeset material is that which was set when the material was placed in a box. This function is therefore required inside any “color safe” box to ensure that the box may be inserted in a location where the foreground color has been altered, while preserving the color used in the box.

4 Drawing

The drawing functions provided here are *highly* experimental. They are inspired heavily by the system layer of pgf (most have the same interface as the same functions in the latter’s `\pgf@sys@...` namespace). They are intended to form the basis for higher level drawing interfaces, which themselves are likely to be further abstracted for user access. Again, this model is heavily inspired by pgf and Tikz.

These low level drawing interfaces abstract from the driver raw requirements but still require an appreciation of the concepts of PostScript/PDF/SVG graphic creation.

```
\__driver_draw_begin: \__driver_draw_begin:  
\__driver_draw_end: <content>  
\__driver_draw_end:
```

Defines a drawing environment. This will be a scope for the purposes of the graphics state. Depending on the driver, other set up may or may not take place here. The natural size of the *<content>* should be zero from the TeX perspective: allowance for the size of the content must be made at a higher level (or indeed this can be skipped if the content is to overlap other material).

```
\__driver_draw_scope_begin: \__driver_draw_scope_begin:  
\__driver_draw_scope_end: <content>  
\__driver_draw_scope_end:
```

Defines a scope for drawing settings and so on. Changes to the graphic state and concepts such as color or linewidth are localised to a scope. This function pair must never be used if an partial path is under construction: such paths must be entirely contained at one unbroken scope level. Note that scopes do not form TeX groups and may not be aligned with them.

4.1 Path construction

<u>_driver_draw_moveto:nn</u>	<u>_driver_draw_move:nn</u> { $\langle x \rangle$ } { $\langle y \rangle$ }
	Moves the current drawing reference point to ($\langle x \rangle$, $\langle y \rangle$); any active transformation matrix will apply.
<u>_driver_draw_lineto:nn</u>	<u>_driver_draw_lineto:nn</u> { $\langle x \rangle$ } { $\langle y \rangle$ }
	Adds a path from the current drawing reference point to ($\langle x \rangle$, $\langle y \rangle$); any active transformation matrix will apply. Note that nothing is drawn until a fill or stroke operation is applied, and that the path may be discarded or used as a clip without appearing itself.
<u>_driver_draw_curveto:nnnnnn</u>	<u>_driver_draw_curveto:nnnnnn</u> { $\langle x_1 \rangle$ } { $\langle y_1 \rangle$ } { $\langle x_2 \rangle$ } { $\langle y_2 \rangle$ } { $\langle x_3 \rangle$ } { $\langle y_3 \rangle$ }
	Adds a Bezier curve path from the current drawing reference point to ($\langle x_3 \rangle$, $\langle y_3 \rangle$), using ($\langle x_1 \rangle$, $\langle y_1 \rangle$) and ($\langle x_2 \rangle$, $\langle y_2 \rangle$) as control points; any active transformation matrix will apply. Note that nothing is drawn until a fill or stroke operation is applied, and that the path may be discarded or used as a clip without appearing itself.
<u>_driver_draw_rectangle:nnnn</u>	<u>_driver_draw_rectangle:nnnn</u> { $\langle x \rangle$ } { $\langle y \rangle$ } { $\langle width \rangle$ } { $\langle height \rangle$ }
	Adds rectangular path from ($\langle x_1 \rangle$, $\langle y_1 \rangle$) of $\langle height \rangle$ and $\langle width \rangle$; any active transformation matrix will apply. Note that nothing is drawn until a fill or stroke operation is applied, and that the path may be discarded or used as a clip without appearing itself.
<u>_driver_draw_closepath:</u>	<u>_driver_draw_closepath:</u>
	Closes an existing path, adding a line from the current point to the start of path. Note that nothing is drawn until a fill or stroke operation is applied, and that the path may be discarded or used as a clip without appearing itself.

4.2 Stroking and filling

<u>_driver_draw_stroke:</u>	<i><path construction></i>
<u>_driver_draw_closestroke:</u>	<u>_driver_draw_stroke:</u>
	Draws a line along the current path, which will also be closed when <u>_driver_draw_closestroke:</u> is used. The nature of the line drawn is influenced by settings for
	<ul style="list-style-type: none">• Line thickness• Stroke color (or the current color if no specific stroke color is set)• Line capping (how non-closed line ends should look)• Join style (how a bend in the path should be rendered)• Dash pattern
	The path may also be used for clipping.

<code>__driver_draw_fill:</code>	<code>__driver_draw_fillstroke:</code>	<code>__driver_draw_fill:</code> <i>(path construction)</i>
Fills the area surrounded by the current path: this will be closed prior to filling if it is not already. The <code>fillstroke</code> version will also stroke the path as described for <code>__driver_draw_stroke</code> . The fill is influenced by the setting for fill color (or the current color if no specific stroke color is set). The path may also be used for clipping. For paths which are self-intersecting or comprising multiple parts, the determination of which areas are inside the path is made using the non-zero winding number rule unless the even-odd rule is active.		
<code>__driver_draw_nonzero_rule:</code>	<code>__driver_draw_nonzero_rule:</code>	<code>__driver_draw_evenodd_rule:</code>
Active either the non-zero winding number or the even-odd rule, respectively, for determining what is inside a fill or clip area. For technical reasons, these command are not influenced by scoping and apply on an ongoing basis.		
<code>__driver_draw_clip:</code>	<code>__driver_draw_clip:</code>	<code>__driver_draw_clip:</code> <i>(path construction)</i>
Indicates that the current path should be used for clipping, such that any subsequent material outside of the path (but within the current scope) will not be shown. This command should be given once a path is complete but before it is stroked or filled (if appropriate). This command is <i>not</i> affected by scoping: it applies to exactly one path as shown.		
<code>__driver_draw_discardpath:</code>	<code>__driver_draw_discardpath:</code>	<code>__driver_draw_discardpath:</code> <i>(path construction)</i>
Discards the current path without stroking or filling. This is primarily useful for paths constructed purely for clipping, as this alone does not end the paths existence.		

4.3 Stroke options

<code>__driver_draw_linewidth:n</code>	<code>__driver_draw_linewidth:n {<dimexpr>}</code>
Sets the width to be used for stroking to <i><dimexpr></i> .	
<code>__driver_draw_dash:nn</code>	<code>__driver_draw_dash:nn {<dash pattern>} {<phase>}</code>
Sets the pattern of dashing to be used when stroking a line. The <i><dash pattern></i> should be a comma-separated list of dimension expressions. This is then interpreted as a series of pairs of line-on and line-off lengths. For example <code>3pt, 4pt</code> means that 3 pt on, 4 pt off, 3 pt on, and so on. A more complex pattern will also repeat: <code>3pt, 4pt, 1pt, 2pt</code> results in 3 pt on, 4 pt off, 1 pt on, 2 pt off, 3 pt on, and so on. An odd number of entries means that the last is repeated, for example <code>3pt</code> is equal to <code>3pt, 3pt</code> . An empty pattern yields a solid line. The <i><phase></i> specifies an offset at the start of the cycle. For example, with a pattern <code>3pt</code> a phase of <code>1pt</code> will mean that the output is 2 pt on, 3 pt off, 3 pt on, 3 pt on, etc.	

```
\__driver_draw_cap_butt:           \__driver_draw_cap_butt:  
\__driver_draw_cap_rectangle:  
\__driver_draw_cap_round:
```

Sets the style of terminal stroke position to one of butt, rectangle or round.

```
\__driver_draw_join_bevel: \__driver_draw_cap_butt:  
\__driver_draw_join_miter:  
\__driver_draw_join_round:
```

Sets the style of stroke joins to one of bevel, miter or round.

```
\__driver_draw_miterlimit:n \__driver_draw_miterlimit:n {\<dimexpr>}
```

Sets the miter limit of lines joined as a miter, as described in the PDF and PostScript manuals.

4.4 Color

```
\__driver_draw_color_cmyk:nnnn      \__driver_draw_color_cmyk:nnnn {\<cyan>} {\<magenta>} {\<yellow>}  
\__driver_draw_color_cmyk_fill:nnnn {\<black>}  
\__driver_draw_color_cmyk_stroke:nnnn
```

Sets the color for drawing to the CMYK values specified, all of which are fp expressions which should evaluate to between 0 and 1. The **fill** and **stroke** versions set only the color for those operations. Note that the general setting is more efficient with some drivers so should in most cases be preferred.

```
\__driver_draw_color_gray:n          \__driver_draw_color_gray:n {\<gray>}  
\__driver_draw_color_gray_fill:n  
\__driver_draw_color_gray_stroke:n
```

Sets the color for drawing to the grayscale value specified, which is fp expressions which should evaluate to between 0 and 1. The **fill** and **stroke** versions set only the color for those operations. Note that the general setting is more efficient with some drivers so should in most cases be preferred.

```
\__driver_draw_color_rgb:nnn        \__driver_draw_color_gray:n {\<red>} {\<green>} {\<blue>}  
\__driver_draw_color_rgb_fill:nnn  
\__driver_draw_color_rgb_stroke:nnn
```

Sets the color for drawing to the RGB values specified, all of which are fp expressions which should evaluate to between 0 and 1. The **fill** and **stroke** versions set only the color for those operations. Note that the general setting is more efficient with some drivers so should in most cases be preferred.

4.5 Inserting T_EX material

```
\__driver_draw_hbox:Nnnnnnnn \__driver_draw_hbox:Nnnnnnnn <box>
{<a>} {<b>} {<c>} {<d>} {<x>} {<y>}
```

Inserts the $\langle box \rangle$ as an hbox with the box reference point placed at (x, y) . The transformation matrix $[abcd]$ will be applied to the box, allowing it to be in synchronisation with any scaling, rotation or skewing applying more generally. Note that T_EX material should not be inserted directly into a drawing as it will not be in the correct location. Also note that as for other drawing elements the box here will have no size from a T_EX perspective.

4.6 Coordinate system transformations

```
\__driver_draw_transformcm:nnnnnn \__driver_draw_transformcm:nnnnnn {<a>} {<b>} {<c>} {<d>}
{<x>} {<y>}
```

Applies the transformation matrix $[abcd]$ and offset vector (x, y) to the current graphic state. This will affect any subsequent items in the same scope but not those already given.

Part XXXIV

Implementation

1 I3bootstrap implementation

```
1  {*initex | package}
2  (@@=expl)
```

1.1 Format-specific code

The very first thing to do is to bootstrap the iniT_EX system so that everything else will actually work. T_EX does not start with some pretty basic character codes set up.

```
3  {*initex}
4  \catcode '\{ = 1 %
5  \catcode '\} = 2 %
6  \catcode '\# = 6 %
7  \catcode '\^ = 7 %
8  /{initex}
```

Tab characters should not show up in the code, but to be on the safe side.

```
9  {*initex}
10 \catcode '\^I = 10 %
11 /{initex}
```

For LuaT_EX, the extra primitives need to be enabled. This is not needed in package mode: common formats have the primitives enabled.

```
12 (*initex)
13 \begingroup\expandafter\expandafter\expandafter\endgroup
14 \expandafter\ifx\csname directlua\endcsname\relax
15 \else
```

```

16     \directlua{tex.enableprimitives("", tex.extraprimitives())}%
17 \fi
18 </initex>

```

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 LuaT_EX 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.

```

19 <*package>
20 \begingroup
21   \expandafter\ifx\csname directlua\endcsname\relax
22   \else
23     \directlua{%
24       local i
25       local t = { }
26       for _,i in pairs(tex.extraprimitives("luatex")) do
27         if string.match(i,"^U") then
28           if not string.match(i,"^Uchar$") then
29             table.insert(t,i)
30           end
31         end
32       end
33       tex.enableprimitives("", t)
34     }%
35   \fi
36 \endgroup
37 </package>

```

1.2 The \pdfstrcmp primitive in X_ET_EX

Only pdfT_EX has a primitive called \pdfstrcmp. The X_ET_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”.

```

38 \begingroup\expandafter\expandafter\expandafter\endgroup
39   \expandafter\ifx\csname pdfstrcmp\endcsname\relax
40   \let\pdfstrcmp\strcmp
41 \fi

```

1.3 Loading support Lua code

When LuaT_EX 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.

```

42 \begingroup\expandafter\expandafter\expandafter\endgroup
43   \expandafter\ifx\csname directlua\endcsname\relax
44   \else
45     \ifnum\luatexversion<70 %
46     \else

```

In package mode a category code table is needed: either use a pre-loaded allocator or provide one using the L^AT_EX 2_E-based generic code. In format mode the table used here can be hard-coded into the Lua.

```

47  {*package}
48      \begingroup\expandafter\expandafter\expandafter\endgroup
49      \expandafter\ifx\csname newcatcodetable\endcsname\relax
50          \input{ltluatex}%
51      \fi
52      \newcatcodetable\ucharcat@table
53      \directlua{
54          l3kernel = l3kernel or {}
55          local charcat_table = \number\ucharcat@table\space
56          l3kernel.charcat_table = charcat_table
57      }%
58  
```

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 will reveal what mode is in operation.

```

60      \ifnum 0%
61          \directlua{
62              if status.ini_version then
63                  tex.write("1")
64              end
65          }>0 %
66          \everyjob\expandafter{%
67              \the\expandafter\everyjob
68              \csname\detokenize{lua_now_x:n}\endcsname{require("expl3")}%
69          }%
70      \fi
71  
```

1.4 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.

```

73  \begingroup
74      \def\next{\endgroup}%
75      \def\ShortText{Required primitives not found}%
76      \def\LongText{%
77          {%
78              LaTeX3 requires the e-TeX primitives and additional functionality as
79              described in the README file.
80              \LineBreak
81              These are available in the engines\LineBreak
82              - pdfTeX v1.40\LineBreak
83              - XeTeX v0.9994\LineBreak
84              - LuaTeX v0.70\LineBreak
85              - e-(u)pTeX mid-2012\LineBreak
86              or later.\LineBreak
87              \LineBreak
88          }%
89      \ifnum0%
90          \expandafter\ifx\csname pdfstrcmp\endcsname\relax

```

```

91      \else
92          \expandafter\ifx\csname pdftexversion\endcsname\relax
93              %
94          \else
95              \ifnum\pdftexversion<140 \else 1\fi
96          \fi
97      \fi
98      \expandafter\ifx\csname directlua\endcsname\relax
99          \else
100             \ifnum\luatexversion<70 \else 1\fi
101         \fi
102         =0 %
103         \newlinechar`^\^J %
104     (*initex)
105         \def\LineBreak{^\^J}%
106         \edef\next
107             {
108                 \errhelp
109                 {%
110                     \LongText
111                     For pdfTeX and XeTeX the '-etex' command-line switch is also
112                     needed.\LineBreak
113                     \LineBreak
114                     Format building will abort!\LineBreak
115                 }%
116                 \errmessage{\ShortText}%
117                 \endgroup
118                 \noexpand\end
119             }%
120     (/initex)
121     (*package)
122         \def\LineBreak{\noexpand\MessageBreak}%
123         \expandafter\ifx\csname PackageError\endcsname\relax
124             \def\LineBreak{^\^J}%
125             \def\PackageError#1#2#3%
126                 {%
127                     \errhelp{#3}%
128                     \errmessage{#1 Error: #2}%
129                 }%
130             \fi
131             \edef\next
132                 {
133                     \noexpand\PackageError{expl3}{\ShortText}
134                     {\LongText Loading of expl3 will abort!}%
135                 \endgroup
136                 \noexpand\endinput
137             }%
138     (/package)
139         \fi
140     \next

```

1.5 Extending allocators

In format mode, allocating registers is handled by `\3alloc`. However, in package mode it's much safer to rely on more general code. For example, 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 $\varepsilon\text{-}\TeX$ extensions and `\pdfstrcmp` or equivalent are available. Thus there is no danger of an “uncontrolled” error if the engine requirements are not met.

For $\text{\LaTeX} 2\varepsilon$ 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.

```
141  (*package)
142  \begingroup
143  \def\@tempa{LaTeX2e}%
144  \def\@next{}%
145  \ifx\fntname\@tempa
146  \expandafter\ifx\csname extrafloats\endcsname\relax
147  \def\@next
148  {%
149  \RequirePackage{etex}%
150  \csname reserveinserts\endcsname{32}%
151  }%
152  \fi
153  \fi
154 \expandafter\endgroup
155 \next
156 </package>
```

1.6 Character data

\TeX needs various pieces of data to be set about characters, in particular which ones to treat as letters and which `\lccode` values apply as these affect hyphenation. It makes most sense to set this and related information up in one place. Whilst for \LaTeX hyphenation patterns can be read anywhere, other engines have to build them into the format and so we *must* do this set up before reading the patterns. For the Unicode engines, there are shared loaders available to obtain the relevant information directly from the Unicode Consortium data files. These need standard (Ini) \TeX category codes and primitive availability and must therefore load *very* early. This has a knock-on effect on the 8-bit set up: it makes sense to do the definitions for those here as well so it is all in one place.

For \XeTeX and \LuaTeX , which are natively Unicode engines, simply load the Unicode data.

```
157  (*initex)
158  \ifdefined\Umathcode
```

```

159     \input load-unicode-data %
160     \input load-unicode-math-classes %
161 \else
```

For the 8-bit engines a font encoding scheme must be chosen. At present, this is the EC (T1) scheme, with the assumption that languages for which this is not appropriate will be used with one of the Unicode engines.

```
162     \begingroup
```

Lower case chars: map to themselves when lower casing and down by "20 when upper casing. (The characters a–z are set up correctly by IniTeX.)

```

163     \def\temp{%
164     \ifnum\count0>\count2 %
165     \else
166         \global\lccode\count0 = \count0 %
167         \global\uccode\count0 = \numexpr\count0 - "20\relax
168         \advance\count0 by 1 %
169         \expandafter\temp
170     \fi
171 }
172 \count0 = "A0 %
173 \count2 = "BC %
174 \temp
175 \count0 = "E0 %
176 \count2 = "FF %
177 \temp
```

Upper case chars: map up by "20 when lower casing, to themselves when upper casing and require an \sfcode of 999. (The characters A–Z are set up correctly by IniTeX.)

```

178     \def\temp{%
179     \ifnum\count0>\count2 %
180     \else
181         \global\lccode\count0 = \numexpr\count0 + "20\relax
182         \global\uccode\count0 = \count0 %
183         \global\sfcodes\count0 = 999 %
184         \advance\count0 by 1 %
185         \expandafter\temp
186     \fi
187 }
188 \count0 = "80 %
189 \count2 = "9C %
190 \temp
191 \count0 = "C0 %
192 \count2 = "DF %
193 \temp
```

A few special cases where things are not as one might expect using the above pattern: dotless-I, dotless-J, dotted-I and d-bar.

```

194     \global\lccode`^\^Y = `^\^Y %
195     \global\uccode`^\^Y = `^I %
196     \global\lccode`^\^Z = `^\^Z %
197     \global\uccode`^\^Y = `^J %
198     \global\lccode"9D = `^i %
199     \global\uccode"9D = "9D %
200     \global\lccode"9E = "9E %
201     \global\uccode"9E = "D0 %
```

Allow hyphenation at a zero-width glyph (used to break up ligatures or to place accents between characters).

```
202     \global\lccode`23 = 23 %
203     \endgroup
204     \fi
```

In all cases it makes sense to set up – to map to itself: this allows hyphenation of the rest of a word following it (suggested by Lars Helström).

```
205 \global\lccode`- = `-
206 </initex>
```

1.7 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` will be a “do nothing” command until `\ExplSyntaxOn` is used. For format mode, there is no need to save category codes so that step is skipped.

```
207 \protected\def\ExplSyntaxOff{}%
208 (*package)
209 \protected\edef\ExplSyntaxOff
210   {%
211     \protected\def\ExplSyntaxOff{}%
212     \catcode 9 = \the\catcode 9\relax
213     \catcode 32 = \the\catcode 32\relax
214     \catcode 34 = \the\catcode 34\relax
215     \catcode 38 = \the\catcode 38\relax
216     \catcode 58 = \the\catcode 58\relax
217     \catcode 94 = \the\catcode 94\relax
218     \catcode 95 = \the\catcode 95\relax
219     \catcode 124 = \the\catcode 124\relax
220     \catcode 126 = \the\catcode 126\relax
221     \endlinechar = \the\endlinechar\relax
222     \chardef\csname\detokenize{l__kernel_expl_bool}\endcsname = 0\relax
223   }%
224 </package>
```

(End definition for `\ExplSyntaxOff`. This function is documented on page 6.)

The code environment is now set up.

```
225 \catcode 9 = 9\relax
226 \catcode 32 = 9\relax
227 \catcode 34 = 12\relax
228 \catcode 38 = 4\relax
229 \catcode 58 = 11\relax
230 \catcode 94 = 7\relax
231 \catcode 95 = 11\relax
232 \catcode 124 = 12\relax
233 \catcode 126 = 10\relax
234 \endlinechar = 32\relax
```

\l__kernel_expl_bool The status for experimental code syntax: this is on at present.

```
235 \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` will alter the definition of `\ExplSyntaxOff` and so in package mode this function should not be used until after the end of the loading process!

```
236 \protected \def \ExplSyntaxOn
237   {
238     \bool_if:NF \l_kernel_expl_bool
239     {
240       \cs_set_protected:Npx \ExplSyntaxOff
241       {
242         \char_set_catcode:n { 9 } { \char_value_catcode:n { 9 } }
243         \char_set_catcode:n { 32 } { \char_value_catcode:n { 32 } }
244         \char_set_catcode:n { 34 } { \char_value_catcode:n { 34 } }
245         \char_set_catcode:n { 38 } { \char_value_catcode:n { 38 } }
246         \char_set_catcode:n { 58 } { \char_value_catcode:n { 58 } }
247         \char_set_catcode:n { 94 } { \char_value_catcode:n { 94 } }
248         \char_set_catcode:n { 95 } { \char_value_catcode:n { 95 } }
249         \char_set_catcode:n { 124 } { \char_value_catcode:n { 124 } }
250         \char_set_catcode:n { 126 } { \char_value_catcode:n { 126 } }
251         \tex_endlinechar:D =
252           \tex_the:D \tex_endlinechar:D \scan_stop:
253         \bool_set_false:N \l_kernel_expl_bool
254         \cs_set_protected:Npn \ExplSyntaxOff { }
255       }
256     }
257     \char_set_catcode_ignore:n { 9 } % tab
258     \char_set_catcode_ignore:n { 32 } % space
259     \char_set_catcode_other:n { 34 } % double quote
260     \char_set_catcode_alignment:n { 38 } % ampersand
261     \char_set_catcode_letter:n { 58 } % colon
262     \char_set_catcode_math_superscript:n { 94 } % circumflex
263     \char_set_catcode_letter:n { 95 } % underscore
264     \char_set_catcode_other:n { 124 } % pipe
265     \char_set_catcode_space:n { 126 } % tilde
266     \tex_endlinechar:D = 32 \scan_stop:
267     \bool_set_true:N \l_kernel_expl_bool
268   }
```

(End definition for `\ExplSyntaxOn`. This function is documented on page 6.)

269 ⟨/initex | package⟩

2 I3names implementation

270 ⟨*initex | package⟩

No prefix substitution here.

271 ⟨@@=⟩

The code here simply renames all of the primitives to new, internal, names. In format mode, it also deletes all of the existing names (although some do come back later).

272 \tex_undefined:D This function does not exist at all, but is the name used by the plain TeX format for an undefined function. So it should be marked here as “taken”.

(End definition for \tex_undefined:D.)

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.

```
272 \let \tex_global:D \global  
273 \let \tex_let:D \let
```

Everything is inside a (rather long) group, which keeps __kernel_primitive:NN trapped.

```
274 \begingroup
```

__kernel_primitive:NN A temporary function to actually do the renaming. This also allows the original names to be removed in format mode.

```
275 \long \def \_\_kernel\_primitive:NN #1#2  
276 {  
277   \tex_global:D \tex_let:D #2 #1  
278   {*initex}  
279   \tex_global:D \tex_let:D #1 \tex_undefined:D  
280   {/initex}  
281 }
```

(End definition for __kernel_primitive:NN.)

To allow extracting “just the names”, a bit of DocStrip fiddling.

```
282 {/initex | package}  
283 {*initex | 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.

```
284 \_\_kernel_primitive:NN \tex_space:D  
285 \_\_kernel_primitive:NN \tex_italiccorrection:D  
286 \_\_kernel_primitive:NN \tex_hyphen:D
```

Now all the other primitives.

287 __kernel_primitive:NN \above	\tex_above:D
288 __kernel_primitive:NN \abovedisplayshortskip	\tex_abovedisplayshortskip:D
289 __kernel_primitive:NN \abovedisplayskip	\tex_abovedisplayskip:D
290 __kernel_primitive:NN \abovewithdelims	\tex_abovewithdelims:D
291 __kernel_primitive:NN \accent	\tex_accent:D
292 __kernel_primitive:NN \adjdemerits	\tex_adjdemerits:D
293 __kernel_primitive:NN \advance	\tex_advance:D
294 __kernel_primitive:NN \afterassignment	\tex_afterassignment:D
295 __kernel_primitive:NN \aftergroup	\tex_aftergroup:D
296 __kernel_primitive:NN \atop	\tex_atop:D
297 __kernel_primitive:NN \atopwithdelims	\tex_atopwithdelims:D
298 __kernel_primitive:NN \badness	\tex_badness:D
299 __kernel_primitive:NN \baselineskip	\tex_baselineskip:D
300 __kernel_primitive:NN \batchmode	\tex_batchmode:D
301 __kernel_primitive:NN \begingroup	\tex_begingroup:D
302 __kernel_primitive:NN \belowdisplayshortskip	\tex_belowdisplayshortskip:D
303 __kernel_primitive:NN \belowdisplayskip	\tex_belowdisplayskip:D
304 __kernel_primitive:NN \binoppenalty	\tex_binoppenalty:D
305 __kernel_primitive:NN \botmark	\tex_botmark:D

```

306  \__kernel_primitive:NN \box
307  \__kernel_primitive:NN \boxmaxdepth
308  \__kernel_primitive:NN \brokenpenalty
309  \__kernel_primitive:NN \catcode
310  \__kernel_primitive:NN \char
311  \__kernel_primitive:NN \chardef
312  \__kernel_primitive:NN \cleaders
313  \__kernel_primitive:NN \closein
314  \__kernel_primitive:NN \closeout
315  \__kernel_primitive:NN \clubpenalty
316  \__kernel_primitive:NN \copy
317  \__kernel_primitive:NN \count
318  \__kernel_primitive:NN \countdef
319  \__kernel_primitive:NN \cr
320  \__kernel_primitive:NN \crcr
321  \__kernel_primitive:NN \csname
322  \__kernel_primitive:NN \day
323  \__kernel_primitive:NN \deadcycles
324  \__kernel_primitive:NN \def
325  \__kernel_primitive:NN \defaulthyphenchar
326  \__kernel_primitive:NN \defaultskewchar
327  \__kernel_primitive:NN \delcode
328  \__kernel_primitive:NN \delimiter
329  \__kernel_primitive:NN \delimiterfactor
330  \__kernel_primitive:NN \delimitershortfall
331  \__kernel_primitive:NN \dimen
332  \__kernel_primitive:NN \dimendef
333  \__kernel_primitive:NN \discretionary
334  \__kernel_primitive:NN \displayindent
335  \__kernel_primitive:NN \displaylimits
336  \__kernel_primitive:NN \displaystyle
337  \__kernel_primitive:NN \displaywidowpenalty
338  \__kernel_primitive:NN \displaywidth
339  \__kernel_primitive:NN \divide
340  \__kernel_primitive:NN \doublehyphendemerits
341  \__kernel_primitive:NN \dp
342  \__kernel_primitive:NN \dump
343  \__kernel_primitive:NN \edef
344  \__kernel_primitive:NN \else
345  \__kernel_primitive:NN \emergencystretch
346  \__kernel_primitive:NN \end
347  \__kernel_primitive:NN \endcsname
348  \__kernel_primitive:NN \endgroup
349  \__kernel_primitive:NN \endinput
350  \__kernel_primitive:NN \endlinechar
351  \__kernel_primitive:NN \eqno
352  \__kernel_primitive:NN \errhelp
353  \__kernel_primitive:NN \errmessage
354  \__kernel_primitive:NN \errorcontextlines
355  \__kernel_primitive:NN \errorstopmode
356  \__kernel_primitive:NN \escapechar
357  \__kernel_primitive:NN \everycr
358  \__kernel_primitive:NN \everydisplay
359  \__kernel_primitive:NN \everyhbox
                                         \tex_box:D
                                         \tex_boxmaxdepth:D
                                         \tex_brokenpenalty:D
                                         \tex_catcode:D
                                         \tex_char:D
                                         \tex_chardef:D
                                         \tex_cleaders:D
                                         \tex_closein:D
                                         \tex_closeout:D
                                         \tex_clubpenalty:D
                                         \tex_copy:D
                                         \tex_count:D
                                         \tex_countdef:D
                                         \tex_cr:D
                                         \tex_crcr:D
                                         \tex_csname:D
                                         \tex_day:D
                                         \tex_deadcycles:D
                                         \tex_def:D
                                         \tex_defaulthyphenchar:D
                                         \tex_defaultskewchar:D
                                         \tex_delcode:D
                                         \tex_delimiter:D
                                         \tex_delimiterfactor:D
                                         \tex_delimitershortfall:D
                                         \tex_dimen:D
                                         \tex_dimendef:D
                                         \tex_discretionary:D
                                         \tex_displayindent:D
                                         \tex_displaylimits:D
                                         \tex_displaystyle:D
                                         \tex_displaywidowpenalty:D
                                         \tex_displaywidth:D
                                         \tex_divide:D
                                         \tex_doublehyphendemerits:D
                                         \tex_dp:D
                                         \tex_dump:D
                                         \tex_edef:D
                                         \tex_else:D
                                         \tex_emergencystretch:D
                                         \tex_end:D
                                         \tex_endcsname:D
                                         \tex_endgroup:D
                                         \tex_endinput:D
                                         \tex_endlinechar:D
                                         \tex_eqno:D
                                         \tex_errhelp:D
                                         \tex_errmessage:D
                                         \tex_errorcontextlines:D
                                         \tex_errorstopmode:D
                                         \tex_escapechar:D
                                         \tex_everycr:D
                                         \tex_everydisplay:D
                                         \tex_everyhbox:D

```

```

360  \__kernel_primitive:NN \everyjob
361  \__kernel_primitive:NN \everymath
362  \__kernel_primitive:NN \everypar
363  \__kernel_primitive:NN \everyvbox
364  \__kernel_primitive:NN \exhyphenpenalty
365  \__kernel_primitive:NN \expandafter
366  \__kernel_primitive:NN \fam
367  \__kernel_primitive:NN \fi
368  \__kernel_primitive:NN \finalhyphendemerits
369  \__kernel_primitive:NN \firstmark
370  \__kernel_primitive:NN \floatingpenalty
371  \__kernel_primitive:NN \font
372  \__kernel_primitive:NN \fontdimen
373  \__kernel_primitive:NN \fontname
374  \__kernel_primitive:NN \futurelet
375  \__kernel_primitive:NN \gdef
376  \__kernel_primitive:NN \global
377  \__kernel_primitive:NN \globaldefs
378  \__kernel_primitive:NN \halign
379  \__kernel_primitive:NN \hangafter
380  \__kernel_primitive:NN \hangindent
381  \__kernel_primitive:NN \hbadness
382  \__kernel_primitive:NN \hbox
383  \__kernel_primitive:NN \hfil
384  \__kernel_primitive:NN \hfill
385  \__kernel_primitive:NN \hfilneg
386  \__kernel_primitive:NN \hfuzz
387  \__kernel_primitive:NN \hoffset
388  \__kernel_primitive:NN \holdinginserts
389  \__kernel_primitive:NN \hrule
390  \__kernel_primitive:NN \hsize
391  \__kernel_primitive:NN \hskip
392  \__kernel_primitive:NN \hss
393  \__kernel_primitive:NN \ht
394  \__kernel_primitive:NN \hyphenation
395  \__kernel_primitive:NN \hyphenchar
396  \__kernel_primitive:NN \hyphenpenalty
397  \__kernel_primitive:NN \if
398  \__kernel_primitive:NN \ifcase
399  \__kernel_primitive:NN \ifcat
400  \__kernel_primitive:NN \ifdim
401  \__kernel_primitive:NN \ifeof
402  \__kernel_primitive:NN \iffalse
403  \__kernel_primitive:NN \ifhbox
404  \__kernel_primitive:NN \ifhmode
405  \__kernel_primitive:NN \ifinner
406  \__kernel_primitive:NN \ifmmode
407  \__kernel_primitive:NN \ifnum
408  \__kernel_primitive:NN \ifodd
409  \__kernel_primitive:NN \iftrue
410  \__kernel_primitive:NN \ifvbox
411  \__kernel_primitive:NN \ifvmode
412  \__kernel_primitive:NN \ifvoid
413  \__kernel_primitive:NN \ifx

\text{everyjob:D}
\text{everymath:D}
\text{everypar:D}
\text{everyvbox:D}
\text{exhyphenpenalty:D}
\text{expandafter:D}
\text{fam:D}
\text{fi:D}
\text{finalhyphendemerits:D}
\text{firstmark:D}
\text{floatingpenalty:D}
\text{font:D}
\text{fontdimen:D}
\text{fontname:D}
\text{futurelet:D}
\text{gdef:D}
\text{global:D}
\text{globaldefs:D}
\text{halign:D}
\text{hangafter:D}
\text{hangindent:D}
\text{hbadness:D}
\text{hbox:D}
\text{hfil:D}
\text{hfill:D}
\text{hfilneg:D}
\text{hfuzz:D}
\text{hoffset:D}
\text{holdinginserts:D}
\text{hrule:D}
\text{hsize:D}
\text{hskip:D}
\text{hss:D}
\text{ht:D}
\text{hyphenation:D}
\text{hyphenchar:D}
\text{hyphenpenalty:D}
\text{if:D}
\text{ifcase:D}
\text{ifcat:D}
\text{ifdim:D}
\text{ifeof:D}
\text{iffalse:D}
\text{ifhbox:D}
\text{ifhmode:D}
\text{ifinner:D}
\text{ifmmode:D}
\text{ifnum:D}
\text{ifodd:D}
\text{iftrue:D}
\text{ifvbox:D}
\text{ifvmode:D}
\text{ifvoid:D}
\text{ifx:D}

```

```

414  \__kernel_primitive:NN \ignorespaces
415  \__kernel_primitive:NN \immediate
416  \__kernel_primitive:NN \indent
417  \__kernel_primitive:NN \input
418  \__kernel_primitive:NN \inputlineno
419  \__kernel_primitive:NN \insert
420  \__kernel_primitive:NN \insertpenalties
421  \__kernel_primitive:NN \interlinepenalty
422  \__kernel_primitive:NN \jobname
423  \__kernel_primitive:NN \kern
424  \__kernel_primitive:NN \language
425  \__kernel_primitive:NN \lastbox
426  \__kernel_primitive:NN \lastkern
427  \__kernel_primitive:NN \lastpenalty
428  \__kernel_primitive:NN \lastskip
429  \__kernel_primitive:NN \lccode
430  \__kernel_primitive:NN \leaders
431  \__kernel_primitive:NN \left
432  \__kernel_primitive:NN \lefthyphenmin
433  \__kernel_primitive:NN \leftskip
434  \__kernel_primitive:NN \leqno
435  \__kernel_primitive:NN \let
436  \__kernel_primitive:NN \limits
437  \__kernel_primitive:NN \linepenalty
438  \__kernel_primitive:NN \lineskip
439  \__kernel_primitive:NN \lineskiplimit
440  \__kernel_primitive:NN \long
441  \__kernel_primitive:NN \looseness
442  \__kernel_primitive:NN \lower
443  \__kernel_primitive:NN \lowercase
444  \__kernel_primitive:NN \mag
445  \__kernel_primitive:NN \mark
446  \__kernel_primitive:NN \mathaccent
447  \__kernel_primitive:NN \mathbin
448  \__kernel_primitive:NN \mathchar
449  \__kernel_primitive:NN \mathchardef
450  \__kernel_primitive:NN \mathchoice
451  \__kernel_primitive:NN \mathclose
452  \__kernel_primitive:NN \mathcode
453  \__kernel_primitive:NN \mathinner
454  \__kernel_primitive:NN \mathop
455  \__kernel_primitive:NN \mathopen
456  \__kernel_primitive:NN \mathord
457  \__kernel_primitive:NN \mathpunct
458  \__kernel_primitive:NN \mathrel
459  \__kernel_primitive:NN \mathsurround
460  \__kernel_primitive:NN \maxdeadcycles
461  \__kernel_primitive:NN \maxdepth
462  \__kernel_primitive:NN \meaning
463  \__kernel_primitive:NN \medmuskip
464  \__kernel_primitive:NN \message
465  \__kernel_primitive:NN \mkern
466  \__kernel_primitive:NN \month
467  \__kernel_primitive:NN \moveleft
                                         \tex_ignorespaces:D
                                         \tex_immediate:D
                                         \tex_indent:D
                                         \tex_input:D
                                         \tex_inputlineno:D
                                         \tex_insert:D
                                         \tex_insertpenalties:D
                                         \tex_interlinepenalty:D
                                         \tex_jobname:D
                                         \tex_kern:D
                                         \tex_language:D
                                         \tex_lastbox:D
                                         \tex_lastkern:D
                                         \tex_lastpenalty:D
                                         \tex_lastskip:D
                                         \tex_lccode:D
                                         \tex_leaders:D
                                         \tex_left:D
                                         \tex_lefthyphenmin:D
                                         \tex_leftskip:D
                                         \tex_leqno:D
                                         \tex_let:D
                                         \tex_limits:D
                                         \tex_linepenalty:D
                                         \tex_lineskip:D
                                         \tex_lineskiplimit:D
                                         \tex_long:D
                                         \tex_looseness:D
                                         \tex_lower:D
                                         \tex_lowercase:D
                                         \tex_mag:D
                                         \tex_mark:D
                                         \tex_mathaccent:D
                                         \tex_mathbin:D
                                         \tex_mathchar:D
                                         \tex_mathchardef:D
                                         \tex_mathchoice:D
                                         \tex_mathclose:D
                                         \tex_mathcode:D
                                         \tex_mathinner:D
                                         \tex_mathop:D
                                         \tex_mathopen:D
                                         \tex_mathord:D
                                         \tex_mathpunct:D
                                         \tex_mathrel:D
                                         \tex_mathsurround:D
                                         \tex_maxdeadcycles:D
                                         \tex_maxdepth:D
                                         \tex_meaning:D
                                         \tex_medmuskip:D
                                         \tex_message:D
                                         \tex_mkern:D
                                         \tex_month:D
                                         \tex_moveleft:D

```

```

468  \__kernel_primitive:NN \moveright
469  \__kernel_primitive:NN \mskip
470  \__kernel_primitive:NN \multiply
471  \__kernel_primitive:NN \muskip
472  \__kernel_primitive:NN \muskipdef
473  \__kernel_primitive:NN \newlinechar
474  \__kernel_primitive:NN \noalign
475  \__kernel_primitive:NN \noboundary
476  \__kernel_primitive:NN \noexpand
477  \__kernel_primitive:NN \noindent
478  \__kernel_primitive:NN \nolimits
479  \__kernel_primitive:NN \nonscript
480  \__kernel_primitive:NN \nonstopmode
481  \__kernel_primitive:NN \nulldelimiterspace
482  \__kernel_primitive:NN \nullfont
483  \__kernel_primitive:NN \number
484  \__kernel_primitive:NN \omit
485  \__kernel_primitive:NN \openin
486  \__kernel_primitive:NN \openout
487  \__kernel_primitive:NN \or
488  \__kernel_primitive:NN \outer
489  \__kernel_primitive:NN \output
490  \__kernel_primitive:NN \outputpenalty
491  \__kernel_primitive:NN \over
492  \__kernel_primitive:NN \overfullrule
493  \__kernel_primitive:NN \overline
494  \__kernel_primitive:NN \overwithdelims
495  \__kernel_primitive:NN \pagedepth
496  \__kernel_primitive:NN \pagefilllstretch
497  \__kernel_primitive:NN \pagefillstretch
498  \__kernel_primitive:NN \pagefilstretch
499  \__kernel_primitive:NN \pagegoal
500  \__kernel_primitive:NN \pageshrink
501  \__kernel_primitive:NN \pagestretch
502  \__kernel_primitive:NN \pagetotal
503  \__kernel_primitive:NN \par
504  \__kernel_primitive:NN \parfillskip
505  \__kernel_primitive:NN \parindent
506  \__kernel_primitive:NN \parshape
507  \__kernel_primitive:NN \parskip
508  \__kernel_primitive:NN \patterns
509  \__kernel_primitive:NN \pausing
510  \__kernel_primitive:NN \penalty
511  \__kernel_primitive:NN \postdisplaypenalty
512  \__kernel_primitive:NN \predisplaypenalty
513  \__kernel_primitive:NN \predisplaysize
514  \__kernel_primitive:NN \pretolerance
515  \__kernel_primitive:NN \prevdepth
516  \__kernel_primitive:NN \prevgraf
517  \__kernel_primitive:NN \radical
518  \__kernel_primitive:NN \raise
519  \__kernel_primitive:NN \read
520  \__kernel_primitive:NN \relax
521  \__kernel_primitive:NN \relpenalty

```

\tex_moveright:D
\text_msip:D
\text_multiply:D
\text_muskip:D
\text_muskipdef:D
\text_newlinechar:D
\text_noalign:D
\text_noboundary:D
\text_noexpand:D
\text_noindent:D
\text_nolimits:D
\text_nonscript:D
\text_nonstopmode:D
\text_nulldelimiterspace:D
\text_nullfont:D
\text_number:D
\text_omit:D
\text_openin:D
\text_openout:D
\text_or:D
\text_outer:D
\text_output:D
\text_outputpenalty:D
\text_over:D
\text_overfullrule:D
\text_overline:D
\text_overwithdelims:D
\text_pagedepth:D
\text_pagefilllstretch:D
\text_pagefillstretch:D
\text_pagefilstretch:D
\text_pagegoal:D
\text_pageshrink:D
\text_pagestretch:D
\text_pagetotal:D
\text_par:D
\text_parfillskip:D
\text_parindent:D
\text_parshape:D
\text_parskip:D
\text_patterns:D
\text_pausing:D
\text_penalty:D
\text_postdisplaypenalty:D
\text_predisplaypenalty:D
\text_predisplaysize:D
\text_pretolerance:D
\text_prevdepth:D
\text_prevgraf:D
\text_radical:D
\text_raise:D
\text_read:D
\text_relax:D
\text_relpolicy:D

```

522  \__kernel_primitive:NN \right
523  \__kernel_primitive:NN \righthyphenmin
524  \__kernel_primitive:NN \rightskip
525  \__kernel_primitive:NN \romannumeral
526  \__kernel_primitive:NN \scriptfont
527  \__kernel_primitive:NN \scriptscriptfont
528  \__kernel_primitive:NN \scriptscriptstyle
529  \__kernel_primitive:NN \scriptspace
530  \__kernel_primitive:NN \scriptstyle
531  \__kernel_primitive:NN \scrollmode
532  \__kernel_primitive:NN \setbox
533  \__kernel_primitive:NN \setlanguage
534  \__kernel_primitive:NN \sfcode
535  \__kernel_primitive:NN \shipout
536  \__kernel_primitive:NN \show
537  \__kernel_primitive:NN \showbox
538  \__kernel_primitive:NN \showboxbreadth
539  \__kernel_primitive:NN \showboxdepth
540  \__kernel_primitive:NN \showlists
541  \__kernel_primitive:NN \showthe
542  \__kernel_primitive:NN \skewchar
543  \__kernel_primitive:NN \skip
544  \__kernel_primitive:NN \skipdef
545  \__kernel_primitive:NN \spacefactor
546  \__kernel_primitive:NN \spaceskip
547  \__kernel_primitive:NN \span
548  \__kernel_primitive:NN \special
549  \__kernel_primitive:NN \splitbotmark
550  \__kernel_primitive:NN \splitfirstmark
551  \__kernel_primitive:NN \splitmaxdepth
552  \__kernel_primitive:NN \splittopskip
553  \__kernel_primitive:NN \string
554  \__kernel_primitive:NN \tabskip
555  \__kernel_primitive:NN \textfont
556  \__kernel_primitive:NN \textstyle
557  \__kernel_primitive:NN \the
558  \__kernel_primitive:NN \thickmuskip
559  \__kernel_primitive:NN \thinmuskip
560  \__kernel_primitive:NN \time
561  \__kernel_primitive:NN \toks
562  \__kernel_primitive:NN \toksdef
563  \__kernel_primitive:NN \tolerance
564  \__kernel_primitive:NN \topmark
565  \__kernel_primitive:NN \topskip
566  \__kernel_primitive:NN \tracingcommands
567  \__kernel_primitive:NN \tracinglostchars
568  \__kernel_primitive:NN \tracingmacros
569  \__kernel_primitive:NN \tracingonline
570  \__kernel_primitive:NN \tracingoutput
571  \__kernel_primitive:NN \tracingpages
572  \__kernel_primitive:NN \tracingparagraphs
573  \__kernel_primitive:NN \tracingrestores
574  \__kernel_primitive:NN \tracingstats
575  \__kernel_primitive:NN \uccode

\text_right:D
\text_righthyphenmin:D
\text_rightskip:D
\text_roman numeral:D
\text_scriptfont:D
\text_scriptscriptfont:D
\text_scriptscriptstyle:D
\text_scriptspace:D
\text_scriptstyle:D
\text_scrollmode:D
\text_setbox:D
\text_setlanguage:D
\text_sfcode:D
\text_shipout:D
\text_show:D
\text_showbox:D
\text_showboxbreadth:D
\text_showboxdepth:D
\text_showlists:D
\text_showthe:D
\text_skewchar:D
\text_skip:D
\text_skipdef:D
\text_spacefactor:D
\text_spaceskip:D
\text_span:D
\text_special:D
\text_splitbotmark:D
\text_splitfirstmark:D
\text_splitmaxdepth:D
\text_splittopskip:D
\text_string:D
\text_tabskip:D
\text_textfont:D
\text_textstyle:D
\text_the:D
\text_thickmuskip:D
\text_thinmuskip:D
\text_time:D
\text_toks:D
\text_toksdef:D
\text_tolerance:D
\text_topmark:D
\text_topskip:D
\text_tracingcommands:D
\text_tracinglostchars:D
\text_tracingmacros:D
\text_tracingonline:D
\text_tracingoutput:D
\text_tracingpages:D
\text_tracingparagraphs:D
\text_tracingrestores:D
\text_tracingstats:D
\text_uccode:D

```

```

576  \__kernel_primitive:NN \uchyph          \tex_uchyph:D
577  \__kernel_primitive:NN \underline        \tex_underline:D
578  \__kernel_primitive:NN \unhbox           \tex_unhbox:D
579  \__kernel_primitive:NN \unhcopy         \tex_unhcopy:D
580  \__kernel_primitive:NN \unkern          \tex_unkern:D
581  \__kernel_primitive:NN \unpenalty       \tex_unpenalty:D
582  \__kernel_primitive:NN \unskip           \tex_unskip:D
583  \__kernel_primitive:NN \unvbox           \tex_unvbox:D
584  \__kernel_primitive:NN \unvcopy         \tex_unvcopy:D
585  \__kernel_primitive:NN \uppercase        \tex_uppercase:D
586  \__kernel_primitive:NN \vadjust          \tex_vadjust:D
587  \__kernel_primitive:NN \valign           \tex_valign:D
588  \__kernel_primitive:NN \vbadness         \tex_vbadness:D
589  \__kernel_primitive:NN \vbox              \tex_vbox:D
590  \__kernel_primitive:NN \vcenter          \tex_vcenter:D
591  \__kernel_primitive:NN \vfil              \tex_vfil:D
592  \__kernel_primitive:NN \vfill             \tex_vfill:D
593  \__kernel_primitive:NN \vfilneg          \tex_vfilneg:D
594  \__kernel_primitive:NN \vfuzz             \tex_vfuzz:D
595  \__kernel_primitive:NN \voffset          \tex_voffset:D
596  \__kernel_primitive:NN \vrule             \tex_vrule:D
597  \__kernel_primitive:NN \vsize             \tex_vsize:D
598  \__kernel_primitive:NN \vskip              \tex_vskip:D
599  \__kernel_primitive:NN \vsplit            \tex_vsplits:D
600  \__kernel_primitive:NN \vss               \tex_vss:D
601  \__kernel_primitive:NN \vtop              \tex_vtop:D
602  \__kernel_primitive:NN \wd                \tex_wd:D
603  \__kernel_primitive:NN \widowpenalty     \tex_widowpenalty:D
604  \__kernel_primitive:NN \write             \tex_write:D
605  \__kernel_primitive:NN \xdef              \tex_xdef:D
606  \__kernel_primitive:NN \xleaders          \tex_xleaders:D
607  \__kernel_primitive:NN \xspaceskip       \tex_xspaceskip:D
608  \__kernel_primitive:NN \year              \tex_year:D

```

Since L^AT_EX3 requires at least the ε-T_EX extensions, we also rename the additional primitives. These are all given the prefix `\etex_`.

```

609  \__kernel_primitive:NN \beginL          \etex_beginL:D
610  \__kernel_primitive:NN \beginR          \etex_beginR:D
611  \__kernel_primitive:NN \botmarks        \etex_botmarks:D
612  \__kernel_primitive:NN \clubpenalties   \etex_clubpenalties:D
613  \__kernel_primitive:NN \currentgrouplevel \etex_currentgrouplevel:D
614  \__kernel_primitive:NN \currentgroupype  \etex_currentgroupype:D
615  \__kernel_primitive:NN \currentifbranch  \etex_currentifbranch:D
616  \__kernel_primitive:NN \currentiflevel   \etex_currentiflevel:D
617  \__kernel_primitive:NN \currentiftype    \etex_currentiftype:D
618  \__kernel_primitive:NN \detokenize       \etex_detokenize:D
619  \__kernel_primitive:NN \dimexpr          \etex_dimexpr:D
620  \__kernel_primitive:NN \displaywidowpenalties \etex_displaywidowpenalties:D
621  \__kernel_primitive:NN \endL            \etex_endL:D
622  \__kernel_primitive:NN \endR            \etex_endR:D
623  \__kernel_primitive:NN \eTeXrevision    \etex_eTeXrevision:D
624  \__kernel_primitive:NN \eTeXversion      \etex_eTeXversion:D
625  \__kernel_primitive:NN \everyeof         \etex_everyeof:D
626  \__kernel_primitive:NN \firstmarks       \etex_firstmarks:D

```

```

627  \__kernel_primitive:NN \fontchardp          \etex_fontchardp:D
628  \__kernel_primitive:NN \fontcharht         \etex_fontcharht:D
629  \__kernel_primitive:NN \fontcharic          \etex_fontcharic:D
630  \__kernel_primitive:NN \fontcharwd          \etex_fontcharwd:D
631  \__kernel_primitive:NN \glueexpr           \etex_glueexpr:D
632  \__kernel_primitive:NN \glueshrink         \etex_glueshrink:D
633  \__kernel_primitive:NN \glueshrinkorder    \etex_glueshrinkorder:D
634  \__kernel_primitive:NN \gluestretch         \etex_gluestretch:D
635  \__kernel_primitive:NN \gluestretchorder   \etex_gluestretchorder:D
636  \__kernel_primitive:NN \gluetomu           \etex_gluetomu:D
637  \__kernel_primitive:NN \ifcsname          \etex_ifcsname:D
638  \__kernel_primitive:NN \ifdefined          \etex_ifdefined:D
639  \__kernel_primitive:NN \iffontchar         \etex_iffontchar:D
640  \__kernel_primitive:NN \interactionmode    \etex_interactionmode:D
641  \__kernel_primitive:NN \interlinepenalties \etex_interlinepenalties:D
642  \__kernel_primitive:NN \lastlinefit        \etex_lastlinefit:D
643  \__kernel_primitive:NN \lastnodetype       \etex_lastnodetype:D
644  \__kernel_primitive:NN \marks              \etex_marks:D
645  \__kernel_primitive:NN \middle             \etex_middle:D
646  \__kernel_primitive:NN \muexpr             \etex_muexpr:D
647  \__kernel_primitive:NN \mutoglu            \etex_mutoglu:D
648  \__kernel_primitive:NN \numexpr            \etex_numexpr:D
649  \__kernel_primitive:NN \pagediscards      \etex_pagediscards:D
650  \__kernel_primitive:NN \parshapedimen     \etex_parshapedimen:D
651  \__kernel_primitive:NN \parshapeindent     \etex_parshapeindent:D
652  \__kernel_primitive:NN \parshape length   \etex_parshape length:D
653  \__kernel_primitive:NN \predisplaydirection \etex_predisplaydirection:D
654  \__kernel_primitive:NN \protected          \etex_protected:D
655  \__kernel_primitive:NN \readline           \etex_readline:D
656  \__kernel_primitive:NN \savinghyphcodes   \etex_savinghyphcodes:D
657  \__kernel_primitive:NN \savingvdiscards   \etex_savingvdiscards:D
658  \__kernel_primitive:NN \scantokens         \etex_scantokens:D
659  \__kernel_primitive:NN \showgroups        \etex_showgroups:D
660  \__kernel_primitive:NN \showifs            \etex_showifs:D
661  \__kernel_primitive:NN \showtokens        \etex_showtokens:D
662  \__kernel_primitive:NN \splitbotmarks     \etex_splitbotmarks:D
663  \__kernel_primitive:NN \spliddiscards     \etex_spliddiscards:D
664  \__kernel_primitive:NN \splitfirstmarks  \etex_splitfirstmarks:D
665  \__kernel_primitive:NN \TeXeTstate        \etex_TeXXeTstate:D
666  \__kernel_primitive:NN \topmarks           \etex_topmarks:D
667  \__kernel_primitive:NN \tracingassigns   \etex_tracingassigns:D
668  \__kernel_primitive:NN \tracinggroups    \etex_tracinggroups:D
669  \__kernel_primitive:NN \tracingifs        \etex_tracingifs:D
670  \__kernel_primitive:NN \tracingnesting   \etex_tracingnesting:D
671  \__kernel_primitive:NN \tracingscantokens \etex_tracingscantokens:D
672  \__kernel_primitive:NN \unexpanded        \etex_unexpanded:D
673  \__kernel_primitive:NN \unless             \etex_unless:D
674  \__kernel_primitive:NN \widowpenalties    \etex_widowpenalties:D

```

The newer primitives are more complex: there are an awful lot of them, and we don't use them all at the moment. So the following is selective, based on those also available in `LuaTeX` or used in `expl3`. In the case of the `pdfTeX` primitives, we retain `pdf` at the start of the names *only* for directly PDF-related primitives, as there are a lot of `pdfTeX` primitives that start `\pdf...` but are not related to PDF output. These ones related to

PDF output or only work in PDF mode.

```

728  \__kernel_primitive:NN \pdfstartlink
729  \__kernel_primitive:NN \pdfstartthread
730  \__kernel_primitive:NN \pdfsuppressptexinfo:D
731  \__kernel_primitive:NN \pdfthread:D
732  \__kernel_primitive:NN \pdfthreadmargin:D
733  \__kernel_primitive:NN \pdftrailer:D
734  \__kernel_primitive:NN \pdfuniqueresname:D
735  \__kernel_primitive:NN \pdfvorigin:D
736  \__kernel_primitive:NN \pdfxform:D
737  \__kernel_primitive:NN \pdfxformattr:D
738  \__kernel_primitive:NN \pdfxformname:D
739  \__kernel_primitive:NN \pdfxformresources:D
740  \__kernel_primitive:NN \pdfximage:D
741  \__kernel_primitive:NN \pdfximagebbox:D

```

While these are not.

```

742  \__kernel_primitive:NN \ifpdfabsdim:D
743  \__kernel_primitive:NN \ifpdfabsnum:D
744  \__kernel_primitive:NN \ifpdfprimitive:D
745  \__kernel_primitive:NN \pdfadjustspacing:D
746  \__kernel_primitive:NN \pdfcopyfont:D
747  \__kernel_primitive:NN \pdfdraftmode:D
748  \__kernel_primitive:NN \pdfeachlinedepth:D
749  \__kernel_primitive:NN \pdfeachlineheight:D
750  \__kernel_primitive:NN \pdffirstlineheight:D
751  \__kernel_primitive:NN \pdffontexpand:D
752  \__kernel_primitive:NN \pdffontsize:D
753  \__kernel_primitive:NN \pdfignoreddimen:D
754  \__kernel_primitive:NN \pdfinsertht:D
755  \__kernel_primitive:NN \pdflastlinedepth:D
756  \__kernel_primitive:NN \pdflastxpos:D
757  \__kernel_primitive:NN \pdflastypos:D
758  \__kernel_primitive:NN \pdfmapfile:D
759  \__kernel_primitive:NN \pdfmapline:D
760  \__kernel_primitive:NN \pdfnoligatures:D
761  \__kernel_primitive:NN \pdfnormaldeviate:D
762  \__kernel_primitive:NN \pdfpageheight:D
763  \__kernel_primitive:NN \pdfpagewidth:D
764  \__kernel_primitive:NN \pdfpkmode:D
765  \__kernel_primitive:NN \pdfpkresolution:D
766  \__kernel_primitive:NN \pdfprimitive:D
767  \__kernel_primitive:NN \pdfprotrudechars:D
768  \__kernel_primitive:NN \pdfpxdimen:D
769  \__kernel_primitive:NN \pdfrandomseed:D
770  \__kernel_primitive:NN \pdfsavepos:D
771  \__kernel_primitive:NN \pdfstrcmp:D
772  \__kernel_primitive:NN \pdfsetrandomseed:D
773  \__kernel_primitive:NN \pdfshellescape:D
774  \__kernel_primitive:NN \pdftracingfonts:D
775  \__kernel_primitive:NN \pdfuniformdeviate:D

```

The version primitives are not related to PDF mode but are related to pdfTEX so retain the full prefix.

```

776  \__kernel_primitive:NN \pdftexbanner:D
777  \__kernel_primitive:NN \pdftexrevision:D

```

```
778  \__kernel_primitive:NN \pdftexversion          \pdftex_pdftexversion:D
```

These ones appear in pdfTeX but don't have pdf in the name at all. (\synctex is odd as it's really not from pdfTeX but from SyncTeX!)

779 __kernel_primitive:NN \efcode	\pdftex_efcode:D
780 __kernel_primitive:NN \ifinccsname	\pdftex_ifinccsname:D
781 __kernel_primitive:NN \leftmarginkern	\pdftex_leftmarginkern:D
782 __kernel_primitive:NN \letterspacefont	\pdftex_letterspacefont:D
783 __kernel_primitive:NN \lpcode	\pdftex_lpcode:D
784 __kernel_primitive:NN \quitvmode	\pdftex_quitvmode:D
785 __kernel_primitive:NN \rightmarginkern	\pdftex_rightmarginkern:D
786 __kernel_primitive:NN \rancode	\pdftex_rancode:D
787 __kernel_primitive:NN \synctex	\pdftex_synctex:D
788 __kernel_primitive:NN \tagcode	\pdftex_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.

```
789 </initex | names | package>
790 <*initex | package>
791   \tex_long:D \tex_def:D \use_ii:nn #1#2 {#2}
792   \tex_long:D \tex_def:D \use_none:n #1 { }
793   \tex_long:D \tex_def:D \__kernel_primitive:NN #1#2
794   {
795     \etex_ifdefined:D #1
796       \tex_expandafter:D \use_ii:nn
797     \tex_fi:D
798       \use_none:n { \tex_global:D \tex_let:D #2 #1 }
799   <*initex>
800     \tex_global:D \tex_let:D #1 \tex_undefined:D
801   </initex>
802   }
803 </initex | package>
804 <*initex | names | package>
```

XeTeX-specific primitives. Note that XeTeX's \strcmp is handled earlier and is “rolled up” into \pdfstrcmp. With the exception of the version primitives these don't carry XeTeX through into the “base” name. A few cross-compatibility names which lack the pdf of the original are handled later.

805 __kernel_primitive:NN \suppressfontnotfounderror	\xetex_suppressfontnotfounderror:D
806 __kernel_primitive:NN \XeTeXcharclass	\xetex_charclass:D
807 __kernel_primitive:NN \XeTeXcharglyph	\xetex_charglyph:D
808 __kernel_primitive:NN \XeTeXcountfeatures	\xetex_countfeatures:D
809 __kernel_primitive:NN \XeTeXcountglyphs	\xetex_countglyphs:D
810 __kernel_primitive:NN \XeTeXcountselectors	\xetex_countselectors:D
811 __kernel_primitive:NN \XeTeXcountvariations	\xetex_countvariations:D
812 __kernel_primitive:NN \XeTeXdefaultencoding	\xetex_defaultencoding:D
813 __kernel_primitive:NN \XeTeXdashbreakstate	\xetex_dashbreakstate:D
814 __kernel_primitive:NN \XeTeXfeaturecode	\xetex_featurecode:D
815 __kernel_primitive:NN \XeTeXfeaturename	\xetex_featurename:D
816 __kernel_primitive:NN \XeTeXfindfeaturebyname	\xetex_findfeaturebyname:D
817 __kernel_primitive:NN \XeTeXfindselectorbyname	\xetex_findselectorbyname:D
818 __kernel_primitive:NN \XeTeXfindvariationbyname	\xetex_findvariationbyname:D
819 __kernel_primitive:NN \XeTeXfirstfontchar	\xetex_firstfontchar:D

```

820  \__kernel_primitive:NN \XeTeXfonttype
821  \__kernel_primitive:NN \XeTeXgenerateactualtext
822  \__kernel_primitive:NN \XeTeXglyph
823  \__kernel_primitive:NN \XeTeXglyphbounds
824  \__kernel_primitive:NN \XeTeXglyphindex
825  \__kernel_primitive:NN \XeTeXglyphname
826  \__kernel_primitive:NN \XeTeXinputencoding
827  \__kernel_primitive:NN \XeTeXinputnormalization
828  \__kernel_primitive:NN \XeTeXinterchartokenstate
829  \__kernel_primitive:NN \XeTeXinterchartoks
830  \__kernel_primitive:NN \XeTeXisdefaultselector
831  \__kernel_primitive:NN \XeTeXisexclusivefeature
832  \__kernel_primitive:NN \XeTeXlastfontchar
833  \__kernel_primitive:NN \XeTeXlinebreakskip
834  \__kernel_primitive:NN \XeTeXlinebreaklocale
835  \__kernel_primitive:NN \XeTeXlinebreakpenalty
836  \__kernel_primitive:NN \XeTeXOTcountfeatures
837  \__kernel_primitive:NN \XeTeXOTcountlanguages
838  \__kernel_primitive:NN \XeTeXOTcountsheets
839  \__kernel_primitive:NN \XeTeXOTfeaturetag
840  \__kernel_primitive:NN \XeTeXOTlanguage tag
841  \__kernel_primitive:NN \XeTeXOTscripttag
842  \__kernel_primitive:NN \XeTeXpdffile
843  \__kernel_primitive:NN \XeTeXpdfpagecount
844  \__kernel_primitive:NN \XeTeXpicfile
845  \__kernel_primitive:NN \XeTeXselectorname
846  \__kernel_primitive:NN \XeTeXtracingfonts
847  \__kernel_primitive:NN \XeTeXupwardsmode
848  \__kernel_primitive:NN \XeTeXuseglyphmetrics
849  \__kernel_primitive:NN \XeTeXvariation
850  \__kernel_primitive:NN \XeTeXvariationdefault
851  \__kernel_primitive:NN \XeTeXvariationmax
852  \__kernel_primitive:NN \XeTeXvariationmin
853  \__kernel_primitive:NN \XeTeXvariationname
854  \__kernel_primitive:NN \XeTeXrevision
855  \__kernel_primitive:NN \XeTeXversion

```

The version primitives retain XeTeX.

```

854  \__kernel_primitive:NN \XeTeXrevision
855  \__kernel_primitive:NN \XeTeXversion

```

Primitives from pdfTeX that XeTeX renames: also helps with LuaTeX.

```

856  \__kernel_primitive:NN \ifprimitive
857  \__kernel_primitive:NN \primitive
858  \__kernel_primitive:NN \shellescape

```

Primitives from LuaTeX, some of which have been ported back to XeTeX. Notice that \expanded was intended for pdfTeX 1.50 but as that was not released we call this a LuaTeX primitive.

```

859  \__kernel_primitive:NN \alignmark
860  \__kernel_primitive:NN \aligntab
861  \__kernel_primitive:NN \attribute
862  \__kernel_primitive:NN \attributedef
863  \__kernel_primitive:NN \automatichyphenpenalty
864  \__kernel_primitive:NN \begincsname
865  \__kernel_primitive:NN \catcodetable
866  \__kernel_primitive:NN \clearmarks

```

```

\lualatex_alignmark:D
\lualatex_aligntab:D
\lualatex_attribute:D
\lualatex_attributedef:D
\lualatex_automatichyphenpenalty:D
\lualatex_begincsname:D
\lualatex_catcodetable:D
\lualatex_clearmarks:D

```

```

867  \__kernel_primitive:NN \crampeddisplaystyle           \luatex_crampeddisplaystyle:D
868  \__kernel_primitive:NN \crampedscriptscriptstyle    \luatex_crampedscriptscriptstyle:D
869  \__kernel_primitive:NN \crampedscriptstyle          \luatex_crampedscriptstyle:D
870  \__kernel_primitive:NN \crampedtextstyle            \luatex_crampedtextstyle:D
871  \__kernel_primitive:NN \directlua                 \luatex_directlua:D
872  \__kernel_primitive:NN \dviextension               \luatex_dviextension:D
873  \__kernel_primitive:NN \dvifeedback              \luatex_dvifeedback:D
874  \__kernel_primitive:NN \dvivariable               \luatex_dvivariable:D
875  \__kernel_primitive:NN \etoksapp                \luatex_etoksapp:D
876  \__kernel_primitive:NN \etokspre                \luatex_etokspre:D
877  \__kernel_primitive:NN \explicithyphenpenalty   \luatex_explicithyphenpenalty:D
878  \__kernel_primitive:NN \expanded                  \luatex_expanded:D
879  \__kernel_primitive:NN \firstvalidlanguage       \luatex_firstvalidlanguage:D
880  \__kernel_primitive:NN \fontid                   \luatex_fontid:D
881  \__kernel_primitive:NN \formatname              \luatex_formatname:D
882  \__kernel_primitive:NN \hjcode                  \luatex_hjcode:D
883  \__kernel_primitive:NN \hpack                   \luatex_hpack:D
884  \__kernel_primitive:NN \hyphenationbounds        \luatex_hyphenationbounds:D
885  \__kernel_primitive:NN \hyphenationmin          \luatex_hyphenationmin:D
886  \__kernel_primitive:NN \hyphenpenaltymode       \luatex_hyphenpenaltymode:D
887  \__kernel_primitive:NN \gleaders               \luatex_gleaders:D
888  \__kernel_primitive:NN \initcatcodetable       \luatex_initcatcodetable:D
889  \__kernel_primitive:NN \lastnamedcs            \luatex_lastnamedcs:D
890  \__kernel_primitive:NN \latelua                \luatex_latelua:D
891  \__kernel_primitive:NN \letcharcode             \luatex_letcharcode:D
892  \__kernel_primitive:NN \luaescapestring        \luatex_luaescapestring:D
893  \__kernel_primitive:NN \luafunction            \luatex_luafunction:D
894  \__kernel_primitive:NN \luatexbanner           \luatex_luatexbanner:D
895  \__kernel_primitive:NN \luatexdatatestamp      \luatex_luatexdatatestamp:D
896  \__kernel_primitive:NN \luatexrevision         \luatex_luatexrevision:D
897  \__kernel_primitive:NN \luatexversion          \luatex_luatexversion:D
898  \__kernel_primitive:NN \mathdisplayskipmode    \luatex_mathdisplayskipmode:D
899  \__kernel_primitive:NN \matheqnogapstep        \luatex_matheqnogapstep:D
900  \__kernel_primitive:NN \mathnolimitsmode        \luatex_mathnolimitsmode:D
901  \__kernel_primitive:NN \mathoption              \luatex_mathoption:D
902  \__kernel_primitive:NN \mathrulesfam           \luatex_mathrulesfam:D
903  \__kernel_primitive:NN \mathscriptsmode        \luatex_mathscriptsmode:D
904  \__kernel_primitive:NN \mathstyle               \luatex_mathstyle:D
905  \__kernel_primitive:NN \mathsurroundmode       \luatex_mathsurroundmode:D
906  \__kernel_primitive:NN \mathsurroundskip       \luatex_mathsurroundskip:D
907  \__kernel_primitive:NN \nohrule                \luatex_noahrule:D
908  \__kernel_primitive:NN \nokerns                \luatex_nokerns:D
909  \__kernel_primitive:NN \noligs                 \luatex_noligs:D
910  \__kernel_primitive:NN \nospaces              \luatex_nospaces:D
911  \__kernel_primitive:NN \novrule                \luatex_novrule:D
912  \__kernel_primitive:NN \outputbox              \luatex_outputbox:D
913  \__kernel_primitive:NN \pagebottomoffset     \luatex_pagebottomoffset:D
914  \__kernel_primitive:NN \pageleftoffset       \luatex_pageleftoffset:D
915  \__kernel_primitive:NN \pagerightoffset     \luatex_pagerightoffset:D
916  \__kernel_primitive:NN \pagetopoffset        \luatex_pagetopoffset:D
917  \__kernel_primitive:NN \pdfextension          \luatex_pdfextension:D
918  \__kernel_primitive:NN \pdffeedback          \luatex_pdffeedback:D
919  \__kernel_primitive:NN \pdfvariable          \luatex_pdfvariable:D
920  \__kernel_primitive:NN \postexhyphenchar     \luatex_postexhyphenchar:D

```

```

921  \__kernel_primitive:NN \posthyphenchar          \luatek_posthyphenchar:D
922  \__kernel_primitive:NN \predisplaygapfactor   \luatek_predisplaygapfactor:D
923  \__kernel_primitive:NN \preexhyphenchar        \luatek_preexhyphenchar:D
924  \__kernel_primitive:NN \prehyphenchar          \luatek_prehyphenchar:D
925  \__kernel_primitive:NN \savecatcodetable      \luatek_savecatcodetable:D
926  \__kernel_primitive:NN \scantexttokens        \luatek_scantexttokens:D
927  \__kernel_primitive:NN \setfontid              \luatek_setfontid:D
928  \__kernel_primitive:NN \shapemode              \luatek_shapemode:D
929  \__kernel_primitive:NN \suppressifcsnameerror \luatek_suppressifcsnameerror:D
930  \__kernel_primitive:NN \suppresslongerror     \luatek_suppresslongerror:D
931  \__kernel_primitive:NN \suppressmathparerror  \luatek_suppressmathparerror:D
932  \__kernel_primitive:NN \suppressoutererror    \luatek_suppressoutererror:D
933  \__kernel_primitive:NN \toksapp               \luatek_toksapp:D
934  \__kernel_primitive:NN \tokspre               \luatek_tokspre:D
935  \__kernel_primitive:NN \tpack                 \luatek_tpck:D
936  \__kernel_primitive:NN \vpack                 \luatek_vpack:D

```

Slightly more awkward are the directional primitives in LuaTeX. These come from Omega/Aleph, but we do not support those engines and so it seems most sensible to treat them as LuaTeX primitives for prefix purposes. One here is “new” but fits into the general set.

```

937  \__kernel_primitive:NN \bodydir               \luatek_bodydir:D
938  \__kernel_primitive:NN \boxdir                \luatek_boxdir:D
939  \__kernel_primitive:NN \leftghost            \luatek_leftghost:D
940  \__kernel_primitive:NN \linedir               \luatek_linedir:D
941  \__kernel_primitive:NN \localbrokenpenalty  \luatek_localbrokenpenalty:D
942  \__kernel_primitive:NN \localinterlinepenalty \luatek_localinterlinepenalty:D
943  \__kernel_primitive:NN \localleftbox         \luatek_localleftbox:D
944  \__kernel_primitive:NN \localrightbox        \luatek_localrightbox:D
945  \__kernel_primitive:NN \mathdir               \luatek_mathdir:D
946  \__kernel_primitive:NN \pagedir              \luatek_pagedir:D
947  \__kernel_primitive:NN \pardir               \luatek_pardir:D
948  \__kernel_primitive:NN \rightghost           \luatek_rightghost:D
949  \__kernel_primitive:NN \textdir              \luatek_textdir:D

```

Primitives from pdfTeX that LuaTeX renames.

```

950  \__kernel_primitive:NN \adjustspacing        \pdftex_adjustspacing:D
951  \__kernel_primitive:NN \copyfont             \pdftex_copyfont:D
952  \__kernel_primitive:NN \draftmode            \pdftex_draftmode:D
953  \__kernel_primitive:NN \expandglyphsinfont \pdftex_fontexpand:D
954  \__kernel_primitive:NN \ifabsdim             \pdftex_ifabsdim:D
955  \__kernel_primitive:NN \ifabsnum             \pdftex_ifabsnum:D
956  \__kernel_primitive:NN \ignoreregularitiesinfont \pdftex_ignoreregularitiesinfont:D
957  \__kernel_primitive:NN \insertrht            \pdftex_insertrht:D
958  \__kernel_primitive:NN \lastsavedboxresourceindex \pdftex_pdflastxform:D
959  \__kernel_primitive:NN \lastsavedimageresourceindex \pdftex_pdflastximage:D
960  \__kernel_primitive:NN \lastsavedimageresourcepages \pdftex_pdflastximagepages:D
961  \__kernel_primitive:NN \lastxpos              \pdftex_lastxpos:D
962  \__kernel_primitive:NN \lastypos              \pdftex_lastypos:D
963  \__kernel_primitive:NN \normaldeviate        \pdftex_normaldeviate:D
964  \__kernel_primitive:NN \outputmode            \pdftex_pdfoutput:D
965  \__kernel_primitive:NN \pageheight            \pdftex_pageheight:D
966  \__kernel_primitive:NN \pagewidth             \pdftex_pagewidth:D
967  \__kernel_primitive:NN \protrudechars        \pdftex_protrudechars:D
968  \__kernel_primitive:NN \pxdimen              \pdftex_pxdimen:D

```

```

969  \__kernel_primitive:NN \randomseed          \pdftex_randomseed:D
970  \__kernel_primitive:NN \useboxresource     \pdftex_pdfrefxform:D
971  \__kernel_primitive:NN \useimageresource   \pdftex_pdfrefximage:D
972  \__kernel_primitive:NN \savepos             \pdftex_savepos:D
973  \__kernel_primitive:NN \saveboxresource    \pdftex_pdxform:D
974  \__kernel_primitive:NN \saveimageresource \pdftex_pdximage:D
975  \__kernel_primitive:NN \setrandomseed      \pdftex_setrandomseed:D
976  \__kernel_primitive:NN \tracingfonts       \pdftex_tracingfonts:D
977  \__kernel_primitive:NN \uniformdeviate     \pdftex_uniformdeviate:D

```

The set of Unicode math primitives were introduced by Xe^{tex} and Lua^{tex} in a somewhat complex fashion: a few first as \XeTeX... which were then renamed with Lua^{tex} having a lot more. These names now all start \U... and mainly \Umath.... To keep things somewhat clear we therefore prefix all of these as \utex... (introduced by a Unicode ^{tex} engine) and drop \U(math) from the names. Where there is a related ^{tex}90 primitive or where it really seems required we keep the math part of the name.

```

978  \__kernel_primitive:NN \Uchar              \utex_char:D
979  \__kernel_primitive:NN \Ucharcat           \utex_charcat:D
980  \__kernel_primitive:NN \Udelcode           \utex_delcode:D
981  \__kernel_primitive:NN \Udelcodenum        \utex_delcodenum:D
982  \__kernel_primitive:NN \Udelimiter         \utex_delimiter:D
983  \__kernel_primitive:NN \Udelimiterover     \utex_delimiterover:D
984  \__kernel_primitive:NN \Udelimiterunder    \utex_delimiterunder:D
985  \__kernel_primitive:NN \Uhextensible       \utex_hextensible:D
986  \__kernel_primitive:NN \Umathaccent         \utex_mathaccent:D
987  \__kernel_primitive:NN \Umathaxis           \utex_mathaxis:D
988  \__kernel_primitive:NN \Umathbinbinspacing  \utex_binbinspacing:D
989  \__kernel_primitive:NN \Umathbinspacing     \utex_binclosespacing:D
990  \__kernel_primitive:NN \Umathbininnerspacing \utex_bininnerspacing:D
991  \__kernel_primitive:NN \Umathbinopenspacing \utex_binopenspacing:D
992  \__kernel_primitive:NN \Umathbinopspacing   \utex_binopspacing:D
993  \__kernel_primitive:NN \Umathbinordspacing  \utex_binordspacing:D
994  \__kernel_primitive:NN \Umathbinpunctspacing \utex_binpunctspacing:D
995  \__kernel_primitive:NN \Umathbinrelspacing  \utex_binrelspacing:D
996  \__kernel_primitive:NN \Umathchar            \utex_mathchar:D
997  \__kernel_primitive:NN \Umathcharclass       \utex_mathcharclass:D
998  \__kernel_primitive:NN \Umathchardef         \utex_mathchardef:D
999  \__kernel_primitive:NN \Umathcharfam         \utex_mathcharfam:D
1000 \__kernel_primitive:NN \Umathcharnum         \utex_mathcharnum:D
1001 \__kernel_primitive:NN \Umathcharnumdef      \utex_mathcharnumdef:D
1002 \__kernel_primitive:NN \Umathcharslot        \utex_mathcharslot:D
1003 \__kernel_primitive:NN \Umathclosebinspacing \utex_closebinspacing:D
1004 \__kernel_primitive:NN \Umathcloseclosespacing \utex_closeclosespacing:D
1005 \__kernel_primitive:NN \Umathcloseinnerspacing \utex_closeinnerspacing:D
1006 \__kernel_primitive:NN \Umathcloseopenspacing \utex_closeopenspacing:D
1007 \__kernel_primitive:NN \Umathcloseopspacing  \utex_closeopspacing:D
1008 \__kernel_primitive:NN \Umathcloseordspacing \utex_closeordspacing:D
1009 \__kernel_primitive:NN \Umathclosepunctspacing \utex_closepunctspacing:D
1010 \__kernel_primitive:NN \Umathcloserelspacing \utex_closerelspacing:D
1011 \__kernel_primitive:NN \Umathcode             \utex_mathcode:D
1012 \__kernel_primitive:NN \Umathcodenum         \utex_mathcodenum:D
1013 \__kernel_primitive:NN \Umathconnectoroverlapmin \utex_connectoroverlapmin:D
1014 \__kernel_primitive:NN \Umathfractiondelsize \utex_fractiondelsize:D
1015 \__kernel_primitive:NN \Umathfractiondenomdown \utex_fractiondenomdown:D

```

```

1016  \__kernel_primitive:NN \Umathfractiondenomvgap
1017  \__kernel_primitive:NN \Umathfractionnumup
1018  \__kernel_primitive:NN \Umathfractionnumvgap
1019  \__kernel_primitive:NN \Umathfractionrule
1020  \__kernel_primitive:NN \Umathinnerbinspacing
1021  \__kernel_primitive:NN \Umathinnerclosespacing
1022  \__kernel_primitive:NN \Umathinnerinnerspacing
1023  \__kernel_primitive:NN \Umathinneropenspacing
1024  \__kernel_primitive:NN \Umathinneropspacing
1025  \__kernel_primitive:NN \Umathinnerordspacing
1026  \__kernel_primitive:NN \Umathinnerpunctspacing
1027  \__kernel_primitive:NN \Umathinnerrelspacing
1028  \__kernel_primitive:NN \Umathlimitabovebgap
1029  \__kernel_primitive:NN \Umathlimitabovekern
1030  \__kernel_primitive:NN \Umathlimitabovevgap
1031  \__kernel_primitive:NN \Umathlimitbelowbgap
1032  \__kernel_primitive:NN \Umathlimitbelowkern
1033  \__kernel_primitive:NN \Umathlimitbelowvgap
1034  \__kernel_primitive:NN \Umathnolimitssubfactor
1035  \__kernel_primitive:NN \Umathnolimitsupfactor
1036  \__kernel_primitive:NN \Umathpbinspacing
1037  \__kernel_primitive:NN \Umathpclosespacing
1038  \__kernel_primitive:NN \Umathopenbinspacing
1039  \__kernel_primitive:NN \Umathopenclosespacing
1040  \__kernel_primitive:NN \Umathopeninnerspacing
1041  \__kernel_primitive:NN \Umathopenopenspacing
1042  \__kernel_primitive:NN \Umathopenopspacing
1043  \__kernel_primitive:NN \Umathopenordspacing
1044  \__kernel_primitive:NN \Umathopenpunctspacing
1045  \__kernel_primitive:NN \Umathopenrelspacing
1046  \__kernel_primitive:NN \Umathoperatorsize
1047  \__kernel_primitive:NN \Umathopinnerspacing
1048  \__kernel_primitive:NN \Umathopopenspacing
1049  \__kernel_primitive:NN \Umathopopspacing
1050  \__kernel_primitive:NN \Umathopordspacing
1051  \__kernel_primitive:NN \Umathoppunctspacing
1052  \__kernel_primitive:NN \Umathoprelspacing
1053  \__kernel_primitive:NN \Umathordbinspacing
1054  \__kernel_primitive:NN \Umathordclosespacing
1055  \__kernel_primitive:NN \Umathorddinerspacing
1056  \__kernel_primitive:NN \Umathordopenspacing
1057  \__kernel_primitive:NN \Umathordopsspacing
1058  \__kernel_primitive:NN \Umathordordspacing
1059  \__kernel_primitive:NN \Umathordpunctspacing
1060  \__kernel_primitive:NN \Umathordrelspacing
1061  \__kernel_primitive:NN \Umathoverbarkern
1062  \__kernel_primitive:NN \Umathoverbarrule
1063  \__kernel_primitive:NN \Umathoverbarvgap
1064  \__kernel_primitive:NN \Umathoverdelimiterbgap
1065  \__kernel_primitive:NN \Umathoverdelimitervgap
1066  \__kernel_primitive:NN \Umathpunctbinspacing
1067  \__kernel_primitive:NN \Umathpunctclosespacing
1068  \__kernel_primitive:NN \Umathpunctinnerspacing
1069  \__kernel_primitive:NN \Umathpunctopenspacing
                                         \utex_fractiondenomvgap:D
                                         \utex_fractionnumup:D
                                         \utex_fractionnumvgap:D
                                         \utex_fractionrule:D
                                         \utex_innerbinspacing:D
                                         \utex_innerclosespacing:D
                                         \utex_innerinnerspacing:D
                                         \utex_inneropenspacing:D
                                         \utex_inneropspacing:D
                                         \utex_innerordspacing:D
                                         \utex_innerpunctspacing:D
                                         \utex_innerrelspacing:D
                                         \utex_limitabovebgap:D
                                         \utex_limitabovekern:D
                                         \utex_limitabovevgap:D
                                         \utex_limitbelowbgap:D
                                         \utex_limitbelowkern:D
                                         \utex_limitbelowvgap:D
                                         \utex_nolimitsubfactor:D
                                         \utex_nolimitsupfactor:D
                                         \utex_opbinspacing:D
                                         \utex_ocpspacing:D
                                         \utex_openbinspacing:D
                                         \utex_openclosespacing:D
                                         \utex_openinnerspacing:D
                                         \utex_openopenspacing:D
                                         \utex_openopspacing:D
                                         \utex_openordspacing:D
                                         \utex_openpunctspacing:D
                                         \utex_openrelspacing:D
                                         \utex_operatorsize:D
                                         \utex_opinnerspacing:D
                                         \utex_opopenspacing:D
                                         \utex_opopspacing:D
                                         \utex_opordspacing:D
                                         \utex_oppunctspacing:D
                                         \utex_oprelspacing:D
                                         \utex_ordbinspacing:D
                                         \utex_ordclosespacing:D
                                         \utex_orddinerspacing:D
                                         \utex_ordopenspacing:D
                                         \utex_ordopsspacing:D
                                         \utex_ordordspacing:D
                                         \utex_ordinalpunctspacing:D
                                         \utex_ordinalrelspacing:D
                                         \utex_overbarkern:D
                                         \utex_overbarrule:D
                                         \utex_overbarvgap:D
                                         \utex_overdelimiterbgap:D
                                         \utex_overdelimitervgap:D
                                         \utex_punctbinspacing:D
                                         \utex_punctclosespacing:D
                                         \utex_punctinnerspacing:D
                                         \utex_punctopenspacing:D

```

```

1070  \__kernel_primitive:NN \Umathpunctopspacing
1071  \__kernel_primitive:NN \Umathpunctordspacing
1072  \__kernel_primitive:NN \Umathpunctpunctspacing
1073  \__kernel_primitive:NN \Umathpunctrelspacing
1074  \__kernel_primitive:NN \Umathquad
1075  \__kernel_primitive:NN \Umathradicaldegreeafter
1076  \__kernel_primitive:NN \Umathradicaldegreebefore
1077  \__kernel_primitive:NN \Umathradicaldegreeraise
1078  \__kernel_primitive:NN \Umathradicalkern
1079  \__kernel_primitive:NN \Umathradicalrule
1080  \__kernel_primitive:NN \Umathradicalvgap
1081  \__kernel_primitive:NN \Umathrelbinspacing
1082  \__kernel_primitive:NN \Umathrelclosespacing
1083  \__kernel_primitive:NN \Umathrelinnerspacing
1084  \__kernel_primitive:NN \Umathrelopspacing
1085  \__kernel_primitive:NN \Umathrellospacing
1086  \__kernel_primitive:NN \Umathrelordspacing
1087  \__kernel_primitive:NN \Umathrelpunctspacing
1088  \__kernel_primitive:NN \Umathrelrelspacing
1089  \__kernel_primitive:NN \Umathskewedfractionhgap
1090  \__kernel_primitive:NN \Umathskewedfractionvgap
1091  \__kernel_primitive:NN \Umathspaceafterscript
1092  \__kernel_primitive:NN \Umathstackdenomdown
1093  \__kernel_primitive:NN \Umathstacknumup
1094  \__kernel_primitive:NN \Umathstackvgap
1095  \__kernel_primitive:NN \Umathsubshiftdown
1096  \__kernel_primitive:NN \Umathsubshiftdrop
1097  \__kernel_primitive:NN \Umathsubsupshiftdown
1098  \__kernel_primitive:NN \Umathsubsupvgap
1099  \__kernel_primitive:NN \Umathsubtopmax
1100  \__kernel_primitive:NN \Umathsupbottommin
1101  \__kernel_primitive:NN \Umathsupshiftdrop
1102  \__kernel_primitive:NN \Umathsupshiftup
1103  \__kernel_primitive:NN \Umathsupsubbottommax
1104  \__kernel_primitive:NN \Umathunderbarkern
1105  \__kernel_primitive:NN \Umathunderbarrule
1106  \__kernel_primitive:NN \Umathunderbarvgap
1107  \__kernel_primitive:NN \Umathunderdelimitergap
1108  \__kernel_primitive:NN \Umathunderdelimitervgap
1109  \__kernel_primitive:NN \Uoverdelimiter
1110  \__kernel_primitive:NN \Urational
1111  \__kernel_primitive:NN \Uroot
1112  \__kernel_primitive:NN \Uskewed
1113  \__kernel_primitive:NN \Uskewedwithdelims
1114  \__kernel_primitive:NN \Ustack
1115  \__kernel_primitive:NN \Ustartdisplaymath
1116  \__kernel_primitive:NN \Ustartmath
1117  \__kernel_primitive:NN \Ustopdisplaymath
1118  \__kernel_primitive:NN \Ustopmath
1119  \__kernel_primitive:NN \Usubscript
1120  \__kernel_primitive:NN \Usuperscript
1121  \__kernel_primitive:NN \Underdelimiter
1122  \__kernel_primitive:NN \Uvextensible
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
1151
1152
1153
1154
1155
1156
1157
1158
1159
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
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1579
1580
1581
1582
1583
1584
1585
1586
1587
1587
1588
1589
1589
1590
1591
1592
1593
1594
1595
1596
1597
1597
1598
1599
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1687
1688
1689
1689
1690
1691
1692
1693
1694
1695
1696
1697
1697
1698
1699
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1728
1729
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1738
1739
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1748
1749
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1768
1769
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1778
1779
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1788
1789
1789
1790
1791
1792
1793
1794
1795
1796
1797
1797
1798
1798
1799
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1818
1819
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1828
1829
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1838
1839
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1848
1849
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1868
1869
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1878
1879
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1888
1889
1889
1890
1891
1892
1893
1894
1895
1896
1897
1897
1898
1898
1899
1899
1900
1901
1902
1903
1904
1905
1906
1907
1907
1908
1908
1909
1909
1910
1911
1912
1913
1914
1915
1916
1917
1917
1918
1918
1919
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1928
1929
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1938
1939
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1948
1949
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1968
1969
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1978
1979
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1988
1989
1989
1990
1991
1992
1993
1994
1995
1996
1997
1997
1998
1998
1999
1999
2000
2001
2002
2003
2004
2005
2006
2007
2007
2008
2008
2009
2009
2010
2011
2012
2013
2014
2015
2016
2017
2017
2018
2018
2019
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2028
2029
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2038
2039
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2048
2049
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2068
2069
2069
2070
2071
2072
2073
2074
2075
2076
2077
2077
2078
2078
2079
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2088
2089
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2099
2100
2101
2102
2103
2104
2105
2106
2107
2107
2108
2108
2109
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2119
2120
2120
2121
2121
2122
2123
2124
2125
2126
2127
2128
2128
2129
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2138
2139
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2148
2149
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2168
2169
2169
2170
2171
2172
2173
2174
2175
2176
2177
2177
2178
2178
2179
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2188
2189
2189
2190
2191
2192
2193
2194
2195
2196
2197
2197
2198
2198
2199
2199
2200
2201
2202
2203
2204
2205
2206
2207
2207
2208
2208
2209
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2218
2219
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2228
2229
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2238
2239
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2248
2249
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2268
2269
2269
2270
2271
2272
2273
2274
2275
2276
2277
2277
2278
2278
2279
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2288
2289
2289
2290
2291
2292
2293
2294
2295
2296
2297
2298
2298
2299
2299
2300
2301
2302
2303
2304
2305
2306
2307
2307
2308
2308
2309
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2318
2319
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2328
2329
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2338
2339
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2348
2349
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2359
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2368
2369
2369
2370
2371
2372
2373
2374
2375
2376
2377
2377
2378
2378
2379
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2388
2389
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2398
2399
2399
2400
2401
2402
2403
2404
2405
2406
2407
2407
2408
2408
2409
2409
2410
2411
2412
2413
2414
2415
2416
2417
2418
2418
2419
2419
2420
2421
2422
2423
2424
2425
2426
2427
2428
2428
2429
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2438
2439
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2448
2449
2449
2450
2451
2452
2453
2454
2455
2456
2457
2458
2459
2459
2460
2461
2462
2463
2464
2465
2466
2467
2468
2468
2469
2469
2470
2471
2472
2473
2474
2475
2476
2477
2477
2478
2478
2479
2479
2480
2481
2482
2483
2484
2485
2486
2487
2488
2488
2489
2489
2490
2491
2492
2493
2494
2495
2496
2497
2498
2498
2499
2499
2500
2501
2502
2503
2504
2505
2506
2507
2507
2508
2508
2509
2509
2510
2511
2512
2513
2514
2515
2516
2517
2518
2518
2519
2519
2520
2521
2522
2523
2524
2525
2526
2527
2528
2528
2529
2529
2530
2531
2532
2533
2534
2535
2536
2537
2538
2538
2539
2539
2540
2541
2542
2543
2544
2545
2546
2547
2548
2548
2549
2549
2550
2551
2552
2553
2554
2555
2556
2557
2558
2559
2559
2560
2561
2562
2563
2564
2565
2566
2567
2568
2568
2569
2569
2570
2571
2572
2573
2574
2575
2576
2577
2577
2578
2578
2579
2579
2580
2581
2582
2583
2584
2585
2586
2587
2588
2588
2589
2589
2590
2591
2592
2593
2594
2595
2596
2597
2598
2598
2599
2599
2600
2601
2602
2603
2604
2605
2606
2607
2607
2608
2608
2609
2609
2610
2611
2612
2613
2614
2615
2616
2617
2618
2618
2619
2619
2620
2621
2622
2623
2624
2625
2626
2627
2628
2628
2629
2629
2630
2631
2632
2633
2634
2635
2636
2637
2638
2638
2639
2639
2640
2641
2642
2643
2644
2645
2646
2647
2648
2648
2649
2649
2650
2651
2652
2653
2654
2655
2656
2657
2658
2659
2659
2660
2661
2662
2663
2664
2665
2666
2667
2668
2668
2669
2669
2670
2671
2672
2673
2674
2675
2676
2677
2677
2678
2678
2679
2679
2680
2681
2682
2683
2684
2685
2686
2687
2688
2688
2689
2689
2690
2691
2692
2693
2694
2695
2696
2697
2698
2698
2699
2699
2700
2701
2702
2703
2704
2705
2706
2707
2707
2708
2708
2709
2709
2710
2711
2712
2713
2714
2715
2716
2717
2718
2718
2719
2719
2720
2721
2722
2723
2724
2725
2726
2727
2728
2728
2729
2729
2730
2731
2732
2733
2734
2735
2736
2737
2738
2738
2739
2739
2740
2741
2742
2743
2744
2745
2746
2747
2748
2748
2749
2749
2750
2751
2752
2753
2754
2755
2756
2757
2758
2759
2759
2760
2761
2762
2763
2764
2765
2766
2767
2768
2768
2769
2769
2770
2771
2772
2773
2774
2775
2776
2777
2778
2779
2779
2780
2781
2782
2783
2784
2785
2786
2787
2788
2788
2789
2789
2790
2791
2792
2793
2794
2795
2796
2797
2798
2798
2799
2799
2800
2801
2802
2803
2804
2805
2806
2807
2807
2808
2808
2809
2809
2810
2811
2812
2813
2814
2815
2816
2817
2818
2818
2819
2819
2820
2821
2822
2823
2824
2825
2826
2827
2828
2828
2829
2829
2830
2831
2832
2833
2834
2835
2836
2837
2838
2838
2839
2839
2840
2841
2842
2843
2844
2845
2846
2847
2848
2848
2849
2849
2850
2851
2852
2853
2854
2855
2856
2857
2858
2859
2859
2860
2861
2862
2863
2864
2865
2866
2867
2868
2868
2869
2869
2870
2871
2872
2873
2874
2875
2876
2877
2878
2879
2879
2880
2881
2882
2883
2884
2885
2886
2887
2888
2888
2889
2889
2890
2891

```

Primitives from pTeX.

```

1123  \__kernel_primitive:NN \autospacing
1124  \__kernel_primitive:NN \autoxspacing
1125  \__kernel_primitive:NN \dtou
1126  \__kernel_primitive:NN \euc
1127  \__kernel_primitive:NN \ifdbox
1128  \__kernel_primitive:NN \ifddir
1129  \__kernel_primitive:NN \ifmdir
1130  \__kernel_primitive:NN \iftbox
1131  \__kernel_primitive:NN \iftdir
1132  \__kernel_primitive:NN \ifybox
1133  \__kernel_primitive:NN \ifydir
1134  \__kernel_primitive:NN \inhibitglue
1135  \__kernel_primitive:NN \inhibitxspcode
1136  \__kernel_primitive:NN \jcharwidowpenalty
1137  \__kernel_primitive:NN \jfam
1138  \__kernel_primitive:NN \jfont
1139  \__kernel_primitive:NN \jis
1140  \__kernel_primitive:NN \kanjiskip
1141  \__kernel_primitive:NN \kansuji
1142  \__kernel_primitive:NN \kansujichar
1143  \__kernel_primitive:NN \kcatcode
1144  \__kernel_primitive:NN \kuten
1145  \__kernel_primitive:NN \noautospacing
1146  \__kernel_primitive:NN \noautoxspacing
1147  \__kernel_primitive:NN \postbreakpenalty
1148  \__kernel_primitive:NN \prebreakpenalty
1149  \__kernel_primitive:NN \showmode
1150  \__kernel_primitive:NN \sjis
1151  \__kernel_primitive:NN \tate
1152  \__kernel_primitive:NN \tbaselineshift
1153  \__kernel_primitive:NN \tfont
1154  \__kernel_primitive:NN \xkanjiskip
1155  \__kernel_primitive:NN \xspcode
1156  \__kernel_primitive:NN \ybaselineshift
1157  \__kernel_primitive:NN \yoko

```

Primitives from upTeX.

```

1158  \__kernel_primitive:NN \disablecjktoken
1159  \__kernel_primitive:NN \enablecjktoken
1160  \__kernel_primitive:NN \forcecjktoken
1161  \__kernel_primitive:NN \kchar
1162  \__kernel_primitive:NN \kchardef
1163  \__kernel_primitive:NN \kuten
1164  \__kernel_primitive:NN \ucs

```

End of the “just the names” part of the source.

```

1165  </initex | names | package>
1166  <*initex | package>

```

The job is done: close the group (using the primitive renamed!).

```
1167 \tex_endgroup:D
```

LATEX_{2ε} will have moved a few primitives, so these are sorted out. A convenient test for LATEX_{2ε} is the \@@end saved primitive.

```

1168  {*package)
1169  \etex_ifdefined:D \@end
1170  \tex_let:D \tex_end:D          \@end
1171  \tex_let:D \tex_everydisplay:D \frozen@everydisplay
1172  \tex_let:D \tex_everymath:D   \frozen@everymath
1173  \tex_let:D \tex_hyphen:D     \chph
1174  \tex_let:D \tex_input:D      \cinput
1175  \tex_let:D \tex_italiccorrection:D \citaliccorr
1176  \tex_let:D \tex_underline:D \cunderline

```

Some tidying up is needed for `\(pdf)tracingfonts`. Newer LuaTeX has this simply as `\tracingfonts`, but that will have been 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 `\@ctracingfonts` 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.)

```

1177  \tex_let:D \pdftex_tracingfonts:D \tex_undefined:D
1178  \etex_ifdefined:D \pdftracingfonts
1179  \tex_let:D \pdftex_tracingfonts:D \pdftracingfonts
1180  \tex_else:D
1181  \etex_ifdefined:D \luatex_directlua:D
1182  \luatex_directlua:D { tex.enableprimitives("@@", {"tracingfonts"}) }
1183  \tex_let:D \pdftex_tracingfonts:D \luatextracingfonts
1184  \tex_fi:D
1185  \tex_fi:D
1186  \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!

```

1187  \etex_ifdefined:D \luatexsuppressfontnotfounderror
1188  \tex_let:D \luatex_alignmark:D          \luatexalignmark
1189  \tex_let:D \luatex_aligntab:D         \luatexaligntab
1190  \tex_let:D \luatex_attribute:D        \luatexattribute
1191  \tex_let:D \luatex_attributedef:D    \luatexattributedef
1192  \tex_let:D \luatex_catcodetable:D   \luatexcatcodetable
1193  \tex_let:D \luatex_clearmarks:D     \luatexclearmarks
1194  \tex_let:D \luatex_crampeddisplaystyle:D \luatexcrampeddisplaystyle
1195  \tex_let:D \luatex_crampedscriptrscriptstyle:D \luatexcrampedscriptrscriptstyle
1196  \tex_let:D \luatex_crampedscriptrstyle:D \luatexcrampedscriptrstyle
1197  \tex_let:D \luatex_crampedtextstyle:D \luatexcrampedtextstyle
1198  \tex_let:D \luatex_fontid:D          \luatexfontid
1199  \tex_let:D \luatex_formatname:D     \luatexformatname
1200  \tex_let:D \luatex_gleaders:D       \luatexgleaders
1201  \tex_let:D \luatex_initcatcodetable:D \luatexinitcatcodetable
1202  \tex_let:D \luatex_latelua:D        \luatexlatelua
1203  \tex_let:D \luatex_luaescapestring:D \luatexluaescapestring
1204  \tex_let:D \luatex_luafunction:D    \luatexluafunction
1205  \tex_let:D \luatex_mathstyle:D      \luatexmathstyle
1206  \tex_let:D \luatex_nokerns:D       \luatexnokerns
1207  \tex_let:D \luatex_noligs:D        \luatexnoligs
1208  \tex_let:D \luatex_outputbox:D     \luatexoutputbox
1209  \tex_let:D \luatex_pageleftoffset:D \luatexpageleftoffset
1210  \tex_let:D \luatex_pagetopoffset:D \luatexpagetopoffset

```

```

1211  \tex_let:D \luatex_postexhyphenchar:D          \luatexpostexhyphenchar
1212  \tex_let:D \luatex_posthyphenchar:D           \luatexposthyphenchar
1213  \tex_let:D \luatex_preehyphenchar:D          \luatexpreexhyphenchar
1214  \tex_let:D \luatex_prehyphenchar:D           \luatexprehyphenchar
1215  \tex_let:D \luatex_savecatcodetable:D        \luatexsavecatcodetable
1216  \tex_let:D \luatex_scantexttokens:D         \luatexscantexttokens
1217  \tex_let:D \luatex_suppressifcsnameerror:D   \luatexsuppressifcsnameerror
1218  \tex_let:D \luatex_suppresslongerror:D       \luatexsuppresslongerror
1219  \tex_let:D \luatex_suppressmathparerror:D    \luatexsuppressmathparerror
1220  \tex_let:D \luatex_suppressoutererror:D      \luatexsuppressoutererror
1221  \tex_let:D \utex_char:D                      \luatexUchar
1222  \tex_let:D \xetex_suppressfontnotfounderror:D \luatexsuppressfontnotfounderror

```

Which also covers those slightly odd ones.

```

1223  \tex_let:D \luatex_bodydir:D                 \luatexbodydir
1224  \tex_let:D \luatex_boxdir:D                  \luatexboxdir
1225  \tex_let:D \luatex_leftghost:D              \luatexleftghost
1226  \tex_let:D \luatex_localbrokenpenalty:D   \luatexlocalbrokenpenalty
1227  \tex_let:D \luatex_localinterlinepenalty:D \luatexlocalinterlinepenalty
1228  \tex_let:D \luatex_localleftbox:D          \luatexlocalleftbox
1229  \tex_let:D \luatex_localrightbox:D        \luatexlocalrightbox
1230  \tex_let:D \luatex_mathdir:D               \luatexmathdir
1231  \tex_let:D \luatex_pagebottomoffset:D     \luatexpagebottomoffset
1232  \tex_let:D \luatex_pagedir:D                \luatexpagedir
1233  \tex_let:D \pdftex_pageheight:D            \luatexpageheight
1234  \tex_let:D \luatex_pagerightoffset:D      \luatexpagerightoffset
1235  \tex_let:D \pdftex_pagewidth:D             \luatexpagewidth
1236  \tex_let:D \luatex_pardir:D                \luatexpardir
1237  \tex_let:D \luatex_rightghost:D            \luatexrightghost
1238  \tex_let:D \luatex_textdir:D               \luatextextdir
1239  \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!

```

1240  \tex_ifnum:D 0
1241  \etex_ifdefined:D \pdftex_pdftexversion:D 1 \tex_fi:D
1242  \etex_ifdefined:D \luatex_luatexversion:D 1 \tex_fi:D
1243  = 0 %
1244  \tex_let:D \pdftex_mapfile:D \tex_undefined:D
1245  \tex_let:D \pdftex_mapline:D \tex_undefined:D
1246  \tex_fi:D
1247  </package>

```

Older XeTeX versions use \XeTeX as the prefix for the Unicode math primitives it knows. That is tided up here (we support XeTeX versions from 0.9994 but this change was in 0.9999).

```

1248  (*initex | package)
1249  \etex_ifdefined:D \XeTeXdelcode
1250  \tex_let:D \utex_delcode:D                  \XeTeXdelcode
1251  \tex_let:D \utex_delcodenum:D             \XeTeXdelcodenum
1252  \tex_let:D \utex_delimiter:D              \XeTeXdelimiter
1253  \tex_let:D \utex_mathaccent:D            \XeTeXmathaccent
1254  \tex_let:D \utex_mathchar:D               \XeTeXmathchar
1255  \tex_let:D \utex_mathchardef:D           \XeTeXmathchardef
1256  \tex_let:D \utex_mathcharnum:D          \XeTeXmathcharnum

```

```

1257   \tex_let:D \utex_mathcharnumdef:D \XeTeXmathcharnumdef
1258   \tex_let:D \utex_mathcode:D           \XeTeXmathcode
1259   \tex_let:D \utex_mathcodenum:D     \XeTeXmathcodenum
1260 \tex_fi:D

```

Up to v0.80, LuaTEX defines the pdfTEX version data: rather confusing. Removing them means that `\pdftex_pdftexversion:D` is a marker for pdfTEX alone: useful in engine-dependent code later.

```

1261 \etex_ifdefined:D \luatex_luatexversion:D
1262   \tex_let:D \pdftex_pdftexbanner:D \tex_undefined:D
1263   \tex_let:D \pdftex_pdftexrevision:D \tex_undefined:D
1264   \tex_let:D \pdftex_pdftexversion:D \tex_undefined:D
1265 \tex_fi:D
1266 </initex | package>

```

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.

```

1267 <*package>
1268 \etex_ifdefined:D \normalend
1269   \tex_let:D \tex_end:D           \normalend
1270   \tex_let:D \tex_everyjob:D     \normaleveryjob
1271   \tex_let:D \tex_input:D       \normalinput
1272   \tex_let:D \tex_language:D    \normallanguage
1273   \tex_let:D \tex_mathop:D      \normalmathop
1274   \tex_let:D \tex_month:D      \normalmonth
1275   \tex_let:D \tex_outer:D       \normalouter
1276   \tex_let:D \tex_over:D        \normalover
1277   \tex_let:D \tex_vcenter:D     \normalvcenter
1278   \tex_let:D \etex_unexpanded:D \normalunexpanded
1279   \tex_let:D \luatex_expanded:D \normalexpanded
1280 \tex_fi:D
1281 \etex_ifdefined:D \normalitaliccorrection
1282   \tex_let:D \tex_hoffset:D      \normalhoffset
1283   \tex_let:D \tex_italiccorrection:D \normalitaliccorrection
1284   \tex_let:D \tex_voffset:D      \normalvoffset
1285   \tex_let:D \etex_showtokens:D \normalshowtokens
1286   \tex_let:D \luatex_bodydir:D  \spac_directions_normal_body_dir
1287   \tex_let:D \luatex_pagedir:D  \spac_directions_normal_page_dir
1288 \tex_fi:D
1289 \etex_ifdefined:D \normalleft
1290   \tex_let:D \tex_left:D \normalleft
1291   \tex_let:D \tex_middle:D \normalmiddle
1292   \tex_let:D \tex_right:D \normalright
1293 \tex_fi:D
1294 </package>
1295 </initex | package>

```

3 I3basics implementation

```

1296 <*initex | package>

```

3.1 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 do a few now, just to get started.⁶

\if_true: Then some conditionals.

```

\if_false: 1297 \tex_let:D \if_true:           \tex_iftrue:D
            \or: 1298 \tex_let:D \if_false:       \tex_iffalse:D
            \else: 1299 \tex_let:D \or:          \tex_or:D
            \fi: 1300 \tex_let:D \else:         \tex_else:D
\reverse_if:N 1301 \tex_let:D \fi:           \tex_fi:D
            \if:w 1302 \tex_let:D \reverse_if:N \etex_unless:D
\if_charcode:w 1303 \tex_let:D \if:w          \tex_if:D
\if_catcode:w 1304 \tex_let:D \if_charcode:w \tex_if:D
\if_meaning:w 1305 \tex_let:D \if_catcode:w \tex_ifcat:D
            1306 \tex_let:D \if_meaning:w \tex_ifx:D

```

(End definition for \if_true: and others. These functions are documented on page 21.)

\if_mode_math: T_EX lets us detect some if its modes.

```

\if_mode_horizontal: 1307 \tex_let:D \if_mode_math:      \tex_ifemode:D
\if_mode_vertical: 1308 \tex_let:D \if_mode_horizontal: \tex_ifhmode:D
\if_mode_inner: 1309 \tex_let:D \if_mode_vertical: \tex_ifvmode:D
            1310 \tex_let:D \if_mode_inner: \tex_ifinner:D

```

(End definition for \if_mode_math: and others. These functions are documented on page 21.)

\if_cs_exist:N Building csnames and testing if control sequences exist.

```

\if_cs_exist:w 1311 \tex_let:D \if_cs_exist:N \etex_ifdefined:D
            \cs:w 1312 \tex_let:D \if_cs_exist:w \etex_ifcsname:D
\cs_end: 1313 \tex_let:D \cs:w \tex_csname:D
            1314 \tex_let:D \cs_end: \tex_endcsname:D

```

(End definition for \if_cs_exist:N and others. These functions are documented on page 21.)

\exp_after:wN The five \exp_ functions are used in the l3expan module where they are described.

```

\exp_not:N 1315 \tex_let:D \exp_after:wN \tex_expandafter:D
\exp_not:n 1316 \tex_let:D \exp_not:N \tex_noexpand:D
            1317 \tex_let:D \exp_not:n \etex_unexpanded:D
            1318 \tex_let:D \exp:w \tex_roman numeral:D
            1319 \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 30.)

\token_to_meaning:N Examining a control sequence or token.

```

\cs_meaning:N 1320 \tex_let:D \token_to_meaning:N \tex_meaning:D
                1321 \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 112.)

⁶This 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.

```

\tl_to_str:n Making strings.
\token_to_str:N 1322 \tex_let:D \tl_to_str:n      \etex_detokenize:D
                  1323 \tex_let:D \token_to_str:N \tex_string:D

(End definition for \tl_to_str:n and \token_to_str:N. These functions are documented on page 41.)
```

\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: 1324 \tex_let:D \scan_stop:      \tex_relax:D
\group_end:   1325 \tex_let:D \group_begin: \tex_begingroup:D
              1326 \tex_let:D \group_end:    \tex_endgroup:D

(End definition for \scan_stop:, \group_begin:, and \group_end:. These functions are documented on page 9.)
```

\if_int_compare:w For integers.

```

\__int_to_roman:w 1327 \tex_let:D \if_int_compare:w \tex_ifnum:D
                  1328 \tex_let:D \__int_to_roman:w \tex_roman numeral:D

(End definition for \if_int_compare:w and \__int_to_roman:w. These functions are documented on page 79.)
```

\group_insert_after:N Adding material after the end of a group.

```

1329 \tex_let:D \group_insert_after:N \tex_aftergroup:D

(End definition for \group_insert_after:N. This function is documented on page 9.)
```

\exp_args:Nc Discussed in l3expan, but needed much earlier.

```

\exp_args:cc 1330 \tex_long:D \tex_def:D \exp_args:Nc #1#2
              { \exp_after:wN #1 \cs:w #2 \cs_end: }
              1332 \tex_long:D \tex_def:D \exp_args:cc #1#2
              { \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 27.)
```

\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:Nc) 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.

```

1334 \tex_def:D \token_to_str:c { \exp_args:Nc \token_to_str:N }
1335 \tex_long:D \tex_def:D \cs_meaning:c #1
1336 {
1337   \if_cs_exist:w #1 \cs_end:
1338     \exp_after:wN \use_i:nn
1339   \else:
1340     \exp_after:wN \use_ii:nn
1341   \fi:
1342   { \exp_args:Nc \cs_meaning:N {#1} }
1343   { \tl_to_str:n {undefined} }
1344 }
1345 \tex_let:D \token_to_meaning:c = \cs_meaning:c

(End definition for \token_to_meaning:c, \token_to_str:c, and \cs_meaning:c. These functions are documented on page 112.)
```

3.2 Defining some constants

\c_zero

We need the constant \c_zero 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!

```
1346 \tex_chardef:D \c_zero = 0 ~
```

(End definition for \c_zero. This variable is documented on page 78.)

\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.

```
1347 \etex_ifdefined:D \luatex_luatexversion:D  
1348   \tex_chardef:D \c_max_register_int = 65 535 ~  
1349   \tex_else:D  
1350     \tex_mathchardef:D \c_max_register_int = 32 767 ~  
1351   \tex_if:D
```

(End definition for \c_max_register_int. This variable is documented on page 78.)

3.3 Defining functions

We start by providing functions for the typical definition functions. First the local ones.

\cs_set_nopar:Npn
\cs_set_nopar:Npx

\cs_set:Npn
\cs_set:Npx

\cs_set_protected_nopar:Npn

\cs_set_protected_nopar:Npx

\cs_set_protected:Npn

\cs_set_protected:Npx

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.

```
1352 \tex_let:D \cs_set_nopar:Npn           \tex_def:D  
1353 \tex_let:D \cs_set_nopar:Npx           \tex_eodef:D  
1354 \etex_protected:D \tex_long:D \tex_def:D \cs_set:Npn  
1355   { \tex_long:D \tex_def:D }  
1356 \etex_protected:D \tex_long:D \tex_def:D \cs_set:Npx  
1357   { \tex_long:D \tex_eodef:D }  
1358 \etex_protected:D \tex_long:D \tex_def:D \cs_set_protected_nopar:Npn  
1359   { \etex_protected:D \tex_def:D }  
1360 \etex_protected:D \tex_long:D \tex_def:D \cs_set_protected_nopar:Npx  
1361   { \etex_protected:D \tex_eodef:D }  
1362 \etex_protected:D \tex_long:D \tex_def:D \cs_set_protected:Npn  
1363   { \etex_protected:D \tex_long:D \tex_def:D }  
1364 \etex_protected:D \tex_long:D \tex_def:D \cs_set_protected:Npx  
1365   { \etex_protected:D \tex_long:D \tex_eodef:D }
```

(End definition for \cs_set_nopar:Npn and others. These functions are documented on page 11.)

\cs_gset_nopar:Npn

\cs_gset_nopar:Npx

\cs_gset:Npn

\cs_gset:Npx

\cs_gset_protected_nopar:Npn

\cs_gset_protected_nopar:Npx

\cs_gset_protected:Npn

\cs_gset_protected:Npx

Global versions of the above functions.

```
1366 \tex_let:D \cs_gset_nopar:Npn           \tex_gdef:D  
1367 \tex_let:D \cs_gset_nopar:Npx           \tex_xdef:D  
1368 \cs_set_protected:Npn \cs_gset:Npn  
1369   { \tex_long:D \tex_gdef:D }  
1370 \cs_set_protected:Npn \cs_gset:Npx  
1371   { \tex_long:D \tex_xdef:D }  
1372 \cs_set_protected:Npn \cs_gset_protected_nopar:Npn  
1373   { \etex_protected:D \tex_gdef:D }  
1374 \cs_set_protected:Npn \cs_gset_protected_nopar:Npx  
1375   { \etex_protected:D \tex_xdef:D }
```

```

1376 \cs_set_protected:Npn \cs_gset_protected:Npn
1377   { \etex_protected:D \tex_long:D \tex_gdef:D }
1378 \cs_set_protected:Npn \cs_gset_protected:Npx
1379   { \etex_protected:D \tex_long:D \tex_xdef:D }

```

(End definition for `\cs_gset_nopar:Npn` and others. These functions are documented on page 12.)

3.4 Selecting tokens

- `\l__exp_internal_tl` Scratch token list variable for `\b3exp`, used by `\use:x`, used in defining conditionals. We don't use `tl` methods because `\b3basics` is loaded earlier.

```
1380 \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.

```
1381 \cs_set:Npn \use:c #1 { \cs:w #1 \cs_end: }
```

(End definition for `\use:c`. This function is documented on page 16.)

- `\use:x` Fully expands its argument and passes it to the input stream. Uses the reserved `\l__-exp_internal_tl` which will be set up in `\b3exp`.

```

1382 \cs_set_protected:Npn \use:x #1
1383   {
1384     \cs_set_nopar:Npx \l__exp_internal_tl {#1}
1385     \l__exp_internal_tl
1386   }

```

(End definition for `\use:x`. This function is documented on page 19.)

- `\use:n` These macros grab their arguments and returns them back to the input (with outer braces removed).

```

\use:nn 1387 \cs_set:Npn \use:n    #1      {#1}
\use:nnn 1388 \cs_set:Npn \use:nn  #1#2    {#1#2}
          1389 \cs_set:Npn \use:nnn #1#2#3  {#1#2#3}
          1390 \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 17.)

- `\use_i:nn` The equivalent to L^AT_EX 2 _{ε} 's `\@firstoftwo` and `\@secondoftwo`.

```

\use_i:nn 1391 \cs_set:Npn \use_i:nn  #1#2 {#1}
          1392 \cs_set:Npn \use_i:nn  #1#2 {#2}

```

(End definition for `\use_i:nn` and `\use_i:nnn`. These functions are documented on page 18.)

- `\use_i:nnn` We also need something for picking up arguments from a longer list.

```

\use_i:nnn 1393 \cs_set:Npn \use_i:nnn  #1#2#3 {#1}
\use_ii:nnn 1394 \cs_set:Npn \use_ii:nnn  #1#2#3 {#2}
\use_i_ii:nnn 1395 \cs_set:Npn \use_i_ii:nnn #1#2#3 {#3}
\use_i:nnnn 1396 \cs_set:Npn \use_i:nnnn #1#2#3 {#1#2}
\use_ii:nnnn 1397 \cs_set:Npn \use_ii:nnnn #1#2#3#4 {#1}
\use_iii:nnnn 1398 \cs_set:Npn \use_iii:nnnn #1#2#3#4 {#2}
\use_iv:nnnn 1399 \cs_set:Npn \use_iv:nnnn #1#2#3#4 {#3}
          1400 \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 18.)

`\use_none_delimit_by_q_nil:w` Functions that gobble everything until they see either `\q_nil`, `\q_stop`, or `\q_recursion_stop`, respectively.

```
1401 \cs_set:Npn \use_none_delimit_by_q_nil:w #1 \q_nil { }
1402 \cs_set:Npn \use_none_delimit_by_q_stop:w #1 \q_stop { }
1403 \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 19.)

`\use_i_delimit_by_q_nil: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.

```
1404 \cs_set:Npn \use_i_delimit_by_q_nil:nw #1#2 \q_nil {#1}
1405 \cs_set:Npn \use_i_delimit_by_q_stop:nw #1#2 \q_stop {#1}
1406 \cs_set:Npn \use_i_delimit_by_q_recursion_stop:nw #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 19.)

3.5 Gobbling tokens from input

`\use_none:n` 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:nnnnnn`, this is very non-intuitive to the programmer who will assume that expanding such a function once will take care of gobbling all the tokens in one go.

```
1407 \cs_set:Npn \use_none:n      #1           { }
1408 \cs_set:Npn \use_none:nn     #1#2          { }
1409 \cs_set:Npn \use_none:nnn    #1#2#3         { }
1410 \cs_set:Npn \use_none:nnnn   #1#2#3#4        { }
1411 \cs_set:Npn \use_none:nnnnn  #1#2#3#4#5       { }
1412 \cs_set:Npn \use_none:nnnnnn #1#2#3#4#5#6      { }
1413 \cs_set:Npn \use_none:nnnnnnn #1#2#3#4#5#6#7     { }
1414 \cs_set:Npn \use_none:nnnnnnnn #1#2#3#4#5#6#7#8    { }
1415 \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 18.)

3.6 Conditional processing and definitions

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` will expand fully any `\else:` and the `\fi:` that are waiting to be discarded, before reaching the `\exp_end:` which will leave the expansion null. 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.

```
1416 \cs_set:Npn \prg_return_true:  
1417   { \exp_after:wN \use_i:nn \exp:w }  
1418 \cs_set:Npn \prg_return_false:  
1419   { \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 will work.

(End definition for `\prg_return_true:` and `\prg_return_false:`. These functions are documented on page 91.)

`\prg_set_conditional:Npnn` 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 `{<name>} {<signature>} {boolean} {set or new} {maybe protected} {parameters} {TF, ...} {code}` to the auxiliary function responsible for defining all conditionals.

```
1420 \cs_set_protected:Npn \prg_set_conditional:Npnn  
1421   { \__prg_generate_conditional_parm:nnNpnn { set } { } }  
1422 \cs_set_protected:Npn \prg_new_conditional:Npnn  
1423   { \__prg_generate_conditional_parm:nnNpnn { new } { } }  
1424 \cs_set_protected:Npn \prg_set_protected_conditional:Npnn  
1425   { \__prg_generate_conditional_parm:nnNpnn { set } { _protected } }  
1426 \cs_set_protected:Npn \prg_new_protected_conditional:Npnn  
1427   { \__prg_generate_conditional_parm:nnNpnn { new } { _protected } }  
1428 \cs_set_protected:Npn \__prg_generate_conditional_parm:nnNpnn #1#2#3#4#  
1429   {  
1430     \__cs_split_function:NN #3 \__prg_generate_conditional:nnNnnnnn  
1431     {#1} {#2} {#4}  
1432   }
```

(End definition for `\prg_set_conditional:Npnn` and others. These functions are documented on page 89.)

`\prg_set_conditional:Nnn` 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 `{<name>} {<signature>} {boolean} {set or new} {maybe protected} {parameters} {TF, ...} {code}` to the auxiliary function responsible for defining all conditionals. If

the $\langle\text{signature}\rangle$ 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.

```

1433 \cs_set_protected:Npn \prg_set_conditional:Nnn
1434   { \__prg_generate_conditional_count:nnNnn { set } { } }
1435 \cs_set_protected:Npn \prg_new_conditional:Nnn
1436   { \__prg_generate_conditional_count:nnNnn { new } { } }
1437 \cs_set_protected:Npn \prg_set_protected_conditional:Nnn
1438   { \__prg_generate_conditional_count:nnNnn { set } { _protected } }
1439 \cs_set_protected:Npn \prg_new_protected_conditional:Nnn
1440   { \__prg_generate_conditional_count:nnNnn { new } { _protected } }
1441 \cs_set_protected:Npn \__prg_generate_conditional_count:nnNnn #1#2#3
1442   {
1443     \__cs_split_function>NN #3 \__prg_generate_conditional_count:nnNnnnn
1444     {#1} {#2}
1445   }
1446 \cs_set_protected:Npn \__prg_generate_conditional_count:nnNnnnn #1#2#3#4#5
1447   {
1448     \__cs_parm_from_arg_count:nnF
1449     { \__prg_generate_conditional:nnNnnnnn {#1} {#2} #3 {#4} {#5} }
1450     { \tl_count:n {#2} }
1451     {
1452       \__msg_kernel_error:nnxx { kernel } { bad-number-of-arguments }
1453       { \token_to_str:c { #1 : #2 } }
1454       { \tl_count:n {#2} }
1455       \use_none:nn
1456     }
1457   }

```

(End definition for `\prg_set_conditional:Nnn` and others. These functions are documented on page 89.)

```
\__prg_generate_conditional:nnNnnnnn
\__prg_generate_conditional:nnnnnnw
```

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.

```

1458 \cs_set_protected:Npn \__prg_generate_conditional:nnNnnnnn #1#2#3#4#5#6#7#8
1459   {
1460     \if_meaning:w \c_false_bool #3
1461     \__msg_kernel_error:nnx { kernel } { missing-colon }
1462     { \token_to_str:c {#1} }
1463     \exp_after:wN \use_none:nn
1464   \fi:
1465   \use:x
1466   {
1467     \exp_not:N \__prg_generate_conditional:nnnnnnw
1468     \exp_not:n { {#4} {#5} {#1} {#2} {#6} {#8} }
1469     \tl_to_str:n {#7}
1470     \exp_not:n { , \q_recursion_tail , \q_recursion_stop }

```

```

1471     }
1472 }
```

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:nnnnnnn cleans up. Otherwise, the error message is removed by the variant form.

```

1473 \cs_set_protected:Npn \__prg_generate_conditional:nnnnnnw #1#2#3#4#5#6#7 ,
1474 {
1475     \if_meaning:w \q_recursion_tail #7
1476         \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
1477     \fi:
1478     \use:c { __prg_generate_ #7 _form:wnnnnnn }
1479         \tl_if_empty:nF {#7}
1480     {
1481         \__msg_kernel_error:nnxx
1482             { kernel } { conditional-form-unknown }
1483             {#7} { \token_to_str:c { #3 : #4 } }
1484     }
1485     \use_none:nnnnnnn
1486     \q_stop
1487     {#1} {#2} {#3} {#4} {#5} {#6}
1488     \__prg_generate_conditional:nnnnnnw {#1} {#2} {#3} {#4} {#5} {#6}
1489 }
```

(End definition for __prg_generate_conditional:nnNnnnnn and __prg_generate_conditional:nnnnnnw.)

```
\__prg_generate_p_form:wnnnnnn
\__prg_generate_TF_form:wnnnnnn
\__prg_generate_T_form:wnnnnnn
\__prg_generate_F_form:wnnnnnn
```

How to generate the various forms. Those functions take the following arguments: 1: set or new, 2: empty or _protected, 3: function name 4: signature, 5: parameter text (or empty), 6: replacement. 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 variant 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.

```

1490 \cs_set_protected:Npn \__prg_generate_p_form:wnnnnnn
1491     #1 \q_stop #2#3#4#5#6#7
1492 {
1493     \if_meaning:w \scan_stop: #3 \scan_stop:
1494         \exp_after:wN \use_i:nn
1495     \else:
1496         \exp_after:wN \use_ii:nn
1497     \fi:
1498     {
1499         \exp_args:cc { cs_ #2 #3 :Npn } { #4 _p: #5 } #6
1500             { #7 \exp_end: \c_true_bool \c_false_bool }
1501     }
1502     {
1503         \__msg_kernel_error:nnx { kernel } { protected-predicate }
1504             { \token_to_str:c { #4 _p: #5 } }
1505     }
1506 }
1507 \cs_set_protected:Npn \__prg_generate_T_form:wnnnnnn
1508     #1 \q_stop #2#3#4#5#6#7
```

```

1509  {
1510    \exp_args:cc { cs_ #2 #3 :Npn } { #4 : #5 T } #6
1511    { #7 \exp_end: \use:n \use_none:n }
1512  }
1513 \cs_set_protected:Npn \__prg_generate_F_form:wnnnnnn
1514   #1 \q_stop #2#3#4#5#6#7
1515  {
1516    \exp_args:cc { cs_ #2 #3 :Npn } { #4 : #5 F } #6
1517    { #7 \exp_end: { } }
1518  }
1519 \cs_set_protected:Npn \__prg_generate_TF_form:wnnnnnn
1520   #1 \q_stop #2#3#4#5#6#7
1521  {
1522    \exp_args:cc { cs_ #2 #3 :Npn } { #4 : #5 TF } #6
1523    { #7 \exp_end: }
1524  }

```

(End definition for `__prg_generate_p_form:wnnnnnn` and others.)

`\prg_set_eq_conditional:NNn`

`\prg_new_eq_conditional:NNn`

`__prg_set_eq_conditional:NNNn`

The setting-equal functions. Split both functions and feed `{(name1)} {(signature1)}` `{(boolean1)} {(name2)} {(signature2)}` `{(boolean2)}` `<copying function>` `<conditions>`, `\q_recursion_tail`, `\q_recursion_stop` to a first auxiliary.

```

1525 \cs_set_protected:Npn \prg_set_eq_conditional:NNn
1526   { \__prg_set_eq_conditional:NNNn \cs_set_eq:cc }
1527 \cs_set_protected:Npn \prg_new_eq_conditional:NNn
1528   { \__prg_set_eq_conditional:NNNn \cs_new_eq:cc }
1529 \cs_set_protected:Npn \__prg_set_eq_conditional:NNNn #1#2#3#4
1530  {
1531    \use:x
1532    {
1533      \exp_not:N \__prg_set_eq_conditional:nnNnnNNw
1534      \__cs_split_function:NN #2 \prg_do_nothing:
1535      \__cs_split_function:NN #3 \prg_do_nothing:
1536      \exp_not:N #1
1537      \tl_to_str:n {#4}
1538      \exp_not:n { , \q_recursion_tail , \q_recursion_stop }
1539    }
1540  }

```

(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 90.)

`__prg_set_eq_conditional:nnNnnNNw`

`__prg_set_eq_conditional_loop:mnnnNw`

`__prg_set_eq_conditional_p_form:nnm`

`__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 `{(name1)} {(signature1)}` `{(name2)} {(signature2)}` `<copying function>` 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.

```

1541 \cs_set_protected:Npn \__prg_set_eq_conditional:nnNnnNNw #1#2#3#4#5#6
1542  {
1543    \if_meaning:w \c_false_bool #3
1544      \__msg_kernel_error:nnx { kernel } { missing-colon }
1545      { \token_to_str:c {#1} }

```

```

1546      \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
1547      \fi:
1548      \if_meaning:w \c_false_bool #6
1549          \_msg_kernel_error:n { kernel } { missing-colon }
1550          { \token_to_str:c {#4} }
1551          \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
1552      \fi:
1553      \__prg_set_eq_conditional_loop:nnnnNw {#1} {#2} {#4} {#5}
1554  }
1555 \cs_set_protected:Npn \__prg_set_eq_conditional_loop:nnnnNw #1#2#3#4#5#6 ,
1556  {
1557      \if_meaning:w \q_recursion_tail #6
1558          \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
1559      \fi:
1560      \use:c { __prg_set_eq_conditional_ #6 _form:wNnnnn }
1561          \tl_if_empty:nF {#6}
1562  {
1563      \__msg_kernel_error:nnxx
1564          { kernel } { conditional-form-unknown }
1565          {#6} { \token_to_str:c { #1 : #2 } }
1566  }
1567      \use_none:nnnnn
1568      \q_stop
1569      #5 {#1} {#2} {#3} {#4}
1570      \__prg_set_eq_conditional_loop:nnnnNw {#1} {#2} {#3} {#4} #5
1571  }
1572 \cs_set:Npn \__prg_set_eq_conditional_p_form:wNnnnn #1 \q_stop #2#3#4#5#6
1573  {
1574      \__chk_if_exist_cs:c { #5 _p : #6      }
1575      #2 { #3 _p : #4      } { #5 _p : #6      }
1576  }
1577 \cs_set:Npn \__prg_set_eq_conditional_TF_form:wNnnnn #1 \q_stop #2#3#4#5#6
1578  {
1579      \__chk_if_exist_cs:c { #5      : #6 TF }
1580      #2 { #3      : #4 TF } { #5      : #6 TF }
1581  }
1582 \cs_set:Npn \__prg_set_eq_conditional_T_form:wNnnnn #1 \q_stop #2#3#4#5#6
1583  {
1584      \__chk_if_exist_cs:c { #5      : #6 T  }
1585      #2 { #3      : #4 T } { #5      : #6 T  }
1586  }
1587 \cs_set:Npn \__prg_set_eq_conditional_F_form:wNnnnn #1 \q_stop #2#3#4#5#6
1588  {
1589      \__chk_if_exist_cs:c { #5      : #6 F  }
1590      #2 { #3      : #4 F } { #5      : #6 F  }
1591  }

```

(End definition for `__prg_set_eq_conditional:nnNnnNNw` 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 1592 \tex_chardef:D \c_true_bool = 1 ~  

                1593 \tex_chardef:D \c_false_bool = 0 ~
```

(End definition for `\c_true_bool` and `\c_false_bool`. These variables are documented on page 20.)

3.7 Dissecting a control sequence

`\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 characters, 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 \a` 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_roman numeral:D`. The second case is that the escape character is not printable. Then the `\if:w` test is unfinished after reading a the space from `\token_to_str:N \a`, 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_roman numeral:D` with `\c_zero`. 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` to the character 0. The initial `\tex_roman numeral: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_roman numeral:D`. In all three cases, `\cs_to_str:N` takes two expansion steps to be fully expanded.

```
1594 \cs_set:Npn \cs_to_str:N  

1595 {
```

We implement the expansion scheme using `\tex_roman numeral:D` terminating it with `\c_zero` 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` so we make this dependency explicit.

```
1596 \tex_roman numeral:D  

1597     \if:w \token_to_str:N \ \_cs_to_str:w \fi:  

1598     \exp_after:wN \_cs_to_str:N \token_to_str:N  

1599 }
1600 \cs_set:Npn \_cs_to_str:N #1 { \c_zero }
1601 \cs_set:Npn \_cs_to_str:w #1 \_cs_to_str:N  

1602 { - \_int_value:w \fi: \exp_after:wN \c_zero }
```

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`. These functions are documented on page 17.)

`_cs_split_function>NN`

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 `<true>` or `<false>` is returned with `<true>` for when there is a colon in the function and `<false>` if there is not. Lastly, the second argument of `_cs_split_function>NN` is supposed to be a function taking three variables, one for name, one for signature, and one for the boolean. For example, `_cs_split_function>NN \foo_bar:cnx \use_i:nnn` as input becomes `\use_i:nnn {foo_bar} {cnx} \c_true_bool`.

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 `\q_mark`, then `\c_true_bool` as #3, and #4 cleans up until `\q_stop`. Otherwise, the #1 contains the function name and `\q_mark \c_true_bool`, #2 is empty, #3 is `\c_false_bool`, and #4 cleans up. In both cases, #5 is the `<processor>`. The second auxiliary trims the trailing `\q_mark` from the function name if present (that is, if the original function had no colon).

```

1603 \cs_set:Npx \_cs_split_function>NN #1
1604   {
1605     \exp_not:N \exp_after:wN \exp_not:N \exp_after:wN
1606     \exp_not:N \exp_after:wN \exp_not:N \_cs_split_function_auxi:w
1607     \exp_not:N \cs_to_str:N #1 \exp_not:N \q_mark \c_true_bool
1608     \token_to_str:N : \exp_not:N \q_mark \c_false_bool
1609     \exp_not:N \q_stop
1610   }
1611 \use:x
1612   {
1613     \cs_set:Npn \exp_not:N \_cs_split_function_auxi:w
1614       ##1 \token_to_str:N : ##2 \exp_not:N \q_mark ##3##4 \exp_not:N \q_stop ##5
1615   }
1616   { \_cs_split_function_auxii:w #5 #1 \q_mark \q_stop {#2} #3 }
1617 \cs_set:Npn \_cs_split_function_auxii:w #1#2 \q_mark #3 \q_stop
1618   { #1 {#2} }
```

(End definition for `_cs_split_function>NN`, `_cs_split_function_auxi:w`, and `_cs_split_function_auxii:w`.)

`_cs_get_function_name:N`

`_cs_get_function_signature:N`

```

1619 \cs_set:Npn \_cs_get_function_name:N #1
1620   { \_cs_split_function>NN #1 \use_i:nnn }
1621 \cs_set:Npn \_cs_get_function_signature:N #1
1622   { \_cs_split_function>NN #1 \use_i:nnn }
```

(End definition for `_cs_get_function_name:N` and `_cs_get_function_signature:N`.)

3.8 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 **\cs_if_exist_p:c** 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:cTF 1623 \prg_set_conditional:Npnn \cs_if_exist:N #1 { p , T , F , TF }
1624 {
1625     \if_meaning:w #1 \scan_stop:
1626         \prg_return_false:
1627     \else:
1628         \if_cs_exist:N #1
1629             \prg_return_true:
1630         \else:
1631             \prg_return_false:
1632         \fi:
1633     \fi:
1634 }
```

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.

```

\cs_if_exist:c 1635 \prg_set_conditional:Npnn \cs_if_exist:c #1 { p , T , F , TF }
1636 {
1637     \if_cs_exist:w #1 \cs_end:
1638         \exp_after:wN \use_i:nn
1639     \else:
1640         \exp_after:wN \use_ii:nn
1641     \fi:
1642 {
1643     \exp_after:wN \if_meaning:w \cs:w #1 \cs_end: \scan_stop:
1644         \prg_return_false:
1645     \else:
1646         \prg_return_true:
1647     \fi:
1648 }
1649 \prg_return_false:
1650 }
```

(End definition for `\cs_if_exist:NTF`. This function is documented on page 20.)

\cs_if_free_p:N The logical reversal of the above.

```

\cs_if_free_p:c 1651 \prg_set_conditional:Npnn \cs_if_free:N #1 { p , T , F , TF }
\cs_if_free:cTF 1652 {
1653     \if_meaning:w #1 \scan_stop:
1654         \prg_return_true:
1655     \else:
1656         \if_cs_exist:N #1
1657             \prg_return_false:
```

```

1658     \else:
1659         \prg_return_true:
1660     \fi:
1661 \fi:
1662 }
1663 \prg_set_conditional:Npnn \cs_if_free:c #1 { p , T , F , TF }
1664 {
1665     \if_cs_exist:w #1 \cs_end:
1666         \exp_after:wN \use_i:nn
1667     \else:
1668         \exp_after:wN \use_ii:nn
1669     \fi:
1670     {
1671         \exp_after:wN \if_meaning:w \cs:w #1 \cs_end: \scan_stop:
1672         \prg_return_true:
1673     \else:
1674         \prg_return_false:
1675     \fi:
1676 }
1677 { \prg_return_true: }
1678 }

```

(End definition for `\cs_if_free:NTF`. This function is documented on page 20.)

\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.

```

1679 \cs_set:Npn \cs_if_exist_use:NTF #1#2
1680 { \cs_if_exist:NTF #1 { #1 #2 } }
1681 \cs_set:Npn \cs_if_exist_use:NF #1
1682 { \cs_if_exist:NTF #1 { #1 } }
1683 \cs_set:Npn \cs_if_exist_use:NT #1 #2
1684 { \cs_if_exist:NTF #1 { #1 #2 } { } }
1685 \cs_set:Npn \cs_if_exist_use:N #1
1686 { \cs_if_exist:NTF #1 { #1 } { } }
1687 \cs_set:Npn \cs_if_exist_use:cTF #1#2
1688 { \cs_if_exist:cTF {#1} { \use:c {#1} #2 } }
1689 \cs_set:Npn \cs_if_exist_use:cF #1
1690 { \cs_if_exist:cTF {#1} { \use:c {#1} } }
1691 \cs_set:Npn \cs_if_exist_use:cT #1#2
1692 { \cs_if_exist:cTF {#1} { \use:c {#1} #2 } { } }
1693 \cs_set:Npn \cs_if_exist_use:c #1
1694 { \cs_if_exist:cTF {#1} { \use:c {#1} } { } }

```

(End definition for `\cs_if_exist_use:NTF`. This function is documented on page 16.)

3.9 Defining and checking (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 definitions here are only temporary, they will be redefined later on.

`\iow_log:x` We define a routine to write only to the log file. And a similar one for writing to both the log file and the terminal. These will be redefined later by `\3io`.

```
1695 \cs_set_protected:Npn \iow_log:x
1696   { \tex_immediate:D \tex_write:D -1 }
1697 \cs_set_protected:Npn \iow_term:x
1698   { \tex_immediate:D \tex_write:D 16 }
```

(End definition for `\iow_log:x` and `\iow_term:x`. These functions are documented on page 142.)

`__chk_log:x` This function is used to write some information to the log file in case the `log-function` option is set. Otherwise its argument is ignored. Using this function rather than directly using `\iow_log:x` allows for `__chk_suspend_log:` which disables such messages until the matching `__chk_resume_log:`. These two commands are used to improve the logging for complicated datatypes. They should come in pairs, which can be nested. The function `\exp_not:o` is defined in `\3expan` later on but `__chk_suspend_log:` and `__chk_resume_log:` are not used before that point.

```
1699 (*initex)
1700 \cs_set_protected:Npn \__chk_log:x { \use_none:n }
1701 \cs_set_protected:Npn \__chk_suspend_log: { }
1702 \cs_set_protected:Npn \__chk_resume_log: { }
1703 
```

(/initex)

```
1704 (*package)
1705 \tex_ifodd:D \l@expl@log@functions@bool
1706   \cs_set_protected:Npn \__chk_log:x { \iow_log:x }
1707   \cs_set_protected:Npn \__chk_suspend_log:
1708   {
1709     \cs_set_protected:Npx \__chk_resume_log:
1710   {
1711     \cs_set_protected:Npn \__chk_resume_log:
1712       { \exp_not:o { \__chk_resume_log: } }
1713     \cs_set_protected:Npn \__chk_log:x
1714       { \exp_not:o { \__chk_log:x } }
1715   }
1716   \cs_set_protected:Npn \__chk_log:x { \use_none:n }
1717 }
1718 \cs_set_protected:Npn \__chk_resume_log: { }
1719 \else:
1720   \cs_set_protected:Npn \__chk_log:x { \use_none:n }
1721   \cs_set_protected:Npn \__chk_suspend_log: { }
1722   \cs_set_protected:Npn \__chk_resume_log: { }
1723 \fi:
1724 
```

(/package)

(End definition for `__chk_log:x`, `__chk_suspend_log:`, and `__chk_resume_log:`.)

`__msg_kernel_error:nxxx` If an internal error occurs before L^AT_EX3 has loaded `\3msg` 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.

```
1725 \cs_set_protected:Npn \__msg_kernel_error:nxxx #1#2#3#4
```

```

1726  {
1727    \tex_newlinechar:D = '\^J \tex_relax:D
1728    \tex_errmessage:D
1729    {
1730      !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!~! ^J
1731      Argh,~internal~LaTeX3~error! ^J ^J
1732      Module ~ #1 , ~ message~name~"#2": ^J
1733      Arguments~'#3'~and~'#4' ^J ^J
1734      This~is~one~for~The~LaTeX3~Project:~bailing~out
1735    }
1736    \tex_end:D
1737  }
1738  \cs_set_protected:Npn \__msg_kernel_error:nnx #1#2#3
1739  { \__msg_kernel_error:nxxx {#1} {#2} {#3} { } }
1740  \cs_set_protected:Npn \__msg_kernel_error:nn #1#2
1741  { \__msg_kernel_error:nxxx {#1} {#2} { } { } }

(End definition for \__msg_kernel_error:nxx, \__msg_kernel_error:nnx, and \__msg_kernel_error:nn.)
```

\msg_line_context: Another one from l3msg which will be altered later.

```

1742  \cs_set:Npn \msg_line_context:
1743  { on-line~ \tex_the:D \tex_inputlineno:D }
```

(End definition for \msg_line_context:. This function is documented on page 129.)

__chk_if_free_cs:N This command is called by \cs_new_nopar:Npn and \cs_new_eq:NN etc. to make sure 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!

```

1744  \cs_set_protected:Npn \__chk_if_free_cs:N #1
1745  {
1746    \cs_if_free:NF #1
1747    {
1748      \__msg_kernel_error:nxxx { kernel } { command-already-defined }
1749      { \token_to_str:N #1 } { \token_to_meaning:N #1 }
1750    }
1751  }
1752  /*package*/
1753  \tex_ifodd:D \l@expl@log@functions@bool
1754  \cs_set_protected:Npn \__chk_if_free_cs:N #1
1755  {
1756    \cs_if_free:NF #1
1757    {
1758      \__msg_kernel_error:nxxx { kernel } { command-already-defined }
1759      { \token_to_str:N #1 } { \token_to_meaning:N #1 }
1760    }
1761    \__chk_log:x { Defining~\token_to_str:N #1~ \msg_line_context: }
1762  }
1763  \fi:
1764  /*package*/
1765  \cs_set_protected:Npn \__chk_if_free_cs:c
1766  { \exp_args:Nc \__chk_if_free_cs:N }
```

(End definition for __chk_if_free_cs:N.)

__chk_if_exist_var:N Create the checking function for variable definitions when the option is set.

```
1767  {*package}
1768  \tex_ifodd:D \l@expl@check@declarations@bool
1769  \cs_set_protected:Npn \__chk_if_exist_var:N #1
1770  {
1771      \cs_if_exist:NF #1
1772      {
1773          \__msg_kernel_error:nnx { check } { non-declared-variable }
1774          { \token_to_str:N #1 }
1775      }
1776  }
1777 \fi:
1778 
```

(End definition for `__chk_if_exist_var:N`.)

__chk_if_exist_cs:N This function issues an error message when the control sequence in its argument does not exist.

```
1779 \cs_set_protected:Npn \__chk_if_exist_cs:N #1
1780 {
1781     \cs_if_exist:NF #1
1782     {
1783         \__msg_kernel_error:nnx { kernel } { command-not-defined }
1784         { \token_to_str:N #1 }
1785     }
1786 }
1787 \cs_set_protected:Npn \__chk_if_exist_cs:c
1788 { \exp_args:Nc \__chk_if_exist_cs:N }
```

(End definition for `__chk_if_exist_cs:N`.)

3.10 More new definitions

\cs_new_nopar:Npn Function which check that the control sequence is free before defining it.

```
1789 \cs_set:Npn \__cs_tmp:w #1#2
1790 {
1791     \cs_set_protected:Npn #1 ##1
1792     {
1793         \__chk_if_free_cs:N ##1
1794         #2 ##1
1795     }
1796 }
1797 \__cs_tmp:w \cs_new_nopar:Npn           \cs_gset_nopar:Npn
1798 \__cs_tmp:w \cs_new_nopar:Npx          \cs_gset_nopar:Npx
1799 \__cs_tmp:w \cs_new:Npn                \cs_gset:Npn
1800 \__cs_tmp:w \cs_new:Npx                \cs_gset:Npx
1801 \__cs_tmp:w \cs_new_protected_nopar:Npn \cs_gset_protected_nopar:Npn
1802 \__cs_tmp:w \cs_new_protected_nopar:Npx \cs_gset_protected_nopar:Npx
1803 \__cs_tmp:w \cs_new_protected:Npn       \cs_gset_protected:Npn
1804 \__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 11.)

`\cs_set_nopar:cpn`
`\cs_set_nopar:cpx`
`\cs_gset_nopar:cpn`
`\cs_gset_nopar:cpx`
`\cs_new_nopar:cpn`
`\cs_new_nopar:cpx`

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<string><rep-text>` will turn `<string>` into a csname and then assign `<rep-text>` to it by using `\cs_set_nopar:Npn`. This means that there might be a parameter string between the two arguments.

```

1805 \cs_set:Npn __cs_tmp:w #1#2
1806   { \cs_new_protected_nopar:Npn #1 { \exp_args:Nc #2 } }
1807 __cs_tmp:w \cs_set_nopar:cpn \cs_set_nopar:Npn
1808 __cs_tmp:w \cs_set_nopar:cpx \cs_set_nopar:Npx
1809 __cs_tmp:w \cs_gset_nopar:cpn \cs_gset_nopar:Npn
1810 __cs_tmp:w \cs_gset_nopar:cpx \cs_gset_nopar:Npx
1811 __cs_tmp:w \cs_new_nopar:cpn \cs_new_nopar:Npn
1812 __cs_tmp:w \cs_new_nopar:cpx \cs_new_nopar:Npx

```

(End definition for `\cs_set_nopar:cpn` and others. These functions are documented on page 11.)

`\cs_set:cpn`
`\cs_set:cpx`
`\cs_gset:cpn`
`\cs_gset:cpx`
`\cs_new:cpn`
`\cs_new:cpx`

Variants of the `\cs_set:Npn` versions which make a csname out of the first arguments. We may also do this globally.

```

1813 __cs_tmp:w \cs_set:cpn \cs_set:Npn
1814 __cs_tmp:w \cs_set:cpx \cs_set:Npx
1815 __cs_tmp:w \cs_gset:cpn \cs_gset:Npn
1816 __cs_tmp:w \cs_gset:cpx \cs_gset:Npx
1817 __cs_tmp:w \cs_new:cpn \cs_new:Npn
1818 __cs_tmp:w \cs_new:cpx \cs_new:Npx

```

(End definition for `\cs_set:cpn` and others. These functions are documented on page 11.)

`\cs_set_protected_nopar:cpn`
`\cs_set_protected_nopar:cpx`
`\cs_gset_protected_nopar:cpn`
`\cs_gset_protected_nopar:cpx`
`\cs_new_protected_nopar:cpn`
`\cs_new_protected_nopar:cpx`

Variants of the `\cs_set_protected_nopar:Npn` versions which make a csname out of the first arguments. We may also do this globally.

```

1819 __cs_tmp:w \cs_set_protected_nopar:cpn \cs_set_protected_nopar:Npn
1820 __cs_tmp:w \cs_set_protected_nopar:cpx \cs_set_protected_nopar:Npx
1821 __cs_tmp:w \cs_gset_protected_nopar:cpn \cs_gset_protected_nopar:Npn
1822 __cs_tmp:w \cs_gset_protected_nopar:cpx \cs_gset_protected_nopar:Npx
1823 __cs_tmp:w \cs_new_protected_nopar:cpn \cs_new_protected_nopar:Npn
1824 __cs_tmp:w \cs_new_protected_nopar:cpx \cs_new_protected_nopar:Npx

```

(End definition for `\cs_set_protected_nopar:cpn` and others. These functions are documented on page 12.)

`\cs_set_protected:cpn`
`\cs_set_protected:cpx`
`\cs_gset_protected:cpn`
`\cs_gset_protected:cpx`
`\cs_new_protected:cpn`
`\cs_new_protected:cpx`

Variants of the `\cs_set_protected:Npn` versions which make a csname out of the first arguments. We may also do this globally.

```

1825 __cs_tmp:w \cs_set_protected:cpn \cs_set_protected:Npn
1826 __cs_tmp:w \cs_set_protected:cpx \cs_set_protected:Npx
1827 __cs_tmp:w \cs_gset_protected:cpn \cs_gset_protected:Npn
1828 __cs_tmp:w \cs_gset_protected:cpx \cs_gset_protected:Npx
1829 __cs_tmp:w \cs_new_protected:cpn \cs_new_protected:Npn
1830 __cs_tmp:w \cs_new_protected:cpx \cs_new_protected:Npx

```

(End definition for `\cs_set_protected:cpn` and others. These functions are documented on page 11.)

3.11 Copying definitions

\cs_set_eq:NN These macros allow us to copy the definition of a control sequence to another control sequence.
\cs_set_eq:cN
\cs_set_eq:Nc
\cs_set_eq:cc
\cs_gset_eq:NN The = sign allows us to define funny char tokens like = itself or \sqcup with this function.
\cs_gset_eq:cN For the definition of `\c_space_char{~}` to work we need the ~ after the =.
\cs_gset_eq:Nc
\cs_gset_eq:NN `\cs_set_eq:NN` is long to avoid problems with a literal argument of `\par`. While `\cs_new_eq:NN` will probably never be correct with a first argument of `\par`, define it long in order to throw an “already defined” error rather than “runaway argument”.
\cs_gset_eq:cc
\cs_new_eq:NN
\cs_new_eq:cN
\cs_new_eq:Nc
\cs_new_eq:cc

```

1831 \cs_new_protected:Npn \cs_set_eq:NN #1 { \tex_let:D #1 =~ }
1832 \cs_new_protected:Npn \cs_set_eq:cN { \exp_args:Nc \cs_set_eq:NN }
1833 \cs_new_protected:Npn \cs_set_eq:Nc { \exp_args:NNc \cs_set_eq:NN }
1834 \cs_new_protected:Npn \cs_set_eq:cc { \exp_args:Ncc \cs_set_eq:NN }
1835 \cs_new_protected:Npn \cs_gset_eq:NN { \tex_global:D \cs_set_eq:NN }
1836 \cs_new_protected:Npn \cs_gset_eq:Nc { \exp_args:NNc \cs_gset_eq:NN }
1837 \cs_new_protected:Npn \cs_gset_eq:cN { \exp_args:Nc \cs_gset_eq:NN }
1838 \cs_new_protected:Npn \cs_gset_eq:cc { \exp_args:Ncc \cs_gset_eq:NN }
1839 \cs_new_protected:Npn \cs_new_eq:NN #1
1840 {
1841     \__chk_if_free_cs:N #1
1842     \tex_global:D \cs_set_eq:NN #1
1843 }
1844 \cs_new_protected:Npn \cs_new_eq:cN { \exp_args:Nc \cs_new_eq:NN }
1845 \cs_new_protected:Npn \cs_new_eq:Nc { \exp_args:NNc \cs_new_eq:NN }
1846 \cs_new_protected:Npn \cs_new_eq:cc { \exp_args:Ncc \cs_new_eq:NN }
```

(End definition for `\cs_set_eq:NN`, `\cs_gset_eq:NN`, and `\cs_new_eq:NN`. These functions are documented on page 15.)

3.12 Undefining functions

\cs_undefine:N The following function is used to free the main memory from the definition of some function that isn’t in use any longer. The c 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 #1 is unbalanced in this matter.
\cs_undefine:c

```

1847 \cs_new_protected:Npn \cs_undefine:N #1
1848     { \cs_gset_eq:NN #1 \tex_undefined:D }
1849 \cs_new_protected:Npn \cs_undefine:c #1
1850 {
1851     \if_cs_exist:w #1 \cs_end:
1852         \exp_after:wN \use:n
1853     \else:
1854         \exp_after:wN \use_none:n
1855     \fi:
1856     { \cs_gset_eq:cN {#1} \tex_undefined:D }
1857 }
```

(End definition for `\cs_undefine:N`. This function is documented on page 15.)

3.13 Generating parameter text from argument count

_cs_parm_from_arg_count:nnf LATEX3 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.

```

1858 \cs_set_protected:Npn \__cs_parm_from_arg_count:nnF #1#2
1859 {
1860     \exp_args:Nx \__cs_parm_from_arg_count_test:nnF
1861     {
1862         \exp_after:wN \exp_not:n
1863         \if_case:w \__int_eval:w #2 \__int_eval_end:
1864         {
1865             \or: { ##1 }
1866             \or: { ##1##2 }
1867             \or: { ##1##2##3 }
1868             \or: { ##1##2##3##4 }
1869             \or: { ##1##2##3##4##5 }
1870             \or: { ##1##2##3##4##5##6 }
1871             \or: { ##1##2##3##4##5##6##7 }
1872             \or: { ##1##2##3##4##5##6##7##8 }
1873             \or: { ##1##2##3##4##5##6##7##8##9 }
1874             \else: { \c_false_bool }
1875             \fi:
1876         }
1877         {#1}
1878     }
1879 \cs_set_protected:Npn \__cs_parm_from_arg_count_test:nnF #1#2
1880 {
1881     \if_meaning:w \c_false_bool #1
1882     \exp_after:wN \use_ii:nn
1883     \else:
1884     \exp_after:wN \use_i:nn
1885     \fi:
1886     {#2 {#1} }
1887 }
```

(End definition for `__cs_parm_from_arg_count:nnF` and `__cs_parm_from_arg_count_test:nnF`.)

3.14 Defining functions from a given number of arguments

`__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 `\tl_count:n` if there is a signature, otherwise `-1` arguments to signal an error. We need a variant form right away.

```

1888 \cs_new:Npn \__cs_count_signature:N #1
1889   { \int_eval:n { \__cs_split_function:NN #1 \__cs_count_signature:nnN } }
1890 \cs_new:Npn \__cs_count_signature:nnN #1#2#3
1891   {
1892     \if_meaning:w \c_true_bool #3
1893     \tl_count:n {#2}
1894     \else:
1895       -1
```

```

1896     \fi:
1897   }
1898 \cs_new:Npn \__cs_count_signature:c
1899   { \exp_args:Nc \__cs_count_signature:N }

(End definition for \__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

```

1900 \cs_new_protected:Npn \cs_generate_from_arg_count:NNnn #1#2#3#4
1901   {
1902     \__cs_parm_from_arg_count:nnF { \use:nnn #2 #1 } {#3}
1903     {
1904       \__msg_kernel_error:nnxx { kernel } { bad-number-of-arguments }
1905       { \token_to_str:N #1 } { \int_eval:n {#3} }
1906       \use_none:n
1907     }
1908     {#4}
1909   }

```

A variant form we need right away, plus one which is used elsewhere but which is most logically created here.

```

1910 \cs_new_protected:Npn \cs_generate_from_arg_count:cNnn
1911   { \exp_args:Nc \cs_generate_from_arg_count:NNnn }
1912 \cs_new_protected:Npn \cs_generate_from_arg_count:Ncnn
1913   { \exp_args:NNc \cs_generate_from_arg_count:NNnn }

```

(End definition for \cs_generate_from_arg_count:NNnn. This function is documented on page 14.)

3.15 Using the signature to define functions

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}.

\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_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

We want to define \cs_set:Nn as

```

\cs_set_protected:Npn \cs_set:Nn #1#2
{
  \cs_generate_from_arg_count:NNnn #1 \cs_set:Npn
    { \__cs_count_signature:N #1 } {#2}
}

```

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.

```

1914 \cs_set:Npn \__cs_tmp:w #1#2#3
1915   {
1916     \cs_new_protected:cp { cs_ #1 : #2 }

```

```

1917     {
1918         \exp_not:N \__cs_generate_from_signature:NNn
1919         \exp_after:wN \exp_not:N \cs:w cs_ #1 : #3 \cs_end:
1920     }
1921 }
1922 \cs_new_protected:Npn \__cs_generate_from_signature:NNn #1#2
1923 {
1924     \__cs_split_function:NN #2 \__cs_generate_from_signature:nnNNNn
1925     #1 #2
1926 }
1927 \cs_new_protected:Npn \__cs_generate_from_signature:nnNNNn #1#2#3#4#5#6
1928 {
1929     \bool_if:NTF #3
1930     {
1931         \str_if_eq_x:nnF { }
1932         { \tl_map_function:nN {#2} \__cs_generate_from_signature:n }
1933         {
1934             \__msg_kernel_error:nnx { kernel } { non-base-function }
1935             { \token_to_str:N #5 }
1936         }
1937         \__cs_generate_from_arg_count:NNnn
1938         #5 #4 { \tl_count:n {#2} } {#6}
1939     }
1940     {
1941         \__msg_kernel_error:nnx { kernel } { missing-colon }
1942         { \token_to_str:N #5 }
1943     }
1944 }
1945 \cs_new:Npn \__cs_generate_from_signature:n #1
1946 {
1947     \if:w n #1 \else: \if:w N #1 \else:
1948     \if:w T #1 \else: \if:w F #1 \else: #1 \fi: \fi: \fi: \fi:
1949 }

```

Then we define the 24 variants beginning with N.

```

1950 \__cs_tmp:w { set } { Nn } { Npn }
1951 \__cs_tmp:w { set } { Nx } { Npx }
1952 \__cs_tmp:w { set_nopar } { Nn } { Npn }
1953 \__cs_tmp:w { set_nopar } { Nx } { Npx }
1954 \__cs_tmp:w { set_protected } { Nn } { Npn }
1955 \__cs_tmp:w { set_protected } { Nx } { Npx }
1956 \__cs_tmp:w { set_protected_nopar } { Nn } { Npn }
1957 \__cs_tmp:w { set_protected_nopar } { Nx } { Npx }
1958 \__cs_tmp:w { gset } { Nn } { Npn }
1959 \__cs_tmp:w { gset } { Nx } { Npx }
1960 \__cs_tmp:w { gset_nopar } { Nn } { Npn }
1961 \__cs_tmp:w { gset_nopar } { Nx } { Npx }
1962 \__cs_tmp:w { gset_protected } { Nn } { Npn }
1963 \__cs_tmp:w { gset_protected } { Nx } { Npx }
1964 \__cs_tmp:w { gset_protected_nopar } { Nn } { Npn }
1965 \__cs_tmp:w { gset_protected_nopar } { Nx } { Npx }
1966 \__cs_tmp:w { new } { Nn } { Npn }
1967 \__cs_tmp:w { new } { Nx } { Npx }
1968 \__cs_tmp:w { new_nopar } { Nn } { Npn }
1969 \__cs_tmp:w { new_nopar } { Nx } { Npx }

```

```

1970 \__cs_tmp:w { new_protected }      { Nn } { Npn }
1971 \__cs_tmp:w { new_protected }      { Nx } { Npx }
1972 \__cs_tmp:w { new_protected_nopar } { Nn } { Npn }
1973 \__cs_tmp:w { new_protected_nopar } { Nx } { Npx }

```

(End definition for `\cs_set:Nn` and others. These functions are documented on page 13.)

`\cs_set:cn` The 24 c variants simply use `\exp_args:Nc`.

```

\cs_set:cx
\cs_set:nopar:cn
\cs_set:nopar:cx
\cs_set_protected:cn
\cs_set_protected:cx
\cs_set_protected_nopar:cn
\cs_set_protected_nopar:cx
\cs_gset:cn
\cs_gset:cx
\cs_gset_nopar:cn
\cs_gset_nopar:cx
\cs_gset_protected:cn
\cs_gset_protected:cx
\cs_gset_protected_nopar:cn
\cs_gset_protected_nopar:cx
\cs_new:cn
\cs_new:cx
\cs_new_nopar:cn
\cs_new_nopar:cx
\cs_new_protected:cn
\cs_new_protected:cx
\cs_new_protected_nopar:cn
\cs_new_protected_nopar:cx

```

```

1974 \cs_set:Npn \__cs_tmp:w #1#2
1975 {
1976   \cs_new_protected:cpx { cs_ #1 : c #2 }
1977   {
1978     \exp_not:N \exp_args:Nc
1979     \exp_after:wN \exp_not:N \cs:w cs_ #1 : N #2 \cs_end:
1980   }
1981 }
1982 \__cs_tmp:w { set }           { n }
1983 \__cs_tmp:w { set }           { x }
1984 \__cs_tmp:w { set_nopar }     { n }
1985 \__cs_tmp:w { set_nopar }     { x }
1986 \__cs_tmp:w { set_protected } { n }
1987 \__cs_tmp:w { set_protected } { x }
1988 \__cs_tmp:w { set_protected_nopar } { n }
1989 \__cs_tmp:w { set_protected_nopar } { x }
1990 \__cs_tmp:w { gset }          { n }
1991 \__cs_tmp:w { gset }          { x }
1992 \__cs_tmp:w { gset_nopar }    { n }
1993 \__cs_tmp:w { gset_nopar }    { x }
1994 \__cs_tmp:w { gset_protected } { n }
1995 \__cs_tmp:w { gset_protected } { x }
1996 \__cs_tmp:w { gset_protected_nopar } { n }
1997 \__cs_tmp:w { gset_protected_nopar } { x }
1998 \__cs_tmp:w { new }           { n }
1999 \__cs_tmp:w { new }           { x }
2000 \__cs_tmp:w { new_nopar }     { n }
2001 \__cs_tmp:w { new_nopar }     { x }
2002 \__cs_tmp:w { new_protected } { n }
2003 \__cs_tmp:w { new_protected } { x }
2004 \__cs_tmp:w { new_protected_nopar } { n }
2005 \__cs_tmp:w { new_protected_nopar } { x }

```

(End definition for `\cs_set:cn` and others. These functions are documented on page 13.)

3.16 Checking control sequence equality

`\cs_if_eq_p:NN` Check if two control sequences are identical.

```

\cs_if_eq_p:cN
\cs_if_eq_p:Nc
\cs_if_eq_p:cc
\cs_if_eq:NNTF
\cs_if_eq:cNTF
\cs_if_eq:NcTF
\cs_if_eq:cCTF

```

```

2006 \prg_new_conditional:Npnn \cs_if_eq:NN #1#2 { p , T , F , TF }
2007 {
2008   \if_meaning:w #1#2
2009     \prg_return_true: \else: \prg_return_false: \fi:
2010   }
2011 \cs_new:Npn \cs_if_eq_p:cN { \exp_args:Nc \cs_if_eq_p:NN }
2012 \cs_new:Npn \cs_if_eq:cNTF { \exp_args:Nc \cs_if_eq:NNTF }
2013 \cs_new:Npn \cs_if_eq:cNT { \exp_args:Nc \cs_if_eq:NNT }

```

```

2014 \cs_new:Npn \cs_if_eq:cNF { \exp_args:Nc \cs_if_eq:NNF }
2015 \cs_new:Npn \cs_if_eq_p:Nc { \exp_args:NNc \cs_if_eq_p:NN }
2016 \cs_new:Npn \cs_if_eq:NcTF { \exp_args:NNc \cs_if_eq:NNTF }
2017 \cs_new:Npn \cs_if_eq:NcT { \exp_args:NNc \cs_if_eq:NNT }
2018 \cs_new:Npn \cs_if_eq:NcF { \exp_args:NNc \cs_if_eq:NNF }
2019 \cs_new:Npn \cs_if_eq_p:cc { \exp_args:Ncc \cs_if_eq_p:NN }
2020 \cs_new:Npn \cs_if_eq:ccTF { \exp_args:Ncc \cs_if_eq:NNTF }
2021 \cs_new:Npn \cs_if_eq:ccT { \exp_args:Ncc \cs_if_eq:NNT }
2022 \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 20.)

3.17 Diagnostic functions

`__kernel_register_show:N` Simply using the `\show` primitive does not allow for line-wrapping, so instead use `__msg_show_variable>NNNnn` (defined in `l3msg`). This checks that the variable exists (using `\cs_if_exist:NTF`), then displays the third argument, namely `>~<variable>=<value>`. We expand the value before-hand as otherwise some integers (such as `\currentgrouplevel` or `\currentgroupstyle`) altered by the line-wrapping code would show wrong values.

```

2023 \cs_new_protected:Npn \__kernel_register_show:N #1
2024   { \exp_args:No \__kernel_register_show_aux:nN { \tex_the:D #1 } #1 }
2025 \cs_new_protected:Npn \__kernel_register_show_aux:nN #1#2
2026   {
2027     \__msg_show_variable>NNNnn #2 \cs_if_exist:NTF ? { }
2028     { > ~ \token_to_str:N #2 = #1 }
2029   }
2030 \cs_new_protected:Npn \__kernel_register_show:c
2031   { \exp_args:Nc \__kernel_register_show:N }

```

(End definition for `__kernel_register_show:N` and `__kernel_register_show_aux:n`.)

`__kernel_register_log:N` Redirect the output of `__kernel_register_show:N` to the log.

```

2032 \cs_new_protected:Npn \__kernel_register_log:N
2033   { \__msg_log_next: \__kernel_register_show:N }
2034 \cs_new_protected:Npn \__kernel_register_log:c
2035   { \exp_args:Nc \__kernel_register_log:N }

```

(End definition for `__kernel_register_log:N`.)

`\cs_show:N` 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 `\iow_wrap:nnnN` 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` command also converts its argument to a control sequence within a group to avoid showing `\relax` for undefined control sequences.

```

2036 \cs_new_protected:Npn \cs_show:N #1
2037   {
2038     \group_begin:
2039       \int_set:Nn \tex_escapechar:D { '\\" }
2040       \exp_args:NNx
2041     \group_end:

```

```

2042     \__msg_show_wrap:n { > ~ \token_to_str:N #1 = \cs_meaning:N #1 }
2043   }
2044 \cs_new_protected:Npn \cs_show:c
2045   { \group_begin: \exp_args:NNc \group_end: \cs_show:N }

```

(End definition for `\cs_show:N`. This function is documented on page 16.)

\cs_log:N Use `\cs_show:N` or `\cs_show:c` after calling `__msg_log_next:` to redirect their output
\cs_log:c to the log file only. Note that `\cs_log:c` is not just a variant of `\cs_log:N` as the csname
 should be turned to a control sequence within a group (see `\cs_show:c`).

```

2046 \cs_new_protected:Npn \cs_log:N { \__msg_log_next: \cs_show:N }
2047 \cs_new_protected:Npn \cs_log:c { \__msg_log_next: \cs_show:c }

```

(End definition for `\cs_log:N`. This function is documented on page 16.)

3.18 Doing nothing functions

\prg_do_nothing: This does not fit anywhere else!

```
2048 \cs_new_nopar:Npn \prg_do_nothing: { }
```

(End definition for `\prg_do_nothing:`. This function is documented on page 9.)

3.19 Breaking out of mapping functions

__prg_break_point:Nn In inline mappings, the nesting level must be reset at the end of the mapping, even when
__prg_map_break:Nn 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.

```

2049 \cs_new_eq:NN \__prg_break_point:Nn \use_ii:nn
2050 \cs_new:Npn \__prg_map_break:Nn #1#2#3 \__prg_break_point:Nn #4#5
2051   {
2052     #5
2053     \if_meaning:w #1 #4
2054       \exp_after:wN \use_iii:nnn
2055     \fi:
2056     \__prg_map_break:Nn #1 {#2}
2057   }

```

(End definition for `__prg_break_point:Nn` and `__prg_map_break:Nn`.)

__prg_break_point: Very simple analogues of `__prg_break_point:Nn` and `__prg_map_break:Nn`, for use
__prg_break: in fast short-term recursions which are not mappings, do not need to support nesting,
__prg_break:n and in which nothing has to be done at the end of the loop.

```

2058 \cs_new_eq:NN \__prg_break_point: \prg_do_nothing:
2059 \cs_new:Npn \__prg_break: #1 \__prg_break_point: { }
2060 \cs_new:Npn \__prg_break:n #1#2 \__prg_break_point: {#1}

```

(End definition for `__prg_break_point:`, `__prg_break:`, and `__prg_break:n`.)

```
2061 </initex | package>
```

4 l3expan implementation

```
2062  {*initex | package}
2063  <@@=exp>

\exp_after:wN These are defined in l3basics.
\exp_not:N
\exp_not:n (End definition for \exp_after:wN, \exp_not:N, and \exp_not:n. These functions are documented on page 30.)
```

4.1 General expansion

In this section a general mechanism for defining functions to handle argument handling 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 4.3. In section 4.2 some common cases are coded by a more direct method for efficiency, typically using calls to `\exp_after:wN`.

`\l__exp_internal_t1` This scratch token list variable is defined in `l3basics`, as it is needed “early”. This is just a reminder that is the case!

(End definition for `\l__exp_internal_t1`.)

This code uses internal functions with names that start with `\:::` to perform the expansions. All macros are `long` as this turned out to be desirable 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` `__exp_arg_next:Nnn` `#1` is the result of an expansion step, `#2` is the remaining argument manipulations and `#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 will require 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.

```
2064 \cs_new:Npn \__exp_arg_next:nnn #1#2#3 { #2 \::: { #3 {#1} } }
2065 \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.

```
2066 \cs_new:Npn \::: #1 {#1}
```

(End definition for `\:::`.)

`\:::n` This function is used to skip an argument that doesn’t need to be expanded.

```
2067 \cs_new:Npn \:::n #1 \::: #2#3 { #1 \::: { #2 {#3} } }
```

(End definition for `\:::n`.)

\:::N This function is used to skip an argument that consists of a single token and doesn't need to be expanded.

```
2068 \cs_new:Npn \:::N #1 \::: #2#3 { #1 \::: {#2#3} }
```

(*End definition for \:::N.*)

\:::p This function is used to skip an argument that is delimited by a left brace and doesn't need to be expanded. It should not be wrapped in braces in the result.

```
2069 \cs_new:Npn \:::p #1 \::: #2#3# { #1 \::: {#2#3} }
```

(*End definition for \:::p.*)

\:::c This function is used to skip an argument that is turned into a control sequence without expansion.

```
2070 \cs_new:Npn \:::c #1 \::: #2#3
  { \exp_after:wN \__exp_arg_next:Nnn \cs:w #3 \cs_end: {#1} {#2} }
```

(*End definition for \:::c.*)

\:::o This function is used to expand an argument once.

```
2072 \cs_new:Npn \:::o #1 \::: #2#3
  { \exp_after:wN \__exp_arg_next:nnn \exp_after:wN {#3} {#1} {#2} }
```

(*End definition for \:::o.*)

\:::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 or a space. We introduce `\exp_stop_f:` to mark such an end of expansion marker. In the example shown earlier the scanning was stopped once TeX had fully expanded `\cs_set_eq:Nc \aaa { b \l_tmpa_tl b }` into `\cs_set_eq:NN \aaa = \blurb` which then turned out to contain the non-expandable token `\cs_set_eq:NN`. Since the expansion of `\exp:w \exp_end_continue_f:w` is `<null>`, 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.

```
2074 \cs_new:Npn \:::f #1 \::: #2#3
  {
 2075   \exp_after:wN \__exp_arg_next:nn
 2076   \exp_after:wN { \exp:w \exp_end_continue_f:w #3 }
 2077   {#1} {#2}
 2078 }
 2079 \use:nn { \cs_new_eq:NN \exp_stop_f: } { ~ }
```

(*End definition for \:::f and \exp_stop_f:.*)

\:::x This function is used to expand an argument fully.

```
2081 \cs_new_protected:Npn \:::x #1 \::: #2#3
  {
 2082   \cs_set_nopar:Npx \l__exp_internal_tl { {#3} }
 2083   \exp_after:wN \__exp_arg_next:nnn \l__exp_internal_tl {#1} {#2}
 2084 }
 2085 }
```

(*End definition for \:::x.*)

\::v These functions return the value of a register, i.e., one of `t1`, `cclist`, `int`, `skip`, `dim` and **\::V** `muskip`. The V version expects a single token whereas v like c creates a csname 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 will terminate using `\exp_end:`. The argument is returned in braces.

```

2086 \cs_new:Npn \::V #1 :::: #2#3
2087 {
2088   \exp_after:wN \__exp_arg_next:nnn
2089   \exp_after:wN { \exp:w \__exp_eval_register:N #3 }
2090   {#1} {#2}
2091 }
2092 \cs_new:Npn \::v # 1::: #2#3
2093 {
2094   \exp_after:wN \__exp_arg_next:nnn
2095   \exp_after:wN { \exp:w \__exp_eval_register:c {#3} }
2096   {#1} {#2}
2097 }
```

(End definition for `\::v` and `\::V`.)

`__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 do here is try to find out whether the token will expand to something else when hit with `\exp_after:wN`. The technique is to compare the meaning of the register in question when it has been prefixed with `\exp_not:N` and the register itself. If it is a macro, the prefixed `\exp_not:N` will temporarily turn it into the primitive `\scan_stop:`.

```

2098 \cs_new:Npn \__exp_eval_register:N #1
2099 {
2100   \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.

```

2101   \if_meaning:w \scan_stop: #1
2102     \__exp_eval_error_msg:w
2103   \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`.

```

2104   \else:
2105     \exp_after:wN \use_i_i:nnn
2106   \fi:
```

```

2107      \exp_after:wN \exp_end: \tex_the:D #1
2108    }
2109 \cs_new:Npn \__exp_eval_register:c #1
2110   { \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}

2111 \cs_new:Npn \__exp_eval_error_msg:w #1 \tex_the:D #2
2112   {
2113     \fi:
2114     \fi:
2115     \__msg_kernel_expandable_error:n { kernel } { bad-variable } {#2}
2116     \exp_end:
2117   }

```

(End definition for `__exp_eval_register:N` and `__exp_eval_error_msg:w`.)

4.2 Hand-tuned definitions

One of the most important features of these functions is that they are fully expandable and therefore allow to prefix them with `\tex_global:D` for example.

`\exp_args:No` Those lovely runs of expansion!

```

2118 \cs_new:Npn \exp_args:No #1#2 { \exp_after:wN #1 \exp_after:wN {#2} }
2119 \cs_new:Npn \exp_args:NNo #1#2#3
2120   { \exp_after:wN #1 \exp_after:wN #2 \exp_after:wN {#3} }
2121 \cs_new:Npn \exp_args:NNNo #1#2#3#4
2122   { \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 27.)

`\exp_args:Nc` In `l3basics`.

`\exp_args:cc` (End definition for `\exp_args:Nc` and `\exp_args:cc`. These functions are documented on page 27.)

`\exp_args:NNc` Here are the functions that turn their argument into csnames but are expandable.

```

2123 \cs_new:Npn \exp_args:NNc #1#2#3
2124   { \exp_after:wN #1 \exp_after:wN #2 \cs:w # 3\cs_end: }
2125 \cs_new:Npn \exp_args:Ncc #1#2#3
2126   { \exp_after:wN #1 \cs:w #2 \exp_after:wN \cs_end: \cs:w #3 \cs_end: }
2127 \cs_new:Npn \exp_args:Nccc #1#2#3#4
2128   {
2129     \exp_after:wN #1
2130     \cs:w #2 \exp_after:wN \cs_end:
2131     \cs:w #3 \exp_after:wN \cs_end:
2132     \cs:w #4 \cs_end:
2133   }

```

(End definition for `\exp_args:Nc`, `\exp_args:Ncc`, and `\exp_args:Nccc`. These functions are documented on page 28.)

```
\exp_args:Nf
\exp_args:NV
\exp_args:Nv
2134 \cs_new:Npn \exp_args:Nf #1#2
2135   { \exp_after:wN #1 \exp_after:wN { \exp:w \exp_end_continue_f:w #2 } }
2136 \cs_new:Npn \exp_args:Nv #1#2
2137   {
2138     \exp_after:wN #1 \exp_after:wN
2139       { \exp:w \__exp_eval_register:c {#2} }
2140   }
2141 \cs_new:Npn \exp_args:NV #1#2
2142   {
2143     \exp_after:wN #1 \exp_after:wN
2144       { \exp:w \__exp_eval_register:N #2 }
2145   }
```

(End definition for `\exp_args:Nf`, `\exp_args:NV`, and `\exp_args:Nv`. These functions are documented on page 27.)

`\exp_args>NNV` 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.

```
\exp_args:NNV
\exp_args:NNv
\exp_args:NNf
\exp_args:Ncf
\exp_args:Nco
2146 \cs_new:Npn \exp_args>NNf #1#2#3
2147   {
2148     \exp_after:wN #1
2149     \exp_after:wN #2
2150     \exp_after:wN { \exp:w \exp_end_continue_f:w #3 }
2151   }
2152 \cs_new:Npn \exp_args>NNv #1#2#3
2153   {
2154     \exp_after:wN #1
2155     \exp_after:wN #2
2156     \exp_after:wN { \exp:w \__exp_eval_register:c {#3} }
2157   }
2158 \cs_new:Npn \exp_args>NNV #1#2#3
2159   {
2160     \exp_after:wN #1
2161     \exp_after:wN #2
2162     \exp_after:wN { \exp:w \__exp_eval_register:N #3 }
2163   }
2164 \cs_new:Npn \exp_args:Nco #1#2#3
2165   {
2166     \exp_after:wN #1
2167     \cs:w #2 \exp_after:wN \cs_end:
2168     \exp_after:wN {#3}
2169   }
2170 \cs_new:Npn \exp_args:Ncf #1#2#3
2171   {
2172     \exp_after:wN #1
2173     \cs:w #2 \exp_after:wN \cs_end:
2174     \exp_after:wN { \exp:w \exp_end_continue_f:w #3 }
2175   }
2176 \cs_new:Npn \exp_args:NNV #1#2#3
```

```

2177   {
2178     \exp_after:wN #1
2179     \exp_after:wN { \exp:w \exp_after:wN
2180       \_\_exp_eval_register:N \exp_after:wN #2 \exp_after:wN }
2181     \exp_after:wN { \exp:w \_\_exp_eval_register:N #3 }
2182   }

```

(End definition for `\exp_args:NNV` and others. These functions are documented on page 28.)

`\exp_args:Ncc0` A few more that we can hand-tune.

```

2183 \cs_new:Npn \exp_args:NNNV #1#2#3#4
2184   {
2185     \exp_after:wN #1
2186     \exp_after:wN #2
2187     \exp_after:wN #3
2188     \exp_after:wN { \exp:w \_\_exp_eval_register:N #4 }
2189   }
2190 \cs_new:Npn \exp_args:NcNc #1#2#3#4
2191   {
2192     \exp_after:wN #1
2193     \cs:w #2 \exp_after:wN \cs_end:
2194     \exp_after:wN #3
2195     \cs:w #4 \cs_end:
2196   }
2197 \cs_new:Npn \exp_args:NcNo #1#2#3#4
2198   {
2199     \exp_after:wN #1
2200     \cs:w #2 \exp_after:wN \cs_end:
2201     \exp_after:wN #3
2202     \exp_after:wN {#4}
2203   }
2204 \cs_new:Npn \exp_args:Ncco #1#2#3#4
2205   {
2206     \exp_after:wN #1
2207     \cs:w #2 \exp_after:wN \cs_end:
2208     \cs:w #3 \exp_after:wN \cs_end:
2209     \exp_after:wN {#4}
2210   }

```

(End definition for `\exp_args:Ncc0` and others. These functions are documented on page 28.)

4.3 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 are all not long: they don't actually take any arguments themselves.

```

\exp_args:Nx
2211 \cs_new_protected:Npn \exp_args:Nx { \:::x \::: }

```

(End definition for `\exp_args:Nx`. This function is documented on page 27.)

\exp_args:Nnc Here are the actual function definitions, using the helper functions above.

```

2212 \cs_new:Npn \exp_args:Nnc { \::n \::c \::: }
2213 \cs_new:Npn \exp_args:Nfo { \::f \::o \::: }
2214 \cs_new:Npn \exp_args:Nff { \::f \::f \::: }
2215 \cs_new:Npn \exp_args:Nnf { \::n \::f \::: }
2216 \cs_new:Npn \exp_args:Nno { \::n \::o \::: }
2217 \cs_new:Npn \exp_args:Nv { \::n \::V \::: }
2218 \cs_new:Npn \exp_args:Noo { \::o \::o \::: }
2219 \cs_new:Npn \exp_args:Nof { \::o \::f \::: }
2220 \cs_new:Npn \exp_args:Noc { \::o \::c \::: }
2221 \cs_new_protected:Npn \exp_args>NNx { \::N \::x \::: }
2222 \cs_new_protected:Npn \exp_args:Ncx { \::c \::x \::: }
2223 \cs_new_protected:Npn \exp_args:Nnx { \::n \::x \::: }
2224 \cs_new_protected:Npn \exp_args:Nox { \::o \::x \::: }
2225 \cs_new_protected:Npn \exp_args:Nxo { \::x \::o \::: }
2226 \cs_new_protected:Npn \exp_args:Nxx { \::x \::x \::: }
```

(End definition for \exp_args:Nnc and others. These functions are documented on page 28.)

```

\exp_args:NNno
\exp_args:NNoo
\exp_args:NNnc
\exp_args:Nnno
\exp_args:Nooo
\exp_args:NNNx
\exp_args:NNnx
\exp_args:NNox
\exp_args:Nnnx
\exp_args:Nnox
\exp_args:Nccx
\exp_args:Ncnx
\exp_args:Noox

2227 \cs_new:Npn \exp_args:NNno { \::N \::n \::o \::: }
2228 \cs_new:Npn \exp_args:NNoo { \::N \::o \::o \::: }
2229 \cs_new:Npn \exp_args:NNnc { \::n \::n \::c \::: }
2230 \cs_new:Npn \exp_args:Nnno { \::n \::n \::o \::: }
2231 \cs_new:Npn \exp_args:Nooo { \::o \::o \::o \::: }
2232 \cs_new_protected:Npn \exp_args:NNNx { \::N \::N \::x \::: }
2233 \cs_new_protected:Npn \exp_args:NNnx { \::N \::n \::x \::: }
2234 \cs_new_protected:Npn \exp_args:NNox { \::N \::o \::x \::: }
2235 \cs_new_protected:Npn \exp_args:Nnnx { \::n \::n \::x \::: }
2236 \cs_new_protected:Npn \exp_args:Nnox { \::n \::o \::x \::: }
2237 \cs_new_protected:Npn \exp_args:Nccx { \::c \::c \::x \::: }
2238 \cs_new_protected:Npn \exp_args:Ncnx { \::c \::n \::x \::: }
2239 \cs_new_protected:Npn \exp_args:Noox { \::o \::o \::x \::: }
```

(End definition for \exp_args:NNno and others. These functions are documented on page 28.)

4.4 Last-unbraced versions

There are a few places where the last argument needs to be available unbraced. First some helper macros.

```

__exp_arg_last_unbraced:nn
  \::f_unbraced
  \::o_unbraced
  \::V_unbraced
  \::v_unbraced
  \::x_unbraced
  \::x_unbraced

2240 \cs_new:Npn __exp_arg_last_unbraced:nn #1#2 { #2#1 }
2241 \cs_new:Npn \::f_unbraced \::: #1#2
2242 {
2243   \exp_after:wN __exp_arg_last_unbraced:nn
2244   \exp_after:wN { \exp:w \exp_end_continue_f:w #2 } {#1}
2245 }
2246 \cs_new:Npn \::o_unbraced \::: #1#2
2247 { \exp_after:wN __exp_arg_last_unbraced:nn \exp_after:wN {#2} {#1} }
2248 \cs_new:Npn \::V_unbraced \::: #1#2
2249 {
2250   \exp_after:wN __exp_arg_last_unbraced:nn
2251   \exp_after:wN { \exp:w __exp_eval_register:N #2 } {#1}
2252 }
2253 \cs_new:Npn \::v_unbraced \::: #1#2
```

```

2254 {
2255   \exp_after:wN \__exp_arg_last_unbraced:nn
2256   \exp_after:wN { \exp:w \__exp_eval_register:c {#2} } {#1}
2257 }
2258 \cs_new_protected:Npn \::x_unbraced \::: #1#2
2259 {
2260   \cs_set_nopar:Npx \l__exp_internal_tl { \exp_not:n {#1} #2 }
2261   \l__exp_internal_tl
2262 }

```

(End definition for `__exp_arg_last_unbraced:nn` and others.)

\exp_last_unbraced:NV Now the business end: most of these are hand-tuned for speed, but the general system is in place.

```

2263 \cs_new:Npn \exp_last_unbraced:NV #1#2
2264 { \exp_after:wN #1 \exp:w \__exp_eval_register:N #2 }
2265 \cs_new:Npn \exp_last_unbraced:Nv #1#2
2266 { \exp_after:wN #1 \exp:w \__exp_eval_register:c {#2} }
2267 \cs_new:Npn \exp_last_unbraced:No #1#2 { \exp_after:wN #1 #2 }
2268 \cs_new:Npn \exp_last_unbraced:Nf #1#2
2269 { \exp_after:wN #1 \exp:w \exp_end_continue_f:w #2 }
2270 \cs_new:Npn \exp_last_unbraced:Nco #1#2#3
2271 { \exp_after:wN #1 \cs:w #2 \exp_after:wN \cs_end: #3 }
2272 \cs_new:Npn \exp_last_unbraced:NcV #1#2#3
2273 {
2274   \exp_after:wN #1
2275   \cs:w #2 \exp_after:wN \cs_end:
2276   \exp:w \__exp_eval_register:N #3
2277 }
2278 \cs_new:Npn \exp_last_unbraced:NNV #1#2#3
2279 {
2280   \exp_after:wN #1
2281   \exp_after:wN #2
2282   \exp:w \__exp_eval_register:N #3
2283 }
2284 \cs_new:Npn \exp_last_unbraced:NNo #1#2#3
2285 { \exp_after:wN #1 \exp_after:wN #2 #3 }
2286 \cs_new:Npn \exp_last_unbraced:NNNV #1#2#3#4
2287 {
2288   \exp_after:wN #1
2289   \exp_after:wN #2
2290   \exp_after:wN #3
2291   \exp:w \__exp_eval_register:N #4
2292 }
2293 \cs_new:Npn \exp_last_unbraced:NNNo #1#2#3#4
2294 { \exp_after:wN #1 \exp_after:wN #2 \exp_after:wN #3 #4 }
2295 \cs_new:Npn \exp_last_unbraced:Nno { ::n ::o_unbraced \::: }
2296 \cs_new:Npn \exp_last_unbraced:Noo { ::o ::o_unbraced \::: }
2297 \cs_new:Npn \exp_last_unbraced:Nfo { ::f ::o_unbraced \::: }
2298 \cs_new:Npn \exp_last_unbraced:NnNo { ::n ::N ::o_unbraced \::: }
2299 \cs_new_protected:Npn \exp_last_unbraced:Nx { ::x_unbraced \::: }

```

(End definition for `\exp_last_unbraced:NV` and others. These functions are documented on page 29.)

\exp_last_two_unbraced:Noo If #2 is a single token then this can be implemented as

`__exp_last_two_unbraced:noN`

```
\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.

```
2300 \cs_new:Npn \exp_last_two_unbraced:Noo #1#2#3
2301   { \exp_after:wN \__exp_last_two_unbraced:noN \exp_after:wN {#3} {#2} #1 }
2302 \cs_new:Npn \__exp_last_two_unbraced:noN #1#2#3
2303   { \exp_after:wN #3 #2 #1 }
```

(End definition for `\exp_last_two_unbraced:Noo` and `_exp_last_two_unbraced:noN`. These functions are documented on page [29](#).)

4.5 Preventing expansion

```
\exp_not:o
\exp_not:c
\exp_not:f
\exp_not:V
\exp_not:v

2304 \cs_new:Npn \exp_not:o #1 { \etex_unexpanded:D \exp_after:wN {#1} }
2305 \cs_new:Npn \exp_not:c #1 { \exp_after:wN \exp_not:N \cs:w #1 \cs_end: }
2306 \cs_new:Npn \exp_not:f #1
2307 { \etex_unexpanded:D \exp_after:wN { \exp:w \exp_end_continue_f:w #1 } }
2308 \cs_new:Npn \exp_not:V #1
2309 {
2310     \etex_unexpanded:D \exp_after:wN
2311     { \exp:w \__exp_eval_register:N #1 }
2312 }
2313 \cs_new:Npn \exp_not:v #1
2314 {
2315     \etex_unexpanded:D \exp_after:wN
2316     { \exp:w \__exp_eval_register:c {#1} }
2317 }
```

(End definition for \exp_{not:o} and others. These functions are documented on page 31.)

4.6 Controlled expansion

`\exp:w` To trigger a sequence of “arbitrary” many expansions we need a method to invoke `TEX`’s expansion mechanism in such a way that a) we are able to stop it in a controlled manner and b) that 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 `TEX`; 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_roman numeral: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.

2318 %\cs_new_eq:NN \exp:w \tex_roman numeral:D

So 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 will immediately stop \tex{romannumeral:D}'s search for a number.

`%\int const:Np \exp end: { 0 }`

(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 TeX’s syntax for a `\{number\}` will continue searching for an optional space (and it will continue expansion doing that) — see TeXbook page 269 for details.

```
2320 \tex_catcode:D '^^@=13
2321 \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.⁷

```
2322 \cs_new:Npn ^^@{\expnERROR}
2323 \cs_new:Npn \exp_end_continue_f:nw #1 { '^^@ #1 }
2324 \tex_catcode:D '^^@=15
```

(End definition for `\exp:w` and others. These functions are documented on page 31.)

4.7 Defining function variants

```
2325 <@=cs>
```

`\cs_generate_variant:Nn` #1 : Base form of a function; e.g., `\tl_set:Nn`
#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.

```
2326 \cs_new_protected:Npn \cs_generate_variant:Nn #1#2
2327 {
2328     \_chk_if_exist_cs:N #1
2329     \_cs_generate_variant:N #1
2330     \exp_after:wN \_cs_split_function:NN
2331     \exp_after:wN #1
2332     \exp_after:wN \_cs_generate_variant:nnNN
2333     \exp_after:wN #1
2334     \tl_to_str:n {#2} , \scan_stop: , \q_recursion_stop
2335 }
```

(End definition for `\cs_generate_variant:Nn`. This function is documented on page 25.)

`_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 all primitive TeX conditionals are expandable.

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 fist occurrence of `macro`. The `ww` auxiliary removes everything in the meaning string after the first `ma`. We

⁷Need to get a real error message.

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 `\q_mark` is taken as an argument of the `wwNw` auxiliary, and #3 is `\cs_new_protected:Npx`, otherwise it is `\cs_new:Npx`.

```

2336 \cs_new_protected:Npx \__cs_generate_variant:N #1
2337 {
2338   \exp_not:N \exp_after:wN \exp_not:N \if_meaning:w
2339   \exp_not:N \exp_not:N #1 #1
2340   \cs_set_eq:NN \exp_not:N \__cs_tmp:w \cs_new_protected:Npx
2341   \exp_not:N \else:
2342     \exp_not:N \exp_after:wN \exp_not:N \__cs_generate_variant:ww
2343     \exp_not:N \token_to_meaning:N #1 \tl_to_str:n { ma }
2344     \exp_not:N \q_mark
2345     \exp_not:N \q_mark \cs_new_protected:Npx
2346     \tl_to_str:n { pr }
2347     \exp_not:N \q_mark \cs_new:Npx
2348     \exp_not:N \q_stop
2349     \exp_not:N \fi:
2350   }
2351 \use:x
2352 {
2353   \cs_new_protected:Npn \exp_not:N \__cs_generate_variant:ww
2354   ##1 \tl_to_str:n { ma } ##2 \exp_not:N \q_mark
2355 }
2356 { \__cs_generate_variant:wwNw #1 }
2357 \use:x
2358 {
2359   \cs_new_protected:Npn \exp_not:N \__cs_generate_variant:wwNw
2360   ##1 \tl_to_str:n { pr } ##2 \exp_not:N \q_mark
2361   ##3 ##4 \exp_not:N \q_stop
2362 }
2363 { \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.

```

2364 \cs_new_protected:Npn \__cs_generate_variant:nnNN #1#2#3#4
2365 {
2366   \if_meaning:w \c_false_bool #3
2367     \__msg_kernel_error:nnx { kernel } { missing-colon }
2368     { \token_to_str:c {#1} }
2369     \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
2370   \fi:
```

```

2371     \__cs_generate_variant:Nnnw #4 {#1}{#2}
2372 }

```

(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}, it would be better to 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 should be defined using \exp_args:Nnx, not \exp_args:Nox, to avoid double o expansion.
- Lastly, \cs_generate_variant:Nn \foo:on {xn} should trigger an error, because we do not have a means to replace o-expansion by x-expansion.

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 two cases: 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 in the form ⟨processed variant signature⟩ \q_mark ⟨errors⟩ \q_stop ⟨base function⟩ ⟨new function⟩. If all went well, ⟨errors⟩ is empty; otherwise, it is a kernel error message, followed by some clean-up code (\use_none:nnnn).

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.

```

2373 \cs_new_protected:Npn \__cs_generate_variant:Nnnw #1#2#3#4 ,
2374 {
2375     \if_meaning:w \scan_stop: #4
2376     \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
2377     \fi:
2378     \use:x
2379     {
2380         \exp_not:N \__cs_generate_variant:wwNN
2381         \__cs_generate_variant_loop:nNwN { }
2382         #4
2383         \__cs_generate_variant_loop_end:nwwwNNnn
2384         \q_mark
2385         #3 ~
2386         { ~ { } \fi: \__cs_generate_variant_loop_long:wNNnn } ~

```

```

2387     { }
2388     \q_stop
2389     \exp_not:N #1 {#2} {#4}
2390   }
2391   \__cs_generate_variant:Nnnw #1 {#2} {#3}
2392 }

```

(End definition for `__cs_generate_variant:Nnnw`.)

```

\__cs_generate_variant_loop:nNwN
\__cs_generate_variant_loop_same:w
\__cs_generate_variant_loop_end:nwwwNNnn
\__cs_generate_variant_loop_long:wNNnn
\__cs_generate_variant_loop_invalid>NNwNNnn

```

#1 : Last few (consecutive) letters common between the base and variant (in fact, `__cs_generate_variant_same:N` *<letter>* for each letter).

#2 : Next variant letter.

#3 : Remainder of variant form.

#4 : Next base letter.

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 with a base letter of `N` or `n`. 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 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 neither `n` nor `N`. Again, 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.

```

2393 \cs_new:Npn \__cs_generate_variant_loop:nNwN #1#2#3 \q_mark #4
2394 {
2395   \if:w #2 #4
2396     \exp_after:wN \__cs_generate_variant_loop_same:w
2397   \else:

```

```

2398 \if:w N #4 \else:
2399   \if:w n #4 \else:
2400     \__cs_generate_variant_loop_invalid>NNwNNnn #4#2
2401   \fi:
2402   \fi:
2403 \fi:
2404 #1
2405 \prg_do_nothing:
2406 #2
2407 \__cs_generate_variant_loop:nNwN { } #3 \q_mark
2408 }
2409 \cs_new:Npn \__cs_generate_variant_loop_same:w
2410   #1 \prg_do_nothing: #2#3#4
2411 {
2412   #3 { #1 \__cs_generate_variant_same:N #2 }
2413 }
2414 \cs_new:Npn \__cs_generate_variant_loop_end:nwwwNNnn
2415   #1#2 \q_mark #3 ~ #4 \q_stop #5#6#7#8
2416 {
2417   \scan_stop: \scan_stop: \fi:
2418   \exp_not:N \q_mark
2419   \exp_not:N \q_stop
2420   \exp_not:N #6
2421   \exp_not:c { #7 : #8 #1 #3 }
2422 }
2423 \cs_new:Npn \__cs_generate_variant_loop_long:wNNnn #1 \q_stop #2#3#4#5
2424 {
2425   \exp_not:n
2426   {
2427     \q_mark
2428     \__msg_kernel_error:nnxx { kernel } { variant-too-long }
2429     {#5} { \token_to_str:N #3 }
2430     \use_none:nnnn
2431     \q_stop
2432     #3
2433     #3
2434   }
2435 }
2436 \cs_new:Npn \__cs_generate_variant_loop_invalid>NNwNNnn
2437   #1#2 \fi: \fi: \fi: #3 \q_stop #4#5#6#7
2438 {
2439   \fi: \fi: \fi:
2440   \exp_not:n
2441   {
2442     \q_mark
2443     \__msg_kernel_error:nnxxxx { kernel } { invalid-variant }
2444     {#7} { \token_to_str:N #5 } {#1} {#2}
2445     \use_none:nnnn
2446     \q_stop
2447     #5
2448     #5
2449   }
2450 }

```

(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.

```
2451 \cs_new:Npn \_\_cs_generate_variant\_same:N #1
2452 {
2453     \if:w N #1
2454     N
2455     \else:
2456         \if:w p #1
2457         p
2458         \else:
2459             n
2460             \fi:
2461         \fi:
2462 }
```

(End definition for `__cs_generate_variant_same:N`.)

```
\_\_cs_generate_variant:wwNN
```

If the variant form has already been defined, log its existence. 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.

```
2463 \cs_new_protected:Npn \_\_cs_generate_variant:wwNN
2464     #1 \q_mark #2 \q_stop #3#4
2465 {
2466     #2
2467     \cs_if_free:NTF #4
2468     {
2469         \group_begin:
2470             \_\_cs_generate_internal_variant:n {#1}
2471             \_\_cs_tmp:w #4 { \exp_not:c { \exp_args:N #1 } \exp_not:N #3 }
2472         \group_end:
2473     }
2474     {
2475         \_\_chk_log:x
2476         {
2477             Variant~\token_to_str:N #4~%
2478             already~defined;~ not~ changing~ it~ \msg_line_context:
2479         }
2480     }
2481 }
```

(End definition for `__cs_generate_variant:wwNN`.)

```
\_\_cs_generate_internal_variant:n
\_\_cs_generate_internal_variant:wwN
\_\_cs_generate_internal_variant_loop:n
```

Test if `\exp_args:N #1` is already defined and if not define it via the `\:::` commands using the chars in `#1`. If `#1` contains an `x` (this is the place where having converted the original comma-list argument to a string is very important), the result should be protected, and the next variant to be defined using that internal variant should be protected.

```
2482 \cs_new_protected:Npx \_\_cs_generate_internal_variant:n #1
2483 {
2484     \exp_not:N \_\_cs_generate_internal_variant:wwNnn
2485     #1 \exp_not:N \q_mark
2486     { \cs_set_eq:NN \exp_not:N \_\_cs_tmp:w \cs_new_protected:Npx }
2487     \cs_new_protected:cpx
```

```

2488     \token_to_str:N x \exp_not:N \q_mark
2489     {
2490     \cs_new:cpx
2491     \exp_not:N \q_stop
2492     { \exp_args:N #1 }
2493     {
2494     \exp_not:N \__cs_generate_internal_variant_loop:n #1
2495     { : \exp_not:N \use_i:nn }
2496     }
2497   }
2498 \use:x
2499   {
2500   \cs_new_protected:Npn \exp_not:N \__cs_generate_internal_variant:wwnNwnn
2501     ##1 \token_to_str:N x ##2 \exp_not:N \q_mark
2502     ##3 ##4 ##5 \exp_not:N \q_stop ##6 ##7
2503   }
2504   {
2505     #3
2506     \cs_if_free:cT {#6} { #4 {#6} {#7} }
2507   }

```

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.

```

2508 \cs_new:Npn \__cs_generate_internal_variant_loop:n #1
2509   {
2510     \exp_after:wN \exp_not:N \cs:w :: #1 \cs_end:
2511     \__cs_generate_internal_variant_loop:n
2512   }

```

(End definition for `__cs_generate_internal_variant:n`, `__cs_generate_internal_variant:wwn`, and `__cs_generate_internal_variant_loop:n`.)

```
2513 </initex | package>
```

5 l3tl implementation

```

2514 <*initex | package>
2515 <@@=tl>

```

A token list variable is a TeX macro that holds tokens. By using the ε -TeX primitive `\unexpanded` inside a TeX `\edef` it is possible to store any tokens, including `#`, in this way.

5.1 Functions

\tl_new:N Creating new token list variables is a case of checking for an existing definition and doing the definition.

```

2516 \cs_new_protected:Npn \tl_new:N #1
2517   {
2518     \__chk_if_free_cs:N #1
2519     \cs_gset_eq:NN #1 \c_empty_tl
2520   }
2521 \cs_generate_variant:Nn \tl_new:N { c }

```

(End definition for `\tl_new:N`. This function is documented on page 34.)

\tl_const:Nn Constants are also easy to generate.

```
2522 \cs_new_protected:Npn \tl_const:Nn #1#2
2523 {
2524     \__chk_if_free_cs:N #1
2525     \cs_gset_nopar:Npx #1 { \exp_not:n {#2} }
2526 }
2527 \cs_new_protected:Npn \tl_const:Nx #1#2
2528 {
2529     \__chk_if_free_cs:N #1
2530     \cs_gset_nopar:Npx #1 {#2}
2531 }
2532 \cs_generate_variant:Nn \tl_const:Nn { c }
2533 \cs_generate_variant:Nn \tl_const:Nx { c }
```

(End definition for `\tl_const:Nn`. This function is documented on page 34.)

\tl_clear:N Clearing a token list variable means setting it to an empty value. Error checking will be sorted out by the parent function.

```
2534 \cs_new_protected:Npn \tl_clear:N #1
2535 { \tl_set_eq:NN #1 \c_empty_tl }
2536 \cs_new_protected:Npn \tl_gclear:N #1
2537 { \tl_gset_eq:NN #1 \c_empty_tl }
2538 \cs_generate_variant:Nn \tl_clear:N { c }
2539 \cs_generate_variant:Nn \tl_gclear:N { c }
```

(End definition for `\tl_clear:N` and `\tl_gclear:N`. These functions are documented on page 34.)

\tl_clear_new:N Clearing a token list variable means setting it to an empty value. Error checking will be sorted out by the parent function.

```
2540 \cs_new_protected:Npn \tl_clear_new:N #1
2541 { \tl_if_exist:NTF #1 { \tl_clear:N #1 } { \tl_new:N #1 } }
2542 \cs_new_protected:Npn \tl_gclear_new:N #1
2543 { \tl_if_exist:NTF #1 { \tl_gclear:N #1 } { \tl_new:N #1 } }
2544 \cs_generate_variant:Nn \tl_clear_new:N { c }
2545 \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 35.)

\tl_set_eq:NN For setting token list variables equal to each other.

```
2546 \cs_new_eq:NN \tl_set_eq:NN \cs_set_eq:NN
2547 \cs_new_eq:NN \tl_set_eq:cN \cs_set_eq:cN
2548 \cs_new_eq:NN \tl_set_eq:Nc \cs_set_eq:Nc
2549 \cs_new_eq:NN \tl_set_eq:cc \cs_set_eq:cc
2550 \cs_new_eq:NN \tl_gset_eq:NN \cs_gset_eq:NN
2551 \cs_new_eq:NN \tl_gset_eq:cN \cs_gset_eq:cN
2552 \cs_new_eq:NN \tl_gset_eq:Nc \cs_gset_eq:Nc
2553 \cs_new_eq:NN \tl_gset_eq:cc \cs_gset_eq:cc
```

(End definition for `\tl_set_eq:NN` and `\tl_gset_eq:NN`. These functions are documented on page 35.)

\tl_concat:NNN Concatenating token lists is easy.

```

2554 \cs_new_protected:Npn \tl_concat:NNN #1#2#3
2555   { \tl_set:Nx #1 { \exp_not:o {#2} \exp_not:o {#3} } }
2556 \cs_new_protected:Npn \tl_gconcat:NNN #1#2#3
2557   { \tl_gset:Nx #1 { \exp_not:o {#2} \exp_not:o {#3} } }
2558 \cs_generate_variant:Nn \tl_concat:NNN { ccc }
2559 \cs_generate_variant:Nn \tl_gconcat:NNN { ccc }

(End definition for \tl_concat:NNN and \tl_gconcat:NNN. These functions are documented on page 35.)
```

\tl_if_exist_p:N Copies of the `cs` functions defined in `l3basics`.

```

2560 \prg_new_eq_conditional:Nnn \tl_if_exist:N \cs_if_exist:N { TF , T , F , p }
2561 \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 35.)
```

5.2 Constant token lists

\c_empty_tl Never full. We need to define that constant before using `\tl_new:N`.

```
2562 \tl_const:Nn \c_empty_tl { }
```

(End definition for `\c_empty_tl`. This variable is documented on page 46.)

\c_space_tl A space as a token list (as opposed to as a character).

```
2563 \tl_const:Nn \c_space_tl { ~ }
```

(End definition for `\c_space_tl`. This variable is documented on page 46.)

5.3 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:N` version is done “by hand” as it is used quite a lot.

```

2564 \cs_new_protected:Npn \tl_set:Nn #1#2
2565   { \cs_set_nopar:Npx #1 { \exp_not:n {#2} } }
2566 \cs_new_protected:Npn \tl_set:N #1#2
2567   { \cs_set_nopar:Npx #1 { \exp_not:o {#2} } }
2568 \cs_new_protected:Npn \tl_set:Nx #1#2
2569   { \cs_set_nopar:Npx #1 {#2} }
2570 \cs_new_protected:Npn \tl_gset:Nn #1#2
2571   { \cs_gset_nopar:Npx #1 { \exp_not:n {#2} } }
2572 \cs_new_protected:Npn \tl_gset:N #1#2
2573   { \cs_gset_nopar:Npx #1 { \exp_not:o {#2} } }
\tl_gset:Nn
2574 \cs_new_protected:Npn \tl_gset:Nx #1#2
2575   { \cs_gset_nopar:Npx #1 {#2} }
2576 \cs_generate_variant:Nn \tl_set:Nn { NV , Nv , Nf }
2577 \cs_generate_variant:Nn \tl_set:Nx { c }
2578 \cs_generate_variant:Nn \tl_set:Nn { c, co , cV , cv , cf }
2579 \cs_generate_variant:Nn \tl_gset:Nn { NV , Nv , Nf }
2580 \cs_generate_variant:Nn \tl_gset:Nx { c }
2581 \cs_generate_variant:Nn \tl_gset:Nn { c, co , cV , cv , cf }

(End definition for \tl_set:Nn and \tl_gset:Nn. These functions are documented on page 35.)
```

\tl_put_left:Nn

```
2582 \cs_new_protected:Npn \tl_put_left:Nn #1#2
2583   { \cs_set_nopar:Npx #1 { \exp_not:n {#2} \exp_not:o #1 } }
2584 \cs_new_protected:Npn \tl_put_left:NV #1#2
2585   { \cs_set_nopar:Npx #1 { \exp_not:V #2 \exp_not:o #1 } }
2586 \cs_new_protected:Npn \tl_put_left:No #1#2
2587   { \cs_set_nopar:Npx #1 { \exp_not:o {#2} \exp_not:o #1 } }
2588 \cs_new_protected:Npn \tl_put_left:Nx #1#2
2589   { \cs_set_nopar:Npx #1 { #2 \exp_not:o #1 } }
2590 \cs_new_protected:Npn \tl_gput_left:Nn #1#2
2591   { \cs_gset_nopar:Npx #1 { \exp_not:n {#2} \exp_not:o #1 } }
2592 \cs_new_protected:Npn \tl_gput_left:NV #1#2
2593   { \cs_gset_nopar:Npx #1 { \exp_not:V #2 \exp_not:o #1 } }
2594 \cs_new_protected:Npn \tl_gput_left:No #1#2
2595   { \cs_gset_nopar:Npx #1 { \exp_not:o {#2} \exp_not:o #1 } }
2596 \cs_new_protected:Npn \tl_gput_left:Nx #1#2
2597   { \cs_gset_nopar:Npx #1 { #2 \exp_not:o {#1} } }
2598 \cs_generate_variant:Nn \tl_put_left:Nn { c }
2599 \cs_generate_variant:Nn \tl_put_left:NV { c }
2600 \cs_generate_variant:Nn \tl_put_left:No { c }
2601 \cs_generate_variant:Nn \tl_put_left:Nx { c }
2602 \cs_generate_variant:Nn \tl_gput_left:Nn { c }
2603 \cs_generate_variant:Nn \tl_gput_left:NV { c }
2604 \cs_generate_variant:Nn \tl_gput_left:No { c }
2605 \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 35.)

\tl_put_right:Nn

```
2606 \cs_new_protected:Npn \tl_put_right:Nn #1#2
2607   { \cs_set_nopar:Npx #1 { \exp_not:o #1 \exp_not:n {#2} } }
2608 \cs_new_protected:Npn \tl_put_right:NV #1#2
2609   { \cs_set_nopar:Npx #1 { \exp_not:o #1 \exp_not:V #2 } }
2610 \cs_new_protected:Npn \tl_put_right:No #1#2
2611   { \cs_set_nopar:Npx #1 { \exp_not:o #1 \exp_not:o {#2} } }
2612 \cs_new_protected:Npn \tl_put_right:Nx #1#2
2613   { \cs_set_nopar:Npx #1 { \exp_not:o #1 #2 } }
2614 \cs_new_protected:Npn \tl_gput_right:Nn #1#2
2615   { \cs_gset_nopar:Npx #1 { \exp_not:o #1 \exp_not:n {#2} } }
2616 \cs_new_protected:Npn \tl_gput_right:NV #1#2
2617   { \cs_gset_nopar:Npx #1 { \exp_not:o #1 \exp_not:V #2 } }
2618 \cs_new_protected:Npn \tl_gput_right:No #1#2
2619   { \cs_gset_nopar:Npx #1 { \exp_not:o #1 \exp_not:o {#2} } }
2620 \cs_new_protected:Npn \tl_gput_right:Nx #1#2
2621   { \cs_gset_nopar:Npx #1 { \exp_not:o {#1} #2 } }
2622 \cs_generate_variant:Nn \tl_put_right:Nn { c }
2623 \cs_generate_variant:Nn \tl_put_right:NV { c }
2624 \cs_generate_variant:Nn \tl_put_right:No { c }
2625 \cs_generate_variant:Nn \tl_put_right:Nx { c }
2626 \cs_generate_variant:Nn \tl_gput_right:Nn { c }
2627 \cs_generate_variant:Nn \tl_gput_right:NV { c }
2628 \cs_generate_variant:Nn \tl_gput_right:No { c }
2629 \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 35.)

When used as a package, there is an option to be picky and to check definitions exist. This part of the process is done now, so that variable types based on `tl` (for example `clist`, `seq` and `prop`) will inherit the appropriate definitions. No `\tl_map_...` yet as the mechanisms are not fully in place. Thus instead do a more low level set up for a mapping, as in `l3basics`.

```

2630  (*package)
2631  \tex_ifodd:D \l@expl@check@declarations@bool
2632    \cs_set_protected:Npn \__cs_tmp:w #1
2633    {
2634      \if_meaning:w \q_recursion_tail #1
2635        \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
2636      \fi:
2637      \use:x
2638      {
2639        \cs_set_protected:Npn #1 \exp_not:n { ##1 ##2 }
2640        {
2641          \__chk_if_exist_var:N \exp_not:n {##1}
2642          \exp_not:o { #1 {##1} {##2} }
2643        }
2644      }
2645      \__cs_tmp:w
2646    }
2647  \__cs_tmp:w
2648    \tl_set:Nn \tl_set:No \tl_set:Nx
2649    \tl_gset:Nn \tl_gset:No \tl_gset:Nx
2650    \tl_put_left:Nn \tl_put_left:NV
2651    \tl_put_left:No \tl_put_left:Nx
2652    \tl_gput_left:Nn \tl_gput_left:NV
2653    \tl_gput_left:No \tl_gput_left:Nx
2654    \tl_put_right:Nn \tl_put_right:NV
2655    \tl_put_right:No \tl_put_right:Nx
2656    \tl_gput_right:Nn \tl_gput_right:NV
2657    \tl_gput_right:No \tl_gput_right:Nx
2658    \q_recursion_tail \q_recursion_stop
2659  
```

The two `set_eq` functions are done by hand as the internals there are a bit different.

```

2660  (*package)
2661    \cs_set_protected:Npn \tl_set_eq:NN #1#2
2662    {
2663      \__chk_if_exist_var:N #1
2664      \__chk_if_exist_var:N #2
2665      \cs_set_eq:NN #1 #2
2666    }
2667  \cs_set_protected:Npn \tl_gset_eq:NN #1#2
2668  {
2669    \__chk_if_exist_var:N #1
2670    \__chk_if_exist_var:N #2
2671    \cs_gset_eq:NN #1 #2
2672  }
2673 
```

There is also a need to check all three arguments of the `concat` functions: a token list #2 or #3 equal to `\scan_stop`: would lead to problems later on.

```

2674  {*package}
2675    \cs_set_protected:Npn \tl_concat:NNN #1#2#3
2676    {
2677      \__chk_if_exist_var:N #1
2678      \__chk_if_exist_var:N #2
2679      \__chk_if_exist_var:N #3
2680      \tl_set:Nx #1 { \exp_not:o {#2} \exp_not:o {#3} }
2681    }
2682    \cs_set_protected:Npn \tl_gconcat:NNN #1#2#3
2683    {
2684      \__chk_if_exist_var:N #1
2685      \__chk_if_exist_var:N #2
2686      \__chk_if_exist_var:N #3
2687      \tl_gset:Nx #1 { \exp_not:o {#2} \exp_not:o {#3} }
2688    }
2689  \tex_if:D
2690  
```

5.4 Reassigning token list category codes

- `\c_tl_rescan_marker_t1` 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.

```

2691 \tl_const:Nx \c_tl_rescan_marker_t1 { : \token_to_str:N : }

(End definition for \c_tl_rescan_marker_t1.)
```

- `\tl_set_rescan:Nnn` These functions use a common auxiliary. After some initial setup explained below, and
`\tl_set_rescan:Nno` the user setup #3 (followed by `\scan_stop`: to be safe), the tokens are rescanned by `_t1_set_rescan:n`
`\tl_set_rescan:Nnx` and stored into `\l_t1_internal_a_t1`, then passed to #1#2 outside
`\tl_set_rescan:cnn` the group after expansion. The auxiliary `_t1_set_rescan:n` is defined later: in the
`\tl_set_rescan:cno` simplest case, this auxiliary calls `_t1_set_rescan_multi:n`, whose code is included
`\tl_set_rescan:cnx` here to help understand the approach.
`\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 ...
```

- The standard solution is to use an x-expanding assignment and set `\everyeof` to `\exp_not:N` to suppress the error at the end of the file. Since the rescanned tokens should not be expanded, they will be taken as a delimited argument of an auxiliary which wraps them in `\exp_not:n` (in fact `\exp_not:o`, as there is a `\prg_do_nothing`: to avoid losing braces). The delimiter cannot appear within the rescanned token list because it contains twice the same character, with different catcodes.

The difference between single-line and multiple-line files complicates the story, as explained below.

```

2692 \cs_new_protected:Npn \tl_set_rescan:Nnn
2693   { \_t1_set_rescan>NNnn \tl_set:Nn }
2694 \cs_new_protected:Npn \tl_gset_rescan:Nnn
2695   { \_t1_set_rescan>NNnn \tl_gset:Nn }
```

```

2696 \cs_new_protected:Npn \tl_rescan:nn
2697   { \__tl_set_rescan:NNnn \prg_do_nothing: \use:n }
2698 \cs_new_protected:Npn \__tl_set_rescan:NNnn #1#2#3#4
2699   {
2700     \tl_if_empty:nTF {#4}
2701     {
2702       \group_begin:
2703       #3
2704       \group_end:
2705       #1 #2 { }
2706     }
2707   {
2708     \group_begin:
2709       \exp_args:No \etex_everyeof:D { \c__tl_rescan_marker_tl \exp_not:N }
2710       \int_compare:nNnT \tex_endlinechar:D = { 32 }
2711       { \int_set:Nn \tex_endlinechar:D { -1 } }
2712       \tex_newlinechar:D \tex_endlinechar:D
2713       #3 \scan_stop:
2714       \exp_args:No \__tl_set_rescan:n { \tl_to_str:n {#4} }
2715       \exp_args:NNNo
2716       \group_end:
2717       #1 #2 \l__tl_internal_a_tl
2718     }
2719   }
2720 \cs_new_protected:Npn \__tl_set_rescan_multi:n #1
2721   {
2722     \tl_set:Nx \l__tl_internal_a_tl
2723     {
2724       \exp_after:wN \__tl_rescan:w
2725       \exp_after:wN \prg_do_nothing:
2726       \etex_scantokens:D {#1}
2727     }
2728   }
2729 \exp_args:Nno \use:nn
2730   { \cs_new:Npn \__tl_rescan:w #1 } \c__tl_rescan_marker_tl
2731   { \exp_not:o {#1} }
2732 \cs_generate_variant:Nn \tl_set_rescan:Nnn { Nno , Nnx }
2733 \cs_generate_variant:Nn \tl_set_rescan:Nnn { c , cno , cnx }
2734 \cs_generate_variant:Nn \tl_gset_rescan:Nnn { Nno , Nnx }
2735 \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 37.)

```

\__tl_set_rescan:n
\__tl_set_rescan:NnTF
\__tl_set_rescan_single:nn
\__tl_set_rescan_single_aux:nn

```

This function calls `__tl_set_rescan_multiple:n` or `__tl_set_rescan_single:nn` { } 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 `\newlinechar` character. If `\newlinechar` is out of range, the argument is assumed to be a single line.

The case of multiple lines is a straightforward application of `\scantokens` as described above. 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.

For a single line, no `\endlinechar` should be added, so it will be 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), 12 (other) and 13 (active) are accepted, as these are 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). Once a valid character is found, run some code very similar to `_t1_set_rescan_multi:n`, except that `_t1_rescan:w` must be redefined to also remove the additional character (with the appropriate catcode). Getting the delimiter with the right catcode requires using `\scantokens` inside an `x`-expansion, hence using the previous definition of `_t1_rescan:w` as well. The odd `\exp_not:N \use:n` ensures that the trailing `\exp_not:N` in `\everyeof` does not prevent the expansion of `\c_t1_rescan_marker_t1`, but rather of a closing brace (this does nothing). If no valid character is found, similar code is ran, and the only difference is that trailing spaces are not preserved (bear in mind that this only happens if no character between 39 and 127 has catcode letter, other or active).

There is also some work to preserve leading spaces: test whether the first character (given by `\str_head:n`, with an extra space to circumvent a limitation of `f`-expansion) has catcode 10 and add what `TEX` would add in the middle of a line for any sequence of such characters: a single space with catcode 10 and character code 32.

```

2736 \group_begin:
2737   \tex_catcode:D '\^@ = 12 \scan_stop:
2738   \cs_new_protected:Npn \_t1_set_rescan:n #1
2739   {
2740     \int_compare:nNnTF \tex_newlinechar:D < 0
2741     { \use_i:nn }
2742     {
2743       \char_set_lccode:nn { 0 } { \tex_newlinechar:D }
2744       \tex_lowercase:D { \_t1_set_rescan:NnTF ^@ } {#1}
2745     }
2746     { \_t1_set_rescan_multi:n }
2747     { \_t1_set_rescan_single:nn { ' } }
2748   {#1}
2749 }
2750 \cs_new_protected:Npn \_t1_set_rescan:NnTF #1#2
2751   { \tl_if_in:nnTF {#2} {#1} }
2752 \cs_new_protected:Npn \_t1_set_rescan_single:nn #1
2753 {
2754   \int_compare:nNnTF
2755   { \char_value_catcode:n { '#1 } / 3 } = 4
2756   { \_t1_set_rescan_single_aux:nn {#1} }
2757   {
2758     \int_compare:nNnTF { '#1 } < { '\~ }
2759     {

```

```

2760           \char_set_lccode:n { 0 } { '#1 + 1 }
2761           \tex_lowercase:D { \__tl_set_rescan_single:nn { ^~@ } }
2762       }
2763       { \__tl_set_rescan_single_aux:nn { } }
2764   }
2765 }
2766 \cs_new_protected:Npn \__tl_set_rescan_single_aux:nn #1#2
2767 {
2768     \int_set:Nn \tex_endlinechar:D { -1 }
2769     \use:x
2770     {
2771         \exp_not:N \use:n
2772         {
2773             \exp_not:n { \cs_set:Npn \__tl_rescan:w ##1 }
2774             \exp_after:wN \__tl_rescan:w
2775             \exp_after:wN \prg_do_nothing:
2776             \etex_scantokens:D {#1}
2777         }
2778         \c__tl_rescan_marker_tl
2779     }
2780     { \exp_not:o {##1} }
2781     \tl_set:Nx \l__tl_internal_a_tl
2782     {
2783         \int_compare:nNnT
2784         {
2785             \char_value_catcode:n
2786             { \exp_last_unbraced:Nf ` \str_head:n {#2} ~ }
2787         }
2788         = { 10 } { ~ }
2789         \exp_after:wN \__tl_rescan:w
2790         \exp_after:wN \prg_do_nothing:
2791         \etex_scantokens:D { #2 #1 }
2792     }
2793 }
2794 \group_end:

```

(End definition for `__tl_set_rescan:n` and others.)

5.5 Modifying token list variables

`\tl_replace_all:Nnn`
`\tl_replace_all:cnn`
`\tl_greplace_all:Nnn`
`\tl_greplace_all:cnn`
`\tl_replace_once:Nnn`
`\tl_replace_once:cnn`
`\tl_greplace_once:Nnn`
`\tl_greplace_once:cnn`

All of the `replace` functions call `__tl_replace:NnNNNnn` with appropriate arguments. The first two arguments are explained later. The next controls whether the replacement function calls itself (`__tl_replace_next:w`) or stops (`__tl_replace_wrap:w`) after 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 `{⟨tl var⟩} {⟨pattern⟩} {⟨replacement⟩}` provided by the user. When describing the auxiliary functions below, we denote the contents of the `⟨tl var⟩` by `⟨token list⟩`.

```

2795 \cs_new_protected:Npn \tl_replace_once:Nnn
2796   { \__tl_replace:NnNNNnn \q_mark ? \__tl_replace_wrap:w \tl_set:Nx }
2797 \cs_new_protected:Npn \tl_greplace_once:Nnn
2798   { \__tl_replace:NnNNNnn \q_mark ? \__tl_replace_wrap:w \tl_gset:Nx }
2799 \cs_new_protected:Npn \tl_replace_all:Nnn
2800   { \__tl_replace:NnNNNnn \q_mark ? \__tl_replace_next:w \tl_set:Nx }

```

```

2801 \cs_new_protected:Npn \tl_greplace_all:Nnn
2802   { \__tl_replace:NnNNNnn \q_mark ? \__tl_replace_next:w \tl_gset:Nx }
2803 \cs_generate_variant:Nn \tl_replace_once:Nnn { c }
2804 \cs_generate_variant:Nn \tl_greplace_once:Nnn { c }
2805 \cs_generate_variant:Nn \tl_replace_all:Nnn { c }
2806 \cs_generate_variant:Nn \tl_greplace_all:Nnn { c }

```

(End definition for `\tl_replace_all:Nnn` and others. These functions are documented on page 36.)

```

\__tl_replace:NnNNNnn
\__tl_replace_auxi:NnnNNNnn
\__tl_replace_auxii:nNNNnn
  \__tl_replace_next:w
  \__tl_replace_wrap:w

```

To implement the actual replacement auxiliary `__tl_replace_auxii:nNNNnn` we will need a `<delimiter>` with the following properties:

- all occurrences of the `<pattern> #6` in “`<token list> <delimiter>`” belong to the `<token list>` and have no overlap with the `<delimiter>`,
- the first occurrence of the `<delimiter>` in “`<token list> <delimiter>`” is the trailing `<delimiter>`.

We first find the building blocks for the `<delimiter>`, namely two tokens `<A>` and `` such that `<A>` does not appear in `#6` and `#6` is not `` (this condition is trivial if `#6` has more than one token). Then we consider the delimiters “`<A>`” and “`<A> <A>n <A>n `”, for $n \geq 1$, where `<A>n` denotes n copies of `<A>`, and we choose as our `<delimiter>` the first one which is not in the `<token list>`.

Every delimiter in the set obeys the first condition: `#6` does not contain `<A>` hence cannot be overlapping with the `<token list>` and the `<delimiter>`, and it cannot be within the `<delimiter>` since it would have to be in one of the two `` 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 `<delimiter>` we choose does not appear in the `<token list>`. Additionally, the set of delimiters is such that a `<token list>` of n tokens can contain at most $O(n^{1/2})$ of them, hence we find a `<delimiter>` 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 “`<A>`” in the list of delimiters to try, so that the `<delimiter>` will simply be `\q_mark` in the most common situation where neither the `<token list>` nor the `<pattern>` contains `\q_mark`.

Let us now ahead, optimizing for this most common case. First, two special cases: an empty `<pattern> #6` is an error, and if `#1` is absent from both the `<token list> #5` and the `<pattern> #6` then we can use it as the `<delimiter>` through `__tl_replace_auxii:nNNNnn {#1}`. Otherwise, we end up calling `__tl_replace:NnNNNnn` repeatedly with the first two arguments `\q_mark {?}, \? {??}, \?? {???`, and so on, until `#6` does not contain the control sequence `#1`, which we take as our `<A>`. 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 `` to be `\q_nil` or `\q_stop` such that it is not equal to `#6`.

The `__tl_replace_auxii:NnnNNNnn` auxiliary receives `{<A>}` and `{<A>n }` as its arguments, initially with $n = 1$. If “`<A> <A>n <A>n `” is in the `<token list>` then increase n and try again. Once it is not anymore in the `<token list>` we take it as our `<delimiter>` and pass this to the auxii auxiliary.

```

2807 \cs_new_protected:Npn \__tl_replace:NnNNNnn #1#2#3#4#5#6#7
2808   {
2809     \tl_if_empty:nTF {#6}

```

```

2810 {
2811     \__msg_kernel_error:nnx { kernel } { empty-search-pattern }
2812     { \tl_to_str:n {#7} }
2813 }
2814 {
2815     \tl_if_in:onTF { #5 #6 } {#1}
2816     {
2817         \tl_if_in:nnTF {#6} {#1}
2818         { \exp_args:Nc \__tl_replace:NnNNNnn {#2} {#2?} }
2819         {
2820             \quark_if_nil:nTF {#6}
2821             { \__tl_replace_auxi:NnnNNNnn #5 {#1} { #1 \q_stop } }
2822             { \__tl_replace_auxi:NnnNNNnn #5 {#1} { #1 \q_nil } }
2823         }
2824     }
2825     { \__tl_replace_auxii:nNNNnn {#1} }
2826     #3#4#5 {#6} {#7}
2827 }
2828 }
2829 \cs_new_protected:Npn \__tl_replace_auxi:NnnNNNnn #1#2#3
2830 {
2831     \tl_if_in:NnTF #1 { #2 #3 #3 }
2832     { \__tl_replace_auxi:NnnNNNnn #1 { #2 #3 } {#2} }
2833     { \__tl_replace_auxi:NnnNNNnn { #2 #3 #3 } }
2834 }

```

The auxiliary `__tl_replace_auxii:nNNNnn` receives the following arguments: `{<delimiter>} {<function>} {<assignment>} {<tl var>} {<pattern>} {<replacement>}`. All of its work is done between `\group_align_safe_begin:` and `\group_align_safe_end:` to avoid issues in alignments. It does the actual replacement within `#3 #4 {...}`, an x-expanding `<assignment>` #3 to the `<tl var>` #4. The auxiliary `__tl_replace_next:w` is called, followed by the `<token list>`, some tokens including the `<delimiter>` #1, followed by the `<pattern>` #5. This auxiliary finds an argument delimited by #5 (the presence of a trailing #5 avoids runaway arguments) and calls `__tl_replace_wrap:w` to test whether this #5 is found within the `<token list>` or is the trailing one.

If on the one hand it is found within the `<token list>`, then ##1 cannot contain the `<delimiter>` #1 that we worked so hard to obtain, thus `__tl_replace_wrap:w` gets ##1 as its own argument ##1, and protects it against the x-expanding assignment. It also finds `\exp_not:n` as ##2 and does nothing to it, thus letting through `\exp_not:n` {<replacement>} into the assignment. Note that `__tl_replace_next:w` and `__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 unbrates the `<ending code>`

which removes what remains (essentially the *<delimiter>* and *<replacement>*).

```
2835 \cs_new_protected:Npn \__tl_replace_auxii:nNNNnn #1#2#3#4#5#6
2836  {
2837      \group_align_safe_begin:
2838      \cs_set:Npn \__tl_replace_wrap:w ##1 #1 ##2
2839          { \exp_not:o { \use_none:nn ##1 } ##2 }
2840      \cs_set:Npx \__tl_replace_next:w ##1 #5
2841          {
2842              \exp_not:N \__tl_replace_wrap:w ##1
2843              \exp_not:n { #1 }
2844              \exp_not:n { \exp_not:n {#6} }
2845              \exp_not:n { #2 { } { } }
2846          }
2847      #3 #4
2848      {
2849          \exp_after:wN \__tl_replace_next:w
2850          \exp_after:wN { \exp_after:wN }
2851          \exp_after:wN { \exp_after:wN }
2852          #4
2853          #1
2854          {
2855              \if_false: { \fi: }
2856              \exp_after:wN \use_none:n \exp_after:wN { \if_false: } \fi:
2857          }
2858          #5
2859      }
2860      \group_align_safe_end:
2861  }
2862 \cs_new_eq:NN \__tl_replace_wrap:w ?
2863 \cs_new_eq:NN \__tl_replace_next:w ?
```

(End definition for *__tl_replace:NnNNNnn* and others.)

\tl_remove_once:Nn Removal is just a special case of replacement.

```
2864 \cs_new_protected:Npn \tl_remove_once:Nn #1#2
2865     { \tl_replace_once:Nnn #1 {#2} { } }
2866 \cs_new_protected:Npn \tl_gremove_once:Nn #1#2
2867     { \tl_greplace_once:Nnn #1 {#2} { } }
2868 \cs_generate_variant:Nn \tl_remove_once:Nn { c }
2869 \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 36.)

\tl_remove_all:Nn Removal is just a special case of replacement.

```
2870 \cs_new_protected:Npn \tl_remove_all:Nn #1#2
2871     { \tl_replace_all:Nnn #1 {#2} { } }
2872 \cs_new_protected:Npn \tl_gremove_all:Nn #1#2
2873     { \tl_greplace_all:Nnn #1 {#2} { } }
2874 \cs_generate_variant:Nn \tl_remove_all:Nn { c }
2875 \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 36.)

5.6 Token list conditionals

\tl_if_blank_p:n TeX skips spaces when reading a non-delimited arguments. Thus, a *(token list)* is blank if and only if `\use_none:n <token list>` is empty after one expansion. The auxiliary `_tl_if_empty_return:o` is a fast emptiness test, converting its argument to a string (after one expansion) and using the test `\if_meaning:w \q_nil ... \q_nil`.

```

2876 \prg_new_conditional:Npnn \tl_if_blank:n #1 { p , T , F , TF }
2877   { \_tl_if_empty_return:o { \use_none:n #1 ? } }
2878 \cs_generate_variant:Nn \tl_if_blank_p:n { V }
2879 \cs_generate_variant:Nn \tl_if_blank:nT { V }
2880 \cs_generate_variant:Nn \tl_if_blank:nF { V }
2881 \cs_generate_variant:Nn \tl_if_blank:nTF { V }
2882 \cs_generate_variant:Nn \tl_if_blank_p:n { o }
2883 \cs_generate_variant:Nn \tl_if_blank:nT { o }
2884 \cs_generate_variant:Nn \tl_if_blank:nF { o }
2885 \cs_generate_variant:Nn \tl_if_blank:nTF { o }

```

(End definition for `\tl_if_blank:nTF` and `_tl_if_blank_p:NNw`. These functions are documented on page 37.)

\tl_if_empty_p:N These functions check whether the token list in the argument is empty and execute the proper code from their argument(s).

```

2886 \prg_new_conditional:Npnn \tl_if_empty:N #1 { p , T , F , TF }
2887   {
2888     \if_meaning:w #1 \c_empty_tl
2889     \prg_return_true:
2890   \else:
2891     \prg_return_false:
2892   \fi:
2893 }
2894 \cs_generate_variant:Nn \tl_if_empty_p:N { c }
2895 \cs_generate_variant:Nn \tl_if_empty:NT { c }
2896 \cs_generate_variant:Nn \tl_if_empty:NF { c }
2897 \cs_generate_variant:Nn \tl_if_empty:NTF { c }

```

(End definition for `\tl_if_empty:NTF`. This function is documented on page 38.)

\tl_if_empty_p:n Convert the argument to a string: this will be empty if and only if the argument is. Then `\if_meaning:w \q_nil ... \q_nil` is true if and only if the string ... is empty. It could be tempting to use `\if_meaning:w \q_nil #1 \q_nil` directly. This fails on a token list starting with `\q_nil` of course but more troubling is the case where argument is a complete conditional such as `\if_true: a \else: b \fi:` because then `\if_true:` is used by `\if_meaning:w`, the test turns out false, the `\else:` executes the false branch, the `\fi:` ends it and the `\q_nil` at the end starts executing...

```

2898 \prg_new_conditional:Npnn \tl_if_empty:n #1 { p , TF , T , F }
2899   {
2900     \exp_after:wN \if_meaning:w \exp_after:wn \q_nil
2901     \tl_to_str:n {#1} \q_nil
2902     \prg_return_true:
2903   \else:
2904     \prg_return_false:
2905   \fi:
2906 }
2907 \cs_generate_variant:Nn \tl_if_empty_p:n { V }

```

```

2908 \cs_generate_variant:Nn \tl_if_empty:nTF { V }
2909 \cs_generate_variant:Nn \tl_if_empty:nT { V }
2910 \cs_generate_variant:Nn \tl_if_empty:nF { V }

```

(End definition for `\tl_if_empty:nTF`. This function is documented on page 38.)

`\tl_if_empty_p:o` The auxiliary function `_tl_if_empty_return:o` is for use in various token list conditionals which reduce to testing if a given token list is empty after applying a simple function 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 many places. Note that this works because `\etex_detokenize:D` expands tokens that follow until reading a catcode 1 (begin-group) token.

```

2911 \cs_new:Npn \_tl_if_empty_return:o #1
2912 {
2913     \exp_after:wN \if_meaning:w \exp_after:wN \q_nil
2914     \etex_detokenize:D \exp_after:wN {#1} \q_nil
2915     \prg_return_true:
2916     \else:
2917         \prg_return_false:
2918     \fi:
2919 }
2920 \prg_new_conditional:Npnn \tl_if_empty:o #1 { p , TF , T , F }
2921     { \_tl_if_empty_return:o {#1} }

```

(End definition for `\tl_if_empty:oTF` and `_tl_if_empty_return:o`. These functions are documented on page 38.)

`\tl_if_eq_p:NN` Returns `\c_true_bool` if and only if the two token list variables are equal.

```

2922 \prg_new_conditional:Npnn \tl_if_eq:NN #1#2 { p , T , F , TF }
2923 {
2924     \if_meaning:w #1 #2
2925         \prg_return_true:
2926     \else:
2927         \prg_return_false:
2928     \fi:
2929 }
2930 \cs_generate_variant:Nn \tl_if_eq_p:NN { Nc , c , cc }
2931 \cs_generate_variant:Nn \tl_if_eq:NNTF { Nc , c , cc }
2932 \cs_generate_variant:Nn \tl_if_eq:NNT { Nc , c , cc }
2933 \cs_generate_variant:Nn \tl_if_eq:NNF { Nc , c , cc }

```

(End definition for `\tl_if_eq:NNTF`. This function is documented on page 38.)

`\tl_if_eq:nnTF` A simple store and compare routine.

```

\l__tl_internal_a_tl
\l__tl_internal_b_tl
2934 \prg_new_protected_conditional:Npnn \tl_if_eq:nn #1#2 { T , F , TF }
2935 {
2936     \group_begin:
2937         \tl_set:Nn \l__tl_internal_a_tl {#1}
2938         \tl_set:Nn \l__tl_internal_b_tl {#2}
2939         \if_meaning:w \l__tl_internal_a_tl \l__tl_internal_b_tl
2940             \group_end:
2941             \prg_return_true:
2942         \else:
2943             \group_end:

```

```

2944         \prg_return_false:
2945     \fi:
2946   }
2947 \tl_new:N \l__tl_internal_a_tl
2948 \tl_new:N \l__tl_internal_b_tl

```

(End definition for `\tl_if_in:nnTF`, `\l__tl_internal_a_tl`, and `\l__tl_internal_b_tl`. These functions are documented on page 38.)

\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`.

```

2949 \cs_new_protected:Npn \tl_if_in:NnT { \exp_args:No \tl_if_in:nnT }
2950 \cs_new_protected:Npn \tl_if_in:NnF { \exp_args:No \tl_if_in:nnF }
2951 \cs_new_protected:Npn \tl_if_in:NnTF { \exp_args:No \tl_if_in:nnTF }
2952 \cs_generate_variant:Nn \tl_if_in:NnT { c }
2953 \cs_generate_variant:Nn \tl_if_in:NnF { c }
2954 \cs_generate_variant:Nn \tl_if_in:NnTF { c }

```

(End definition for `\tl_if_in:NnTF`. This function is documented on page 38.)

\tl_if_in:nnTF Once more, the test relies on the emptiness test for robustness. The function `_tl_-
\tl_if_in:VnTF` 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:nTF` 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 TeX 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:`.

```

2955 \prg_new_protected_conditional:Npnn \tl_if_in:nn #1#2 { T , F , TF }
2956   {
2957     \if_false: { \fi:
2958     \cs_set:Npn \_tl_tmp:w ##1 #2 { }
2959     \tl_if_empty:ctF { \_tl_tmp:w #1 {} {} #2 }
2960     { \prg_return_false: } { \prg_return_true: }
2961     \if_false: } \fi:
2962   }
2963 \cs_generate_variant:Nn \tl_if_in:nnT { V , o , no }
2964 \cs_generate_variant:Nn \tl_if_in:nnF { V , o , no }
2965 \cs_generate_variant:Nn \tl_if_in:nnTF { V , o , no }

```

(End definition for `\tl_if_in:nnTF`. This function is documented on page 38.)

\tl_if_single_p:N Expand the token list and feed it to `\tl_if_single:n`.

\tl_if_single:NTF

```

2966 \cs_new:Npn \tl_if_single_p:N { \exp_args:No \tl_if_single_p:n }
2967 \cs_new:Npn \tl_if_single:NT { \exp_args:No \tl_if_single:nT }
2968 \cs_new:Npn \tl_if_single:NF { \exp_args:No \tl_if_single:nF }
2969 \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 38.)

\tl_if_single_p:n
\tl_if_single:nTF
 $__ \text{tl_if_single_p:n}$
 $__ \text{tl_if_single:nTF}$

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, `\tl_to_str:n` 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:nw` picks the second token in front of it. If #1 is empty, this token will be the trailing ? and the catcode test yields `false`. If #1 has a single item, the token will be ^ and the catcode test yields `true`. Otherwise, it will be one of the characters resulting from `\tl_to_str:n`, and the catcode test yields `false`. Note that `\if_catcode:w` takes care of the expansions, and that `\tl_to_str:n` (the `\detokenize` primitive) actually expands tokens until finding a begin-group token.

```

2970 \prg_new_conditional:Npnn \tl_if_single:n { p , T , F , TF }
2971   {
2972     \if_catcode:w ^ \exp_after:wN \_\_tl_if_single:nw
2973       \tl_to_str:n \exp_after:wN { \use_none:nn #1 ?? } ^ ? \q_stop
2974     \prg_return_true:
2975   \else:
2976     \prg_return_false:
2977   \fi:
2978 }
2979 \cs_new:Npn \_\_tl_if_single:nw #1#2#3 \q_stop {#2}

```

(End definition for `\tl_if_single:nTF` and `__tl_if_single:nTF`. These functions are documented on page 39.)

\tl_case:Nn
\tl_case:cn
\tl_case:NnTF
\tl_case:cnTF
 $__ \text{tl_case:nntF}$
 $__ \text{tl_case:Nw}$
 $__ \text{prg_case_end:nw}$
 $__ \text{tl_case_end:nw}$

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 will always be 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.

```

2980 \cs_new:Npn \tl_case:Nn #1#2
2981   {
2982     \exp:w
2983     \_\_tl_case:NnTF #1 {#2} { } { }
2984   }
2985 \cs_new:Npn \tl_case:NnT #1#2#3
2986   {
2987     \exp:w
2988     \_\_tl_case:NnTF #1 {#2} {#3} { }
2989   }
2990 \cs_new:Npn \tl_case:NnF #1#2#3
2991   {
2992     \exp:w
2993     \_\_tl_case:NnTF #1 {#2} { } {#3}
2994   }
2995 \cs_new:Npn \tl_case:NnTF #1#2
2996   {
2997     \exp:w
2998     \_\_tl_case:NnTF #1 {#2}
2999   }
3000 \cs_new:Npn \_\_tl_case:NnTF #1#2#3#4
3001   { \_\_tl_case:Nw #1 #2 #1 { } \q_mark {#3} \q_mark {#4} \q_stop }
3002 \cs_new:Npn \_\_tl_case:Nw #1#2#3

```

```

3003  {
3004    \tl_if_eq:NNTF #1 #2
3005    { \__tl_case_end:nw {#3} }
3006    { \__tl_case:Nw #1 }
3007  }
3008 \cs_generate_variant:Nn \tl_case:Nn { c }
3009 \cs_generate_variant:Nn \tl_case:NnT { c }
3010 \cs_generate_variant:Nn \tl_case:NnF { c }
3011 \cs_generate_variant:Nn \tl_case:NnTF { c }

```

To tidy up the recursion, there are two outcomes. If there was a hit to one of the cases searched for, then #1 will be the code to insert, #2 will be the *next* case to check on and #3 will be all of the rest of the cases code. That means that #4 will be the **true** branch code, and #5 will be tidy up the spare **\q_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 will be empty, #2 will be the first **\q_mark** and so #4 will be the **false** code (the **true** code is mopped up by #3).

```

3012 \cs_new:Npn \__prg_case_end:nw #1#2#3 \q_mark #4#5 \q_stop
3013 { \exp_end: #1 #4 }
3014 \cs_new_eq:NN \__tl_case_end:nw \__prg_case_end:nw

```

(End definition for **\tl_case:NnTF** and others. These functions are documented on page 39.)

5.7 Mapping to token lists

\tl_map_function:nN
\tl_map_function:NN
\tl_map_function:cN

Expandable loop macro for token lists. These have the advantage of not needing to test if the argument is empty, because if it is, the stop marker will be read immediately and the loop terminated.

```

\__tl_map_function:Nn
3015 \cs_new:Npn \tl_map_function:nN #1#2
3016 {
3017   \__tl_map_function:Nn #2 #1
3018   \q_recursion_tail
3019   \__prg_break_point:Nn \tl_map_break: { }
3020 }
3021 \cs_new:Npn \tl_map_function:NN
3022 { \exp_args:No \tl_map_function:nN }
3023 \cs_new:Npn \__tl_map_function:Nn #1#2
3024 {
3025   \__quark_if_recursion_tail_break:nN {#2} \tl_map_break:
3026   #1 {#2} \__tl_map_function:Nn #1
3027 }
3028 \cs_generate_variant:Nn \tl_map_function:NN { c }

```

(End definition for **\tl_map_function:nN**, **\tl_map_function:NN**, and **__tl_map_function:Nn**. These functions are documented on page 39.)

\tl_map_inline:nn
\tl_map_inline:Nn
\tl_map_inline:cn

The inline functions are straight forward by now. We use a little trick with the counter **\g_prg_map_int** to make them nestable. We can also make use of **__tl_map_function:Nn** from before.

```

3029 \cs_new_protected:Npn \tl_map_inline:nn #1#2
3030 {
3031   \int_gincr:N \g_prg_map_int
3032   \cs_gset_protected:cpn
3033   { __prg_map_ \int_use:N \g_prg_map_int :w } ##1 {#2}

```

```

3034     \exp_args:Nc \__tl_map_function:Nn
3035     { __prg_map_ \int_use:N \g__prg_map_int :w }
3036     #1 \q_recursion_tail
3037     \__prg_break_point:Nn \tl_map_break: { \int_gdecr:N \g__prg_map_int }
3038   }
3039 \cs_new_protected:Npn \tl_map_inline:Nn
3040   { \exp_args:No \tl_map_inline:nn }
3041 \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 39.)

`\tl_map_variable:nNn`
`\tl_map_variable:NNn`

`\tl_map_variable:cNn`
`_tl_map_variable:Nnn`

```

3042 \cs_new_protected:Npn \tl_map_variable:nNn #1#2#3
3043   {
3044     \__tl_map_variable:Nnn #2 {#3} #1
3045     \q_recursion_tail
3046     \__prg_break_point:Nn \tl_map_break: { }
3047   }
3048 \cs_new_protected:Npn \tl_map_variable:NNn
3049   { \exp_args:No \tl_map_variable:nNn }
3050 \cs_new_protected:Npn \_tl_map_variable:Nnn #1#2#3
3051   {
3052     \tl_set:Nn #1 {#3}
3053     \__quark_if_recursion_tail_break:NN #1 \tl_map_break:
3054     \use:n {#2}
3055     \__tl_map_variable:Nnn #1 {#2}
3056   }
3057 \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 40.)

`\tl_map_break:` The break statements use the general `__prg_map_break:Nn`.

`\tl_map_break:n`

```

3058 \cs_new:Npn \tl_map_break:
3059   { \__prg_map_break:Nn \tl_map_break: { } }
3060 \cs_new:Npn \tl_map_break:n
3061   { \__prg_map_break:Nn \tl_map_break: }

```

(End definition for `\tl_map_break:` and `\tl_map_break:n`. These functions are documented on page 40.)

5.8 Using token lists

`\tl_to_str:n` Another name for a primitive: defined in `l3basics`.

`\tl_to_str:V`

```

3062 \cs_generate_variant:Nn \tl_to_str:n { V }

```

(End definition for `\tl_to_str:n`. This function is documented on page 41.)

`\tl_to_str:N` These functions return the replacement text of a token list as a string.

`\tl_to_str:c`

```

3063 \cs_new:Npn \tl_to_str:N #1 { \etex_detokenize:D \exp_after:wN {#1} }
3064 \cs_generate_variant:Nn \tl_to_str:N { c }

```

(End definition for `\tl_to_str:N`. This function is documented on page 41.)

\tl_use:N Token lists which are simply not defined will give a clear TeX error here. No such luck for ones equal to \scan_stop: so instead a test is made and if there is an issue an error is forced.

```

3065 \cs_new:Npn \tl_use:N #1
3066 {
3067     \tl_if_exist:NTF #1 {#1}
3068     {
3069         \__msg_kernel_expandable_error:nnn
3070         { kernel } { bad-variable } {#1}
3071     }
3072 }
3073 \cs_generate_variant:Nn \tl_use:N { c }
```

(End definition for \tl_use:N. This function is documented on page 41.)

5.9 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.

```

3074 \cs_new:Npn \tl_count:n #1
3075 {
3076     \int_eval:n
3077     { 0 \tl_map_function:nN {#1} \__tl_count:n }
3078 }
3079 \cs_new:Npn \tl_count:N #1
3080 {
3081     \int_eval:n
3082     { 0 \tl_map_function:NN #1 \__tl_count:n }
3083 }
3084 \cs_new:Npn \__tl_count:n #1 { + 1 }
3085 \cs_generate_variant:Nn \tl_count:n { V , o }
3086 \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 41.)

\tl_reverse_items:n Reversal of a token list is done by taking one item at a time and putting it after \q_stop.

```

3087 \cs_new:Npn \tl_reverse_items:n #1
3088 {
3089     \__tl_reverse_items:nwNwn #1 ?
3090     \q_mark \__tl_reverse_items:nwNwn
3091     \q_mark \__tl_reverse_items:wn
3092     \q_stop { }
3093 }
3094 \cs_new:Npn \__tl_reverse_items:nwNwn #1 #2 \q_mark #3 #4 \q_stop #5
3095 {
3096     #3 #2
3097     \q_mark \__tl_reverse_items:nwNwn
3098     \q_mark \__tl_reverse_items:wn
3099     \q_stop { {#1} #5 }
3100 }
3101 \cs_new:Npn \__tl_reverse_items:wn #1 \q_stop #2
3102     { \exp_not:o { \use_none:nn #2 } }
```

(End definition for `\tl_reverse_items:n`, `_tl_reverse_items:nwNwn`, and `__tl_reverse_items:wn`. These functions are documented on page 42.)

`\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 `\q_mark`, and whose second argument is a *(continuation)*, which will receive as a braced argument `\use_none:n \q_mark` *(trimmed token list)*. In the case at hand, we take `\exp_not:o` as our continuation, so that space trimming will behave correctly within an x-type expansion.

```

3103 \cs_new:Npn \tl_trim_spaces:n #1
3104   { \_tl_trim_spaces:nn { \q_mark #1 } \exp_not:o }
3105 \cs_new_protected:Npn \tl_trim_spaces:N #1
3106   { \tl_set:Nx #1 { \exp_args:No \tl_trim_spaces:n {#1} } }
3107 \cs_new_protected:Npn \tl_gtrim_spaces:N #1
3108   { \tl_gset:Nx #1 { \exp_args:No \tl_trim_spaces:n {#1} } }
3109 \cs_generate_variant:Nn \tl_trim_spaces:N { c }
3110 \cs_generate_variant:Nn \tl_gtrim_spaces:N { c }

```

(End definition for `\tl_trim_spaces:n`, `\tl_trim_spaces:N`, and `\tl_gtrim_spaces:N`. These functions are documented on page 42.)

`__tl_trim_spaces:nn` 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 `\q_nil`. Removing leading spaces is done with `_tl_trim_spaces_auxi:w`, which loops until `\q_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 `\q_nil` matches the one present in the definition of `\tl_trim_spacs:n`. Then `_tl_trim_spaces_auxiv:w` puts the token list into a group, with `\use_none:n` placed there to gobble a lingering `\q_mark`, and feeds this to the *(continuation)*.

```

3111 \cs_set:Npn \_tl_tmp:w #1
3112   {
3113     \cs_new:Npn \_tl_trim_spaces:nn ##1
3114       {
3115         \_tl_trim_spaces_auxi:w
3116           ##
3117           \q_nil
3118           \q_mark #1 { }
3119           \q_mark \_tl_trim_spaces_auxii:w
3120           \_tl_trim_spaces_auxiii:w
3121           #1 \q_nil
3122           \_tl_trim_spaces_auxiv:w
3123           \q_stop
3124       }
3125     \cs_new:Npn \_tl_trim_spaces_auxi:w ##1 \q_mark #1 ##2 \q_mark ##3
3126       {
3127         ##
3128         \_tl_trim_spaces_auxi:w
3129         \q_mark
3130         ##
3131         \q_mark #1 {##1}
3132       }
3133     \cs_new:Npn \_tl_trim_spaces_auxii:w

```

```

3134      \_tl_trim_spaces_auxi:w \q_mark \q_mark ##1
3135      {
3136          \_tl_trim_spaces_auxiii:w
3137          ##1
3138      }
3139      \cs_new:Npn \_tl_trim_spaces_auxiii:w ##1 #1 \q_nil ##2
3140      {
3141          ##2
3142          ##1 \q_nil
3143          \_tl_trim_spaces_auxiii:w
3144      }
3145      \cs_new:Npn \_tl_trim_spaces_auxiv:w ##1 \q_nil ##2 \q_stop ##3
3146      { ##3 { \use_none:n ##1 } }
3147  }
3148 \_tl_tmp:w { ~ }

(End definition for \_tl_trim_spaces:nn and others.)

```

\tl_sort:Nn Implemented in l3sort.

\tl_sort:cn

\tl_gsort:Nn

\tl_gsort:cn

\tl_sort:nN

5.10 Token by token changes

\q_tl_act_mark

\q_tl_act_stop

The \tl_act functions may be applied to any token list. Hence, we use two private quarks, to allow any token, even quarks, in the token list. Only \q_tl_act_mark and \q_tl_act_stop may not appear in the token lists manipulated by _tl_act:NNNnn functions. The quarks are effectively defined in l3quark.

(End definition for \q_tl_act_mark and \q_tl_act_stop.)

_tl_act:NNNnn

_tl_act_output:n

_tl_act_reverse_output:n

_tl_act_loop:w

_tl_act_normal:NwnNNN

_tl_act_group:nwnNNN

_tl_act_space:wwnNNN

_tl_act_end:w

To help control the expansion, _tl_act:NNNnn should always be proceeded by \exp:w and ends by producing \exp_end: once the result has been obtained. Then loop over tokens, groups, and spaces in #5. The marker \q_tl_act_mark is used both to avoid losing outer braces and to detect the end of the token list more easily. The result is stored as an argument for the dummy function _tl_act_result:n.

```

3149 \cs_new:Npn \_tl_act:NNNnn #1#2#3#4#5
3150  {
3151      \group_align_safe_begin:
3152      \_tl_act_loop:w #5 \q_tl_act_mark \q_tl_act_stop
3153      {##4} #1 #2 #3
3154      \_tl_act_result:n { }
3155  }

```

In the loop, we check how the token list begins and act accordingly. In the “normal” case, we may have reached \q_tl_act_mark, 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 a group or with a space. Some extra work is needed to make _tl_act_space:wwnNNN gobble the space.

```

3156 \cs_new:Npn \_tl_act_loop:w #1 \q_tl_act_stop
3157  {
3158      \tl_if_head_is_N_type:nTF {##1}

```

```

3159 { \__tl_act_normal:NwnNNN }
3160 {
3161     \tl_if_head_is_group:nTF {#1}
3162         { \__tl_act_group:nwnNNN }
3163         { \__tl_act_space:wwnNNN }
3164     }
3165     #1 \q__tl_act_stop
3166 }
3167 \cs_new:Npn \__tl_act_normal:NwnNNN #1 #2 \q__tl_act_stop #3#4
3168 {
3169     \if_meaning:w \q__tl_act_mark #1
3170         \exp_after:wN \__tl_act_end:wn
3171     \fi:
3172     #4 {#3} #1
3173     \__tl_act_loop:w #2 \q__tl_act_stop
3174     {#3} #4
3175 }
3176 \cs_new:Npn \__tl_act_end:wn #1 \__tl_act_result:n #2
3177 {
3178     \group_align_safe_end: \exp_end: #2
3179 \cs_new:Npn \__tl_act_group:nwnNNN #1 #2 \q__tl_act_stop #3#4#5
3180 {
3181     #5 {#3} {#1}
3182     \__tl_act_loop:w #2 \q__tl_act_stop
3183     {#3} #4 #5
3184 }
3185 \exp_last_unbraced:NNo
3186 \cs_new:Npn \__tl_act_space:wwnNNN \c_space_tl #1 \q__tl_act_stop #2#3#4#5
3187 {
3188     #5 {#2}
3189     \__tl_act_loop:w #1 \q__tl_act_stop
3190     {#2} #3 #4 #5
3191 }

```

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.

```

3191 \cs_new:Npn \__tl_act_output:n #1 #2 \__tl_act_result:n #3
3192 {
3193     #2 \__tl_act_result:n { #3 #1 } }
3194 \cs_new:Npn \__tl_act_reverse_output:n #1 #2 \__tl_act_result:n #3
3195 {
3196     #2 \__tl_act_result:n { #1 #3 } }

```

(End definition for `__tl_act:NNNnn` and others.)

\tl_reverse:n The goal here is to reverse without losing spaces nor braces. This is done using the general internal function `__tl_act:NNNnn`. Spaces and “normal” tokens are output on the left of the current output. Grouped tokens are output to the left but without any reversal within the group. All of the internal functions here drop one argument: this is needed by `__tl_act:NNNnn` when changing case (to record which direction the change is in), but not when reversing the tokens.

```

3195 \cs_new:Npn \tl_reverse:n #1
3196 {
3197     \etex_unexpanded:D \exp_after:wN
3198     {
3199         \exp:w
3200         \__tl_act:NNNnn

```

```

3201      \__tl_reverse_normal:nN
3202      \__tl_reverse_group_preserve:nn
3203      \__tl_reverse_space:n
3204      { }
3205      {#1}
3206    }
3207  }
3208 \cs_generate_variant:Nn \tl_reverse:n { o , V }
3209 \cs_new:Npn \__tl_reverse_normal:nN #1#2
3210   { \__tl_act_reverse_output:n {#2} }
3211 \cs_new:Npn \__tl_reverse_group_preserve:nn #1#2
3212   { \__tl_act_reverse_output:n { {#2} } }
3213 \cs_new:Npn \__tl_reverse_space:n #1
3214   { \__tl_act_reverse_output:n { ~ } }

```

(End definition for `\tl_reverse:n` and others. These functions are documented on page 42.)

<code>\tl_reverse:N</code>	This reverses the list, leaving <code>\exp_stop_f:</code> in front, which stops the f-expansion.
<code>\tl_reverse:c</code>	<code>\cs_new_protected:Npn \tl_reverse:N #1</code>
<code>\tl_greverse:N</code>	<code>{ \tl_set:Nx #1 { \exp_args:No \tl_reverse:n { #1 } } }</code>
<code>\tl_greverse:c</code>	<code>\cs_new_protected:Npn \tl_greverse:N #1</code>
	<code>{ \tl_gset:Nx #1 { \exp_args:No \tl_reverse:n { #1 } } }</code>
	<code>\cs_generate_variant:Nn \tl_reverse:N { c }</code>
	<code>\cs_generate_variant:Nn \tl_greverse:N { c }</code>

(End definition for `\tl_reverse:N` and `\tl_greverse:N`. These functions are documented on page 42.)

5.11 The first token from a token list

<code>\tl_head:N</code>	Finding the head of a token list expandably will always strip braces, which is fine as this is consistent with for example mapping to a list. The empty brace groups in <code>\tl_head:n</code> ensure that a blank argument gives an empty result. The result is returned within the <code>\unexpanded</code> primitive. The approach here is to use <code>\if_false:</code> to allow us to use <code>}</code> as the closing delimiter: this is the only safe choice, as any other token would not be able to parse it's own code. Using a marker, we can see if what we are grabbing is exactly the marker, or there is anything else to deal with. Is 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 .
<code>\tl_head:n</code>	
<code>\tl_head:V</code>	
<code>\tl_head:v</code>	
<code>\tl_head:f</code>	
<code>__tl_head_auxi:nw</code>	
<code>__tl_head_auxi:n</code>	
<code>\tl_head:w</code>	
<code>\tl_head:N</code>	
<code>\tl_tail:N</code>	

```

3221 \cs_new:Npn \tl_head:n #1
3222   {
3223     \etex_unexpanded:D
3224     \if_false: { \fi: \__tl_head_auxi:nw #1 { } \q_stop }
3225   }
3226 \cs_new:Npn \__tl_head_auxi:nw #1#2 \q_stop
3227   {
3228     \exp_after:wN \__tl_head_auxi:n \exp_after:wN {
3229       \if_false: } \fi: {#1}
3230   }
3231 \cs_new:Npn \__tl_head_auxi:n #1
3232   {
3233     \exp_after:wN \if_meaning:w \exp_after:wN \q_nil
3234     \tl_to_str:n \exp_after:wN { \use_none:n #1 } \q_nil
3235     \exp_after:wN \use_i:nn

```

```

3236     \else:
3237         \exp_after:wN \use_i:nn
3238     \fi:
3239     {#1}
3240     { \if_false: { \fi: \_tl_head_auxi:nw #1 } }
3241   }
3242 \cs_generate_variant:Nn \tl_head:n { V , v , f }
3243 \cs_new:Npn \tl_head:w #1\q_stop {#1}
3244 \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\q_stop

```

will give the wrong result for `\tl_tail:n { a { bc } }` (the braces will 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.

```

3245 \cs_new:Npn \tl_tail:n #
3246   {
3247     \etex_unexpanded:D
3248     \tl_if_blank:nTF {#1}
3249     { { } }
3250     { \exp_after:wN { \use_none:n #1 } }
3251   }
3252 \cs_generate_variant:Nn \tl_tail:n { V , v , f }
3253 \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 43.)

`\tl_if_head_eq_meaning_p:nN` 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 will result in `\tl_head:w` leaving two tokens: ? which is taken in the `\if_charcode:w` test, and `\use_none:nn`, which ensures that `\prg_return_false:` is returned regardless of whether the charcode test was true or false.

```

3254 \prg_new_conditional:Npnn \tl_if_head_eq_charcode:nN #1#2 { p , T , F , TF }
3255   {

```

```

3256     \if_charcode:w
3257         \exp_not:N #2
3258         \tl_if_head_is_N_type:nTF { #1 ? }
3259     {
3260         \exp_after:wN \exp_not:N
3261         \tl_head:w #1 { ? \use_none:nn } \q_stop
3262     }
3263     { \str_head:n {#1} }
3264     \prg_return_true:
3265 \else:
3266     \prg_return_false:
3267 \fi:
3268 }
3269 \cs_generate_variant:Nn \tl_if_head_eq_charcode_p:nN { f }
3270 \cs_generate_variant:Nn \tl_if_head_eq_charcode:nNTF { f }
3271 \cs_generate_variant:Nn \tl_if_head_eq_charcode:nNT { f }
3272 \cs_generate_variant:Nn \tl_if_head_eq_charcode:nNF { 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 `\prg_return_true:` and `\else:` with `\use_none:nn` in case the catcode test with the (arbitrarily chosen) ? is true.

```

3273 \prg_new_conditional:Npnn \tl_if_head_eq_catcode:nN #1 #2 { p , T , F , TF }
3274 {
3275     \if_catcode:w
3276         \exp_not:N #2
3277         \tl_if_head_is_N_type:nTF { #1 ? }
3278     {
3279         \exp_after:wN \exp_not:N
3280         \tl_head:w #1 { ? \use_none:nn } \q_stop
3281     }
3282     {
3283         \tl_if_head_is_group:nTF {#1}
3284         { \c_group_begin_token }
3285         { \c_space_token }
3286     }
3287     \prg_return_true:
3288 \else:
3289     \prg_return_false:
3290 \fi:
3291 }

```

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 the usual `\prg_return_true:` and `\else..`. 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.

```

3292 \prg_new_conditional:Npnn \tl_if_head_eq_meaning:nN #1#2 { p , T , F , TF }
3293 {
3294     \tl_if_head_is_N_type:nTF { #1 ? }

```

```

3295     { \__tl_if_head_eq_meaning_normal:nN }
3296     { \__tl_if_head_eq_meaning_special:nN }
3297     {#1} #2
3298   }
3299 \cs_new:Npn \__tl_if_head_eq_meaning_normal:nN #1 #2
3300   {
3301     \exp_after:wN \if_meaning:w
3302       \tl_head:w #1 { ?? \use_none:nnn } \q_stop #2
3303       \prg_return_true:
3304     \else:
3305       \prg_return_false:
3306     \fi:
3307   }
3308 \cs_new:Npn \__tl_if_head_eq_meaning_special:nN #1 #2
3309   {
3310     \if_charcode:w \str_head:n {#1} \exp_not:N #2
3311       \exp_after:wN \use:n
3312     \else:
3313       \prg_return_false:
3314       \exp_after:wN \use_none:n
3315     \fi:
3316   {
3317     \if_catcode:w \exp_not:N #2
3318       \tl_if_head_is_group:nTF {#1}
3319         { \c_group_begin_token }
3320         { \c_space_token }
3321       \prg_return_true:
3322     \else:
3323       \prg_return_false:
3324     \fi:
3325   }
3326 }

```

(End definition for `\tl_if_head_eq_meaning:nTF` and others. These functions are documented on page 44.)

`\tl_if_head_is_N_type:p: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:w` produces `~` (and otherwise nothing). In the third case (begin-group token), the lines involving `\exp_after:wN` 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 `*` to the beginning: if `#1` contains primitive conditionals, all of its occurrences must be dealt with before the `\if_catcode:w` tries to skip the `true` branch of the conditional.

```

3327 \prg_new_conditional:Npnn \tl_if_head_is_N_type:n #1 { p , T , F , TF }
3328   {
3329     \if_catcode:w
3330       \if_false: { \fi: \__tl_if_head_is_N_type:w ? #1 ~ }
3331       \exp_after:wN \use_none:n
3332       \exp_after:wN { \exp_after:wN { \token_to_str:N #1 ? } }
3333       * *
3334     \prg_return_true:
3335   \else:
3336     \prg_return_false:

```

```

3337     \fi:
3338   }
3339 \cs_new:Npn \__tl_if_head_is_N_type:w #1 ~
3340   {
3341     \tl_if_empty:cTF { \use_none:n #1 } { ^ } { }
3342     \exp_after:wN \use_none:n \exp_after:wN { \if_false: } \fi:
3343   }

```

(End definition for `\tl_if_head_is_N_type:nTF` and `__tl_if_head_is_N_type:w`. These functions are documented on page 45.)

`\tl_if_head_is_group_p:n` Pass the first token of #1 through `\token_to_str:N`, then check for the brace balance.
`\tl_if_head_is_group:nTF` The extra ? caters for an empty argument.⁸

```

3344 \prg_new_conditional:Npnn \tl_if_head_is_group:n #1 { p , T , F , TF }
3345   {
3346     \if_catcode:w
3347       \exp_after:wN \use_none:n
3348         \exp_after:wN { \exp_after:wN { \token_to_str:N #1 ? } }
3349       *
3350       \prg_return_false:
3351     \else:
3352       \prg_return_true:
3353     \fi:
3354   }

```

(End definition for `\tl_if_head_is_group:nTF`. This function is documented on page 45.)

`\tl_if_head_is_space_p:n` The auxiliary's argument is all that is before the first explicit space in ?#1?~. If that is a single ? 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 TeX in a table, and to allow for removing what remains of the token list after its first space. The `\exp:w` and `\exp_end:` ensure that the result of a single step of expansion directly yields a balanced token list (no trailing closing brace).

```

3355 \prg_new_conditional:Npnn \tl_if_head_is_space:n #1 { p , T , F , TF }
3356   {
3357     \exp:w \if_false: { \fi:
3358       \__tl_if_head_is_space:w ? #1 ? ~ }
3359     }
3360 \cs_new:Npn \__tl_if_head_is_space:w #1 ~
3361   {
3362     \tl_if_empty:cTF { \use_none:n #1 }
3363     { \exp_after:wN \exp_end: \exp_after:wN \prg_return_true: }
3364     { \exp_after:wN \exp_end: \exp_after:wN \prg_return_false: }
3365     \exp_after:wN \use_none:n \exp_after:wN { \if_false: } \fi:
3366   }

```

(End definition for `\tl_if_head_is_space:nTF` and `__tl_if_head_is_space:w`. These functions are documented on page 45.)

⁸Bruno: this could be made faster, but we don't: if we hope to ever have an e-type argument, we need all brace "tricks" to happen in one step of expansion, keeping the token list brace balanced at all times.

5.12 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 `\quark_if_recursion_tail_stop:n` terminates the loop, and returns nothing at all.

```

\_tl_item_aux:nn      \cs_new:Npn \tl_item:nn #1#2
3367   {
3368     \exp_args:Nf \_tl_item:nn
3369       { \exp_args:Nf \_tl_item_aux:nn { \int_eval:n {#2} } {#1} }
3370     #1
3371     \q_recursion_tail
3372       \__prg_break_point:
3373     }
3374   \cs_new:Npn \_tl_item_aux:nn #1#2
3375   {
3376     \int_compare:nNnTF {#1} < 0
3377       { \int_eval:n { \tl_count:n {#2} + 1 + #1 } }
3378       {#1}
3379     }
3380   \cs_new:Npn \_tl_item:nn #1#2
3381   {
3382     \__quark_if_recursion_tail_break:nN {#2} \__prg_break:
3383     \int_compare:nNnTF {#1} = 1
3384       { \__prg_break:n { \exp_not:n {#2} } }
3385       { \exp_args:Nf \_tl_item:nn { \int_eval:n { #1 - 1 } } }
3386     }
3387   }
3388 \cs_new:Npn \tl_item:Nn { \exp_args:No \tl_item:nn }
3389 \cs_generate_variant:Nn \tl_item:Nn { c }
```

(End definition for `\tl_item:nn` and others. These functions are documented on page 45.)

5.13 Viewing token lists

\tl_show:N Showing token list variables is done after checking that the variable is defined (see `__kernel_register_show:N`).

```

3390 \cs_new_protected:Npn \tl_show:N #1
3391   {
3392     \__msg_show_variable>NNNnn #1 \tl_if_exist:NTF ? { }
3393       { > ~ \token_to_str:N #1 = \tl_to_str:N #1 }
3394   }
3395 \cs_generate_variant:Nn \tl_show:N { c }
```

(End definition for `\tl_show:N`. This function is documented on page 45.)

\tl_show:n The `__msg_show_wrap:n` internal function performs line-wrapping and shows the result using the `\etex_showtokens:D` primitive. Since `\tl_to_str:n` is expanded within the line-wrapping code, the escape character is always a backslash.

```

3396 \cs_new_protected:Npn \tl_show:n #1
3397   { \__msg_show_wrap:n { > ~ \tl_to_str:n {#1} } }
```

(End definition for `\tl_show:n`. This function is documented on page 46.)

```

\tl_log:N Redirect output of \tl_show:N and \tl_show:n to the log.
\tl_log:c 3398 \cs_new_protected:Npn \tl_log:N
\tl_log:n 3399 { \__msg_log_next: \tl_show:N }
3400 \cs_generate_variant:Nn \tl_log:N { c }
3401 \cs_new_protected:Npn \tl_log:n
3402 { \__msg_log_next: \tl_show:n }

```

(End definition for `\tl_log:N` and `\tl_log:n`. These functions are documented on page 46.)

5.14 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.

```

3403 \tl_new:N \g_tmpa_tl
3404 \tl_new:N \g_tmpb_tl

```

(End definition for `\g_tmpa_tl` and `\g_tmpb_tl`. These variables are documented on page 46.)

`\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.

```

3405 \tl_new:N \l_tmpa_tl
3406 \tl_new:N \l_tmpb_tl

```

(End definition for `\l_tmpa_tl` and `\l_tmpb_tl`. These variables are documented on page 46.)

5.15 Deprecated functions

```

\tl_to_lowercase:n For removal after 2017-12-31.
\tl_to_uppercase:n 3407 \cs_new_protected:Npn \tl_to_lowercase:n #1
3408 {
3409   \__msg_kernel_warning:nnnn { kernel } { deprecated-command }
3410   { 2017-12-31 }
3411   { \token_to_str:N \tl_to_lowercase:n }
3412   { }
3413   \cs_gset_eq:NN \tl_to_lowercase:n \tex_lowercase:D
3414   \tex_lowercase:D {#1}
3415 }
3416 \cs_new_protected:Npn \tl_to_uppercase:n #1
3417 {
3418   \__msg_kernel_warning:nnnn { kernel } { deprecated-command }
3419   { 2017-12-31 }
3420   { \token_to_str:N \tl_to_uppercase:n }
3421   { }
3422   \cs_gset_eq:NN \tl_to_uppercase:n \tex_uppercase:D
3423   \tex_uppercase:D {#1}
3424 }

```

(End definition for `\tl_to_lowercase:n` and `\tl_to_uppercase:n`.)

```

3425 </initex | package>

```

6 l3str implementation

```
3426  {*initex | package}
3427  <@@=str>
```

6.1 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
\str_new:c hand.

\str_use:N 3428 \group_begin:
\str_use:c 3429   \cs_set_protected:Npn \__str_tmp:n #1
\str_clear:N 3430   {
\str_clear:c 3431     \tl_if_blank:nF {#1}
\str_gclear:N 3432     {
\str_gclear:c 3433       \cs_new_eq:cc { str_ #1 :N } { tl_ #1 :N }
\str_clear_new:N 3434       \exp_args:Nc \cs_generate_variant:Nn { str_ #1 :N } { c }
\str_clear_new:c 3435       \__str_tmp:n
\str_gclear_new:N 3436     }
\str_gclear_new:c 3437   }
\str_gclear_new:c 3438   \__str_tmp:n
\str_set_eq:NN 3439   { new }
\str_set_eq:cN 3440   { use }
\str_set_eq:Nc 3441   { clear }
\str_set_eq:cc 3442   { gclear }
\str_gset_eq:NN 3443   { clear_new }
\str_gset_eq:cN 3444   { gclear_new }
\str_gset_eq:Nc 3445   { }
\str_gset_eq:cc 3446 \group_end:
3447 \cs_new_eq:NN \str_set_eq:NN \tl_set_eq:NN
3448 \cs_new_eq:NN \str_gset_eq:NN \tl_gset_eq:NN
3449 \cs_generate_variant:Nn \str_set_eq:NN { c , Nc , cc }
3450 \cs_generate_variant:Nn \str_gset_eq:NN { c , Nc , cc }
```

(End definition for `\str_new:N` and others. These functions are documented on page 47.)

```
\str_set:Nn Simply convert the token list inputs to <strings>.
\str_set:Nx 3451 \group_begin:
\str_set:cn 3452   \cs_set_protected:Npn \__str_tmp:n #1
\str_set:cx 3453   {
\str_gset:Nn 3454     \tl_if_blank:nF {#1}
\str_gset:Nx 3455     {
\str_gset:cn 3456       \cs_new_protected:cpx { str_ #1 :Nn } ##1##2
\str_gset:cx 3457       { \exp_not:c { tl_ #1 :Nx } ##1 { \exp_not:N \tl_to_str:n {##2} } }
\str_const:Nn 3458       \exp_args:Nc \cs_generate_variant:Nn { str_ #1 :Nn } { Nx , cn , cx }
\str_const:Nx 3459       \__str_tmp:n
\str_const:cn 3460     }
\str_const:cx 3461   }
\str_put_left:Nn 3462   \__str_tmp:n
\str_put_left:Nx 3463   { set }
\str_put_left:cn 3464   { gset }
\str_put_left:cx 3465   { const }
\str_gput_left:Nn 3466   { put_left }
\str_gput_left:Nx 3467   { gput_left }
\str_gput_left:cn 3468   { put_right }
\str_gput_left:cx
\str_put_right:Nn
\str_put_right:Nx
\str_put_right:cn
\str_put_right:cx
\str_gput_right:Nn
\str_gput_right:Nx
\str_gput_right:cn
\str_gput_right:cx
```

```

3469     { gput_right }
3470     { }
3471 \group_end:
```

(End definition for `\str_set:Nn` and others. These functions are documented on page 48.)

6.2 String comparisons

`\str_if_empty_p:N` More copy-paste!

`\str_if_empty_p:c`

```
3472 \prg_new_eq_conditional:Nnn \str_if_exist:N \tl_if_exist:N { p , T , F , TF }
```

`\str_if_empty:NTF`

```
3473 \prg_new_eq_conditional:Nnn \str_if_exist:c \tl_if_exist:c { p , T , F , TF }
```

`\str_if_empty:cTF`

```
3474 \prg_new_eq_conditional:Nnn \str_if_empty:N \tl_if_empty:N { p , T , F , TF }
```

`\str_if_exist_p:N`

```
3475 \prg_new_eq_conditional:Nnn \str_if_empty:c \tl_if_empty:c { p , T , F , TF }
```

`\str_if_exist_p:c`

(End definition for `\str_if_empty:NTF` and `\str_if_exist:NTF`. These functions are documented on page 49.)

`\str_if_exist:cTF`

`_str_if_eq_x:nn`

`_str_escape_x:n`

String comparisons rely on the primitive `\(pdf)strcmp` if available: LuaTeX does not have it, so emulation is required. As the net result is that we do not *always* use the primitive, the correct approach is to wrap up in a function with defined behaviour. That's done by providing a wrapper and then redefining in the LuaTeX case. Note that the necessary Lua code is covered in `l3bootstrap`: long-term this may need to go into a separate Lua file, but at present it's somewhere that spaces are not skipped for ease-of-input. The need to detokenize and force expansion of input arises from the case where a `#` token is used in the input, e.g. `_str_if_eq_x:nn {#} { \tl_to_str:n {#} }`, which otherwise will fail as `\luatex_luaescapestring:D` does not double such tokens.

```

3476 \cs_new:Npn \_str_if_eq_x:nn #1#2 { \pdftex_strcmp:D {#1} {#2} }
3477 \cs_if_exist:NT \luatex_luatexversion:D
3478   {
3479     \cs_set:Npn \_str_if_eq_x:nn #1#2
3480     {
3481       \luatex_directlua:D
3482       {
3483         l3kernel.strptime
3484         (
3485           " \_str_escape_x:n {#1} " ,
3486           " \_str_escape_x:n {#2} "
3487         )
3488       }
3489     }
3490   \cs_new:Npn \_str_escape_x:n #1
3491   {
3492     \luatex_luaescapestring:D
3493     {
3494       \etex_detokenize:D \exp_after:wN { \luatex_expanded:D {#1} }
3495     }
3496   }
3497 }
```

(End definition for `_str_if_eq_x:nn` and `_str_escape_x:n`.)

`_str_if_eq_x_return:nn`

It turns out that we often need to compare a token list with the result of applying some function to it, and return with `\prg_return_true/false:`. This test is similar to `\str_if_eq:nnTF` (see `l3str`), but is hard-coded for speed.

```

3498 \cs_new:Npn \__str_if_eq_x_return:nn #1 #2
3499 {
3500     \if_int_compare:w \__str_if_eq_x:nn {#1} {#2} = 0 \exp_stop_f:
3501         \prg_return_true:
3502     \else:
3503         \prg_return_false:
3504     \fi:
3505 }

```

(End definition for `__str_if_eq_x_return:nn`.)

\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 make life a bit clearer. The `nn` and `xx` versions are created directly as this is most efficient.

```

3506 \prg_new_conditional:Npnn \str_if_eq:nn #1#2 { p , T , F , TF }
3507 {
3508     \if_int_compare:w
3509         \__str_if_eq_x:nn { \exp_not:n {#1} } { \exp_not:n {#2} }
3510         = 0 \exp_stop_f:
3511         \prg_return_true: \else: \prg_return_false: \fi:
3512     }
3513 \cs_generate_variant:Nn \str_if_eq_p:nn { V , o }
3514 \cs_generate_variant:Nn \str_if_eq_p:nn { nV , no , VV }
3515 \cs_generate_variant:Nn \str_if_eq:nnT { V , o }
3516 \cs_generate_variant:Nn \str_if_eq:nnT { nV , no , VV }
3517 \cs_generate_variant:Nn \str_if_eq:nnF { V , o }
3518 \cs_generate_variant:Nn \str_if_eq:nnF { nV , no , VV }
3519 \cs_generate_variant:Nn \str_if_eq:nnTF { V , o }
3520 \cs_generate_variant:Nn \str_if_eq:nnTF { nV , no , VV }
3521 \prg_new_conditional:Npnn \str_if_eq_x:nn #1#2 { p , T , F , TF }
3522 {
3523     \if_int_compare:w \__str_if_eq_x:nn {#1} {#2} = 0 \exp_stop_f:
3524         \prg_return_true: \else: \prg_return_false: \fi:
3525 }

```

(End definition for `\str_if_eq:nnTF` and `\str_if_eq_x:nnTF`. These functions are documented on page 49.)

\str_if_eq_p:NN Note that `\str_if_eq:NN` is different from `\tl_if_eq:NN` because it needs to ignore category codes.

```

3526 \prg_new_conditional:Npnn \str_if_eq:NN #1#2 { p , TF , T , F }
3527 {
3528     \if_int_compare:w \__str_if_eq_x:nn { \tl_to_str:N #1 } { \tl_to_str:N #2 }
3529         = 0 \exp_stop_f: \prg_return_true: \else: \prg_return_false: \fi:
3530     }
3531 \cs_generate_variant:Nn \str_if_eq:NNT { c , Nc , cc }
3532 \cs_generate_variant:Nn \str_if_eq:NNF { c , Nc , cc }
3533 \cs_generate_variant:Nn \str_if_eq:NNTF { c , Nc , cc }
3534 \cs_generate_variant:Nn \str_if_eq_p:NN { c , Nc , cc }

```

(End definition for `\str_if_eq:NNTF`. This function is documented on page 49.)

\str_case:nn Much the same as `\tl_case:nn(TF)` here: just a change in the internal comparison.

```

\str_case:on 3535 \cs_new:Npn \str_case:nn #1#2
\str_case:nV 3536 {
\str_case:nv
\str_case:nnTF
\str_case:onTF
\str_case:nVTF
\str_case:nvTF
\str_case_x:nn
\str_case_x:nnTF
\__str_case:nntf
\__str_case_x:nntf
\__str_case:nw

```

```

3537     \exp:w
3538     \__str_case:nnTF {#1} {#2} { } { }
3539   }
3540 \cs_new:Npn \str_case:nnT #1#2#3
3541   {
3542     \exp:w
3543     \__str_case:nnTF {#1} {#2} {#3} { }
3544   }
3545 \cs_new:Npn \str_case:nnF #1#2
3546   {
3547     \exp:w
3548     \__str_case:nnTF {#1} {#2} { }
3549   }
3550 \cs_new:Npn \str_case:nnTF #1#2
3551   {
3552     \exp:w
3553     \__str_case:nnTF {#1} {#2}
3554   }
3555 \cs_new:Npn \__str_case:nnTF #1#2#3#4
3556   { \__str_case:nw {#1} #2 {#1} { } \q_mark {#3} \q_mark {#4} \q_stop }
3557 \cs_generate_variant:Nn \str_case:nn { o , nV , nv }
3558 \cs_generate_variant:Nn \str_case:nnT { o , nV , nv }
3559 \cs_generate_variant:Nn \str_case:nnF { o , nV , nv }
3560 \cs_generate_variant:Nn \str_case:nnTF { o , nV , nv }
3561 \cs_new:Npn \__str_case:nw #1#2#3
3562   {
3563     \str_if_eq:nnTF {#1} {#2}
3564     { \__str_case_end:nw {#3} }
3565     { \__str_case:nw {#1} }
3566   }
3567 \cs_new:Npn \str_case_x:nn #1#2
3568   {
3569     \exp:w
3570     \__str_case_x:nnTF {#1} {#2} { } { }
3571   }
3572 \cs_new:Npn \str_case_x:nnT #1#2#3
3573   {
3574     \exp:w
3575     \__str_case_x:nnTF {#1} {#2} {#3} { }
3576   }
3577 \cs_new:Npn \str_case_x:nnF #1#2
3578   {
3579     \exp:w
3580     \__str_case_x:nnTF {#1} {#2} { }
3581   }
3582 \cs_new:Npn \str_case_x:nnTF #1#2
3583   {
3584     \exp:w
3585     \__str_case_x:nnTF {#1} {#2}
3586   }
3587 \cs_new:Npn \__str_case_x:nnTF #1#2#3#4
3588   { \__str_case_x:nw {#1} #2 {#1} { } \q_mark {#3} \q_mark {#4} \q_stop }
3589 \cs_new:Npn \__str_case_x:nw #1#2#3
3590   {

```

```

3591     \str_if_eq_x:nnTF {#1} {#2}
3592     { \__str_case_end:nw {#3} }
3593     { \__str_case_x:nw {#1} }
3594   }
3595 \cs_new_eq:NN \__str_case_end:nw \__prg_case_end:nw

```

(End definition for `\str_case:nnTF` and others. These functions are documented on page 49.)

6.3 Accessing specific characters in a string

`__str_to_other:n`
`__str_to_other_loop:w`
`__str_to_other_end:w`

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 `\q_mark` and `\q_stop` markers. 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 `\q_mark` and the first A (well, there is also the need to remove a space).

```

3596 \cs_new:Npn \__str_to_other:n #1
3597   {
3598     \exp_after:wN \__str_to_other_loop:w
3599     \tl_to_str:n {#1} ~ A ~ A ~ A ~ A ~ A ~ A ~ A ~ \q_mark \q_stop
3600   }
3601 \group_begin:
3602 \tex_lccode:D '\* = '\ %
3603 \tex_lccode:D 'A = '\A
3604 \tex_lowercase:D
3605   {
3606     \group_end:
3607     \cs_new:Npn \__str_to_other_loop:w
3608       #1 ~ #2 ~ #3 ~ #4 ~ #5 ~ #6 ~ #7 ~ #8 ~ #9 \q_stop
3609   {
3610     \if_meaning:w A #8
3611       \__str_to_other_end:w
3612     \fi:
3613     \__str_to_other_loop:w
3614       #9 #1 * #2 * #3 * #4 * #5 * #6 * #7 * #8 * \q_stop
3615   }
3616   \cs_new:Npn \__str_to_other_end:w \fi: #1 \q_mark #2 * A #3 \q_stop
3617   { \fi: #2 }
3618 }

```

(End definition for `__str_to_other:n`, `__str_to_other_loop:w`, and `__str_to_other_end:w`.)

`__str_to_other_fast:n`
`__str_to_other_fast_loop:w`
`__str_to_other_fast_end:w`

The difference with `__str_to_other:n` is that the converted part is left in the input stream, making these commands only restricted-expandable.

```

3619 \cs_new:Npn \__str_to_other_fast:n #1
3620   {
3621     \exp_after:wN \__str_to_other_fast_loop:w \tl_to_str:n {#1} ~
3622     A ~ A ~ A ~ A ~ A ~ A ~ A ~ A ~ \q_stop
3623   }
3624 \group_begin:
3625 \tex_lccode:D '\* = '\ %
3626 \tex_lccode:D 'A = '\A
3627 \tex_lowercase:D
3628   {

```

```

3629     \group_end:
3630     \cs_new:Npn \__str_to_other_fast_loop:w
3631         #1 ~ #2 ~ #3 ~ #4 ~ #5 ~ #6 ~ #7 ~ #8 ~ #9 ~
3632     {
3633         \if_meaning:w A #9
3634             \__str_to_other_fast_end:w
3635         \fi:
3636         #1 * #2 * #3 * #4 * #5 * #6 * #7 * #8 * #9
3637             \__str_to_other_fast_loop:w *
3638     }
3639     \cs_new:Npn \__str_to_other_fast_end:w #1 * A #2 \q_stop {#1}
3640 }

```

(End definition for `__str_to_other_fast:n`, `__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.

```

3641 \cs_new:Npn \str_item:Nn { \exp_args:No \str_item:nn }
3642 \cs_generate_variant:Nn \str_item:Nn { c }
3643 \cs_new:Npn \str_item:nn #1#2
3644 {
3645     \exp_args:Nf \tl_to_str:n
3646     {
3647         \exp_args:Nf \__str_item:nn
3648             { \__str_to_other:n {#1} } {#2}
3649     }
3650 }
3651 \cs_new:Npn \str_item_ignore_spaces:nn #1
3652     { \exp_args:No \__str_item:nn { \tl_to_str:n {#1} } }
3653 \cs_new:Npn \__str_item:nn #1#2
3654 {
3655     \exp_after:wN \__str_item:w
3656     \__int_value:w \__int_eval:w #2 \exp_after:wN ;
3657     \__int_value:w \__str_count:n {#1} ;
3658     #1 \q_stop
3659 }
3660 \cs_new:Npn \__str_item:w #1; #2;
3661 {
3662     \int_compare:nNnTF {#1} < 0
3663     {
3664         \int_compare:nNnTF {#1} < {-#2}
3665             { \use_none_delimit_by_q_stop:w }
3666             {
3667                 \exp_after:wN \use_i_delimit_by_q_stop:nw

```

```

3668     \exp:w \exp_after:wN \__str_skip_exp_end:w
3669         \__int_value:w \__int_eval:w #1 + #2 ;
3670     }
3671 }
3672 {
3673     \int_compare:nNnTF {#1} > {#2}
3674         { \use_none_delimit_by_q_stop:w }
3675         {
3676             \exp_after:wN \use_i_delimit_by_q_stop:nw
3677             \exp:w \__str_skip_exp_end:w #1 ; { }
3678         }
3679     }
3680 }

```

(End definition for `\str_item:Nn` and others. These functions are documented on page 51.)

`__str_skip_exp_end:w` 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).

```

3681 \cs_new:Npn \__str_skip_exp_end:w #1;
3682 {
3683     \if_int_compare:w #1 > 8 \exp_stop_f:
3684         \exp_after:wN \__str_skip_loop:wNNNNNNNN
3685     \else:
3686         \exp_after:wN \__str_skip_end:w
3687         \__int_value:w \__int_eval:w
3688     \fi:
3689     #1 ;
3690 }
3691 \cs_new:Npn \__str_skip_loop:wNNNNNNNN #1; #2#3#4#5#6#7#8#9
3692     { \exp_after:wN \__str_skip_exp_end:w \__int_value:w \__int_eval:w #1 - 8 ; }
3693 \cs_new:Npn \__str_skip_end:w #1 ;
3694 {
3695     \exp_after:wN \__str_skip_end:NNNNNNNN
3696     \if_case:w #1 \exp_stop_f: \or: \or: \or: \or: \or: \or: \or:
3697 }
3698 \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`
`\str_range:nmm`
`\str_range_ignore_spaces:nmm`
`__str_range:nmm`
`__str_range:w`
`__str_range:nww`

Sanitize the string. Then evaluate the arguments. At this stage we also decrement the `(start index)`, since our goal is to know how many characters should be removed. Then limit the range to be non-negative and at most the length of the string (this avoids needing to check for the end of the string when grabbing characters), shifting negative numbers by the appropriate amount. Afterwards, skip characters, then keep some more, and finally drop the end of the string.

```
3699 \cs_new:Npn \str_range:Nnn { \exp_args:No \str_range:nmm }
```

```

3700 \cs_generate_variant:Nn \str_range:Nnn { c }
3701 \cs_new:Npn \str_range:nnn #1#2#3
3702 {
3703     \exp_args:Nf \tl_to_str:n
3704     {
3705         \exp_args:Nf \__str_range:nnn
3706         { \__str_to_other:n {#1} } {#2} {#3}
3707     }
3708 }
3709 \cs_new:Npn \str_range_ignore_spaces:nnn #1
3710 { \exp_args:No \__str_range:nnn { \tl_to_str:n {#1} } }
3711 \cs_new:Npn \__str_range:nnn #1#2#3
3712 {
3713     \exp_after:wN \__str_range:w
3714     \__int_value:w \__str_count:n {#1} \exp_after:wN ;
3715     \__int_value:w \__int_eval:w #2 - 1 \exp_after:wN ;
3716     \__int_value:w \__int_eval:w #3 ;
3717     #1 \q_stop
3718 }
3719 \cs_new:Npn \__str_range:w #1; #2; #3;
3720 {
3721     \exp_args:Nf \__str_range:nnw
3722     { \__str_range_normalize:nn {#2} {#1} }
3723     { \__str_range_normalize:nn {#3} {#1} }
3724 }
3725 \cs_new:Npn \__str_range:nnw #1#2
3726 {
3727     \exp_after:wN \__str_collect_delimit_by_q_stop:w
3728     \__int_value:w \__int_eval:w #2 - #1 \exp_after:wN ;
3729     \exp:w \__str_skip_exp_end:w #1 ;
3730 }

```

(End definition for `\str_range:Nnn` and others. These functions are documented on page 52.)

`__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 $[0, \#2]$.

```

3731 \cs_new:Npn \__str_range_normalize:nn #1#2
3732 {
3733     \int_eval:n
3734     {
3735         \if_int_compare:w #1 < 0 \exp_stop_f:
3736         \if_int_compare:w #1 < -#2 \exp_stop_f:
3737             0
3738         \else:
3739             #1 + #2 + 1
3740         \fi:
3741     \else:
3742         \if_int_compare:w #1 < #2 \exp_stop_f:
3743             #1
3744         \else:
3745             #2
3746         \fi:

```

```

3747         \fi:
3748     }
3749 }

```

(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:nnnnnnnnw

```

Collects `max(#1,0)` characters, and removes everything else until `\q_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:nnnnnnnnw` are some `\or:`, followed by an `\fi:`, followed by #1 characters from the input stream. Simply leaving this in the input stream will close the conditional properly and the `\or:` disappear.

```

3750 \cs_new:Npn \_\_str_collect_delimit_by_q_stop:w #1;
3751   { \_\_str_collect_loop:wn #1 ; { } }
3752 \cs_new:Npn \_\_str_collect_loop:wn #1 ;
3753   {
3754     \if_int_compare:w #1 > 7 \exp_stop_f:
3755       \exp_after:wN \_\_str_collect_loop:wnNNNNNNN
3756     \else:
3757       \exp_after:wN \_\_str_collect_end:wn
3758     \fi:
3759     #1 ;
3760   }
3761 \cs_new:Npn \_\_str_collect_loop:wnNNNNNNN #1; #2 #3#4#5#6#7#8#9
3762   {
3763     \exp_after:wN \_\_str_collect_loop:wn
3764     \_\_int_value:w \_\_int_eval:w #1 - 7 ;
3765     { #2 #3#4#5#6#7#8#9 }
3766   }
3767 \cs_new:Npn \_\_str_collect_end:wn #1 ;
3768   {
3769     \exp_after:wN \_\_str_collect_end:nnnnnnnnw
3770     \if_case:w \if_int_compare:w #1 > 0 \exp_stop_f: #1 \else: 0 \fi: \exp_stop_f:
3771     \or: \or: \or: \or: \or: \or: \or: \or: \fi:
3772   }
3773 \cs_new:Npn \_\_str_collect_end:nnnnnnnnw #1#2#3#4#5#6#7#8 #9 \q_stop
3774   { #1#2#3#4#5#6#7#8 }

```

(End definition for `_str_collect_delimit_by_q_stop:w` and others.)

6.4 Counting characters

```

\str_count_spaces:N
\str_count_spaces:c
\str_count_spaces:n
\_\_str_count_spaces_loop:w

```

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.

```

3775 \cs_new:Npn \str_count_spaces:N
3776   { \exp_args:No \str_count_spaces:n }
3777 \cs_generate_variant:Nn \str_count_spaces:N { c }
3778 \cs_new:Npn \str_count_spaces:n #1
3779   {
3780     \int_eval:n
3781     {

```

```

3782         \exp_after:wN \__str_count_spaces_loop:w
3783         \tl_to_str:n {#1} ~
3784         X 7 ~ X 6 ~ X 5 ~ X 4 ~ X 3 ~ X 2 ~ X 1 ~ X 0 ~ X -1 ~
3785         \q_stop
3786     }
3787 }
3788 \cs_new:Npn \__str_count_spaces_loop:w #1~#2~#3~#4~#5~#6~#7~#8~#9~
3789 {
3790     \if_meaning:w X #9
3791     \use_i_delimit_by_q_stop:nw
3792     \fi:
3793     9 + \__str_count_spaces_loop:w
3794 }

```

(End definition for `\str_count_spaces:N`, `\str_count_spaces:n`, and `__str_count_spaces_loop:w`. These functions are documented on page 50.)

`\str_count:N` To count characters in a string we could first escape all spaces using `__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.

```

3795 \cs_new:Npn \str_count:N { \exp_args:No \str_count:n }
3796 \cs_generate_variant:Nn \str_count:N { c }
3797 \cs_new:Npn \str_count:n #1
3798 {
3799     \__str_count_aux:n
3800     {
3801         \str_count_spaces:n {#1}
3802         + \exp_after:wN \__str_count_loop:NNNNNNNNN \tl_to_str:n {#1}
3803     }
3804 }
3805 \cs_new:Npn \__str_count:n #1
3806 {
3807     \__str_count_aux:n
3808     { \__str_count_loop:NNNNNNNNN #1 }
3809 }
3810 \cs_new:Npn \str_count_ignore_spaces:n #1
3811 {
3812     \__str_count_aux:n
3813     { \exp_after:wN \__str_count_loop:NNNNNNNNN \tl_to_str:n {#1} }
3814 }
3815 \cs_new:Npn \__str_count_aux:n #1
3816 {
3817     \int_eval:n
3818     {
3819         #1
3820         { X 8 } { X 7 } { X 6 }
3821         { X 5 } { X 4 } { X 3 }

```

```

3822     { X 2 } { X 1 } { X 0 }
3823     \q_stop
3824   }
3825 }
3826 \cs_new:Npn \__str_count_loop:NNNNNNNN #1#2#3#4#5#6#7#8#9
3827 {
3828   \if_meaning:w X #9
3829     \exp_after:wN \use_none_delimit_by_q_stop:w
3830   \fi:
3831   9 + \__str_count_loop:NNNNNNNN
3832 }

```

(End definition for `\str_count:N` and others. These functions are documented on page 50.)

6.5 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`. 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, `\use_i_delimit_by_q_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 `\use_i_delimit_by_q_stop:nw`.

```

3833 \cs_new:Npn \str_head:N { \exp_args:No \str_head:n }
3834 \cs_generate_variant:Nn \str_head:N { c }
3835 \cs_new:Npn \str_head:n #1
3836   {
3837     \exp_after:wN \__str_head:w
3838     \tl_to_str:n {#1}
3839     { { } } ~ \q_stop
3840   }
3841 \cs_new:Npn \__str_head:w #1 ~ %
3842   { \use_i_delimit_by_q_stop:nw #1 { ~ } }
3843 \cs_new:Npn \str_head_ignore_spaces:n #1
3844   {
3845     \exp_after:wN \use_i_delimit_by_q_stop:nw
3846     \tl_to_str:n {#1} { } \q_stop
3847 }

```

(End definition for `\str_head:N` and others. These functions are documented on page 51.)

`\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 `\q_mark`. One can check that an empty (or blank) string yields an empty tail.

```

3848 \cs_new:Npn \str_tail:N { \exp_args:No \str_tail:n }
3849   \cs_generate_variant:Nn \str_tail:N { c }
3850   \cs_new:Npn \str_tail:n #1
3851   {
3852     \exp_after:wN \__str_tail_auxi:w
3853     \reverse_if:N \if_charcode:w
3854     \scan_stop: \tl_to_str:n {#1} X X \q_stop
3855   }
3856 \cs_new:Npn \__str_tail_auxi:w #1 X #2 \q_stop { \fi: #1 }
3857 \cs_new:Npn \str_tail_ignore_spaces:n #1
3858   {
3859     \exp_after:wN \__str_tail_auxii:w
3860     \tl_to_str:n {#1} \q_mark \q_mark \q_stop
3861   }
3862 \cs_new:Npn \__str_tail_auxii:w #1 #2 \q_mark #3 \q_stop { #2 }

```

(End definition for `\str_tail:N` and others. These functions are documented on page 51.)

6.6 String manipulation

```

\str_fold_case:n
\str_fold_case:v
\str_lower_case:n
\str_lower_case:f
\str_upper_case:n
\str_upper_case: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
\__str_lookup_lower:N
\__str_lookup_upper:N
\__str_lookup_fold:N
\use:x

```

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.

```

3863 \cs_new:Npn \str_fold_case:n #1 { \__str_change_case:nn {#1} { fold } }
3864 \cs_new:Npn \str_lower_case:n #1 { \__str_change_case:nn {#1} { lower } }
3865 \cs_new:Npn \str_upper_case:n #1 { \__str_change_case:nn {#1} { upper } }
3866 \cs_generate_variant:Nn \str_fold_case:n { V }
3867 \cs_generate_variant:Nn \str_lower_case:n { f }
3868 \cs_generate_variant:Nn \str_upper_case:n { f }
3869 \cs_new:Npn \__str_change_case:nn #1
3870   {
3871     \exp_after:wN \__str_change_case_aux:nn \exp_after:wN
3872     { \tl_to_str:n {#1} }
3873   }
3874 \cs_new:Npn \__str_change_case_aux:nn #1#2
3875   {
3876     \__str_change_case_loop:nw {#2} #1 \q_recursion_tail \q_recursion_stop
3877     \__str_change_case_result:n { }
3878   }
3879 \cs_new:Npn \__str_change_case_output:nw #1#2 \__str_change_case_result:n #3
3880   { #2 \__str_change_case_result:n { #3 #1 } }
3881 \cs_generate_variant:Nn \__str_change_case_output:nw { f }
3882 \cs_new:Npn \__str_change_case_end:wn #1 \__str_change_case_result:n #2 { #2 }
3883 \cs_new:Npn \__str_change_case_loop:nw #1#2 \q_recursion_stop
3884   {
3885     \tl_if_head_is_space:nTF {#2}
3886     { \__str_change_case_space:n }
3887     { \__str_change_case_char:nN }
3888     {#1} #2 \q_recursion_stop
3889   }
3890 \use:x

```

```

3891 { \cs_new:Npn \exp_not:N \__str_change_case_space:n ##1 \c_space_tl }
3892 {
3893     \__str_change_case_output:nw { ~ }
3894     \__str_change_case_loop:nw {#1}
3895 }
3896 \cs_new:Npn \__str_change_case_char:nN #1#2
3897 {
3898     \quark_if_recursion_tail_stop_do:Nn #2
3899     { \__str_change_case_end:wn }
3900     \cs_if_exist:cTF { c__unicode_ #1 _ #2 _tl }
3901     {
3902         \__str_change_case_output:fw
3903         { \tl_to_str:c { c__unicode_ #1 _ #2 _tl } }
3904     }
3905     { \__str_change_case_char_aux:nN {#1} #2 }
3906     \__str_change_case_loop:nw {#1}
3907 }

```

For Unicode engines there's a look up to see if the current character has a valid one-to-one case change mapping. That's not needed for 8-bit engines: as they don't have `\utex_char:D` all of the changes they can make are hard-coded and so already picked up above.

```

3908 \cs_if_exist:NTF \utex_char:D
3909 {
3910     \cs_new:Npn \__str_change_case_char_aux:nN #1#2
3911     {
3912         \int_compare:nNnTF { \use:c { __str_lookup_ #1 :N } #2 } = { 0 }
3913         { \__str_change_case_output:nw {#2} }
3914         {
3915             \__str_change_case_output:fw
3916             { \utex_char:D \use:c { __str_lookup_ #1 :N } #2 ~ }
3917         }
3918     }
3919     \cs_new_protected:Npn \__str_lookup_lower:N #1 { \tex_lccode:D '#1 }
3920     \cs_new_protected:Npn \__str_lookup_upper:N #1 { \tex_uccode:D '#1 }
3921     \cs_new_eq:NN \__str_lookup_fold:N \__str_lookup_lower:N
3922 }
3923 {
3924     \cs_new:Npn \__str_change_case_char_aux:nN #1#2
3925     { \__str_change_case_output:nw {#2} }
3926 }

```

(End definition for `\str_fold_case:n` and others. These functions are documented on page 54.)

`\c_ampersand_str` For all of those strings, use `\cs_to_str:N` to get characters with the correct category code without worries

`\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`

```

3935 \str_const:Nx \c_hash_str      { \cs_to_str:N \# }
3936 \str_const:Nx \c_percent_str   { \cs_to_str:N \% }
3937 \str_const:Nx \c_tilde_str     { \cs_to_str:N \~ }
3938 \str_const:Nx \c_underscore_str { \cs_to_str:N \_ }

```

(End definition for `\c_ampersand_str` and others. These variables are documented on page 55.)

`\l_tmpa_str` Scratch strings.

```

3939 \str_new:N \l_tmpa_str
3940 \str_new:N \l_tmpb_str
3941 \str_new:N \g_tmpa_str
3942 \str_new:N \g_tmpb_str

```

(End definition for `\l_tmpa_str` and others. These variables are documented on page 55.)

6.7 Viewing strings

`\str_show:n` Displays a string on the terminal.

```

3943 \cs_new_eq:NN \str_show:n \tl_show:n
3944 \cs_new_eq:NN \str_show:N \tl_show:N
3945 \cs_generate_variant:Nn \str_show:N { c }

```

(End definition for `\str_show:n` and `\str_show:N`. These functions are documented on page 54.)

6.8 Unicode data for case changing

```
3946 ⟨@@=unicode⟩
```

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.

The data required for cross-module manipulations is loaded here: currently this means for `str` and `tl` functions. As such, the prefix used is not `str` but rather `unicode`. For performance (as the entire data set must be read during each run) and as this code comes somewhat early in the load process, there is quite a bit of low-level code here.

As only the data needs to remain at the end of this process, everything is set up inside a group.

```
3947 \group_begin:
```

A read stream is needed. The I/O module is not yet in place and we do not want to use up a stream. We therefore use a known free one in format mode or look for the next free one in package mode (covers plain, L^AT_EX 2_< and ConTeXt MkII and MkIV).

```

3948 (*initex)
3949   \tex_chardef:D \g_unicode_data_ior = 0 \scan_stop:
3950   ⟨/initex⟩
3951   (*package)
3952   \tex_chardef:D \g_unicode_data_ior
3953   \etex_numexpr:D
3954   \cs_if_exist:NTF \lastallocatedread
3955   { \lastallocatedread }
3956   {
3957   \cs_if_exist:NTF \c_syst_last_allocated_read
3958   { \c_syst_last_allocated_read }

```

```

3959         { \tex_count:D 16 ~ }
3960     }
3961     + 1
3962     \scan_stop:
3963 </package>

```

Set up to read each file. As they use C-style comments, there is a need to deal with #. At the same time, spaces are important so they need to be picked up as they are important. Beyond that, the current category code scheme works fine. With no I/O loop available, hard-code one that will work quickly.

```

3964     \cs_set_protected:Npn \__unicode_map_inline:n #1
3965     {
3966         \group_begin:
3967         \tex_catcode:D '\# = 12 \scan_stop:
3968         \tex_catcode:D '\_ = 10 \scan_stop:
3969         \tex_openin:D \g__unicode_data_ior = #1 \scan_stop:
3970         \cs_if_exist:NT \utex_char:D
3971         { \__unicode_map_loop: }
3972         \tex_closein:D \g__unicode_data_ior
3973         \group_end:
3974     }
3975     \cs_set_protected:Npn \__unicode_map_loop:
3976     {
3977         \tex_ifeof:D \g__unicode_data_ior
3978         \exp_after:wN \use_none:n
3979         \else:
3980         \exp_after:wN \use:n
3981         \fi:
3982         {
3983             \tex_read:D \g__unicode_data_ior to \l__unicode_tmp_tl
3984             \if_meaning:w \c_empty_tl \l__unicode_tmp_tl
3985             \else:
3986                 \exp_after:wN \__unicode_parse:w \l__unicode_tmp_tl \q_stop
3987             \fi:
3988             \__unicode_map_loop:
3989         }
3990     }

```

The lead-off parser for each line is common for all of the files. If the line starts with a # it's a comment. There's one special comment line to look out for in `SpecialCasing.txt` as we want to ignore everything after it. As this line does not appear in any other sources and the test is quite quick (there are relatively few comment lines), it can be present in all of the passes.

```

3991     \cs_set_protected:Npn \__unicode_parse:w #1#2 \q_stop
3992     {
3993         \reverse_if:N \if:w \c_hash_str #1
3994         \__unicode_parse_auxi:w #1#2 \q_stop
3995         \else:
3996             \if_int_compare:w \__str_if_eq_x:nn
3997                 { \exp_not:n {#2} } { ~Conditional~Mappings~ } = 0 \exp_stop_f:
3998                 \cs_set_protected:Npn \__unicode_parse:w ##1 \q_stop { }
3999             \fi:
4000         \fi:
4001     }

```

Storing each exception is always done in the same way: create a constant token list which expands to exactly the mapping. These will have the category codes “now” (so should be letters) but will be detokenized for string use.

```

4002   \cs_set_protected:Npn \__unicode_store:nnnnn #1#2#3#4#5
4003   {
4004     \tl_const:cx { c__unicode_ #2 _ \utex_char:D "#1 _tl }
4005     {
4006       \utex_char:D "#3 ~
4007       \utex_char:D "#4 ~
4008       \tl_if_blank:nF {#5}
4009       { \utex_char:D "#5 }
4010     }
4011   }

```

Parse the main Unicode data file for title case exceptions (the one-to-one lower and upper case mappings it contains will all be covered by the TeX data).

```

4012   \cs_set_protected:Npn \__unicode_parse_auxi:w
4013     #1 ; #2 ; #3 ; #4 ; #5 ; #6 ; #7 ; #8 ; #9 ;
4014     { \__unicode_parse_auxii:w #1 ; }
4015   \cs_set_protected:Npn \__unicode_parse_auxii:w
4016     #1 ; #2 ; #3 ; #4 ; #5 ; #6 ; #7 \q_stop
4017     {
4018       \tl_if_blank:nF {#7}
4019       {
4020         \if_int_compare:w \__str_if_eq_x:nn { #5 ~ } {#7} = 0 \exp_stop_f:
4021         \else:
4022           \tl_const:cx
4023             { c__unicode_title_ \utex_char:D "#1 _tl }
4024             { \utex_char:D "#7 }
4025         \fi:
4026       }
4027     }
4028   \__unicode_map_inline:n { UnicodeData.txt }

```

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.

```

4029   \cs_set_protected:Npn \__unicode_parse_auxi:w #1 ;~ #2 ;~ #3 ; #4 \q_stop
4030   {
4031     \if_int_compare:w \__str_if_eq_x:nn {#2} { C } = 0 \exp_stop_f:
4032     \if_int_compare:w \tex_lccode:D "#1 = "#3 \scan_stop:
4033     \else:
4034       \tl_const:cx
4035         { c__unicode_fold_ \utex_char:D "#1 _tl }
4036         { \utex_char:D "#3 ~ }
4037     \fi:
4038   \else:
4039     \if_int_compare:w \__str_if_eq_x:nn {#2} { F } = 0 \exp_stop_f:
4040       \__unicode_parse_auxii:w #1 ~ #3 ~ \q_stop
4041     \fi:
4042     \fi:
4043   }
4044   \cs_set_protected:Npn \__unicode_parse_auxii:w #1 ~ #2 ~ #3 ~ #4 \q_stop
4045   { \__unicode_store:nnnnn {#1} { fold } {#2} {#3} {#4} }

```

```
4046     \__unicode_map_inline:n { CaseFolding.txt }
```

For upper and lower casing special situations, there is a bit more to do as we also have title casing to consider.

```
4047     \cs_set_protected:Npn \__unicode_parse_auxii:w #1 ;~ #2 ;~ #3 ;~ #4 ;~ #5 \q_stop
4048     {
4049         \use:n { \__unicode_parse_auxii:w #1 ~ lower ~ #2 ~ } ~ \q_stop
4050         \use:n { \__unicode_parse_auxii:w #1 ~ upper ~ #4 ~ } ~ \q_stop
4051         \if_int_compare:w \__str_if_eq_x:nn {#3} {#4} = 0 \exp_stop_f:
4052         \else:
4053             \use:n { \__unicode_parse_auxii:w #1 ~ title ~ #3 ~ } ~ \q_stop
4054         \fi:
4055     }
4056     \cs_set_protected:Npn \__unicode_parse_auxii:w #1 ~ #2 ~ #3 ~ #4 ~ #5 \q_stop
4057     {
4058         \tl_if_empty:nF {#4}
4059         { \__unicode_store:nnnn {#1} {#2} {#3} {#4} {#5} }
4060     }
4061     \__unicode_map_inline:n { SpecialCasing.txt }
```

For the 8-bit engines, the above does nothing but there is some set up needed. There is no expandable character generator primitive so some alternative is needed. As we've not used up hash space for the above, we can go for the fast approach here of one name per letter. Keeping folding and lower casing separate makes the use later a bit easier.

```
4062     \cs_if_exist:NF \utex_char:D
4063     {
4064         \cs_set_protected:Npn \__unicode_tmp>NN #1#2
4065         {
4066             \if_meaning:w \q_recursion_tail #2
4067                 \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
4068             \fi:
4069             \tl_const:cn { c__unicode_fold_ #1 _tl } {#2}
4070             \tl_const:cn { c__unicode_lower_ #1 _tl } {#2}
4071             \tl_const:cn { c__unicode_upper_ #2 _tl } {#1}
4072             \__unicode_tmp>NN
4073         }
4074         \__unicode_tmp>NN
4075         AaBbCcDdEeFfGgHhIiJjKkLlMmNnOoPpQqRrSsTtUuVvWwXxYyZz
4076         ? \q_recursion_tail \q_recursion_stop
4077     }
```

All done: tidy up.

```
4078 \group_end:
4079 </initex | package>
```

7 **I3seq** implementation

The following test files are used for this code: m3seq002, m3seq003.

```
4080 <*initex | package>
4081 <@@=seq>
```

A sequence is a control sequence whose top-level expansion is of the form “\s__ - seq __seq_item:n {\langle item₁ \rangle} ... __seq_item:n {\langle item_n \rangle}”, with a leading scan mark followed by *n* items of the same form. An earlier implementation used the structure

`\seq_elt:w` ($item_1$) `\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.

\s__seq The variable is defined in the l3quark module, loaded later.

(*End definition for \s__seq.*)

__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.

```
4082 \cs_new:Npn \_\_seq\_item:n
4083 {
4084     \__msg_kernel_expandable_error:nn { kernel } { misused-sequence }
4085     \use_none:n
4086 }
```

(*End definition for __seq_item:n.*)

\l__seq_internal_a_tl Scratch space for various internal uses.

```
4087 \tl_new:N \l\_\_seq\_internal\_a\_tl
4088 \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.

```
4089 \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.

```
4090 \tl_const:Nn \c\_empty_seq { \s__seq }
```

(*End definition for \c_empty_seq. This variable is documented on page 66.*)

7.1 Allocation and initialisation

\seq_new:N Sequences are initialized to `\c_empty_seq`.

```
4091 \cs_new_protected:Npn \seq_new:N #1
4092 {
4093     \__chk_if_free_cs:N #1
4094     \cs_gset_eq:NN #1 \c_empty_seq
4095 }
4096 \cs_generate_variant:Nn \seq_new:N { c }
```

(*End definition for \seq_new:N. This function is documented on page 57.*)

\seq_clear:N Clearing a sequence is similar to setting it equal to the empty one.

```
4097 \cs_new_protected:Npn \seq_clear:N #1
4098 { \seq_set_eq:NN #1 \c_empty_seq }
4099 \cs_generate_variant:Nn \seq_clear:N { c }
4100 \cs_new_protected:Npn \seq_gclear:N #1
4101 { \seq_gset_eq:NN #1 \c_empty_seq }
4102 \cs_generate_variant:Nn \seq_gclear:N { c }
```

(*End definition for \seq_clear:N and \seq_gclear:N. These functions are documented on page 57.*)

\seq_clear_new:N Once again we copy code from the token list functions.

```
4103 \cs_new_protected:Npn \seq_clear_new:N #1
4104   { \seq_if_exist:NTF #1 { \seq_clear:N #1 } { \seq_new:N #1 } }
4105   \cs_generate_variant:Nn \seq_clear_new:N { c }
4106   \cs_new_protected:Npn \seq_gclear_new:N #1
4107   { \seq_if_exist:NTF #1 { \seq_gclear:N #1 } { \seq_new:N #1 } }
4108   \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 57.)

\seq_set_eq:NN Copying a sequence is the same as copying the underlying token list.

```
4109 \cs_new_eq:NN \seq_set_eq:NN \tl_set_eq:NN
4110 \cs_new_eq:NN \seq_set_eq:Nc \tl_set_eq:Nc
4111 \cs_new_eq:NN \seq_set_eq:cc \tl_set_eq:cc
4112 \cs_new_eq:NN \seq_set_eq:cc \tl_set_eq:cc
4113 \cs_new_eq:NN \seq_gset_eq:NN \tl_gset_eq:NN
4114 \cs_new_eq:NN \seq_gset_eq:Nc \tl_gset_eq:Nc
4115 \cs_new_eq:NN \seq_gset_eq:cc \tl_gset_eq:cc
4116 \cs_new_eq:NN \seq_gset_eq:cc \tl_gset_eq:cc
```

(End definition for **\seq_set_eq:NN** and **\seq_gset_eq:NN**. These functions are documented on page 57.)

\seq_set_from_clist:NN Setting a sequence from a comma-separated list is done using a simple mapping.

```
4117 \cs_new_protected:Npn \seq_set_from_clist:NN #1#2
4118   {
4119     \tl_set:Nx #1
4120     { \s__seq \clist_map_function:NN #2 \__seq_wrap_item:n }
4121   }
4122 \cs_new_protected:Npn \seq_set_from_clist:Nn #1#2
4123   {
4124     \tl_set:Nx #1
4125     { \s__seq \clist_map_function:nN {#2} \__seq_wrap_item:n }
4126   }
4127 \cs_new_protected:Npn \seq_gset_from_clist:NN #1#2
4128   {
4129     \tl_gset:Nx #1
4130     { \s__seq \clist_map_function:NN #2 \__seq_wrap_item:n }
4131   }
4132 \cs_new_protected:Npn \seq_gset_from_clist:Nn #1#2
4133   {
4134     \tl_gset:Nx #1
4135     { \s__seq \clist_map_function:nN {#2} \__seq_wrap_item:n }
4136   }
4137 \cs_generate_variant:Nn \seq_set_from_clist:NN { Nc }
4138 \cs_generate_variant:Nn \seq_set_from_clist:NN { c , cc }
4139 \cs_generate_variant:Nn \seq_set_from_clist:Nn { c      }
4140 \cs_generate_variant:Nn \seq_gset_from_clist:NN { Nc }
4141 \cs_generate_variant:Nn \seq_gset_from_clist:NN { c , cc }
4142 \cs_generate_variant:Nn \seq_gset_from_clist:Nn { c      }
```

(End definition for **\seq_set_from_clist:NN** and others. These functions are documented on page 57.)

```

\seq_set_split:Nnn
\seq_set_split:NnV
\seq_gset_split:Nnn
\seq_gset_split:NnV
\__seq_set_split>NNnn
\__seq_set_split_auxi:w
\__seq_set_split_auxii: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.

```

4143 \cs_new_protected:Npn \seq_set_split:Nnn
4144   { \__seq_set_split:NNnn \tl_set:Nx }
4145 \cs_new_protected:Npn \seq_gset_split:Nnn
4146   { \__seq_set_split:NNnn \tl_gset:Nx }
4147 \cs_new_protected:Npn \__seq_set_split:NNnn #1#2#3#4
4148   {
4149     \tl_if_empty:nTF {#3}
4150     {
4151       \tl_set:Nn \l__seq_internal_a_tl
4152         { \tl_map_function:nN {#4} \__seq_wrap_item:n }
4153     }
4154     {
4155       \tl_set:Nn \l__seq_internal_a_tl
4156         {
4157           \__seq_set_split_auxi:w \prg_do_nothing:
4158             #4
4159           \__seq_set_split_end:
4160         }
4161       \tl_replace_all:Nnn \l__seq_internal_a_tl { #3 }
4162       {
4163         \__seq_set_split_end:
4164           \__seq_set_split_auxi:w \prg_do_nothing:
4165         }
4166       \tl_set:Nx \l__seq_internal_a_tl { \l__seq_internal_a_tl }
4167     }
4168   #1 #2 { \s__seq \l__seq_internal_a_tl }
4169 }
4170 \cs_new:Npn \__seq_set_split_auxi:w #1 \__seq_set_split_end:
4171   {
4172     \exp_not:N \__seq_set_split_auxii:w
4173     \exp_args:No \tl_trim_spaces:n {#1}
4174     \exp_not:N \__seq_set_split_end:
4175   }
4176 \cs_new:Npn \__seq_set_split_auxii:w #1 \__seq_set_split_end:
4177   { \__seq_wrap_item:n {#1} }
4178 \cs_generate_variant:Nn \seq_set_split:Nnn { NnV }
4179 \cs_generate_variant:Nn \seq_gset_split:Nnn { NnV }

```

(End definition for `\seq_set_split:Nnn` and others. These functions are documented on page 58.)

```

\seq_concat:NNN
\seq_concat:ccc
\seq_gconcat:NNN
\seq_gconcat:ccc

```

When concatenating sequences, one must remove the leading `\s__seq` of the second

sequence. The result starts with `\s__seq` (of the first sequence), which stops f-expansion.

```

4180 \cs_new_protected:Npn \seq_concat:NNN #1#2#3
4181   { \tl_set:Nf #1 { \exp_after:wN \use_i:nn \exp_after:wN #2 #3 } }
4182 \cs_new_protected:Npn \seq_gconcat:NNN #1#2#3
4183   { \tl_gset:Nf #1 { \exp_after:wN \use_i:nn \exp_after:wN #2 #3 } }
4184 \cs_generate_variant:Nn \seq_concat:NNN { ccc }
4185 \cs_generate_variant:Nn \seq_gconcat:NNN { ccc }

```

(End definition for `\seq_concat:NNN` and `\seq_gconcat:NNN`. These functions are documented on page 58.)

\seq_if_exist_p:N Copies of the `cs` functions defined in `l3basics`.

```

4186 \prg_new_eq_conditional>NNn \seq_if_exist:N \cs_if_exist:N
4187   { TF , T , F , p }
4188 \prg_new_eq_conditional>NNn \seq_if_exist:c \cs_if_exist:c
4189   { TF , T , F , p }

```

(End definition for `\seq_if_exist:NTF`. This function is documented on page 58.)

7.2 Appending data to either end

\seq_put_left:Nn

\seq_put_left:NV

\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

\seq_gput_left:Nn

\seq_gput_left:NV

\seq_gput_left:Nv

\seq_gput_left:No

\seq_gput_left:Nx

\seq_gput_left:cn

\seq_gput_left:cV

\seq_gput_left:cv

\seq_gput_left:co

\seq_gput_left:cx

_seq_put_left_aux:w

When adding to the left of a sequence, remove `\s__seq`. This is done by `_seq_put_left_aux:w`, which also stops f-expansion.

```

4190 \cs_new_protected:Npn \seq_put_left:Nn #1#2
4191   {
4192     \tl_set:Nx #1
4193     {
4194       \exp_not:n { \s__seq \_seq_item:n {#2} }
4195       \exp_not:f { \exp_after:wN \_seq_put_left_aux:w #1 }
4196     }
4197   }
4198 \cs_new_protected:Npn \seq_gput_left:Nn #1#2
4199   {
4200     \tl_gset:Nx #1
4201     {
4202       \exp_not:n { \s__seq \_seq_item:n {#2} }
4203       \exp_not:f { \exp_after:wN \_seq_put_left_aux:w #1 }
4204     }
4205   }
4206 \cs_new:Npn \_seq_put_left_aux:w \s__seq { \exp_stop_f: }
4207 \cs_generate_variant:Nn \seq_put_left:Nn { NV , Nv , No , Nx }
4208 \cs_generate_variant:Nn \seq_put_left:Nn { c , cV , cv , co , cx }
4209 \cs_generate_variant:Nn \seq_gput_left:Nn { NV , Nv , No , Nx }
4210 \cs_generate_variant:Nn \seq_gput_left:Nn { c , cV , cv , co , cx }

```

(End definition for `\seq_put_left:Nn`, `\seq_gput_left:Nn`, and `_seq_put_left_aux:w`. These functions are documented on page 58.)

\seq_put_right:Nn

\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

\seq_gput_right:Nn

\seq_gput_right:NV

\seq_gput_right:Nv

\seq_gput_right:No

\seq_gput_right:Nx

\seq_gput_right:cn

Since there is no trailing marker, adding an item to the right of a sequence simply means wrapping it in `_seq_item:n`.

```

4211 \cs_new_protected:Npn \seq_put_right:Nn #1#2
4212   { \tl_put_right:Nn #1 { \_seq_item:n {#2} } }
4213 \cs_new_protected:Npn \seq_gput_right:Nn #1#2
4214   { \tl_gput_right:Nn #1 { \_seq_item:n {#2} } }

```

```

4215 \cs_generate_variant:Nn \seq_gput_right:Nn { NV , Nv , No , Nx }
4216 \cs_generate_variant:Nn \seq_gput_right:Nn { c , cV , cv , co , cx }
4217 \cs_generate_variant:Nn \seq_put_right:Nn { NV , Nv , No , Nx }
4218 \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 58.)

7.3 Modifying sequences

`__seq_wrap_item:n` This function converts its argument to a proper sequence item in an x-expansion context.

```
4219 \cs_new:Npn \__seq_wrap_item:n #1 { \exp_not:n { \__seq_item:n {#1} } }
```

(End definition for `__seq_wrap_item:n`.)

`\l__seq_remove_seq` An internal sequence for the removal routines.

```
4220 \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.

```

4221 \cs_new_protected:Npn \seq_remove_duplicates:N
4222   { \__seq_remove_duplicates>NN \seq_set_eq:NN }
4223 \cs_new_protected:Npn \seq_gremove_duplicates:N
4224   { \__seq_remove_duplicates>NN \seq_gset_eq:NN }
4225 \cs_new_protected:Npn \__seq_remove_duplicates>NN #1#2
4226   {
4227     \seq_clear:N \l__seq_remove_seq
4228     \seq_map_inline:Nn #2
4229     {
4230       \seq_if_in:NnF \l__seq_remove_seq {##1}
4231         { \seq_put_right:Nn \l__seq_remove_seq {##1} }
4232     }
4233     #1 #2 \l__seq_remove_seq
4234   }
4235 \cs_generate_variant:Nn \seq_remove_duplicates:N { c }
4236 \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 61.)

`\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 will happen repeatedly until the entire sequence has been scanned. The code is set up to avoid needing an intermediate scratch list: the lead-off x-type expansion (`#1 #2 {#2}`) will ensure that nothing is lost.

```

4237 \cs_new_protected:Npn \seq_remove_all:Nn
4238   { \__seq_remove_all_aux:NNn \tl_set:Nx }
4239 \cs_new_protected:Npn \seq_gremove_all:Nn

```

```

4240 { \_seq_remove_all_aux:Nn \tl_gset:Nx }
4241 \cs_new_protected:Npn \_seq_remove_all_aux:Nn #1#2#3
4242 {
4243     \_seq_push_item_def:n
4244     {
4245         \str_if_eq:nnT {##1} {#3}
4246         {
4247             \if_false: { \fi: }
4248             \tl_set:Nn \l__seq_internal_b_tl {##1}
4249             #1 #2
4250             { \if_false: } \fi:
4251             \exp_not:o {#2}
4252             \tl_if_eq:NNT \l__seq_internal_a_tl \l__seq_internal_b_tl
4253             { \use_none:nn }
4254         }
4255         \_seq_wrap_item:n {##1}
4256     }
4257     \tl_set:Nn \l__seq_internal_a_tl {#3}
4258     #1 #2 {#2}
4259     \_seq_pop_item_def:
4260 }
4261 \cs_generate_variant:Nn \seq_remove_all:Nn { c }
4262 \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 [61](#).)

`\seq_reverse:N` Previously, `\seq_reverse:N` was coded by collecting the items in reverse order after an `\seq_reverse:c \exp_stop_f:` marker.

```
\seq_greverse:N  
\seq_greverse:c  
_seq_reverse:NN  
reverse_item:nwn  
  
\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_=  
{  
    #2 \exp_stop_f:  
    \@@_item:n {#1}  
}
```

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.

4263 \cs_new_protected:Npn \seq_reverse:N

```

4264     { \__seq_reverse:NN \tl_set:Nx }
4265     \cs_new_protected:Npn \seq_greverse:N
4266     { \__seq_reverse:NN \tl_gset:Nx }
4267     \cs_new_protected:Npn \__seq_reverse:NN #1 #2
4268     {
4269         \cs_set_eq:NN \__seq_tmp:w \__seq_item:n
4270         \cs_set_eq:NN \__seq_item:n \__seq_reverse_item:nwn
4271         #1 #2 { #2 \exp_not:n f } }
4272         \cs_set_eq:NN \__seq_item:n \__seq_tmp:w
4273     }
4274     \cs_new:Npn \__seq_reverse_item:nwn #1 #2 \exp_not:n #3
4275     {
4276         #2
4277         \exp_not:n { \__seq_item:n {#1} #3 }
4278     }
4279     \cs_generate_variant:Nn \seq_reverse:N { c }
4280     \cs_generate_variant:Nn \seq_greverse:N { c }

```

(End definition for `\seq_reverse:N` and others. These functions are documented on page 61.)

`\seq_sort:Nn` Implemented in `l3sort`.

`\seq_sort:cn`

`\seq_gsort:Nn`

`\seq_gsort:cn`

7.4 Sequence conditionals

`\seq_if_empty_p:N` Similar to token lists, we compare with the empty sequence.

```

4281 \prg_new_conditional:Npnn \seq_if_empty:N #1 { p , T , F , TF }
4282 {
4283     \if_meaning:w #1 \c_empty_seq
4284         \prg_return_true:
4285     \else:
4286         \prg_return_false:
4287     \fi:
4288 }
4289 \cs_generate_variant:Nn \seq_if_empty_p:N { c }
4290 \cs_generate_variant:Nn \seq_if_empty:NT { c }
4291 \cs_generate_variant:Nn \seq_if_empty:NF { c }
4292 \cs_generate_variant:Nn \seq_if_empty:NTF { c }

```

(End definition for `\seq_if_empty:NTF`. This function is documented on page 61.)

`\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, `\seq_if_in:NoTF` if there is no match then the loop will break returning `\prg_return_false:`. Everything is inside a group so that `__seq_item:n` is preserved in nested situations.

```

4293 \prg_new_protected_conditional:Npnn \seq_if_in:Nn #1#2
4294 { T , F , TF }
4295 {
4296     \group_begin:
4297         \tl_set:Nn \l__seq_internal_a_tl {#2}
4298         \cs_set_protected:Npn \__seq_item:n ##1
4299         {

```

```

4300          \tl_set:Nn \l__seq_internal_b_tl {##1}
4301          \if_meaning:w \l__seq_internal_a_tl \l__seq_internal_b_tl
4302              \exp_after:wN \__seq_if_in:
4303          \fi:
4304      }
4305      #1
4306      \group_end:
4307      \prg_return_false:
4308      \__prg_break_point:
4309  }
4310 \cs_new:Npn \__seq_if_in:
4311     { \__prg_break:n { \group_end: \prg_return_true: } }
4312 \cs_generate_variant:Nn \seq_if_in:NnT { NV , Nv , No , Nx }
4313 \cs_generate_variant:Nn \seq_if_in:NnT { c , cV , cv , co , cx }
4314 \cs_generate_variant:Nn \seq_if_in:NnF { NV , Nv , No , Nx }
4315 \cs_generate_variant:Nn \seq_if_in:NnF { c , cV , cv , co , cx }
4316 \cs_generate_variant:Nn \seq_if_in:NnTF { NV , Nv , No , Nx }
4317 \cs_generate_variant:Nn \seq_if_in:NnTF { c , cV , cv , co , cx }

```

(End definition for `\seq_if_in:NnTF` and `__seq_if_in:`. These functions are documented on page 61.)

7.5 Recovering data from sequences

`__seq_pop:NNNN` The two pop functions share their emptiness tests. We also use a common emptiness test for all branching get and pop functions.

```

4318 \cs_new_protected:Npn \__seq_pop:NNNN #1#2#3#4
4319 {
4320     \if_meaning:w #3 \c_empty_seq
4321         \tl_set:Nn #4 { \q_no_value }
4322     \else:
4323         #1#2#3#4
4324     \fi:
4325 }
4326 \cs_new_protected:Npn \__seq_pop_TF:NNNN #1#2#3#4
4327 {
4328     \if_meaning:w #3 \c_empty_seq
4329         % \tl_set:Nn #4 { \q_no_value }
4330         \prg_return_false:
4331     \else:
4332         #1#2#3#4
4333         \prg_return_true:
4334     \fi:
4335 }

```

(End definition for `__seq_pop:NNNN` and `__seq_pop_TF:NNNN`.)

`\seq_get_left:NN` 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

```

4336 \cs_new_protected:Npn \seq_get_left:NN #1#2
4337 {
4338     \tl_set:Nx #2
4339     {
4340         \exp_after:wN \__seq_get_left:wnw

```

```

4341         #1 \__seq_item:n { \q_no_value } \q_stop
4342     }
4343   }
4344 \cs_new:Npn \__seq_get_left:wnw #1 \__seq_item:n #2#3 \q_stop
4345   { \exp_not:n {#2} }
4346 \cs_generate_variant:Nn \seq_get_left:NN { c }

```

(End definition for `\seq_get_left:NN` and `__seq_get_left:wnw`. These functions are documented on page 58.)

`\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.

```

4347 \cs_new_protected:Npn \seq_pop_left:NN
4348   { \__seq_pop>NNNN \__seq_pop_left>NNNN \tl_set:Nn }
4349 \cs_new_protected:Npn \seq_gpop_left:NN
4350   { \__seq_pop>NNNN \__seq_pop_left>NNNN \tl_gset:Nn }
4351 \cs_new_protected:Npn \__seq_pop_left>NNNN #1#2#3
4352   { \exp_after:wN \__seq_pop_left:wnwNNN #2 \q_stop #1#2#3 }
4353 \cs_new_protected:Npn \__seq_pop_left:wnwNNN
4354   #1 \__seq_item:n #2#3 \q_stop #4#5#6
4355   {
4356     #4 #5 { #1 #3 }
4357     \tl_set:Nn #6 {#2}
4358   }
4359 \cs_generate_variant:Nn \seq_pop_left:NN { c }
4360 \cs_generate_variant:Nn \seq_gpop_left:NN { c }

```

(End definition for `\seq_pop_left:NN` and others. These functions are documented on page 59.)

`\seq_get_right:NN`
`\seq_get_right:cN`

`__seq_get_right_loop:nn`

First remove `\s_seq` and prepend `\q_no_value`, then take two arguments at a time. Before the right-hand end of the sequence, this is a brace group followed by `__seq_item:n`, both removed by `\use_none:nn`. At the end of the sequence, the two question marks are taken by `\use_none:nn`, and the assignment is placed before the right-most item. In the next iteration, `__seq_get_right_loop:nn` receives two empty arguments, and `\use_none:nn` stops the loop.

```

4361 \cs_new_protected:Npn \seq_get_right:NN #1#2
4362   {
4363     \exp_after:wN \use_i_ii:nn
4364     \exp_after:wN \__seq_get_right_loop:nn
4365     \exp_after:wN \q_no_value
4366     #1
4367     { ?? \tl_set:Nn #2 }
4368     { } { }
4369   }
4370 \cs_new_protected:Npn \__seq_get_right_loop:nn #1#2
4371   {
4372     \use_none:nn #2 {#1}
4373     \__seq_get_right_loop:nn
4374   }
4375 \cs_generate_variant:Nn \seq_get_right:NN { c }

```

(End definition for `\seq_get_right:NN` and `__seq_get_right_loop:nn`. These functions are documented on page 59.)

```

\seq_pop_right:NN
\seq_pop_right:cN
\seq_gpop_right:NN
\seq_gpop_right:cN
\_\_seq_pop_right:NNN
\_\_seq_pop_right_loop: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: ... \if_false: { \fi: }` construct. Using an x-type expansion and a “non-expanding” definition for `__seq_item:n`, the left-most $n - 1$ entries in a sequence of n items will be 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.

```

4376 \cs_new_protected:Npn \seq_pop_right:NN
4377   { \_\_seq_pop>NNNN \_\_seq_pop_right>NNN \tl_set:Nx }
4378 \cs_new_protected:Npn \seq_gpop_right:NN
4379   { \_\_seq_pop>NNNN \_\_seq_pop_right>NNN \tl_gset:Nx }
4380 \cs_new_protected:Npn \_\_seq_pop_right>NNN #1#2#3
4381   {
4382     \cs_set_eq:NN \_\_seq_tmp:w \_\_seq_item:n
4383     \cs_set_eq:NN \_\_seq_item:n \scan_stop:
4384     #1 #2
4385     { \if_false: } \fi: \s_seq
4386     \exp_after:wN \use_i:nnn
4387     \exp_after:wN \_\_seq_pop_right_loop:nn
4388     #2
4389   {
4390     \if_false: { \fi: }
4391     \tl_set:Nx #3
4392   }
4393   { } \use_none:nn
4394   \cs_set_eq:NN \_\_seq_item:n \_\_seq_tmp:w
4395   }
4396 \cs_new:Npn \_\_seq_pop_right_loop:nn #1#2
4397   {
4398     #2 { \exp_not:n {#1} }
4399     \_\_seq_pop_right_loop:nn
4400   }
4401 \cs_generate_variant:Nn \seq_pop_right:NN { c }
4402 \cs_generate_variant:Nn \seq_gpop_right:NN { c }

```

(End definition for `\seq_pop_right:NN` and others. These functions are documented on page 59.)

```

\seq_get_left:NNTF
\seq_get_left:cNTF

```

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_right:NNTF
\seq_get_right:cNTF

```

```

4403 \prg_new_protected_conditional:Npnn \seq_get_left:NN #1#2 { T , F , TF }
4404   { \_\_seq_pop_TF>NNNN \prg_do_nothing: \seq_get_left:NN #1#2 }
4405 \prg_new_protected_conditional:Npnn \seq_get_right:NN #1#2 { T , F , TF }
4406   { \_\_seq_pop_TF>NNNN \prg_do_nothing: \seq_get_right:NN #1#2 }
4407 \cs_generate_variant:Nn \seq_get_left:NNT { c }
4408 \cs_generate_variant:Nn \seq_get_left:NNF { c }
4409 \cs_generate_variant:Nn \seq_get_left:NNTF { c }
4410 \cs_generate_variant:Nn \seq_get_right:NNT { c }
4411 \cs_generate_variant:Nn \seq_get_right:NNF { c }
4412 \cs_generate_variant:Nn \seq_get_right:NNTF { c }

```

(End definition for `\seq_get_left:NNTF` and `\seq_get_right:NNTF`. These functions are documented on page 60.)

`\seq_pop_left:NNTF` More or less the same for popping.

```

4413 \prg_new_protected_conditional:Npnn \seq_pop_left:NN #1#2 { T , F , TF }
4414   { \__seq_pop_TF:NNNN \__seq_pop_left:NNN \tl_set:Nn #1 #2 }
4415 \prg_new_protected_conditional:Npnn \seq_gpop_left:NN #1#2 { T , F , TF }
4416   { \__seq_pop_TF:NNNN \__seq_pop_left:NNN \tl_gset:Nn #1 #2 }
4417 \prg_new_protected_conditional:Npnn \seq_pop_right:NN #1#2 { T , F , TF }
4418   { \__seq_pop_TF:NNNN \__seq_pop_right:NNN \tl_set:Nx #1 #2 }
4419 \prg_new_protected_conditional:Npnn \seq_gpop_right:NN #1#2 { T , F , TF }
4420   { \__seq_pop_TF:NNNN \__seq_pop_right:NNN \tl_gset:Nx #1 #2 }
4421 \cs_generate_variant:Nn \seq_pop_left:NNT { c }
4422 \cs_generate_variant:Nn \seq_pop_left:NNF { c }
4423 \cs_generate_variant:Nn \seq_pop_left:NNTF { c }
4424 \cs_generate_variant:Nn \seq_gpop_left:NNT { c }
4425 \cs_generate_variant:Nn \seq_gpop_left:NNF { c }
4426 \cs_generate_variant:Nn \seq_gpop_left:NNTF { c }
4427 \cs_generate_variant:Nn \seq_pop_right:NNT { c }
4428 \cs_generate_variant:Nn \seq_pop_right:NNF { c }
4429 \cs_generate_variant:Nn \seq_pop_right:NNTF { c }
4430 \cs_generate_variant:Nn \seq_gpop_right:NNT { c }
4431 \cs_generate_variant:Nn \seq_gpop_right:NNF { c }
4432 \cs_generate_variant:Nn \seq_gpop_right:NNTF { c }

```

(End definition for `\seq_pop_left:NNTF` and others. These functions are documented on page 60.)

`\seq_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 the stop code `{ ? __prg_break: } { }` will be used by the auxiliary, terminating the loop and returning nothing at all.

```

4433 \cs_new:Npn \seq_item:Nn #1
4434   { \exp_after:wN \__seq_item:wNn #1 \q_stop #1 }
4435 \cs_new:Npn \__seq_item:wNn \s__seq #1 \q_stop #2#3
4436   {
4437     \exp_args:Nf \__seq_item:nnn
4438       { \exp_args:Nf \__seq_item:nN { \int_eval:n {#3} } #2 }
4439     #1
4440     { ? \__prg_break: } { }
4441     \__prg_break_point:
4442   }
4443 \cs_new:Npn \__seq_item:nN #1#2
4444   {
4445     \int_compare:nNnTF {#1} < 0
4446       { \int_eval:n { \seq_count:N #2 + 1 + #1 } }
4447       {#1}
4448   }
4449 \cs_new:Npn \__seq_item:nnn #1#2#3
4450   {
4451     \use_none:n #2
4452     \int_compare:nNnTF {#1} = 1
4453       { \__prg_break:n { \exp_not:n {#3} } }
4454       { \exp_args:Nf \__seq_item:nnn { \int_eval:n { #1 - 1 } } }
4455   }
4456 \cs_generate_variant:Nn \seq_item:Nn { c }

```

(End definition for `\seq_item:Nn` and others. These functions are documented on page 59.)

7.6 Mapping to sequences

`\seq_map_break:`
`\seq_map_break:n`

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.

```
4457 \cs_new:Npn \seq_map_break:
4458   { \__prg_map_break:Nn \seq_map_break: { } }
4459 \cs_new:Npn \seq_map_break:n
4460   { \__prg_map_break:Nn \seq_map_break: }
```

(End definition for `\seq_map_break:` and `\seq_map_break:n`. These functions are documented on page 62.)

`\seq_map_function>NN`
`\seq_map_function:cN`
`__seq_map_function:NNn`

The idea here is to apply the code of #2 to each item in the sequence without altering the definition of `__seq_item:n`. This is done as by noting that every odd token in the sequence must be `__seq_item:n`, which can be gobbled by `\use_none:n`. At the end of the loop, #2 is instead ? `\seq_map_break:`, which therefore breaks the loop without needing to do a (relatively-expensive) quark test.

```
4461 \cs_new:Npn \seq_map_function:NN #1#2
4462   {
4463     \exp_after:wN \use_i_i:nnn
4464     \exp_after:wN \__seq_map_function:NNn
4465     \exp_after:wN #2
4466     #1
4467     { ? \seq_map_break: } f }
4468     \__prg_break_point:Nn \seq_map_break: { }
4469   }
4470 \cs_new:Npn \__seq_map_function:NNn #1#2#3
4471   {
4472     \use_none:n #2
4473     #1 {#3}
4474     \__seq_map_function:NNn #1
4475   }
4476 \cs_generate_variant:Nn \seq_map_function:NN { c }
```

(End definition for `\seq_map_function:NN` and `__seq_map_function:NNn`. These functions are documented on page 61.)

`__seq_push_item_def:n`
`__seq_push_item_def:x`
`__seq_push_item_def:`
`__seq_pop_item_def:`

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.

```
4477 \cs_new_protected:Npn \__seq_push_item_def:n
4478   {
4479     \__seq_push_item_def:
4480     \cs_gset:Npn \__seq_item:n ##1
4481   }
4482 \cs_new_protected:Npn \__seq_push_item_def:x
4483   {
4484     \__seq_push_item_def:
4485     \cs_gset:Npx \__seq_item:n ##1
4486   }
4487 \cs_new_protected:Npn \__seq_push_item_def:
```

```

4488   {
4489     \int_gincr:N \g_prg_map_int
4490     \cs_gset_eq:cN { __prg_map_ \int_use:N \g_prg_map_int :w }
4491       \__seq_item:n
4492   }
4493 \cs_new_protected:Npn \__seq_pop_item_def:
4494   {
4495     \cs_gset_eq:Nc \__seq_item:n
4496       { __prg_map_ \int_use:N \g_prg_map_int :w }
4497       \int_gdecr:N \g_prg_map_int
4498   }

```

(End definition for `__seq_push_item_def:n`, `__seq_push_item_def:`, and `__seq_pop_item_def:..`)

`\seq_map_inline:Nn` `\seq_map_inline:cn`

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`.

```

4499 \cs_new_protected:Npn \seq_map_inline:Nn #1#2
4500   {
4501     \__seq_push_item_def:n {#2}
4502     #1
4503     \__prg_break_point:Nn \seq_map_break: { \__seq_pop_item_def: }
4504   }
4505 \cs_generate_variant:Nn \seq_map_inline:Nn { c }

```

(End definition for `\seq_map_inline:Nn`. This function is documented on page 62.)

`\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.

```

4506 \cs_new_protected:Npn \seq_map_variable>NNn #1#2#3
4507   {
4508     \__seq_push_item_def:x
4509       {
4510         \tl_set:Nn \exp_not:N #2 {##1}
4511         \exp_not:n {#3}
4512       }
4513     #1
4514     \__prg_break_point:Nn \seq_map_break: { \__seq_pop_item_def: }
4515   }
4516 \cs_generate_variant:Nn \seq_map_variable:NNn { Nc }
4517 \cs_generate_variant:Nn \seq_map_variable:NNn { c , cc }

```

(End definition for `\seq_map_variable>NNn`. This function is documented on page 62.)

`\seq_count:N` `\seq_count:c` `__seq_count:n`

Counting the items in a sequence 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.

```

4518 \cs_new:Npn \seq_count:N #1
4519   {
4520     \int_eval:n
4521       {
4522         0
4523         \seq_map_function:NN #1 \__seq_count:n
4524       }
4525   }
4526 \cs_new:Npn \__seq_count:n #1 { + 1 }
4527 \cs_generate_variant:Nn \seq_count:N { c }

```

(End definition for `\seq_count:N` and `_seq_count:n`. These functions are documented on page 63.)

7.7 Using sequences

```
\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
```

See `\clist_use:Nnnn` for a general explanation. The main difference is that we use `_seq_item:n` as a delimiter rather than commas. We also need to add `_seq_item:n` at various places, and `\s_seq`.

```
4528 \cs_new:Npn \seq_use:Nnnn #1#2#3#4
4529 {
4530     \seq_if_exist:NTF #1
4531     {
4532         \int_case:nnF { \seq_count:N #1 }
4533         {
4534             { 0 } { }
4535             { 1 } { \exp_after:wN \__seq_use:NNnNnn #1 ? { } { } }
4536             { 2 } { \exp_after:wN \__seq_use:NNnNnn #1 {#2} }
4537         }
4538         {
4539             \exp_after:wN \__seq_use_setup:w #1 \__seq_item:n
4540             \q_mark { \__seq_use:nwwwnwn {#3} }
4541             \q_mark { \__seq_use:nwwn {#4} }
4542             \q_stop { }
4543         }
4544     }
4545     {
4546         \__msg_kernel_expandable_error:nnn
4547         { kernel } { bad-variable } {#1}
4548     }
4549 }
4550 \cs_generate_variant:Nn \seq_use:Nnnn { c }
4551 \cs_new:Npn \__seq_use:NNnNnn #1#2#3#4#5#6 { \exp_not:n { #3 #6 #5 } }
4552 \cs_new:Npn \__seq_use_setup:w \s_seq { \__seq_use:nwwwnwn { } }
4553 \cs_new:Npn \__seq_use:nwwwnwn
4554     #1 \__seq_item:n #2 \__seq_item:n #3 \__seq_item:n #4#5
4555     \q_mark #6#7 \q_stop #8
4556     {
4557         #6 \__seq_item:n {#3} \__seq_item:n {#4} #5
4558         \q_mark {#6} #7 \q_stop { #8 #1 #2 }
4559     }
4560 \cs_new:Npn \__seq_use:nwwn #1 \__seq_item:n #2 #3 \q_stop #4
4561     { \exp_not:n { #4 #1 #2 } }
4562 \cs_new:Npn \seq_use:Nn #1#
4563     { \seq_use:Nnnn #1 {#2} {#2} {#2} }
4564 \cs_generate_variant:Nn \seq_use:Nn { c }
```

(End definition for `\seq_use:Nnnn` and others. These functions are documented on page 63.)

7.8 Sequence stacks

The same functions as for sequences, but with the correct naming.

```
\seq_push:Nn
```

```
\seq_push:NV
```

```
\seq_push:Nv
```

```
\seq_push:No
```

```
\seq_push:Nx
```

```
\seq_push:cn
```

```
\seq_push:cV
```

```
\seq_push:cV
```

```
\seq_push:co
```

```
\seq_push:cX
```

```
\seq_gpush:Nn
```

```
\seq_gpush:NV
```

```
\seq_gpush:Nv
```

```

4566 \cs_new_eq:NN \seq_push:NV \seq_put_left:NV
4567 \cs_new_eq:NN \seq_push:Nv \seq_put_left:Nv
4568 \cs_new_eq:NN \seq_push:No \seq_put_left:No
4569 \cs_new_eq:NN \seq_push:Nx \seq_put_left:Nx
4570 \cs_new_eq:NN \seq_push:cn \seq_put_left:cn
4571 \cs_new_eq:NN \seq_push:cV \seq_put_left:cV
4572 \cs_new_eq:NN \seq_push:cv \seq_put_left:cv
4573 \cs_new_eq:NN \seq_push:co \seq_put_left:co
4574 \cs_new_eq:NN \seq_push:cx \seq_put_left:cx
4575 \cs_new_eq:NN \seq_gpush:Nn \seq_gput_left:Nn
4576 \cs_new_eq:NN \seq_gpush:NV \seq_gput_left:NV
4577 \cs_new_eq:NN \seq_gpush:Nv \seq_gput_left:Nv
4578 \cs_new_eq:NN \seq_gpush:No \seq_gput_left:No
4579 \cs_new_eq:NN \seq_gpush:Nx \seq_gput_left:Nx
4580 \cs_new_eq:NN \seq_gpush:cn \seq_gput_left:cn
4581 \cs_new_eq:NN \seq_gpush:cV \seq_gput_left:cV
4582 \cs_new_eq:NN \seq_gpush:cv \seq_gput_left:cv
4583 \cs_new_eq:NN \seq_gpush:co \seq_gput_left:co
4584 \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 65.)

\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

```

4585 \cs_new_eq:NN \seq_get:NN \seq_get_left:NN
4586 \cs_new_eq:NN \seq_get:cN \seq_get_left:cN
\seq_gpop:NN
4587 \cs_new_eq:NN \seq_gpop:NN \seq_gpop_left:NN
\seq_gpop:cN
4588 \cs_new_eq:NN \seq_gpop:cN \seq_gpop_left:cN
4589 \cs_new_eq:NN \seq_gpop:NN \seq_gpop_left:NN
4590 \cs_new_eq:NN \seq_gpop:cN \seq_gpop_left:cN

```

(End definition for `\seq_get:NN`, `\seq_gpop:NN`, and `\seq_gpop:cN`. These functions are documented on page 64.)

\seq_get:NNTF More copies.

```

\seq_get:cNTF
4591 \prg_new_eq_conditional:NNn \seq_get:NN \seq_get_left:NN { T , F , TF }
\seq_pop:NNTF
4592 \prg_new_eq_conditional:NNn \seq_get:cN \seq_get_left:cN { T , F , TF }
\seq_pop:cNTF
4593 \prg_new_eq_conditional:NNn \seq_pop:NN \seq_pop_left:NN { T , F , TF }
\seq_gpop:NNTF
4594 \prg_new_eq_conditional:NNn \seq_gpop:cN \seq_gpop_left:cN { T , F , TF }
\seq_gpop:cNTF
4595 \prg_new_eq_conditional:NNn \seq_gpop:NN \seq_gpop_left:NN { T , F , TF }
4596 \prg_new_eq_conditional:NNn \seq_gpop:cN \seq_gpop_left:cN { T , F , TF }

```

(End definition for `\seq_get:NNTF`, `\seq_gpop:NNTF`, and `\seq_gpop:cNTF`. These functions are documented on page 64.)

7.9 Viewing sequences

\seq_show:N Apply the general `__msg_show_variable:NNNnn`.

\seq_show:c

```

4597 \cs_new_protected:Npn \seq_show:N #1
4598 {
4599     \__msg_show_variable:NNNnn #1
4600     \seq_if_exist:NTF \seq_if_empty:NTF { seq }
4601     { \seq_map_function:NN #1 \__msg_show_item:n }
4602 }
4603 \cs_generate_variant:Nn \seq_show:N { c }

```

(End definition for `\seq_show:N`. This function is documented on page 67.)

`\seq_log:N` Redirect output of `\seq_show:N` to the log.
`\seq_log:c` 4604 `\cs_new_protected:Npn \seq_log:N`
4605 { `_msg_log_next: \seq_show:N` }
4606 `\cs_generate_variant:Nn \seq_log:N { c }`

(End definition for `\seq_log:N`. This function is documented on page 67.)

7.10 Scratch sequences

`\l_tmpa_seq` Temporary comma list variables.
`\l_tmpb_seq` 4607 `\seq_new:N \l_tmpa_seq`
`\g_tmpa_seq` 4608 `\seq_new:N \l_tmpb_seq`
`\g_tmpb_seq` 4609 `\seq_new:N \g_tmpa_seq`
4610 `\seq_new:N \g_tmpb_seq`

(End definition for `\l_tmpa_seq` and others. These variables are documented on page 67.)

4611 `\langle *initex | package \rangle`

8 l3int implementation

4612 `\langle *initex | package \rangle`
4613 `\langle @@=int \rangle`

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 78.)

`_int_to_roman:w` Done in l3basics.

`\if_int_compare:w` (End definition for `_int_to_roman:w` and `\if_int_compare:w`.)

`\or:` Done in l3basics.

(End definition for `\or:`. This function is documented on page 79.)

`_int_value:w` Here are the remaining primitives for number comparisons and expressions.

`_int_eval:w` 4614 `\cs_new_eq:NN _int_value:w \tex_number:D`
`_int_eval_end:` 4615 `\cs_new_eq:NN _int_eval:w \etex_numexpr:D`
`\if_int_odd:w` 4616 `\cs_new_eq:NN _int_eval_end: \tex_relax:D`
`\if_case:w` 4617 `\cs_new_eq:NN \if_int_odd:w \tex_ifodd:D`
4618 `\cs_new_eq:NN \if_case:w \tex_ifcase:D`

(End definition for `_int_value:w` and others.)

8.1 Integer expressions

\int_eval:n

Wrapper for `__int_eval:w`: can be used in an integer expression or directly in the input stream.

```
4619 \cs_new:Npn \int_eval:n #1
4620   { \__int_value:w \__int_eval:w #1 \__int_eval_end: }
```

(End definition for `\int_eval:n`. This function is documented on page 68.)

\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.

```
4621 \cs_new:Npn \int_abs:n #1
4622   {
4623     \__int_value:w \exp_after:wN \__int_abs:N
4624     \__int_value:w \__int_eval:w #1 \__int_eval_end:
4625     \exp_stop_f:
4626   }
4627 \cs_new:Npn \__int_abs:N #1
4628   { \if_meaning:w - #1 \else: \exp_after:wN #1 \fi: }
4629 \cs_set:Npn \int_max:nn #1#2
4630   {
4631     \__int_value:w \exp_after:wN \__int_maxmin:wwN
4632     \__int_value:w \__int_eval:w #1 \exp_after:wN ;
4633     \__int_value:w \__int_eval:w #2 ;
4634     >
4635     \exp_stop_f:
4636   }
4637 \cs_set:Npn \int_min:nn #1#2
4638   {
4639     \__int_value:w \exp_after:wN \__int_maxmin:wwN
4640     \__int_value:w \__int_eval:w #1 \exp_after:wN ;
4641     \__int_value:w \__int_eval:w #2 ;
4642     <
4643     \exp_stop_f:
4644   }
4645 \cs_new:Npn \__int_maxmin:wwN #1 ; #2 ; #3
4646   {
4647     \if_int_compare:w #1 #3 #2 ~
4648     #1
4649     \else:
4650     #2
4651     \fi:
4652   }
```

(End definition for `\int_abs:n` and others. These functions are documented on page 68.)

\int_div_truncate:nn

\int_div_round:nn

\int_mod:nn

`__int_div_truncate:NwNw`
`__int_mod:ww`

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 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.

```

4653 \cs_new:Npn \int_div_truncate:nn #1#2
4654 {
4655     __int_value:w __int_eval:w
4656     \exp_after:wN __int_div_truncate:NwNw
4657     __int_value:w __int_eval:w #1 \exp_after:wN ;
4658     __int_value:w __int_eval:w #2 ;
4659     __int_eval_end:
4660 }
4661 \cs_new:Npn __int_div_truncate:NwNw #1#2; #3#4;
4662 {
4663     \if_meaning:w 0 #1
4664     0
4665     \else:
4666     (
4667         #1#2
4668         \if_meaning:w - #1 + \else: - \fi:
4669         ( \if_meaning:w - #3 - \fi: #3#4 - 1 ) / 2
4670     )
4671     \fi:
4672     / #3#4
4673 }
```

For the sake of completeness:

```

4674 \cs_new:Npn \int_div_round:nn #1#2
4675 { __int_value:w __int_eval:w ( #1 ) / ( #2 ) __int_eval_end: }
```

Finally there's the modulus operation.

```

4676 \cs_new:Npn \int_mod:nn #1#2
4677 {
4678     __int_value:w __int_eval:w \exp_after:wN __int_mod:ww
4679     __int_value:w __int_eval:w #1 \exp_after:wN ;
4680     __int_value:w __int_eval:w #2 ;
4681     __int_eval_end:
4682 }
4683 \cs_new:Npn __int_mod:ww #1; #2;
4684 { #1 - ( __int_div_truncate:NwNw #1 ; #2 ; ) * #2 }
```

(End definition for `\int_div_truncate:nn` and others. These functions are documented on page 69.)

8.2 Creating and initialising integers

`\int_new:N` Two ways to do this: one for the format and one for the L^AT_EX 2_ε package. In plain T_EX, `\int_new:c` `\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.)

```

4685 (*package)
4686 \cs_new_protected:Npn \int_new:N #1
4687 {
4688     __chk_if_free_cs:N #1
4689     \cs:w newcount \cs_end: #1
4690 }
4691 
```

```
\cs_generate_variant:Nn \int_new:N { c }
```

(End definition for `\int_new:N`. This function is documented on page 69.)

`\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.

```
4693 \cs_new_protected:Npn \int_const:Nn #1#2
4694 {
4695     \int_compare:nNnTF {#2} < \c_zero
4696     {
4697         \int_new:N #1
4698         \int_gset:Nn #1 {#2}
4699     }
4700     {
4701         \int_compare:nNnTF {#2} > \c_max_constdef_int
4702         {
4703             \int_new:N #1
4704             \int_gset:Nn #1 {#2}
4705         }
4706         {
4707             \__chk_if_free_cs:N #1
4708             \tex_global:D \__int_constdef:Nw #1 =
4709                 \__int_eval:w #2 \__int_eval_end:
4710         }
4711     }
4712 }
4713 \cs_generate_variant:Nn \int_const:Nn { c }
4714 \if_int_odd:w 0
4715     \cs_if_exist:NT \luatex_luatexversion:D { 1 }
4716     \cs_if_exist:NT \uptex_disablecjktoken:D
4717         { \if_int_compare:w \ptex_jis:D "2121 = "3000 ~ 1 \fi: }
4718     \cs_if_exist:NT \xetex_XeTeXversion:D { 1 } ~
4719         \cs_if_exist:NTF \uptex_disablecjktoken:D
4720             { \cs_new_eq:NN \__int_constdef:Nw \uptex_kchardef:D }
4721             { \cs_new_eq:NN \__int_constdef:Nw \tex_chardef:D }
4722             \__int_constdef:Nw \c_max_constdef_int 1114111 ~
4723 \else:
4724     \cs_new_eq:NN \__int_constdef:Nw \tex_mathchardef:D
4725     \tex_mathchardef:D \c_max_constdef_int 32767 ~
4726 \fi:
```

(End definition for `\int_const:Nn`, `__int_constdef:Nw`, and `\c_max_constdef_int`. These functions are documented on page 69.)

`\int_zero:N` Functions that reset an `<integer>` register to zero.

```
4727 \cs_new_protected:Npn \int_zero:N #1 { #1 = \c_zero }
4728 \cs_new_protected:Npn \int_gzero:N #1 { \tex_global:D #1 = \c_zero }
4729 \cs_generate_variant:Nn \int_zero:N { c }
4730 \cs_generate_variant:Nn \int_gzero:N { c }
```

(End definition for `\int_zero:N` and `\int_gzero:N`. These functions are documented on page 69.)

`\int_zero_new:N` Create a register if needed, otherwise clear it.

```
4731 \cs_new_protected:Npn \int_zero_new:N #1
4732     { \int_if_exist:NTF #1 { \int_zero:N #1 } { \int_new:N #1 } }
```

`\int_gzero_new:c`

```

4733 \cs_new_protected:Npn \int_gzero_new:N #1
4734   { \int_if_exist:NTF #1 { \int_gzero:N #1 } { \int_new:N #1 } }
4735 \cs_generate_variant:Nn \int_zero_new:N { c }
4736 \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 69.)

\int_set_eq:NN Setting equal means using one integer inside the set function of another.

```

\int_set_eq:cN 4737 \cs_new_protected:Npn \int_set_eq:NN #1#2 { #1 = #2 }
\int_set_eq:Nc 4738 \cs_generate_variant:Nn \int_set_eq:NN { c }
\int_set_eq:cc 4739 \cs_generate_variant:Nn \int_set_eq:NN { Nc , cc }
\int_gset_eq:NN 4740 \cs_new_protected:Npn \int_gset_eq:NN #1#2 { \tex_global:D #1 = #2 }
\int_gset_eq:cN 4741 \cs_generate_variant:Nn \int_gset_eq:NN { c }
\int_gset_eq:Nc 4742 \cs_generate_variant:Nn \int_gset_eq:NN { Nc , cc }
\int_gset_eq:cc

```

(End definition for `\int_set_eq:NN` and `\int_gset_eq:NN`. These functions are documented on page 69.)

\int_if_exist_p:N Copies of the `cs` functions defined in `l3basics`.

```

\int_if_exist_p:c 4743 \prg_new_eq_conditional:NNN \int_if_exist:N \cs_if_exist:N
\int_if_exist:NTF 4744 { TF , T , F , p }
\int_if_exist:cTF 4745 \prg_new_eq_conditional:NNN \int_if_exist:c \cs_if_exist:c
4746 { TF , T , F , p }

```

(End definition for `\int_if_exist:NTF`. This function is documented on page 70.)

8.3 Setting and incrementing integers

\int_add:Nn Adding and subtracting to and from a counter ...

```

\int_add:cn 4747 \cs_new_protected:Npn \int_add:Nn #1#2
\int_gadd:Nn 4748 { \tex_advance:D #1 by \__int_eval:w #2 \__int_eval_end: }
\int_gadd:cn 4749 \cs_new_protected:Npn \int_sub:Nn #1#2
\int_sub:Nn 4750 { \tex_advance:D #1 by - \__int_eval:w #2 \__int_eval_end: }
\int_sub:cn 4751 \cs_new_protected:Npn \int_gadd:Nn
\int_gsub:Nn 4752 { \tex_global:D \int_add:Nn }
\int_gsub:cn 4753 \cs_new_protected:Npn \int_gsub:Nn
4754 { \tex_global:D \int_sub:Nn }
4755 \cs_generate_variant:Nn \int_add:Nn { c }
4756 \cs_generate_variant:Nn \int_gadd:Nn { c }
4757 \cs_generate_variant:Nn \int_sub:Nn { c }
4758 \cs_generate_variant:Nn \int_gsub:Nn { c }

```

(End definition for `\int_add:Nn` and others. These functions are documented on page 70.)

\int_incr:N Incrementing and decrementing of integer registers is done with the following functions.

```

\int_incr:c 4759 \cs_new_protected:Npn \int_incr:N #1
\int_gincr:N 4760 { \tex_advance:D #1 \c_one }
\int_gincr:c 4761 \cs_new_protected:Npn \int_decr:N #1
\int_decr:N 4762 { \tex_advance:D #1 - \c_one }
\int_decr:c 4763 \cs_new_protected:Npn \int_gincr:N
\int_gdecr:N 4764 { \tex_global:D \int_incr:N }
\int_gdecr:c 4765 \cs_new_protected:Npn \int_gdecr:N
4766 { \tex_global:D \int_decr:N }
4767 \cs_generate_variant:Nn \int_incr:N { c }

```

```

4768 \cs_generate_variant:Nn \int_decr:N { c }
4769 \cs_generate_variant:Nn \int_gincr:N { c }
4770 \cs_generate_variant:Nn \int_gdecr:N { c }

```

(End definition for `\int_incr:N` and others. These functions are documented on page 70.)

`\int_set:Nn` As integers are register-based T_EX will issue an error if they are not defined. Thus there is no need for the checking code seen with token list variables.

```

4771 \cs_new_protected:Npn \int_set:Nn #1#2
4772   { #1 ~ \_int_eval:w #2 \_int_eval_end: }
4773 \cs_new_protected:Npn \int_gset:Nn { \tex_global:D \int_set:Nn }
4774 \cs_generate_variant:Nn \int_set:Nn { c }
4775 \cs_generate_variant:Nn \int_gset:Nn { c }

```

(End definition for `\int_set:Nn` and `\int_gset:Nn`. These functions are documented on page 70.)

8.4 Using integers

`\int_use:N` Here is how counters are accessed:

```

4776 \cs_new_eq:NN \int_use:N \tex_the:D

```

We hand-code this for some speed gain:

```

4777 \% \cs_generate_variant:Nn \int_use:N { c }
4778 \cs_new:Npn \int_use:c #1 f \tex_the:D \cs:w #1 \cs_end: }

```

(End definition for `\int_use:N`. This function is documented on page 70.)

8.5 Integer expression conditionals

`_prg_compare_error:` `_prg_compare_error:Nw` 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. The tests first evaluate their left-hand side, with a trailing `_prg_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 T_EX 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 `_prg_compare_error:Nw` gets expanded, cleaning up the end of the test and telling the user what the problem was.

```

4779 \cs_new_protected:Npn \_prg_compare_error:
4780   {
4781     \if_int_compare:w \c_zero \c_zero \fi:
4782     =
4783     \_prg_compare_error:
4784   }
4785 \cs_new:Npn \_prg_compare_error:Nw
4786   #1#2 \q_stop
4787   {
4788     {}
4789     \c_zero \fi:
4790     \_msg_kernel_expandable_error:nnn
4791       { kernel } { unknown-comparison } {#1}
4792     \prg_return_false:
4793   }

```

(End definition for `_prg_compare_error:` and `_prg_compare_error:Nw`.)

```

\int_compare_p:n
\int_compare:nTF
  \_int_compare:w
  \_int_compare:Nw
  \_int_compare:NNw
  \_int_compare:nnN
\_int_compare_end_=:NNw
  \_int_compare_=:NNw
  \_int_compare_<:NNw
  \_int_compare_>:NNw
\_int_compare_==:NNw
\_int_compare_!=:NNw
\_int_compare_<=:NNw
\_int_compare_>=:NNw

```

Comparison tests using a simple syntax where only one set of braces is required, additional operators such as `!=` and `>=` are supported, and multiple comparisons can be performed at once, for instance `0 < 5 <= 1`. The idea is to loop through the argument, finding one operand at a time, and comparing it to the previous one. The looping auxiliary `_int_compare:Nw` reads one `(operand)` and one `(comparison)` symbol, and leaves roughly

```

  ⟨operand⟩ \prg_return_false: \fi:
  \reverse_if:N \if_int_compare:w ⟨operand⟩ ⟨comparison⟩
  \_int_compare:Nw

```

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 will terminate it. If the argument contains no relation symbol, `_prg_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 will be ended by the two odd-looking items `e` and `{=nd_}`, with a trailing `\q_stop` used to grab the entire argument when necessary.

```

4794 \prg_new_conditional:Npnn \int_compare:n #1 { p , T , F , TF }
4795   {
4796     \exp_after:wN \_int_compare:w
4797     \_int_value:w \_int_eval:w #1 \_prg_compare_error:
4798   }
4799 \cs_new:Npn \_int_compare:w #1 \_prg_compare_error:
4800   {
4801     \exp_after:wN \if_false: \_int_value:w
4802       \_int_compare:Nw #1 e { = nd_ } \q_stop
4803   }

```

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 `_prg_compare_error:Nw` raises an error.

```

4804 \cs_new:Npn \_int_compare:Nw #1#2 \q_stop
4805   {
4806     \exp_after:wN \_int_compare:NNw
4807       \_int_to_roman:w - 0 #2 \q_mark
4808       #1#2 \q_stop
4809   }
4810 \cs_new:Npn \_int_compare:NNw #1#2#3 \q_mark

```

```

4811  {
4812    \etex_unexpanded:D
4813    \use:c
4814    {
4815      __int_compare_ \token_to_str:N #1
4816      \if_meaning:w = #2 = \fi:
4817      :NNw
4818    }
4819    \__prg_compare_error:Nw #1
4820  }

```

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.

```

4821 \cs_new:cpn { __int_compare_end_=:NNw } #1#2#3 e #4 \q_stop
4822  {
4823    {#3} \exp_stop_f:
4824    \prg_return_false: \else: \prg_return_true: \fi:
4825  }
4826 \cs_new:Npn \__int_compare:nnN #1#2#3
4827  {
4828    {#2} \exp_stop_f:
4829    \prg_return_false: \exp_after:wN \use_none_delimit_by_q_stop:w
4830    \fi:
4831    #1 #2 #3 \exp_after:wN \__int_compare:Nw \__int_value:w \__int_eval:w
4832  }

```

The actual comparisons are then simple function calls, using the relation as delimiter for a delimited argument and discarding `__prg_compare_error:Nw` (*token*) responsible for error detection.

```

4833 \cs_new:cpn { __int_compare_=:NNw } #1#2#3 =
4834  { \__int_compare:nnN { \reverse_if:N \if_int_compare:w } {#3} = }
4835 \cs_new:cpn { __int_compare_<:NNw } #1#2#3 <
4836  { \__int_compare:nnN { \reverse_if:N \if_int_compare:w } {#3} < }
4837 \cs_new:cpn { __int_compare_>:NNw } #1#2#3 >
4838  { \__int_compare:nnN { \reverse_if:N \if_int_compare:w } {#3} > }
4839 \cs_new:cpn { __int_compare_==:NNw } #1#2#3 ==
4840  { \__int_compare:nnN { \reverse_if:N \if_int_compare:w } {#3} = }
4841 \cs_new:cpn { __int_compare_!=:NNw } #1#2#3 !=
4842  { \__int_compare:nnN { \if_int_compare:w } {#3} = }
4843 \cs_new:cpn { __int_compare_<=:NNw } #1#2#3 <=
4844  { \__int_compare:nnN { \if_int_compare:w } {#3} > }
4845 \cs_new:cpn { __int_compare_>=:NNw } #1#2#3 >=
4846  { \__int_compare:nnN { \if_int_compare:w } {#3} < }

```

(End definition for `\int_compare:nTF` and others. These functions are documented on page 71.)

`\int_compare_p:nNn` More efficient but less natural in typing.

```
4847 \prg_new_conditional:Npnn \int_compare:nNn #1#2#3 { p , T , F , TF }
4848 {
4849     \if_int_compare:w \__int_eval:w #1 #2 \__int_eval:w #3 \__int_eval_end:
4850         \prg_return_true:
4851     \else:
4852         \prg_return_false:
4853     \fi:
4854 }
```

(End definition for `\int_compare:nNnTF`. This function is documented on page 71.)

`\int_case:nn` For integer cases, the first task to fully expand the check condition. The over all idea is
`\int_case:nnTF` then much the same as for `\str_case:nn(TF)` as described in l3basics.

```
4855 \cs_new:Npn \int_case:nnTF #1
4856 {
4857     \exp:w
4858     \exp_args:Nf \__int_case:nnTF { \int_eval:n {#1} }
4859 }
4860 \cs_new:Npn \int_case:nnT #1#2#3
4861 {
4862     \exp:w
4863     \exp_args:Nf \__int_case:nnTF { \int_eval:n {#1} } {#2} {#3} { }
4864 }
4865 \cs_new:Npn \int_case:nnF #1#2
4866 {
4867     \exp:w
4868     \exp_args:Nf \__int_case:nnTF { \int_eval:n {#1} } {#2} { }
4869 }
4870 \cs_new:Npn \int_case:nn #1#2
4871 {
4872     \exp:w
4873     \exp_args:Nf \__int_case:nnTF { \int_eval:n {#1} } {#2} { } { }
4874 }
4875 \cs_new:Npn \__int_case:nnTF #1#2#3#4
4876 { \__int_case:nw {#1} #2 {#1} { } \q_mark {#3} \q_mark {#4} \q_stop }
4877 \cs_new:Npn \__int_case:nw #1#2#3
4878 {
4879     \int_compare:nNnTF {#1} = {#2}
4880     { \__int_case_end:nw {#3} }
4881     { \__int_case:nw {#1} }
4882 }
4883 \cs_new_eq:NN \__int_case_end:nw \__prg_case_end:nw
```

(End definition for `\int_case:nnTF` and others. These functions are documented on page 72.)

`\int_if_odd_p:n` A predicate function.

```
4884 \prg_new_conditional:Npnn \int_if_odd:n #1 { p , T , F , TF }
4885 {
4886     \if_int_odd:w \__int_eval:w #1 \__int_eval_end:
4887         \prg_return_true:
4888     \else:
4889         \prg_return_false:
4890     \fi:
```

```

4891   }
4892 \prg_new_conditional:Npnn \int_if_even:n #1 { p , T , F , TF}
4893   {
4894     \if_int_odd:w \__int_eval:w #1 \__int_eval_end:
4895     \prg_return_false:
4896   \else:
4897     \prg_return_true:
4898   \fi:
4899 }

```

(End definition for `\int_if_odd:nTF` and `\int_if_even:nTF`. These functions are documented on page 72.)

8.6 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.

```

4900 \cs_new:Npn \int_while_do:nn #1#2
4901   {
4902     \int_compare:nT {#1}
4903     {
4904       #2
4905       \int_while_do:nn {#1} {#2}
4906     }
4907   }
4908 \cs_new:Npn \int_until_do:nn #1#2
4909   {
4910     \int_compare:nF {#1}
4911     {
4912       #2
4913       \int_until_do:nn {#1} {#2}
4914     }
4915   }
4916 \cs_new:Npn \int_do_while:nn #1#2
4917   {
4918     #2
4919     \int_compare:nT {#1}
4920     { \int_do_while:nn {#1} {#2} }
4921   }
4922 \cs_new:Npn \int_do_until:nn #1#2
4923   {
4924     #2
4925     \int_compare:nF {#1}
4926     { \int_do_until:nn {#1} {#2} }
4927   }

```

(End definition for `\int_while_do:nn` and others. These functions are documented on page 73.)

`\int_while_do:nNnn` As above but not using the more natural syntax.

```

4928 \cs_new:Npn \int_while_do:nNnn #1#2#3#4
4929   {
4930     \int_compare:nNnT {#1} #2 {#3}
4931     {
4932       #4

```

```

4933         \int_while_do:nNnn {#1} #2 {#3} {#4}
4934     }
4935   }
4936 \cs_new:Npn \int_until_do:nNnn #1#2#3#4
4937   {
4938     \int_compare:nNnF {#1} #2 {#3}
4939     {
4940       #4
4941       \int_until_do:nNnn {#1} #2 {#3} {#4}
4942     }
4943   }
4944 \cs_new:Npn \int_do_while:nNnn #1#2#3#4
4945   {
4946     #4
4947     \int_compare:nNnT {#1} #2 {#3}
4948     { \int_do_while:nNnn {#1} #2 {#3} {#4} }
4949   }
4950 \cs_new:Npn \int_do_until:nNnn #1#2#3#4
4951   {
4952     #4
4953     \int_compare:nNnF {#1} #2 {#3}
4954     { \int_do_until:nNnn {#1} #2 {#3} {#4} }
4955   }

```

(End definition for `\int_while_do:nNnn` and others. These functions are documented on page 73.)

8.7 Integer step functions

`\int_step_function:nnnN`
`_int_step:wwwN`
`_int_step:NnnnN`

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).

```

4956 \cs_new:Npn \int_step_function:nnnN #1#2#3
4957   {
4958     \exp_after:wN \_int_step:wwwN
4959     \_int_value:w \_int_eval:w #1 \exp_after:wN ;
4960     \_int_value:w \_int_eval:w #2 \exp_after:wN ;
4961     \_int_value:w \_int_eval:w #3 ;
4962   }
4963 \cs_new:Npn \_int_step:wwwN #1; #2; #3; #4
4964   {
4965     \int_compare:nNnTF {#2} > \c_zero
4966     { \_int_step:NnnnN > }
4967     {
4968       \int_compare:nNnTF {#2} = \c_zero
4969       {
4970         \_msg_kernel_expandable_error:nnn { kernel } { zero-step } {#4}
4971         \use_none:nnnn
4972       }
4973       { \_int_step:NnnnN < }
4974     }
4975     {#1} {#2} {#3} #4
4976   }

```

```

4977 \cs_new:Npn \__int_step:NnnN #1#2#3#4#5
4978 {
4979     \int_compare:nNnF {#2} #1 {#4}
4980     {
4981         #5 {#2}
4982         \exp_args:NNf \__int_step:NnnN
4983             #1 { \int_eval:n { #2 + #3 } } {#3} {#4} #5
4984     }
4985 }

```

(End definition for `\int_step_function:nnnN`, `\int_step:wwwN`, and `\int_step:NnnnN`. These functions are documented on page 74.)

```

\int_step_inline:nnn
\int_step_variable:nnnN
\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__prg_map_int` before looking for their own break point. The first argument is `\scan_stop:`, so no breaking function will recognize this break point as its own.

```

4986 \cs_new_protected:Npn \int_step_inline:nnn
4987 {
4988     \int_gincr:N \g__prg_map_int
4989     \exp_args:NNc \__int_step>NNnnnn
4990     \cs_gset_protected:Npn
4991         { __prg_map_ \int_use:N \g__prg_map_int :w }
4992 }
4993 \cs_new_protected:Npn \int_step_variable:nnnN #1#2#3#4#5
4994 {
4995     \int_gincr:N \g__prg_map_int
4996     \exp_args:NNc \__int_step>NNnnnn
4997     \cs_gset_protected:Npx
4998         { __prg_map_ \int_use:N \g__prg_map_int :w }
4999         {#1}{#2}{#3}
5000         {
5001             \tl_set:Nn \exp_not:N #4 {##1}
5002             \exp_not:n {#5}
5003         }
5004     }
5005 \cs_new_protected:Npn \__int_step>NNnnnn #1#2#3#4#5#6
5006 {
5007     #1 #2 ##1 {#6}
5008     \int_step_function:nnnN {#3} {#4} {#5} #2
5009     \__prg_break_point:Nn \scan_stop: { \int_gdecr:N \g__prg_map_int }
5010 }

```

(End definition for `\int_step_inline:nnn`, `\int_step_variable:nnnN`, and `__int_step>NNnnnn`. These functions are documented on page 74.)

8.8 Formatting integers

`\int_to_arabic:n`

Nothing exciting here.

```

5011 \cs_new_eq:NN \int_to_arabic:n \int_eval:n

```

(End definition for `\int_to_arabic:n`. This function is documented on page 74.)

\int_to_symbols:nnn **_int_to_symbols:nnnn**

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.

```

5012 \cs_new:Npn \int_to_symbols:nnn #1#2#3
5013 {
5014     \int_compare:nNnTF {#1} > {#2}
5015     {
5016         \exp_args:NNo \exp_args:No \_int_to_symbols:nnnn
5017         {
5018             \int_case:nn
5019                 { 1 + \int_mod:nn {#1 - 1} {#2} } {#3}
5020                 {#3}
5021             }
5022             {#1} {#2} {#3}
5023         }
5024         { \int_case:nn {#1} {#3} }
5025     }
5026 \cs_new:Npn \_int_to_symbols:nnnn #1#2#3#4
5027 {
5028     \exp_args:Nf \int_to_symbols:nnn
5029         { \int_div_truncate:nn {#2 - 1} {#3} } {#3} {#4}
5030         #1
5031     }

```

(End definition for `\int_to_symbols:nnn` and `_int_to_symbols:nnnn`. These functions are documented on page 75.)

\int_to_alpha:n **\int_to_Alph:n** These both use the above function with input functions that make sense for the alphabet in English.

```

5032 \cs_new:Npn \int_to_alpha:n #1
5033 {
5034     \int_to_symbols:nnn {#1} { 26 }
5035     {
5036         { 1 } { a }
5037         { 2 } { b }
5038         { 3 } { c }
5039         { 4 } { d }
5040         { 5 } { e }
5041         { 6 } { f }
5042         { 7 } { g }
5043         { 8 } { h }
5044         { 9 } { i }
5045         { 10 } { j }
5046         { 11 } { k }
5047         { 12 } { l }
5048         { 13 } { m }
5049         { 14 } { n }
5050         { 15 } { o }
5051         { 16 } { p }
5052         { 17 } { q }

```

```

5053 { 18 } { r }
5054 { 19 } { s }
5055 { 20 } { t }
5056 { 21 } { u }
5057 { 22 } { v }
5058 { 23 } { w }
5059 { 24 } { x }
5060 { 25 } { y }
5061 { 26 } { z }
5062 }
5063 }
5064 \cs_new:Npn \int_to_Alph:n #1
5065 {
5066     \int_to_symbols:nnn {#1} { 26 }
5067     {
5068         { 1 } { A }
5069         { 2 } { B }
5070         { 3 } { C }
5071         { 4 } { D }
5072         { 5 } { E }
5073         { 6 } { F }
5074         { 7 } { G }
5075         { 8 } { H }
5076         { 9 } { I }
5077         { 10 } { J }
5078         { 11 } { K }
5079         { 12 } { L }
5080         { 13 } { M }
5081         { 14 } { N }
5082         { 15 } { O }
5083         { 16 } { P }
5084         { 17 } { Q }
5085         { 18 } { R }
5086         { 19 } { S }
5087         { 20 } { T }
5088         { 21 } { U }
5089         { 22 } { V }
5090         { 23 } { W }
5091         { 24 } { X }
5092         { 25 } { Y }
5093         { 26 } { Z }
5094     }
5095 }

```

(End definition for `\int_to_alpha:n` and `\int_to_Alph:n`. These functions are documented on page 75.)

`\int_to_base:nn`
`\int_to_Base:nn`

`_int_to_base:nn`
`_int_to_Base:nn`
`_int_to_base:nnN`
`_int_to_Base:nnN`
`_int_to_base:nnnN`
`_int_to_Base:nnnN`
`_int_to_base:nnmN`
`_int_to_Base:nnmN`
`_int_to_letter:n`
`_int_to_Letter:n`

Converting from base ten (#1) to a second base (#2) starts with computing #1: if it is a complicated calculation, we shouldn't perform it twice. Then check the sign, store it, either - or `\c_empty_tl`, and feed the absolute value to the next auxiliary function.

```

5096 \cs_new:Npn \int_to_base:nn #1
5097     { \exp_args:Nf \_int_to_base:nn { \int_eval:n {#1} } }
5098 \cs_new:Npn \int_to_Base:nn #1
5099     { \exp_args:Nf \_int_to_Base:nn { \int_eval:n {#1} } }
5100 \cs_new:Npn \_int_to_base:nn #1#2

```

```

5101  {
5102    \int_compare:nNnTF {#1} < 0
5103      { \exp_args:No \__int_to_base:nnN { \use_none:n #1 } {#2} - }
5104      { \__int_to_base:nnN {#1} {#2} \c_empty_tl }
5105  }
5106 \cs_new:Npn \__int_to_Base:nn #1#2
5107  {
5108    \int_compare:nNnTF {#1} < 0
5109      { \exp_args:No \__int_to_Base:nnN { \use_none:n #1 } {#2} - }
5110      { \__int_to_Base:nnN {#1} {#2} \c_empty_tl }
5111  }

```

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.

```

5112 \cs_new:Npn \__int_to_base:nnN #1#2#3
5113  {
5114    \int_compare:nNnTF {#1} < {#2}
5115      { \exp_last_unbraced:Nf #3 { \__int_to_letter:n {#1} } }
5116      {
5117        \exp_args:Nf \__int_to_base:nnnN
5118          { \__int_to_letter:n { \int_mod:nn {#1} {#2} } }
5119          {#1}
5120          {#2}
5121          #3
5122      }
5123  }
5124 \cs_new:Npn \__int_to_base:nnnN #1#2#3#4
5125  {
5126    \exp_args:Nf \__int_to_base:nnN
5127      { \int_div_truncate:nn {#2} {#3} }
5128      {#3}
5129      #4
5130      #1
5131  }
5132 \cs_new:Npn \__int_to_Base:nnN #1#2#3
5133  {
5134    \int_compare:nNnTF {#1} < {#2}
5135      { \exp_last_unbraced:Nf #3 { \__int_to_Letter:n {#1} } }
5136      {
5137        \exp_args:Nf \__int_to_Base:nnnN
5138          { \__int_to_Letter:n { \int_mod:nn {#1} {#2} } }
5139          {#1}
5140          {#2}
5141          #3
5142      }
5143  }
5144 \cs_new:Npn \__int_to_Base:nnnN #1#2#3#4
5145  {
5146    \exp_args:Nf \__int_to_Base:nnN
5147      { \int_div_truncate:nn {#2} {#3} }

```

```

5148      {#3}
5149      #4
5150      #1
5151  }

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:.

5152 \cs_new:Npn \__int_to_letter:n #1
5153 {
5154   \exp_after:wN \exp_after:wN
5155   \if_case:w \__int_eval:w #1 - 10 \__int_eval_end:
5156     a
5157     \or: b
5158     \or: c
5159     \or: d
5160     \or: e
5161     \or: f
5162     \or: g
5163     \or: h
5164     \or: i
5165     \or: j
5166     \or: k
5167     \or: l
5168     \or: m
5169     \or: n
5170     \or: o
5171     \or: p
5172     \or: q
5173     \or: r
5174     \or: s
5175     \or: t
5176     \or: u
5177     \or: v
5178     \or: w
5179     \or: x
5180     \or: y
5181     \or: z
5182   \else: \__int_value:w \__int_eval:w #1 \exp_after:wN \__int_eval_end:
5183   \fi:
5184 }
5185 \cs_new:Npn \__int_to_Letter:n #1
5186 {
5187   \exp_after:wN \exp_after:wN
5188   \if_case:w \__int_eval:w #1 - 10 \__int_eval_end:
5189     A
5190     \or: B
5191     \or: C
5192     \or: D
5193     \or: E
5194     \or: F

```

```

5195      \or: G
5196      \or: H
5197      \or: I
5198      \or: J
5199      \or: K
5200      \or: L
5201      \or: M
5202      \or: N
5203      \or: O
5204      \or: P
5205      \or: Q
5206      \or: R
5207      \or: S
5208      \or: T
5209      \or: U
5210      \or: V
5211      \or: W
5212      \or: X
5213      \or: Y
5214      \or: Z
5215      \else: \__int_value:w \__int_eval:w #1 \exp_after:wN \__int_eval_end:
5216      \fi:
5217  }

```

(End definition for `\int_to_base:nn` and others. These functions are documented on page 76.)

`\int_to_bin:n` Wrappers around the generic function.

```

5218  \cs_new:Npn \int_to_bin:n #1
5219  { \int_to_base:nn {#1} { 2 } }
5220  \cs_new:Npn \int_to_hex:n #1
5221  { \int_to_base:nn {#1} { 16 } }
5222  \cs_new:Npn \int_to_Hex:n #1
5223  { \int_to_Base:nn {#1} { 16 } }
5224  \cs_new:Npn \int_to_oct:n #1
5225  { \int_to_base:nn {#1} { 8 } }

```

(End definition for `\int_to_bin:n` and others. These functions are documented on page 75.)

`\int_to_roman:n` The `__int_to_roman:w` primitive creates tokens of category code 12 (other). Usually, what is actually wanted is letters. The approach here is to convert the output of the primitive into letters using appropriate control sequence names. That keeps everything expandable. The loop will be terminated by the conversion of the Q.

```

5226  \cs_new:Npn \int_to_roman:n #1
5227  {
5228    \exp_after:wN \__int_to_roman:N
5229    \__int_to_roman:w \int_eval:n {#1} Q
5230  }
5231  \cs_new:Npn \__int_to_roman:N #1
5232  {
5233    \use:c { __int_to_roman_ #1 :w }
5234    \__int_to_roman:N
5235  }
5236  \cs_new:Npn \int_to_Roman:n #1
5237  {
5238    \__int_to_Roman_1:w
5239    \__int_to_Roman_c:w
5240    \__int_to_Roman_d:w
5241    \__int_to_Roman_m:w
5242    \__int_to_Roman_Q:w
5243  }

```

```

5238     \exp_after:wN \__int_to_Roman_aux:N
5239         \__int_to_roman:w \int_eval:n {#1} Q
5240     }
5241 \cs_new:Npn \__int_to_Roman_aux:N #1
5242 {
5243     \use:c { __int_to_Roman_ #1 :w }
5244     \__int_to_Roman_aux:N
5245 }
5246 \cs_new:Npn \__int_to_roman_i:w { i }
5247 \cs_new:Npn \__int_to_roman_v:w { v }
5248 \cs_new:Npn \__int_to_roman_x:w { x }
5249 \cs_new:Npn \__int_to_roman_l:w { l }
5250 \cs_new:Npn \__int_to_roman_c:w { c }
5251 \cs_new:Npn \__int_to_roman_d:w { d }
5252 \cs_new:Npn \__int_to_roman_m:w { m }
5253 \cs_new:Npn \__int_to_roman_Q:w #1 { }
5254 \cs_new:Npn \__int_to_Roman_i:w { I }
5255 \cs_new:Npn \__int_to_Roman_v:w { V }
5256 \cs_new:Npn \__int_to_Roman_x:w { X }
5257 \cs_new:Npn \__int_to_Roman_l:w { L }
5258 \cs_new:Npn \__int_to_Roman_c:w { C }
5259 \cs_new:Npn \__int_to_Roman_d:w { D }
5260 \cs_new:Npn \__int_to_Roman_m:w { M }
5261 \cs_new:Npn \__int_to_Roman_Q:w #1 { }

```

(End definition for `\int_to_roman:n` and others. These functions are documented on page 76.)

8.9 Converting from other formats to integers

Called as `__int_pass_signs:wn <signs and digits> \q_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 `\q_stop`). More precisely, it deletes any + and passes any - to the input stream, hence should be called in an integer expression.

```

5262 \cs_new:Npn \__int_pass_signs:wn #
5263 {
5264     \if:w + \if:w - \exp_not:N #1 + \fi: \exp_not:N #1
5265         \exp_after:wN \__int_pass_signs:wn
5266     \else:
5267         \exp_after:wN \__int_pass_signs_end:wn
5268         \exp_after:wN #1
5269     \fi:
5270 }
5271 \cs_new:Npn \__int_pass_signs_end:wn #1 \q_stop #2 { #2 #1 }

```

(End definition for `__int_pass_signs:wn` and `__int_pass_signs_end:wn`.)

`\int_from_alpha:n`
`__int_from_alpha:nN`
`__int_from_alpha:N`

First take care of signs then loop through the input using the `recursion` quarks. The `__int_from_alpha:nN` auxiliary collects in its first argument the value obtained so far, and the auxiliary `__int_from_alpha:N` converts one letter to an expression which evaluates to the correct number.

```

5272 \cs_new:Npn \int_from_alpha:n #1
5273 {
5274     \int_eval:n
5275     {

```

```

5276     \exp_after:wN \__int_pass_signs:wn \tl_to_str:n {#1}
5277     \q_stop { \__int_from_alpha:nN { 0 } }
5278     \q_recursion_tail \q_recursion_stop
5279   }
5280 }
5281 \cs_new:Npn \__int_from_alpha:nN #1#2
5282 {
5283   \quark_if_recursion_tail_stop_do:Nn #2 {#1}
5284   \exp_args:Nf \__int_from_alpha:nN
5285   { \int_eval:n { #1 * 26 + \__int_from_alpha:N #2 } }
5286 }
5287 \cs_new:Npn \__int_from_alpha:N #
5288 { '#1 - \int_compare:nNnTF { '#1 } < { 91 } { 64 } { 96 } }

```

(End definition for `\int_from_alpha:n`, `__int_from_alpha:nN`, and `__int_from_alpha:N`. These functions are documented on page 76.)

`\int_from_base:nn`
`__int_from_base:nnN`
`__int_from_base:N`

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.

```

5289 \cs_new:Npn \int_from_base:nn #1#2
5290 {
5291   \int_eval:n
5292   {
5293     \exp_after:wN \__int_pass_signs:wn \tl_to_str:n {#1}
5294     \q_stop { \__int_from_base:nnN { 0 } {#2} }
5295     \q_recursion_tail \q_recursion_stop
5296   }
5297 }
5298 \cs_new:Npn \__int_from_base:nnN #1#2#3
5299 {
5300   \quark_if_recursion_tail_stop_do:Nn #3 {#1}
5301   \exp_args:Nf \__int_from_base:nnN
5302   { \int_eval:n { #1 * #2 + \__int_from_base:N #3 } }
5303   {#2}
5304 }
5305 \cs_new:Npn \__int_from_base:N #
5306 {
5307   \int_compare:nNnTF { '#1 } < { 58 }
5308   { '#1 }
5309   { '#1 - \int_compare:nNnTF { '#1 } < { 91 } { 55 } { 87 } }
5310 }

```

(End definition for `\int_from_base:nn`, `__int_from_base:nnN`, and `__int_from_base:N`. These functions are documented on page 77.)

`\int_from_bin:n`
`\int_from_hex:n`
`\int_from_oct:n`

Wrappers around the generic function.

```

5311 \cs_new:Npn \int_from_bin:n #
5312 { \int_from_base:nn {#1} { 2 } }
5313 \cs_new:Npn \int_from_hex:n #
5314 { \int_from_base:nn {#1} { 16 } }
5315 \cs_new:Npn \int_from_oct:n #
5316 { \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 76.)

```
\c__int_from_roman_i_int  Constants used to convert from Roman numerals to integers.
\c__int_from_roman_v_int  5317 \int_const:cn { c__int_from_roman_i_int } { 1 }
\c__int_from_roman_x_int  5318 \int_const:cn { c__int_from_roman_v_int } { 5 }
\c__int_from_roman_l_int  5319 \int_const:cn { c__int_from_roman_x_int } { 10 }
\c__int_from_roman_c_int  5320 \int_const:cn { c__int_from_roman_l_int } { 50 }
\c__int_from_roman_d_int  5321 \int_const:cn { c__int_from_roman_c_int } { 100 }
\c__int_from_roman_m_int  5322 \int_const:cn { c__int_from_roman_d_int } { 500 }
\c__int_from_roman_I_int  5323 \int_const:cn { c__int_from_roman_m_int } { 1000 }
\c__int_from_roman_V_int  5324 \int_const:cn { c__int_from_roman_I_int } { 1 }
\c__int_from_roman_X_int  5325 \int_const:cn { c__int_from_roman_V_int } { 5 }
\c__int_from_roman_L_int  5326 \int_const:cn { c__int_from_roman_X_int } { 10 }
\c__int_from_roman_C_int  5327 \int_const:cn { c__int_from_roman_L_int } { 50 }
\c__int_from_roman_D_int  5328 \int_const:cn { c__int_from_roman_C_int } { 100 }
\c__int_from_roman_M_int  5329 \int_const:cn { c__int_from_roman_D_int } { 500 }
                                         5330 \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`
`__int_from_roman>NN`
`__int_from_roman_error:w`

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.

```
5331 \cs_new:Npn \int_from_roman:n #1
5332   {
5333     \int_eval:n
5334     {
5335       (
5336         0
5337         \exp_after:wN \_\_int_from_roman>NN \tl_to_str:n {#1}
5338         \q_recursion_tail \q_recursion_tail \q_recursion_stop
5339       )
5340     }
5341   }
5342 \cs_new:Npn \_\_int_from_roman>NN #1#2
5343   {
5344     \quark_if_recursion_tail_stop:N #1
5345     \int_if_exist:cF { c__int_from_roman_ #1 _int }
5346     { \_\_int_from_roman_error:w }
5347     \quark_if_recursion_tail_stop_do:Nn #2
5348     { + \use:c { c__int_from_roman_ #1 _int } }
5349     \int_if_exist:cF { c__int_from_roman_ #2 _int }
5350     { \_\_int_from_roman_error:w }
5351     \int_compare:nNnTF
5352     { \use:c { c__int_from_roman_ #1 _int } }
5353     <
5354     { \use:c { c__int_from_roman_ #2 _int } }
5355     {
5356       + \use:c { c__int_from_roman_ #2 _int }
5357       - \use:c { c__int_from_roman_ #1 _int }
5358       \_\_int_from_roman>NN
5359     }
```

```

5360      {
5361        + \use:c { c__int_from_roman_ #1 _int }
5362        \__int_from_roman:NN #2
5363      }
5364    }
5365 \cs_new:Npn \__int_from_roman_error:w #1 \q_recursion_stop #2
5366   { #2 * 0 - 1 }

(End definition for \int_from_roman:n, \__int_from_roman:NN, and \__int_from_roman_error:w.
These functions are documented on page 77.)

```

8.10 Viewing integer

\int_show:N Diagnostics.

```

\int_show:c 5367 \cs_new_eq:NN \int_show:N \__kernel_register_show:N
\__int_show:nN 5368 \cs_generate_variant:Nn \int_show:N { c }

```

(End definition for \int_show:N and __int_show:nN. These functions are documented on page 77.)

\int_show:n We don't use the TeX primitive \showthe to show integer expressions: this gives a more unified output.

```

5369 \cs_new_protected:Npn \int_show:n
5370   { \__msg_show_wrap:Nn \int_eval:n }

```

(End definition for \int_show:n. This function is documented on page 77.)

\int_log:N Diagnostics.

```

\int_log:c 5371 \cs_new_eq:NN \int_log:N \__kernel_register_log:N
5372 \cs_generate_variant:Nn \int_log:N { c }

```

(End definition for \int_log:N. This function is documented on page 77.)

\int_log:n Redirect output of \int_show:n to the log.

```

5373 \cs_new_protected:Npn \int_log:n
5374   { \__msg_log_next: \int_show:n }

```

(End definition for \int_log:n. This function is documented on page 77.)

8.11 Constant integers

\c_zero Again, in l3basics

(End definition for \c_zero. This variable is documented on page 78.)

\c_one Low-number values not previously defined.

```

\c_two 5375 \int_const:Nn \c_one     { 1 }
\c_three 5376 \int_const:Nn \c_two     { 2 }
\c_four 5377 \int_const:Nn \c_three   { 3 }
\c_five 5378 \int_const:Nn \c_four    { 4 }
\c_six 5379 \int_const:Nn \c_five    { 5 }
\c_seven 5380 \int_const:Nn \c_six    { 6 }
\c_eight 5381 \int_const:Nn \c_seven  { 7 }
\c_nine 5382 \int_const:Nn \c_eight  { 8 }
\c_ten 5383 \int_const:Nn \c_nine   { 9 }
\c_eleven 5384 \int_const:Nn \c_ten   { 10 }
\c_twelve
\c_thirteen
\c_fourteen
\c_fifteen
\c_sixteen

```

```

5385 \int_const:Nn \c_eleven { 11 }
5386 \int_const:Nn \c_twelve { 12 }
5387 \int_const:Nn \c_thirteen { 13 }
5388 \int_const:Nn \c_fourteen { 14 }
5389 \int_const:Nn \c_fifteen { 15 }
5390 \int_const:Nn \c_sixteen { 16 }

```

(End definition for `\c_one` and others. These variables are documented on page 78.)

`\c_thirty_two` One middling value.

```
5391 \int_const:Nn \c_thirty_two { 32 }
```

(End definition for `\c_thirty_two`. This variable is documented on page 78.)

`\c_two_hundred_fifty_five` Two classic mid-range integer constants.

```

\c_two_hundred_fifty_six
5392 \int_const:Nn \c_two_hundred_fifty_five { 255 }
5393 \int_const:Nn \c_two_hundred_fifty_six { 256 }

```

(End definition for `\c_two_hundred_fifty_five` and `\c_two_hundred_fifty_six`. These variables are documented on page 78.)

`\c_one_hundred` Simple runs of powers of ten.

```

\c_one_thousand
\c_ten_thousand
5394 \int_const:Nn \c_one_hundred { 100 }
5395 \int_const:Nn \c_one_thousand { 1000 }
5396 \int_const:Nn \c_ten_thousand { 10000 }

```

(End definition for `\c_one_hundred`, `\c_one_thousand`, and `\c_ten_thousand`. These variables are documented on page 78.)

`\c_max_int` The largest number allowed is $2^{31} - 1$

```
5397 \int_const:Nn \c_max_int { 2 147 483 647 }
```

(End definition for `\c_max_int`. This variable is documented on page 78.)

`\c_max_char_int` The largest character code is 1114111 (hexadecimal 10FFFF) in X_ET_EX and LuaT_EX and 255 in other engines. In many places pT_EX and upT_EX support larger character codes but for instance the values of `\lccode` are restricted to [0, 255].

```

5398 \int_const:Nn \c_max_char_int
5399 {
5400     \if_int_odd:w 0
5401         \cs_if_exist:NT \luatex_luatexversion:D { 1 }
5402         \cs_if_exist:NT \xetex_XeTeXversion:D { 1 } ~
5403         "10FFFF
5404     \else:
5405         "FF
5406     \fi:
5407 }

```

(End definition for `\c_max_char_int`. This variable is documented on page 78.)

8.12 Scratch integers

```
\l_tmpa_int  We provide two local and two global scratch counters, maybe we need more or less.  
\l_tmpb_int  
5408 \int_new:N \l_tmpa_int  
\g_tmpa_int  
5409 \int_new:N \l_tmpb_int  
\g_tmpa_int  
5410 \int_new:N \g_tmpa_int  
5411 \int_new:N \g_tmpb_int
```

(End definition for `\l_tmpa_int` and others. These variables are documented on page 78.)

8.13 Deprecated

`\c_minus_one` The actual allocation mechanism is in `\B3alloc`; it requires `\c_one` to be defined. In package mode, reuse `\m@ne`.

```
5412 /*package)  
5413 \cs_new_eq:NN \c_minus_one \m@ne  
5414 
```

```
5415 /*initex)  
5416 \int_const:Nn \c_minus_one { -1 }  
5417 
```

(End definition for `\c_minus_one`.)

```
5418 
```

9 `\B3intarray` implementation

```
5419 /*initex | package)
```

```
5420 (@@=intarray)
```

9.1 Allocating arrays

`\g_intarray_font_int` Used to assign one font per array.

```
5421 \int_new:N \g_intarray_font_int
```

(End definition for `\g_intarray_font_int`.)

`__intarray_new:Nn` Declare #1 to be a font (arbitrarily `cmr10` at a never-used size). Store the array's size as the `\hyphenchar` of that font and make sure enough `\fontdimen` are allocated, by setting the last one. Then clear any `\fontdimen` that `cmr10` starts with. It seems `\TeX`'s `cmr10` has an extra `\fontdimen` parameter number 8 compared to other engines (for a math font we would replace 8 by 22 or some such).

```
5422 \cs_new_protected:Npn \__intarray_new:Nn #1#2  
5423 {  
5424     \__chk_if_free_cs:N #1  
5425     \int_gincr:N \g_intarray_font_int  
5426     \tex_global:D \tex_font:D #1 = cmr10~at~ \g_intarray_font_int sp \scan_stop:  
5427     \tex_hyphenchar:D #1 = \int_eval:n {#2} \scan_stop:  
5428     \int_compare:nNnT { \tex_hyphenchar:D #1 } > 0  
5429         { \tex_fontdimen:D \tex_hyphenchar:D #1 #1 = 0 sp \scan_stop: }  
5430     \int_step_inline:nnnn { 1 } { 1 } { 8 }  
5431         { \tex_fontdimen:D ##1 #1 = 0 sp \scan_stop: }  
5432 }
```

(End definition for `_intarray_new:Nn`.)

`_intarray_count:N` Size of an array.

```
5433 \cs_new:Npn \_intarray_count:N #1 { \tex_the:D \tex_hyphenchar:D #1 }
```

(End definition for `_intarray_count:N`.)

9.2 Array items

`_intarray_gset:Nnn` Set the appropriate `\fontdimen`. The slow version checks the position and value are within bounds.

```
\_intarray_gset_aux:Nnn
5434 \cs_new_protected:Npn \_intarray_gset_fast:Nnn #1#2#3
5435   { \tex_fontdimen:D \int_eval:n {#2} #1 = \int_eval:n {#3} sp \scan_stop: }
5436 \cs_new_protected:Npn \_intarray_gset:Nnn #1#2#3
5437   {
5438     \exp_args:Nff \_intarray_gset_aux:Nnn #1
5439     { \int_eval:n {#2} } { \int_eval:n {#3} }
5440   }
5441 \cs_new_protected:Npn \_intarray_gset_aux:Nnn #1#2#3
5442   {
5443     \int_compare:nTF { 1 <= #2 <= \_intarray_count:N #1 }
5444     {
5445       \int_compare:nTF { - \c_max_dim <= \int_abs:n {#3} <= \c_max_dim }
5446         { \_intarray_gset_fast:Nnn #1 {#2} {#3} }
5447         {
5448           \_msg_kernel_error:nnxxxx { kernel } { overflow }
5449           { \token_to_str:N #1 } {#2} {#3}
5450           { \int_compare:nNnT {#3} < 0 { - } \int_value:w \c_max_dim }
5451           \_intarray_gset_fast:Nnn #1 {#2}
5452             { \int_compare:nNnT {#3} < 0 { - } \c_max_dim }
5453         }
5454     }
5455   {
5456     \_msg_kernel_error:nnxxxx { kernel } { out-of-bounds }
5457     { \token_to_str:N #1 } {#2} { \_intarray_count:N #1 }
5458   }
5459 }
```

(End definition for `_intarray_gset:Nnn`, `_intarray_gset_fast:Nnn`, and `_intarray_gset_aux:Nnn`.)

`_intarray_item:Nn` Get the appropriate `\fontdimen` and perform bound checks if requested.

```
\_intarray_item_fast:Nn
\_intarray_item_aux:Nn
5460 \cs_new:Npn \_intarray_item_fast:Nn #1#
5461   { \_int_value:w \tex_fontdimen:D \int_eval:n {#2} #1 }
5462 \cs_new:Npn \_intarray_item:Nn #1#
5463   { \exp_args:Nf \_intarray_item_aux:Nn #1 { \int_eval:n {#2} } }
5464 \cs_new:Npn \_intarray_item_aux:Nn #1#
5465   {
5466     \int_compare:nTF { 1 <= #2 <= \_intarray_count:N #1 }
5467       { \_intarray_item_fast:Nn #1 {#2} }
5468     {
5469       \_msg_kernel_expandable_error:nnnnn { kernel } { out-of-bounds }
5470         { \token_to_str:N #1 } {#2} { \_intarray_count:N #1 }
5471       0
5472     }
5473   }
```

```

5472     }
5473   }

(End definition for \__intarray_item:Nn, \__intarray_item_fast:Nn, and \__intarray_item_aux:Nn.)
5474 </initex | package>

```

10 I3flag implementation

```

5475 <*initex | package>
5476 <@C=flag>

```

The following test files are used for this code: m3flag001.

10.1 Non-expandable flag commands

The height h of a flag (initially zero) is stored by setting control sequences of the form $\text{\flag} \langle\text{name}\rangle \langle\text{integer}\rangle$ to \relax for $0 \leq \langle\text{integer}\rangle < h$. When a flag is raised, a “trap” function $\text{\flag} \langle\text{name}\rangle$ 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!

```

5477 \cs_new_protected:Npn \flag_new:n #1
5478   {
5479     \cs_new:cpn { flag~#1 } ##1 ;
5480     { \exp_after:wN \use_none:n \cs:w flag~#1~##1 \cs_end: }
5481   }

```

(End definition for \flag_new:n . This function is documented on page 82.)

\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.

```

5482 \cs_new_protected:Npn \flag_clear:n { \__flag_clear:wn 0 ; }
5483 \cs_new_protected:Npn \__flag_clear:wn #1 ; #2
5484   {
5485     \if_cs_exist:w flag~#2~#1 \cs_end:
5486       \cs_set_eq:cN { flag~#2~#1 } \tex_undefined:D
5487       \exp_after:wN \__flag_clear:wn
5488       \__int_value:w \__int_eval:w 1 + #1
5489     \else:
5490       \use_i:nnn
5491     \fi:
5492   ; {#2}
5493 }

```

(End definition for \flag_clear:n and __flag_clear:wn . These functions are documented on page 82.)

\flag_clear_new:n As for other datatypes, clear the $\langle\text{flag}\rangle$ or create a new one, as appropriate.

```

5494 \cs_new_protected:Npn \flag_clear_new:n #1
5495   { \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 82.)

```

\flag_show:n Show the height (terminal or log file) using appropriate l3msg auxiliaries.
\flag_log:n
5496 \cs_new_protected:Npn \flag_show:n #1
5497 {
5498     \exp_args:Nc \__msg_show_variable:NNNnn { flag~#1 } \cs_if_exist:NTF ? { }
5499     { > ~ flag ~ #1 ~ height = \flag_height:n {#1} }
5500 }
5501 \cs_new_protected:Npn \flag_log:n
5502 { \__msg_log_next: \flag_show:n }

```

(End definition for `\flag_show:n` and `\flag_log:n`. These functions are documented on page 82.)

10.2 Expandable flag commands

```

\flag_if_exist_p:n A flag exist if the corresponding trap \flag <flag name>:n is defined.
\flag_if_exist:nTF
5503 \prg_new_conditional:Npnn \flag_if_exist:n #1 { p , T , F , TF }
5504 {
5505     \cs_if_exist:cTF { flag~#1 }
5506     { \prg_return_true: } { \prg_return_false: }
5507 }

```

(End definition for `\flag_if_exist:nTF`. This function is documented on page 83.)

```

\flag_if_raised_p:n Test if the flag has a non-zero height, by checking the 0 control sequence.
\flag_if_raised:nTF
5508 \prg_new_conditional:Npnn \flag_if_raised:n #1 { p , T , F , TF }
5509 {
5510     \if_cs_exist:w flag~#1~0 \cs_end:
5511     \prg_return_true:
5512     \else:
5513     \prg_return_false:
5514     \fi:
5515 }

```

(End definition for `\flag_if_raised:nTF`. This function is documented on page 83.)

```

\flag_height:n Extract the value of the flag by going through all of the control sequences starting from
\__flag_height_loop:wn 0.
\__flag_height_end:wn
5516 \cs_new:Npn \flag_height:n { \__flag_height_loop:wn 0; }
5517 \cs_new:Npn \__flag_height_loop:wn #1 ; #2
5518 {
5519     \if_cs_exist:w flag~#2~#1 \cs_end:
5520     \exp_after:wN \__flag_height_loop:wn \__int_value:w \__int_eval:w 1 +
5521     \else:
5522     \exp_after:wN \__flag_height_end:wn
5523     \fi:
5524     #1 ; {#2}
5525 }
5526 \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`. These functions are documented on page 83.)

\flag_raise:n Simply apply the trap to the height, after expanding the latter.

```

5527 \cs_new:Npn \flag_raise:n #1
5528 {
5529     \cs:w flag~#1 \exp_after:wN \cs_end:
5530     \__int_value:w \flag_height:n {#1} ;
5531 }
```

(End definition for `\flag_raise:n`. This function is documented on page 83.)

10.3 Option check-declarations

__flag_chk_exist:n In package mode, there is an option to check all variables used are defined. Since flags are not implemented in terms of other variables, we need to add checks by hand. Not all functions need to be patched because some are defined in terms of others.

```

5532 (*package)
5533 \tex_ifodd:D \l@expl@check@declarations@bool
5534   \cs_set_protected:Npn \flag_clear:n #1
5535   {
5536     \exp_args:Nc \__chk_if_exist_var:N { flag~#1 }
5537     \__flag_clear:wn 0 ; {#1}
5538   }
5539 \cs_set:Npn \__flag_chk_exist:n #1
5540   {
5541     \flag_if_exist:nF {#1}
5542     {
5543       \__msg_kernel_expandable_error:nnn
5544         { kernel } { bad-variable } { flag~#1~ }
5545     }
5546   }
5547 \cs_set:Npn \flag_height:n #1
5548   {
5549     \__flag_chk_exist:n {#1}
5550     \__flag_height_loop:wn 0; {#1}
5551   }
5552 \prg_set_conditional:Npnn \flag_if_raised:n #1 { p , T , F , TF }
5553   {
5554     \__flag_chk_exist:n {#1}
5555     \if cs_exist:w flag~#1~0 \cs_end:
5556       \prg_return_true:
5557     \else:
5558       \prg_return_false:
5559     \fi:
5560   }
5561 \fi:
5562 
```

(End definition for `__flag_chk_exist:n`.)

```
5563 </initex | package>
```

11 I3quark implementation

The following test files are used for this code: `m3quark001.lvt`.

```
5564 <*initex | package>
```

11.1 Quarks

5565 ⟨@@=quark⟩

\quark_new:N Allocate a new quark.

5566 \cs_new_protected:Npn \quark_new:N #1 { \tl_const:Nn #1 {#1} }

(End definition for \quark_new:N. This function is documented on page 84.)

\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.

\q_mark
\q_no_value
\q_stop
5567 \quark_new:N \q_nil
5568 \quark_new:N \q_mark
5569 \quark_new:N \q_no_value
5570 \quark_new:N \q_stop

(End definition for \q_nil and others. These variables are documented on page 85.)

\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.

5571 \quark_new:N \q_recursion_tail
5572 \quark_new:N \q_recursion_stop

(End definition for \q_recursion_tail and \q_recursion_stop. These variables are documented on page 86.)

\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 wrapped 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.

5573 \cs_new:Npn \quark_if_recursion_tail_stop:N #1
5574 {
5575 \if_meaning:w \q_recursion_tail #1
5576 \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
5577 \fi:
5578 }
5579 \cs_new:Npn \quark_if_recursion_tail_stop_do:Nn #1
5580 {
5581 \if_meaning:w \q_recursion_tail #1
5582 \exp_after:wN \use_i_delimit_by_q_recursion_stop:nw
5583 \else:
5584 \exp_after:wN \use_none:n
5585 \fi:
5586 }

(End definition for \quark_if_recursion_tail_stop:N and \quark_if_recursion_tail_stop_do:Nn. These functions are documented on page 86.)

```
\quark_if_recursion_tail_stop:n
\quark_if_recursion_tail_stop:o
\quark_if_recursion_tail_stop:nn
\quark_if_recursion_tail_stop:on
\__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`.

```
5587 \cs_new:Npn \quark_if_recursion_tail_stop:n #1
5588 {
5589     \tl_if_empty:oTF
5590     { \__quark_if_recursion_tail:w {} #1 {} ?! \q_recursion_tail ??! }
5591     { \use_none_delimit_by_q_recursion_stop:w }
5592     { }
5593 }
5594 \cs_new:Npn \quark_if_recursion_tail_stop:nn #1
5595 {
5596     \tl_if_empty:oTF
5597     { \__quark_if_recursion_tail:w {} #1 {} ?! \q_recursion_tail ??! }
5598     { \use_i_delimit_by_q_recursion_stop:nn }
5599     { \use_none:n }
5600 }
5601 \cs_new:Npn \__quark_if_recursion_tail:w
5602     #1 \q_recursion_tail #2 ? #3 ?! { #1 #2 }
5603 \cs_generate_variant:Nn \quark_if_recursion_tail_stop:n { o }
5604 \cs_generate_variant:Nn \quark_if_recursion_tail_stop:nn { o }
```

(End definition for `\quark_if_recursion_tail_stop:n`, `\quark_if_recursion_tail_stop:nn`, and `__quark_if_recursion_tail:w`. These functions are documented on page 86.)

```
\__quark_if_recursion_tail_break:NN
\__quark_if_recursion_tail_break:nN
```

Analogs of the `\quark_if_recursion_tail_stop...` functions. Break the mapping using #2.

```
5605 \cs_new:Npn \__quark_if_recursion_tail_break:NN #1#2
5606 {
5607     \if_meaning:w \q_recursion_tail #1
5608     \exp_after:wN #2
5609     \fi:
5610 }
5611 \cs_new:Npn \__quark_if_recursion_tail_break:nN #1#2
5612 {
5613     \tl_if_empty:oTF
5614     { \__quark_if_recursion_tail:w {} #1 {} ?! \q_recursion_tail ??! }
5615     {#2}
5616     { }
5617 }
```

(End definition for `__quark_if_recursion_tail_break:NN` and `__quark_if_recursion_tail_break:nN`.)

```
\quark_if_nil_p:N
\quark_if_nil:NTF
\quark_if_no_value_p:N
\quark_if_no_value_p:c
\quark_if_no_value:NTF
\quark_if_no_value:cTF
```

Here we test if we found a special quark as the first argument. We better start with `\q_no_value` as the first argument since the whole thing may otherwise loop if #1 is wrongly given a string like `aabc` instead of a single token.⁹

```
5618 \prg_new_conditional:Npnn \quark_if_nil:N #1 { p, T , F , TF }
5619 {
5620     \if_meaning:w \q_nil #1
5621     \prg_return_true:
5622     \else:
5623     \prg_return_false:
```

⁹It may still loop in special circumstances however!

```

5624     \fi:
5625   }
5626 \prg_new_conditional:Npnn \quark_if_no_value:N #1 { p, T , F , TF }
5627   {
5628     \if_meaning:w \q_no_value #1
5629       \prg_return_true:
5630     \else:
5631       \prg_return_false:
5632     \fi:
5633   }
5634 \cs_generate_variant:Nn \quark_if_no_value_p:N { c }
5635 \cs_generate_variant:Nn \quark_if_no_value:NT { c }
5636 \cs_generate_variant:Nn \quark_if_no_value:NF { c }
5637 \cs_generate_variant:Nn \quark_if_no_value:NTF { c }

```

(End definition for `\quark_if_nil:NTF` and `\quark_if_no_value:NTF`. These functions are documented on page 85.)

\quark_if_nil_p:n Let us explain `\quark_if_nil:n(TF)`. Expanding `_quark_if_nil:w` once is safe thanks to the trailing `\q_nil ??!`. The result of expanding once is empty if and only if both delimited arguments #1 and #2 are empty and #3 is delimited by the last tokens `?!`. Thanks to the leading {}, the argument #1 is empty if and only if the argument of `\quark_if_nil:n` starts with `\q_nil`. The argument #2 is empty if and only if this `\q_nil` is followed immediately by ? or by {}?, coming either from the trailing tokens in the definition of `\quark_if_nil:n`, or from its argument. In the first case, `_quark_if_nil:w` is followed by {}`\q_nil {} ? !\q_nil ??!`, hence #3 is delimited by the final `?!`, and the test returns `true` as wanted. In the second case, the result is not empty since the first `?!` in the definition of `\quark_if_nil:n` stop #3.

```

5638 \prg_new_conditional:Npnn \quark_if_nil:n #1 { p, T , F , TF }
5639   {
5640     \_tl_if_empty_return:o
5641     { \_quark_if_nil:w {} #1 {} ? ! \q_nil ? ? ! }
5642   }
5643 \cs_new:Npn \_quark_if_nil:w #1 \q_nil #2 ? #3 ? ! { #1 #2 }
5644 \prg_new_conditional:Npnn \quark_if_no_value:n #1 { p, T , F , TF }
5645   {
5646     \_tl_if_empty_return:o
5647     { \_quark_if_no_value:w {} #1 {} ? ! \q_no_value ? ? ! }
5648   }
5649 \cs_new:Npn \_quark_if_no_value:w #1 \q_no_value #2 ? #3 ? ! { #1 #2 }
5650 \cs_generate_variant:Nn \quark_if_nil_p:n { V , o }
5651 \cs_generate_variant:Nn \quark_if_nil:NTF { V , o }
5652 \cs_generate_variant:Nn \quark_if_nil:NT { V , o }
5653 \cs_generate_variant:Nn \quark_if_nil:NF { V , o }

```

(End definition for `\quark_if_nil:NTF` and others. These functions are documented on page 85.)

\q_tl_act_mark These private quarks are needed by l3tl, but that is loaded before the quark module, hence their definition is deferred.

```

5654 \quark_new:N \q_tl_act_mark
5655 \quark_new:N \q_tl_act_stop

```

(End definition for `\q_tl_act_mark` and `\q_tl_act_stop`.)

11.2 Scan marks

5656 ⟨@@=scan⟩

\g__scan_marks_tl The list of all scan marks currently declared.

5657 \tl_new:N \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.

```
5658 \cs_new_protected:Npn \_scan_new:N #1
  {
    \tl_if_in:NnTF \g__scan_marks_tl { #1 }
    {
      \__msg_kernel_error:nnx { kernel } { scanmark-already-defined }
      { \token_to_str:N #1 }
    }
    {
      \tl_gput_right:Nn \g__scan_marks_tl {#1}
      \cs_new_eq:NN #1 \scan_stop:
    }
  }
```

(End definition for _scan_new:N.)

\s__stop We only declare one scan mark here, more can be defined by specific modules.

5670 _scan_new:N \s__stop

(End definition for \s__stop.)

_use_none_delimit_by_s__stop:w Similar to \use_none_delimit_by_q_stop:w.

5671 \cs_new:Npn _use_none_delimit_by_s__stop:w #1 \s__stop { }

(End definition for _use_none_delimit_by_s__stop:w.)

\s__seq This private scan mark is needed by l3seq, but that is loaded before the quark module, hence its definition is deferred.

5672 _scan_new:N \s__seq

(End definition for \s__seq.)

5673 ⟨/initex | package⟩

12 l3prg implementation

The following test files are used for this code: m3prg001.lvt, m3prg002.lvt, m3prg003.lvt.

5674 ⟨*initex | package⟩

12.1 Primitive conditionals

\if_bool:N Those two primitive TeX conditionals are synonyms.

\if_predicate:w 5675 \cs_new_eq:NN \if_bool:N \tex_ifodd:D

5676 \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 96.)

12.2 Defining a set of conditional functions

These are all defined in `!3basics`, 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 89.*)

12.3 The boolean data type

5677 `\@=bool`

Boolean variables have to be initiated when they are created. Other than that there is not much to say here.

5678 `\cs_new_protected:Npn \bool_new:N #1 { \cs_new_eq:NN #1 \c_false_bool }`
5679 `\cs_generate_variant:Nn \bool_new:N { c }`

(*End definition for `\bool_new:N`. This function is documented on page 91.*)

`\bool_set_true:N`

Setting is already pretty easy.

5680 `\cs_new_protected:Npn \bool_set_true:N #1`
5681 `{ \cs_set_eq:NN #1 \c_true_bool }`
5682 `\cs_new_protected:Npn \bool_set_false:N #1`
5683 `{ \cs_set_eq:NN #1 \c_false_bool }`
5684 `\cs_new_protected:Npn \bool_gset_true:N #1`
5685 `{ \cs_gset_eq:NN #1 \c_true_bool }`
5686 `\cs_new_protected:Npn \bool_gset_false:N #1`
5687 `{ \cs_gset_eq:NN #1 \c_false_bool }`
5688 `\cs_generate_variant:Nn \bool_set_true:N { c }`
5689 `\cs_generate_variant:Nn \bool_set_false:N { c }`
5690 `\cs_generate_variant:Nn \bool_gset_true:N { c }`
5691 `\cs_generate_variant:Nn \bool_gset_false:N { c }`

(*End definition for `\bool_set_true:N` and others. These functions are documented on page 91.*)

`\bool_set_eq:NN`

The usual copy code.

5692 `\cs_new_eq:NN \bool_set_eq:NN \cs_set_eq:NN`
5693 `\cs_new_eq:NN \bool_set_eq:Nc \cs_set_eq:Nc`
5694 `\cs_new_eq:NN \bool_set_eq:cN \cs_set_eq:cN`
5695 `\cs_new_eq:NN \bool_set_eq:cc \cs_set_eq:cc`
5696 `\cs_new_eq:NN \bool_gset_eq:NN \cs_gset_eq:NN`
5697 `\cs_new_eq:NN \bool_gset_eq:Nc \cs_gset_eq:Nc`
5698 `\cs_new_eq:NN \bool_gset_eq:cN \cs_gset_eq:cN`
5699 `\cs_new_eq:NN \bool_gset_eq:cc \cs_gset_eq:cc`

(*End definition for `\bool_set_eq:NN` and `\bool_gset_eq:NN`. These functions are documented on page 92.*)

`\bool_set:Nn`

`\bool_set:cn`

`\bool_gset:Nn`

`\bool_gset:cn`

This function evaluates a boolean expression and assigns the first argument the meaning `\c_true_bool` or `\c_false_bool`.

5700 `\cs_new_protected:Npn \bool_set:Nn #1#2`
5701 `{ \tex_chardef:D #1 = \bool_if_p:n {#2} }`
5702 `\cs_new_protected:Npn \bool_gset:Nn #1#2`
5703 `{ \tex_global:D \tex_chardef:D #1 = \bool_if_p:n {#2} }`
5704 `\cs_generate_variant:Nn \bool_set:Nn { c }`
5705 `\cs_generate_variant:Nn \bool_gset:Nn { c }`

(End definition for `\bool_set:Nn` and `\bool_gset:Nn`. These functions are documented on page 92.)

Booleans are not based on token lists but do need checking: this code complements similar material in `l3tl`.

```

5706  (*package)
5707  \if_bool:N \l@expl@check@declarations@bool
5708    \cs_set_protected:Npn \bool_set_true:N #1
5709    {
5710      \__chk_if_exist_var:N #1
5711      \cs_set_eq:NN #1 \c_true_bool
5712    }
5713  \cs_set_protected:Npn \bool_set_false:N #1
5714  {
5715    \__chk_if_exist_var:N #1
5716    \cs_set_eq:NN #1 \c_false_bool
5717  }
5718  \cs_set_protected:Npn \bool_gset_true:N #1
5719  {
5720    \__chk_if_exist_var:N #1
5721    \cs_gset_eq:NN #1 \c_true_bool
5722  }
5723  \cs_set_protected:Npn \bool_gset_false:N #1
5724  {
5725    \__chk_if_exist_var:N #1
5726    \cs_gset_eq:NN #1 \c_false_bool
5727  }
5728  \cs_set_protected:Npn \bool_set_eq:NN #1
5729  {
5730    \__chk_if_exist_var:N #1
5731    \cs_set_eq:NN #1
5732  }
5733  \cs_set_protected:Npn \bool_gset_eq:NN #1
5734  {
5735    \__chk_if_exist_var:N #1
5736    \cs_gset_eq:NN #1
5737  }
5738  \cs_set_protected:Npn \bool_set:Nn #1#2
5739  {
5740    \__chk_if_exist_var:N #1
5741    \tex_chardef:D #1 = \bool_if_p:n {#2}
5742  }
5743  \cs_set_protected:Npn \bool_gset:Nn #1#2
5744  {
5745    \__chk_if_exist_var:N #1
5746    \tex_global:D \tex_chardef:D #1 = \bool_if_p:n {#2}
5747  }
5748 \fi:
5749 
```

\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:NTF `\prg_new_conditional:Npnn \bool_if:N #1 { p , T , F , TF }`
\bool_if:cTF `5750 \prg_new_conditional:Npnn \bool_if:N #1 { p , T , F , TF }`
`5751 {`
`5752 \if_meaning:w \c_true_bool #1`

```

5753     \prg_return_true:
5754     \else:
5755     \prg_return_false:
5756     \fi:
5757 }
5758 \cs_generate_variant:Nn \bool_if_p:N { c }
5759 \cs_generate_variant:Nn \bool_if:NT { c }
5760 \cs_generate_variant:Nn \bool_if:NF { c }
5761 \cs_generate_variant:Nn \bool_if:NTF { c }

```

(End definition for `\bool_if:NTF`. This function is documented on page 92.)

\bool_show:N Show the truth value of the boolean, as true or false.

```

5762 \cs_new_protected:Npn \bool_show:N #1
5763 {
5764     \__msg_show_variable>NNNnn #1 \bool_if_exist:NTF ? { }
5765     { > ~ \token_to_str:N #1 = \__bool_to_str:n {#1} }
5766 }
5767 \cs_new_protected:Npn \bool_show:n
5768 { \__msg_show_wrap:Nn \__bool_to_str:n }
5769 \cs_new:Npn \__bool_to_str:n #1
5770 { \bool_if:nTF {#1} { true } { false } }
5771 \cs_generate_variant:Nn \bool_show:N { c }

```

(End definition for `\bool_show:N`, `\bool_show:n`, and `__bool_to_str:n`. These functions are documented on page 92.)

\bool_log:N Redirect output of `\bool_show:N` to the log.

```

5772 \cs_new_protected:Npn \bool_log:N
5773 { \__msg_log_next: \bool_show:N }
5774 \cs_new_protected:Npn \bool_log:n
5775 { \__msg_log_next: \bool_show:n }
5776 \cs_generate_variant:Nn \bool_log:N { c }

```

(End definition for `\bool_log:N` and `\bool_log:n`. These functions are documented on page 92.)

\l_tmpa_bool A few booleans just if you need them.

```

5777 \bool_new:N \l_tmpa_bool
5778 \bool_new:N \l_tmpb_bool
5779 \bool_new:N \g_tmpa_bool
5780 \bool_new:N \g_tmpb_bool

```

(End definition for `\l_tmpa_bool` and others. These variables are documented on page 92.)

\bool_if_exist_p:N Copies of the `cs` functions defined in `l3basics`.

```

5781 \prg_new_eq_conditional:NNn \bool_if_exist:N \cs_if_exist:N
5782 { TF , T , F , p }
5783 \prg_new_eq_conditional:NNn \bool_if_exist:c \cs_if_exist:c
5784 { TF , T , F , p }

```

(End definition for `\bool_if_exist:NTF`. This function is documented on page 92.)

12.4 Boolean expressions

\bool_if_p:n

\bool_if:nTF

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”. 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 GetNotNext function, which eventually reverses the logic compared to GetNext.
- 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 *<true>* or *<false>*.

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 evaluating the predicates within this sub-expression and break to the nearest Close. Then return *<false>*.

<true>Or Current truth value is true, logical Or seen, stop evaluating the predicates within this sub-expression and break to 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>*.

We introduce an additional Stop operation with the same semantics as the Close operation.

<true>Stop Current truth value is true, return *<true>*.

<false>Stop Current truth value is false, return *<false>*.

The reasons for this follow below.

```
5785 \prg_new_conditional:Npn \bool_if:n #1 { T , F , TF }
5786 {
5787     \if_predicate:w \bool_if_p:n {#1}
5788     \prg_return_true:
5789     \else:
5790     \prg_return_false:
5791     \fi:
5792 }
```

(End definition for `\bool_if:nTF`. This function is documented on page 94.)

```
\bool_if_p:n
\__bool_if_left_parentheses:www
\__bool_if_right_parentheses:www
\__bool_if_or:www
```

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 will be closed at the end of the expression parsing (see S below).

Minimal (“short-circuit”) evaluation of boolean expressions requires skipping to the end of the current parenthesized group when `<true>||` is seen, but to the next `||` or closing parenthesis when `<false>&&` is seen. To avoid having separate functions for the two cases, we transform the boolean expression by doubling each parenthesis and adding parenthesis around each `||`. This ensures that `&&` will bind tighter than `||`.

The replacement is done in three passes, for left and right parentheses and for `||`. At each pass, the part of the expression that has been transformed is stored before `\q_nil`, the rest lies until the first `\q_mark`, followed by an empty brace group. A trailing marker ensures that the auxiliaries’ delimited arguments will not run-away. As long as the delimiter matches inside the expression, material is moved before `\q_nil` and we continue. Afterwards, the trailing marker is taken as a delimiter, #4 is the next auxiliary, immediately followed by a new `\q_nil` delimiter, which indicates that nothing has been treated at this pass. The last step calls `__bool_if_parse:NNNww` which cleans up and triggers the evaluation of the expression itself.

```
5793 \cs_new:Npn \bool_if_p:n #1
5794 {
5795     \group_align_safe_begin:
5796     \__bool_if_left_parentheses:www \q_nil
5797         #1 \q_mark { }
5798         ( \q_mark { \__bool_if_right_parentheses:www \q_nil }
5799             ) \q_mark { \__bool_if_or:www \q_nil }
5800             || \q_mark \__bool_if_parse:NNNww
5801             \q_stop
5802     }
5803 \cs_new:Npn \__bool_if_left_parentheses:www #1 \q_nil #2 ( #3 \q_mark #4
5804     { #4 \__bool_if_left_parentheses:www #1 #2 (( \q_nil #3 \q_mark {#4} )
5805 \cs_new:Npn \__bool_if_right_parentheses:www #1 \q_nil #2 ) #3 \q_mark #4
5806     { #4 \__bool_if_right_parentheses:www #1 #2 ) ) \q_nil #3 \q_mark {#4} }
5807 \cs_new:Npn \__bool_if_or:www #1 \q_nil #2 || #3 \q_mark #4
5808     { #4 \__bool_if_or:www #1 #2 ) || ( \q_nil #3 \q_mark {#4} }
```

(End definition for `\bool_if_p:n` and others. These functions are documented on page 94.)

```
\__bool_if_parse:NNNww
```

After removing extra tokens from the transformation phase, start evaluating. At the end, we will need to finish the special `align_safe` group before finally returning a `\c_true_bool` or `\c_false_bool` as there might otherwise be something left in front in the input stream. For this we call the Stop operation, denoted simply by a S following the last Close operation.

```
5809 \cs_new:Npn \__bool_if_parse:NNNww #1#2#3#4 \q_mark #5 \q_stop
5810 {
5811     \__bool_get_next:NN \use_i:nn (( #4 )) S
5812 }
```

(End definition for `__bool_if_parse:NNNww`.)

```
\__bool_get_next:NN
```

The GetNext operation. This is a switch: if what follows is neither ! nor (, we assume it is a predicate. The first argument is `\use_i:nn` if the logic must eventually be reversed (after a !), otherwise it is `\use_i:nn`. 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.

```

5813 \cs_new:Npn \__bool_get_next:NN #1#2
5814   {
5815     \use:c
5816     {
5817       __bool_
5818       \if_meaning:w !#2 ! \else: \if_meaning:w (#2 ( \else: p \fi: \fi:
5819       :Nw
5820     }
5821   #1 #2
5822 }
```

(End definition for `__bool_get_next:NN`.)

`__bool_! :Nw` The Not operation reverses the logic: discard the `!` token and call the GetNext operation with its first argument reversed.

```

5823 \cs_new:cpn { __bool_! :Nw } #1#2
5824   { \exp_after:wN \__bool_get_next:NN #1 \use_i:nn \use_i:nn }
```

(End definition for `__bool_! :Nw`.)

`__bool_(:Nw` The Open operation starts a sub-expression after discarding the token. This is done by calling GetNext, with a post-processing step which looks for And, Or or Close afterwards.

```

5825 \cs_new:cpn { __bool_( :Nw } #1#2
5826   {
5827     \exp_after:wN \__bool_choose:NNN \exp_after:wN #1
5828     \__int_value:w \__bool_get_next:NN \use_i:nn
5829   }
```

(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.

```

5830 \cs_new:cpn { __bool_p:Nw } #1
5831   { \exp_after:wN \__bool_choose:NNN \exp_after:wN #1 \__int_value:w }
```

(End definition for `__bool_p:Nw`.)

`__bool_choose:NNN` Branching the eight-way switch. The arguments are 1: `\use_i:nn` or `\use_ii:nn`, 2: 0 or 1 encoding the current truth value, 3: the next operation, And, Or, Close or Stop. If `#1` is `\use_ii:nn`, the logic of `#2` must be reversed.

```

5832 \cs_new:Npn \__bool_choose:NNN #1#2#3
5833   {
5834     \use:c
5835     {
5836       __bool_ #3 -
5837       #1 #2 { \if_meaning:w 0 #2 1 \else: 0 \fi: }
5838       :w
5839     }
5840 }
```

(End definition for `__bool_choose:NNN`.)

__bool_)_0:w Closing a group is just about returning the result. The Stop operation is similar except it closes the special alignment group before returning the boolean.

```

\_\_bool\_)_1:w
\_\_bool\_)_0:w      5841 \cs_new:cpn { __bool\_)_0:w } { \c_false_bool }
\_\_bool\_)_1:w      5842 \cs_new:cpn { __bool\_)_1:w } { \c_true_bool }
\_\_bool\_)_0:w      5843 \cs_new:cpn { __bool\_)_0:w } { \group_align_safe_end: \c_false_bool }
\_\_bool\_)_1:w      5844 \cs_new:cpn { __bool\_)_1:w } { \group_align_safe_end: \c_true_bool }

```

(End definition for __bool_)_0:w and others.)

__bool\&_1:w Two cases where we simply continue scanning. We must remove the second & or |.

```

\_\_bool\|_0:w
\_\_bool\&_1:w      5845 \cs_new:cpn { __bool\&_1:w } & { \_\_bool_get_next:NN \use_i:nn }
\_\_bool\|_0:w      5846 \cs_new:cpn { __bool\|_0:w } | { \_\_bool_get_next:NN \use_i:nn }

```

(End definition for __bool\&_1:w and __bool\|_0:w.)

__bool\&_0:w When the truth value has already been decided, we have to throw away the remainder of the current group as we are doing minimal evaluation. This is slightly tricky as there are no braces so we have to play match the () manually.

```

\_\_bool\eval_skip_to_end_auxi:Nw
\_\_bool\eval_skip_to_end_auxii:Nw
\_\_bool\eval_skip_to_end_auxiii:Nw
\_\_bool\eval_skip_to_end_auxi:Nw      5847 \cs_new:cpn { __bool\&_0:w } &
\_\_bool\eval_skip_to_end_auxii:Nw     5848   { \_\_bool_eval_skip_to_end_auxi:Nw \c_false_bool }
\_\_bool\eval_skip_to_end_auxiii:Nw   5849 \cs_new:cpn { __bool\|_1:w } |
\_\_bool\eval_skip_to_end_auxi:Nw     5850   { \_\_bool_eval_skip_to_end_auxi:Nw \c_true_bool }

```

There is always at least one) waiting, namely the outer one. However, we are facing the problem that there may be more than one that need to be finished off and we have to detect the correct number of them. Here is a complicated example showing how this is done. After evaluating the following, we realize we must skip everything after the first And. Note the extra Close at the end.

```
\c_false_bool && ((abc) && xyz) && ((xyz) && (def)))
```

First read up to the first Close. This gives us the list we first read up until the first right parenthesis so we are looking at the token list

```
((abc
```

This contains two Open markers so we must remove two groups. Since no evaluation of the contents is to be carried out, it doesn't matter how we remove the groups as long as we wind up with the correct result. We therefore first remove a () pair and what preceded the Open – but leave the contents as it may contain Open tokens itself – leaving

```
(abc && xyz) && ((xyz) && (def)))
```

Another round of this gives us

```
(abc && xyz
```

which still contains an Open so we remove another () pair, giving us

```
abc && xyz && ((xyz) && (def)))
```

Again we read up to a Close and again find Open tokens:

```
abc && xyz && ((xyz
```

Further reduction gives us

```
(xyz && (def)))
```

and then

```
(xyz && (def
```

with reduction to

```
xyz && (def))
```

and ultimately we arrive at no Open tokens being skipped and we can finally close the group nicely.

```
5851 %% (
5852 \cs_new:Npn \__bool_eval_skip_to_end_auxi:Nw #1#2 )
5853 {
5854     \__bool_eval_skip_to_end_auxii:Nw #1#2 ( % )
5855     \q_no_value \q_stop
5856     {#2}
5857 }
```

If no right parenthesis, then #3 is no_value and we are done, return the boolean #1. If there is, we need to grab a () pair and then recurse

```
5858 \cs_new:Npn \__bool_eval_skip_to_end_auxii:Nw #1#2 ( #3#4 \q_stop #5 % )
5859 {
5860     \quark_if_no_value:NTF #3
5861     {#1}
5862     { \__bool_eval_skip_to_end_auxiii:Nw #1 #5 }
5863 }
```

Keep the boolean, throw away anything up to the (as it is irrelevant, remove a () pair but remember to reinsert #3 as it may contain (tokens!

```
5864 \cs_new:Npn \__bool_eval_skip_to_end_auxiii:Nw #1#2 ( #3 )
5865 { % (
5866     \__bool_eval_skip_to_end_auxi:Nw #1#3 )
5867 }
```

(End definition for __bool_&_0:w 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:n

```
5868 \prg_new_conditional:Npnn \bool_lazy_all:n #1 { p , T , F , TF }
5869 { \__bool_lazy_all:n #1 \q_recursion_tail \q_recursion_stop }
5870 \cs_new:Npn \__bool_lazy_all:n #1
5871 {
5872     \quark_if_recursion_tail_stop_do:nn {#1} { \prg_return_true: }
5873     \bool_if:nF {#1}
5874     { \use_i_delimit_by_q_recursion_stop:nw { \prg_return_false: } }
5875     \__bool_lazy_all:n
5876 }
```

(End definition for \bool_lazy_all:nTF and __bool_lazy_all:n. These functions are documented on page 94.)

`\bool_lazy_and_p:nn` Only evaluate the second expression if the first is `true`.

```

5877 \prg_new_conditional:Npnn \bool_lazy_and:nn #1#2 { p , T , F , TF }
5878 {
5879   \bool_if:nTF {#1}
5880   { \bool_if:nTF {#2} { \prg_return_true: } { \prg_return_false: } }
5881   { \prg_return_false: }
5882 }
```

(End definition for `\bool_lazy_and:nnTF`. This function is documented on page 94.)

`\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`

```

5883 \prg_new_conditional:Npnn \bool_lazy_any:n #1 { p , T , F , TF }
5884 { __bool_lazy_any:n #1 \q_recursion_tail \q_recursion_stop }
5885 \cs_new:Npn __bool_lazy_any:n #1
5886 {
5887   \quark_if_recursion_tail_stop_do:nn {#1} { \prg_return_false: }
5888   \bool_if:nT {#1}
5889   { \use_i_delimit_by_q_recursion_stop:nw { \prg_return_true: } }
5890   __bool_lazy_any:n
5891 }
```

(End definition for `\bool_lazy_any:nTF` and `__bool_lazy_any:n`. These functions are documented on page 94.)

`\bool_lazy_or_p:nn` Only evaluate the second expression if the first is `false`.

`\bool_lazy_or:nTF`

```

5892 \prg_new_conditional:Npnn \bool_lazy_or:nn #1#2 { p , T , F , TF }
5893 {
5894   \bool_if:nTF {#1}
5895   { \prg_return_true: }
5896   { \bool_if:nTF {#2} { \prg_return_true: } { \prg_return_false: } }
5897 }
```

(End definition for `\bool_lazy_or:nTF`. This function is documented on page 94.)

`\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.

```
5898 \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 94.)

`\bool_xor_p:nn` Exclusive or. If the boolean expressions have same truth value, return `false`, otherwise return `true`.

```

5899 \cs_new:Npn \bool_xor_p:nn #1#2
5900 {
5901   \int_compare:nNnTF { \bool_if_p:n {#1} } = { \bool_if_p:n {#2} }
5902   \c_false_bool
5903   \c_true_bool
5904 }
```

(End definition for `\bool_xor_p:nn`. This function is documented on page 94.)

12.5 Logical loops

\bool_while_do:Nn A `while` loop where the boolean is tested before executing the statement. The “while” version executes the code as long as the boolean is true; the “until” version executes the code as long as the boolean is false.

```

5905 \cs_new:Npn \bool_while_do:Nn #1#2
5906   { \bool_if:NT #1 { #2 \bool_while_do:Nn #1 {#2} } }
5907 \cs_new:Npn \bool_until_do:Nn #1#2
5908   { \bool_if:NF #1 { #2 \bool_until_do:Nn #1 {#2} } }
5909 \cs_generate_variant:Nn \bool_while_do:Nn { c }
5910 \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 95.)

\bool_do_while:Nn A `do-while` loop where the body is performed at least once and the boolean is tested after executing the body. Otherwise identical to the above functions.

```

5911 \cs_new:Npn \bool_do_while:Nn #1#2
5912   { #2 \bool_if:NT #1 { \bool_do_while:Nn #1 {#2} } }
5913 \cs_new:Npn \bool_do_until:Nn #1#2
5914   { #2 \bool_if:NF #1 { \bool_do_until:Nn #1 {#2} } }
5915 \cs_generate_variant:Nn \bool_do_while:Nn { c }
5916 \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 95.)

\bool_while_do:nn Loop functions with the test either before or after the first body expansion.

```

5917 \cs_new:Npn \bool_while_do:nn #1#2
5918   {
5919     \bool_if:nT {#1}
5920     {
5921       #2
5922       \bool_while_do:nn {#1} {#2}
5923     }
5924   }
5925 \cs_new:Npn \bool_do_while:nn #1#2
5926   {
5927     #2
5928     \bool_if:nT {#1} { \bool_do_while:nn {#1} {#2} }
5929   }
5930 \cs_new:Npn \bool_until_do:nn #1#2
5931   {
5932     \bool_if:nF {#1}
5933     {
5934       #2
5935       \bool_until_do:nn {#1} {#2}
5936     }
5937   }
5938 \cs_new:Npn \bool_do_until:nn #1#2
5939   {
5940     #2
5941     \bool_if:nF {#1} { \bool_do_until:nn {#1} {#2} }
5942   }
```

(End definition for `\bool_while_do:nn` and others. These functions are documented on page 95.)

12.6 Producing multiple copies

5943 `<@@=prg>`

\prg_replicate:nn This function uses a cascading csname technique by David Kastrup (who else :-)

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 will severely affect 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.

```
5944 \cs_new:Npn \prg_replicate:nn #1
5945   {
5946     \exp:w
5947       \exp_after:wN \__prg_replicate_first:N
5948         \__int_value:w \__int_eval:w #1 \__int_eval_end:
5949       \cs_end:
5950   }
5951 \cs_new:Npn \__prg_replicate:N #1
5952   { \cs:w __prg_replicate_#1 :n \__prg_replicate:N }
5953 \cs_new:Npn \__prg_replicate_first:N #1
5954   { \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.

```
5955 \cs_new:Npn \__prg_replicate_ :n #1 { \cs_end: }
5956 \cs_new:cpn { __prg_replicate_0:n } #1
5957   { \cs_end: {#1#1#1#1#1#1#1#1#1} }
5958 \cs_new:cpn { __prg_replicate_1:n } #1
5959   { \cs_end: {#1#1#1#1#1#1#1#1#1} #1 }
5960 \cs_new:cpn { __prg_replicate_2:n } #1
5961   { \cs_end: {#1#1#1#1#1#1#1#1#1} #1#1 }
5962 \cs_new:cpn { __prg_replicate_3:n } #1
5963   { \cs_end: {#1#1#1#1#1#1#1#1#1} #1#1#1 }
5964 \cs_new:cpn { __prg_replicate_4:n } #1
5965   { \cs_end: {#1#1#1#1#1#1#1#1#1} #1#1#1#1 }
5966 \cs_new:cpn { __prg_replicate_5:n } #1
5967   { \cs_end: {#1#1#1#1#1#1#1#1#1} #1#1#1#1#1 }
5968 \cs_new:cpn { __prg_replicate_6:n } #1
5969   { \cs_end: {#1#1#1#1#1#1#1#1#1} #1#1#1#1#1#1 }
```

```

5970 \cs_new:cpn { __prg_replicate_7:n } #1
5971   { \cs_end: {#1#1#1#1#1#1#1#1} #1#1#1#1#1#1 }
5972 \cs_new:cpn { __prg_replicate_8:n } #1
5973   { \cs_end: {#1#1#1#1#1#1#1#1} #1#1#1#1#1#1 }
5974 \cs_new:cpn { __prg_replicate_9:n } #1
5975   { \cs_end: {#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...

```

5976 \cs_new:cpn { __prg_replicate_first_:-n } #1
5977   {
5978     \exp_end:
5979     \__msg_kernel_expandable_error:nn { kernel } { negative-replication }
5980   }
5981 \cs_new:cpn { __prg_replicate_first_0:n } #1 { \exp_end: }
5982 \cs_new:cpn { __prg_replicate_first_1:n } #1 { \exp_end: #1 }
5983 \cs_new:cpn { __prg_replicate_first_2:n } #1 { \exp_end: #1#1 }
5984 \cs_new:cpn { __prg_replicate_first_3:n } #1 { \exp_end: #1#1#1 }
5985 \cs_new:cpn { __prg_replicate_first_4:n } #1 { \exp_end: #1#1#1#1 }
5986 \cs_new:cpn { __prg_replicate_first_5:n } #1 { \exp_end: #1#1#1#1#1 }
5987 \cs_new:cpn { __prg_replicate_first_6:n } #1 { \exp_end: #1#1#1#1#1#1 }
5988 \cs_new:cpn { __prg_replicate_first_7:n } #1 { \exp_end: #1#1#1#1#1#1#1 }
5989 \cs_new:cpn { __prg_replicate_first_8:n } #1 { \exp_end: #1#1#1#1#1#1#1#1 }
5990 \cs_new:cpn { __prg_replicate_first_9:n } #1 { \exp_end: #1#1#1#1#1#1#1#1#1 }

```

(End definition for `\prg_replicate:nn` and others. These functions are documented on page 96.)

12.7 Detecting TeX's mode

`\mode_if_vertical_p:` For testing vertical mode. Strikes me here on the bus with David, that as long as we `\mode_if_vertical:TF` 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.

```

5991 \prg_new_conditional:Npnn \mode_if_vertical: { p , T , F , TF }
5992   { \if_mode_vertical: \prg_return_true: \else: \prg_return_false: \fi: }

```

(End definition for `\mode_if_vertical:TF`. This function is documented on page 96.)

`\mode_if_horizontal_p:` For testing horizontal mode.

```

5993 \prg_new_conditional:Npnn \mode_if_horizontal: { p , T , F , TF }
5994   { \if_mode_horizontal: \prg_return_true: \else: \prg_return_false: \fi: }

```

(End definition for `\mode_if_horizontal:TF`. This function is documented on page 96.)

`\mode_if_inner_p:` For testing inner mode.

```

5995 \prg_new_conditional:Npnn \mode_if_inner: { p , T , F , TF }
5996   { \if_mode_inner: \prg_return_true: \else: \prg_return_false: \fi: }

```

(End definition for `\mode_if_inner:TF`. This function is documented on page 96.)

`\mode_if_math_p:` For testing math mode. At the beginning of an alignment cell, this should be used only `\mode_if_math:TF` inside a non-expandable function.

```

5997 \prg_new_conditional:Npnn \mode_if_math: { p , T , F , TF }
5998   { \if_mode_math: \prg_return_true: \else: \prg_return_false: \fi: }

```

(End definition for `\mode_if_math:TF`. This function is documented on page 96.)

12.8 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 issues 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 will get some sort of weird error message because the underlying `\futurelet` will store 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 code documented only in Appendix D of *The T_EXbook*... 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.

```
5999 \cs_new:Npn \group_align_safe_begin:
6000   { \if_int_compare:w \if_false: { \fi: ' } = \c_zero \fi: }
6001 \cs_new:Npn \group_align_safe_end:
6002   { \if_int_compare:w '{ = \c_zero } \fi: }
```

(End definition for `\group_align_safe_begin:` and `\group_align_safe_end:..`)
6003 ⟨@=prg⟩

`\g_prg_map_int` A nesting counter for mapping.

```
6004 \int_new:N \g_prg_map_int
6005 (End definition for \g_prg_map_int.)
```

`__prg_break_point:Nn` These are defined in `\3basics`, as they are needed "early". This is just a reminder that is the case!

(End definition for `__prg_break_point:Nn` and `__prg_map_break:Nn.`)

`__prg_break_point:` Also done in `\3basics` as in format mode these are needed within `\3alloc`.

```
\__prg_break:
\__prg_break:n
6005 ⟨/initex | package⟩
```

13 I3clist implementation

The following test files are used for this code: `m3clist002`.

```
6006 ⟨*initex | package⟩
6007 ⟨@=clist⟩
```

`\c_empty_clist` An empty comma list is simply an empty token list.

```
6008 \cs_new_eq:NN \c_empty_clist \c_empty_tl
```

(End definition for `\c_empty_clist`. This variable is documented on page 106.)

`\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`

```
6009 \tl_new:N \l__clist_internal_clist
```

(End definition for `\l__clist_internal_clist.`)

`__clist_tmp:w` A temporary function for various purposes.
6010 `\cs_new_protected:Npn __clist_tmp:w { }`
(End definition for `__clist_tmp:w.`)

13.1 Allocation and initialisation

`\clist_new:N` Internally, comma lists are just token lists.
`\clist_new:c` 6011 `\cs_new_eq:NN \clist_new:N \tl_new:N`
6012 `\cs_new_eq:NN \clist_new:c \tl_new:c`
(End definition for `\clist_new:N`. This function is documented on page 98.)

`\clist_const:Nn` Creating and initializing a constant comma list is done in a way similar to `\clist_set:Nn` and `\clist_gset:Nn`, being careful to strip spaces.
`\clist_const:cn` 6013 `\cs_new_protected:Npn \clist_const:Nn #1#2`
6014 `{ \tl_const:Nx #1 { __clist_trim_spaces:n {#2} } }`
6015 `\cs_generate_variant:Nn \clist_const:Nn { c , Nx , cx }`
(End definition for `\clist_const:Nn`. This function is documented on page 98.)

`\clist_clear:N` Clearing comma lists is just the same as clearing token lists.
`\clist_clear:c` 6016 `\cs_new_eq:NN \clist_clear:N \tl_clear:N`
`\clist_gclear:N` 6017 `\cs_new_eq:NN \clist_clear:c \tl_clear:c`
`\clist_gclear:c` 6018 `\cs_new_eq:NN \clist_gclear:N \tl_gclear:N`
6019 `\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 98.)

`\clist_clear_new:N` Once again a copy from the token list functions.
`\clist_clear_new:c` 6020 `\cs_new_eq:NN \clist_clear_new:N \tl_clear_new:N`
`\clist_gclear_new:N` 6021 `\cs_new_eq:NN \clist_clear_new:c \tl_clear_new:c`
`\clist_gclear_new:c` 6022 `\cs_new_eq:NN \clist_gclear_new:N \tl_gclear_new:N`
6023 `\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 98.)

`\clist_set_eq:NN` Once again, these are simple copies from the token list functions.
`\clist_set_eq:cN` 6024 `\cs_new_eq:NN \clist_set_eq:NN \tl_set_eq:NN`
`\clist_set_eq:Nc` 6025 `\cs_new_eq:NN \clist_set_eq:Nc \tl_set_eq:Nc`
`\clist_set_eq:cc` 6026 `\cs_new_eq:NN \clist_set_eq:cc \tl_set_eq:cc`
`\clist_gset_eq>NN` 6027 `\cs_new_eq:NN \clist_set_eq:cc \tl_set_eq:cc`
`\clist_gset_eq:cN` 6028 `\cs_new_eq:NN \clist_gset_eq:NN \tl_gset_eq:NN`
`\clist_gset_eq:Nc` 6029 `\cs_new_eq:NN \clist_gset_eq:Nc \tl_gset_eq:Nc`
`\clist_gset_eq:cc` 6030 `\cs_new_eq:NN \clist_gset_eq:cN \tl_gset_eq:cN`
6031 `\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 99.)

\clist_set_from_seq:NN Setting a comma list from a comma-separated list is done using a simple mapping. We wrap most items with \exp_not:n, and a comma. Items which contain a comma or a space are surrounded by an extra set of braces. The first comma must be removed, except in the case of an empty comma-list.

```

\clist_set_from_seq:cN
\clist_set_from_seq:Nc
\clist_set_from_seq:cc
\clist_gset_from_seq:NN
\clist_gset_from_seq:cN
\clist_gset_from_seq:Nc
\clist_gset_from_seq:cc
\__clist_set_from_seq:NNNN
  \__clist_wrap_item:n
\__clist_set_from_seq:w

6032 \cs_new_protected:Npn \clist_set_from_seq:NN
6033   { \__clist_set_from_seq:NNNN \clist_clear:N \tl_set:Nx }
6034 \cs_new_protected:Npn \clist_gset_from_seq:NN
6035   { \__clist_set_from_seq:NNNN \clist_gclear:N \tl_gset:Nx }
6036 \cs_new_protected:Npn \__clist_set_from_seq:NNNN #1#2#3#4
6037   {
6038     \seq_if_empty:NTF #4
6039       { #1 #3 }
6040       {
6041         #2 #3
6042         {
6043           \exp_last_unbraced:Nf \use_none:n
6044             { \seq_map_function:NN #4 \__clist_wrap_item:n }
6045         }
6046       }
6047     }
6048 \cs_new:Npn \__clist_wrap_item:n #1
6049   {
6050     ,
6051     \tl_if_empty:oTF { \__clist_set_from_seq:w #1 ~ , #1 ~ }
6052       { \exp_not:n {#1} }
6053       { \exp_not:n { {#1} } }
6054     }
6055 \cs_new:Npn \__clist_set_from_seq:w #1 , #2 ~ { }
6056 \cs_generate_variant:Nn \clist_set_from_seq:NN { Nc }
6057 \cs_generate_variant:Nn \clist_set_from_seq:NN { c , cc }
6058 \cs_generate_variant:Nn \clist_gset_from_seq:NN { Nc }
6059 \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 99.)

Concatenating comma lists is not quite as easy as it seems, as there needs to be the correct addition of a comma to the output. So a little work to do.

```

\clist_concat:NNN
\clist_concat:ccc
\clist_gconcat:NNN
\clist_gconcat:ccc
\__clist_concat:NNNN

6060 \cs_new_protected:Npn \clist_concat:NNN
6061   { \__clist_concat:NNNN \tl_set:Nx }
6062 \cs_new_protected:Npn \clist_gconcat:NNN
6063   { \__clist_concat:NNNN \tl_gset:Nx }
6064 \cs_new_protected:Npn \__clist_concat:NNNN #1#2#3#4
6065   {
6066     #1 #2
6067     {
6068       \exp_not:o #3
6069       \clist_if_empty:NF #3 { \clist_if_empty:NF #4 { , } }
6070       \exp_not:o #4
6071     }
6072   }
6073 \cs_generate_variant:Nn \clist_concat:NNN { ccc }
6074 \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 99.)

```

\clist_if_exist_p:N Copies of the cs functions defined in l3basics.
\clist_if_exist_p:c 6075 \prg_new_eq_conditional:NNN \clist_if_exist:N \cs_if_exist:N
\clist_if_exist:NTF 6076 {TF , T , F , p }
\clist_if_exist:cTF 6077 \prg_new_eq_conditional:NNN \clist_if_exist:c \cs_if_exist:c
6078 {TF , T , F , p }

```

(End definition for `\clist_if_exist:NTF`. This function is documented on page 99.)

13.2 Removing spaces around items

`_clist_trim_spaces_generic:nw` This expands to the `<code>`, followed by a brace group containing the `<item>`, with leading and trailing spaces removed. The calling function is responsible for inserting `\q_mark` in front of the `<item>`, as well as testing for the end of the list. We reuse a `_tl` internal function, whose first argument must start with `\q_mark`. That trims the item #2, then feeds the result (after having to do an o-type expansion) to `_clist_trim_spaces_generic:nn` which places the `<code>` in front of the `<trimmed item>`.

```

6079 \cs_new:Npn \_clist_trim_spaces_generic:nw #1#2 ,
6080 {
6081     \_tl_trim_spaces:nn {#2}
6082     { \exp_args:No \_clist_trim_spaces_generic:nn } {#1}
6083 }
6084 \cs_new:Npn \_clist_trim_spaces_generic:nn #1#2 { #2 {#1} }

```

(End definition for `_clist_trim_spaces_generic:nw` and `_clist_trim_spaces_generic:nn`.)

`_clist_trim_spaces:n` The first argument of `_clist_trim_spaces:nn` is initially empty, and later a comma, namely, as soon as we have added an item to the resulting list. The auxiliary tests for the end of the list, and also prevents empty arguments from finding their way into the output.

```

6085 \cs_new:Npn \_clist_trim_spaces:n #1
6086 {
6087     \_clist_trim_spaces_generic:nw
6088     { \_clist_trim_spaces:nn { } }
6089     \q_mark #1 ,
6090     \q_recursion_tail, \q_recursion_stop
6091 }
6092 \cs_new:Npn \_clist_trim_spaces:nn #1 #2
6093 {
6094     \quark_if_recursion_tail_stop:n {#2}
6095     \tl_if_empty:nTF {#2}
6096     {
6097         \_clist_trim_spaces_generic:nw
6098         { \_clist_trim_spaces:nn {#1} } \q_mark
6099     }
6100     {
6101         #1 \exp_not:n {#2}
6102         \_clist_trim_spaces_generic:nw
6103         { \_clist_trim_spaces:nn { , } } \q_mark
6104     }
6105 }

```

(End definition for `_clist_trim_spaces:n` and `_clist_trim_spaces:nn`.)

13.3 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_put_left:Nn
\clist_lput_left:Nn
\clist_lput_gset:Nn
\clist_lput_gset:Nx
\clist_lput_gset:cn
\clist_lput_gset:cV
\clist_lput_gset:co
\clist_put_left: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_put_left:NNNn
\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

```

6106 \cs_new_protected:Npn \clist_set:Nn #1#2
6107 { \tl_set:Nx #1 { __clist_trim_spaces:n {#2} } }
6108 \cs_new_protected:Npn \clist_gset:Nn #1#2
6109 { \tl_gset:Nx #1 { __clist_trim_spaces:n {#2} } }
6110 \cs_generate_variant:Nn \clist_set:Nn { NV , No , Nx , c , cV , co , cx }
6111 \cs_generate_variant:Nn \clist_gset:Nn { NV , No , Nx , c , cV , co , cx }
(End definition for \clist_set:Nn and \clist_gset:Nn. These functions are documented on page 99.)

Comma lists cannot hold empty values: there are therefore a couple of sanity checks to avoid accumulating commas.

6112 \cs_new_protected:Npn \clist_put_left:Nn
6113 { __clist_put_left:NNNn \clist_concat:NNN \clist_set:Nn }
6114 \cs_new_protected:Npn \clist_gput_left:Nn
6115 { __clist_put_left:NNNn \clist_gconcat:NNN \clist_set:Nn }
6116 \cs_new_protected:Npn __clist_put_left:NNNn #1#2#3#4
6117 {
6118 #2 \l__clist_internal_clist {#4}
6119 #1 #3 \l__clist_internal_clist #3
6120 }
6121 \cs_generate_variant:Nn \clist_put_left:Nn { NV , No , Nx }
6122 \cs_generate_variant:Nn \clist_put_left:Nn { c , cV , co , cx }
6123 \cs_generate_variant:Nn \clist_gput_left:Nn { NV , No , Nx }
6124 \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 99.)

6125 \cs_new_protected:Npn \clist_put_right:Nn
6126 { __clist_put_right:NNNn \clist_concat:NNN \clist_set:Nn }
6127 \cs_new_protected:Npn \clist_gput_right:Nn
6128 { __clist_put_right:NNNn \clist_gconcat:NNN \clist_set:Nn }
6129 \cs_new_protected:Npn __clist_put_right:NNNn #1#2#3#4
6130 {
6131 #2 \l__clist_internal_clist {#4}
6132 #1 #3 #3 \l__clist_internal_clist
6133 }
6134 \cs_generate_variant:Nn \clist_put_right:Nn { NV , No , Nx }
6135 \cs_generate_variant:Nn \clist_put_right:Nn { c , cV , co , cx }
6136 \cs_generate_variant:Nn \clist_gput_right:Nn { NV , No , Nx }
6137 \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 100.)

13.4 Comma lists as stacks

Getting an item from the left of a comma list is pretty easy: just trim off the first item using the comma.

6138 \cs_new_protected:Npn \clist_get:NN #1#2
6139 {

```

6140     \if_meaning:w #1 \c_empty_clist
6141         \tl_set:Nn #2 { \q_no_value }
6142     \else:
6143         \exp_after:wN \__clist_get:wN #1 , \q_stop #2
6144     \fi:
6145 }
6146 \cs_new_protected:Npn \__clist_get:wN #1 , #2 \q_stop #3
6147 { \tl_set:Nn #3 {#1} }
6148 \cs_generate_variant:Nn \clist_get:NN { c }

```

(End definition for `\clist_get:NN` and `__clist_get:wN`. These functions are documented on page 104.)

```

\clist_pop:NN
\clist_pop:cN
\clist_gpop:NN
\clist_gpop:cN
\__clist_pop:NNN
\__clist_pop:wwNNN
\__clist_pop:wN
6149 \cs_new_protected:Npn \clist_pop:NN
6150 { \__clist_pop:NNN \tl_set:Nx }
6151 \cs_new_protected:Npn \clist_gpop:NN
6152 { \__clist_pop:NNN \tl_gset:Nx }
6153 \cs_new_protected:Npn \__clist_pop:NNN #1#2#3
6154 {
6155     \if_meaning:w #2 \c_empty_clist
6156         \tl_set:Nn #3 { \q_no_value }
6157     \else:
6158         \exp_after:wN \__clist_pop:wwNNN #2 , \q_mark \q_stop #1#2#3
6159     \fi:
6160 }
6161 \cs_new_protected:Npn \__clist_pop:wwNNN #1 , #2 \q_stop #3#4#5
6162 {
6163     \tl_set:Nn #5 {#1}
6164     #3 #4
6165     {
6166         \__clist_pop:wN \prg_do_nothing:
6167             #2 \exp_not:o
6168             , \q_mark \use_none:n
6169             \q_stop
6170     }
6171 }
6172 \cs_new:Npn \__clist_pop:wN #1 , \q_mark #2 #3 \q_stop { #2 {#1} }
6173 \cs_generate_variant:Nn \clist_pop:NN { c }
6174 \cs_generate_variant:Nn \clist_gpop:NN { c }

```

(End definition for `\clist_pop:NN` and others. These functions are documented on page 104.)

`\clist_get:NNTF`

`\clist_get:cNTF`

`\clist_pop:NNTF`

`\clist_gpop:NNTF`

`\clist_gpop:cNTF`

`__clist_pop_TF:NNN`

The same, as branching code: very similar to the above.

```

6175 \prg_new_protected_conditional:Npnn \clist_get:NN #1#2 { T , F , TF }
6176 {
6177     \if_meaning:w #1 \c_empty_clist
6178         \prg_return_false:
6179     \else:
6180         \exp_after:wN \__clist_get:wN #1 , \q_stop #2
6181         \prg_return_true:

```

```

6182     \fi:
6183   }
6184 \cs_generate_variant:Nn \clist_get:NNT { c }
6185 \cs_generate_variant:Nn \clist_get:NNF { c }
6186 \cs_generate_variant:Nn \clist_get:NNTF { c }
6187 \prg_new_protected_conditional:Npnn \clist_pop:NN #1#2 { T , F , TF }
6188   { \__clist_pop_TF:NNN \tl_set:Nx #1 #2 }
6189 \prg_new_protected_conditional:Npnn \clist_gpop:NN #1#2 { T , F , TF }
6190   { \__clist_pop_TF:NNN \tl_gset:Nx #1 #2 }
6191 \cs_new_protected:Npn \__clist_pop_TF:NNN #1#2#3
6192   {
6193     \if_meaning:w #2 \c_empty_clist
6194       \prg_return_false:
6195     \else:
6196       \exp_after:wN \__clist_pop:wwNNN #2 , \q_mark \q_stop #1#2#3
6197       \prg_return_true:
6198     \fi:
6199   }
6200 \cs_generate_variant:Nn \clist_pop:NNT { c }
6201 \cs_generate_variant:Nn \clist_pop:NNF { c }
6202 \cs_generate_variant:Nn \clist_pop:NNTF { c }
6203 \cs_generate_variant:Nn \clist_gpop:NNT { c }
6204 \cs_generate_variant:Nn \clist_gpop:NNF { c }
6205 \cs_generate_variant:Nn \clist_gpop:NNTF { c }

```

(End definition for `\clist_get:NNTF` and others. These functions are documented on page [104](#).)

\clist_push:Nn

Pushing to a comma list is the same as adding on the left.

```

6206 \cs_new_eq:NN \clist_push:Nn \clist_put_left:Nn
6207 \cs_new_eq:NN \clist_push:NV \clist_put_left:NV
6208 \cs_new_eq:NN \clist_push:No \clist_put_left:No
6209 \cs_new_eq:NN \clist_push:Nx \clist_put_left:Nx
6210 \cs_new_eq:NN \clist_push:cn \clist_put_left:cn
6211 \cs_new_eq:NN \clist_push:cV \clist_put_left:cV
6212 \cs_new_eq:NN \clist_push:co \clist_put_left:co
6213 \cs_new_eq:NN \clist_push:cX \clist_put_left:cX
6214 \cs_new_eq:NN \clist_gpush:Nn \clist_gput_left:Nn
6215 \cs_new_eq:NN \clist_gpush:NV \clist_gput_left:NV
6216 \cs_new_eq:NN \clist_gpush:No \clist_gput_left:No
6217 \cs_new_eq:NN \clist_gpush:Nx \clist_gput_left:Nx
6218 \cs_new_eq:NN \clist_gpush:cn \clist_gput_left:cn
6219 \cs_new_eq:NN \clist_gpush:cV \clist_gput_left:cV
6220 \cs_new_eq:NN \clist_gpush:co \clist_gput_left:co
6221 \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 [105](#).)

13.5 Modifying comma lists

`\l__clist_internal_remove_clist` An internal comma list for the removal routines.

```

6222 \clist_new:N \l__clist_internal_remove_clist

```

(End definition for `\l__clist_internal_remove_clist`.)

```

\clist_remove_duplicates:N Removing duplicates means making a new list then copying it.
\clist_remove_duplicates:c
\clist_gremove_duplicates:N
\clist_gremove_duplicates:c
    \_clist_remove_duplicates:NN
6223 \cs_new_protected:Npn \clist_remove_duplicates:N
6224     { \_clist_remove_duplicates:NN \clist_set_eq:NN }
6225 \cs_new_protected:Npn \clist_gremove_duplicates:N
6226     { \_clist_remove_duplicates:NN \clist_gset_eq:NN }
6227 \cs_new_protected:Npn \_clist_remove_duplicates:NN #1#2
6228     {
6229         \clist_clear:N \l__clist_internal_remove_clist
6230         \clist_map_inline:Nn #2
6231             {
6232                 \clist_if_in:NnF \l__clist_internal_remove_clist {##1}
6233                     { \clist_put_right:Nn \l__clist_internal_remove_clist {##1} }
6234             }
6235         #1 #2 \l__clist_internal_remove_clist
6236     }
6237 \cs_generate_variant:Nn \clist_remove_duplicates:N { c }
6238 \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 100.)

```

`\clist_remove_all:Nn` The method used here is very similar to `\tl_replace_all:Nnn`. 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 `\q_mark` delimiter used is the one inserted by `_clist_tmp:w`, and `\use_none_delimit_by_q_stop:w` is deleted. At the end, the final `\langle item\rangle` is grabbed, and the argument of `_clist_tmp:w` contains `\q_mark:` in that case, `_clist_remove_all:w` removes the second `\q_mark` (inserted by `_clist_tmp:w`), and lets `\use_none_delimit_by_q_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.

```

6239 \cs_new_protected:Npn \clist_remove_all:Nn
6240     { \_clist_remove_all:NNn \tl_set:Nx }
6241 \cs_new_protected:Npn \clist_gremove_all:Nn
6242     { \_clist_remove_all:NNn \tl_gset:Nx }
6243 \cs_new_protected:Npn \_clist_remove_all:NNn #1#2#3
6244     {
6245         \cs_set:Npn \_clist_tmp:w ##1 , #3 ,
6246             {
6247                 ##1
6248                 , \q_mark , \use_none_delimit_by_q_stop:w ,
6249                 \_clist_remove_all:
6250             }
6251         #1 #2
6252             {
6253                 \exp_after:wN \_clist_remove_all:
6254                 #2 , \q_mark , #3 , \q_stop
6255             }
6256 \clist_if_empty:NF #2

```

```

6257   {
6258     #1 #2
6259     {
6260       \exp_args:No \exp_not:o
6261       { \exp_after:wN \use_none:n #2 }
6262     }
6263   }
6264 }
```

`\cs_new:Npn __clist_remove_all:`
`{ \exp_after:wN __clist_remove_all:w __clist_tmp:w , }`
`\cs_new:Npn __clist_remove_all:w #1 , \q_mark , #2 , { \exp_not:n {#1} }`
`\cs_generate_variant:Nn \clist_remove_all:Nn { c }`
`\cs_generate_variant:Nn \clist_gremove_all:Nn { c }`

(End definition for `\clist_remove_all:Nn` and others. These functions are documented on page 100.)

`\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`

```

6270 \cs_new_protected:Npn \clist_reverse:N #1
6271   { \tl_set:Nx #1 { \exp_args:No \clist_reverse:n {#1} } }
6272 \cs_new_protected:Npn \clist_greverse:N #1
6273   { \tl_gset:Nx #1 { \exp_args:No \clist_reverse:n {#1} } }
6274 \cs_generate_variant:Nn \clist_reverse:N { c }
6275 \cs_generate_variant:Nn \clist_greverse:N { c }
```

(End definition for `\clist_reverse:N` and `\clist_greverse:N`. These functions are documented on page 100.)

`\clist_reverse:n` The reversed token list is built one item at a time, and stored between `\q_stop` and `\q_mark`, in the form of ? followed by zero or more instances of “`\item`,”. We start from a comma list “`\item_1, ..., \item_n`”. During the loop, the auxiliary `__clist_reverse:wwNww` receives “`? \item_i`” as #1, “`\item_{i+1}, ..., \item_n`” as #2, `__clist_reverse:wwNww` as #3, what remains until `\q_stop` as #4, and “`\item_{i-1}, ..., \item_1`,” 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 “`\q_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 `\q_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`.

```

6276 \cs_new:Npn \clist_reverse:n #1
6277   {
6278     \__clist_reverse:wwNww ? #1 ,
6279     \q_mark \__clist_reverse:wwNww ! ,
6280     \q_mark \__clist_reverse_end:ww
6281     \q_stop ? \q_mark
6282   }
6283 \cs_new:Npn \__clist_reverse:wwNww
6284   #1 , #2 \q_mark #3 #4 \q_stop ? #5 \q_mark
6285   { #3 ? #2 \q_mark #3 #4 \q_stop #1 , #5 \q_mark }
6286 \cs_new:Npn \__clist_reverse_end:ww #1 ! #2 , \q_mark
6287   { \exp_not:o { \use_none:n #2 } }
```

(End definition for `\clist_reverse:n`, `_clist_reverse:wwNww`, and `_clist_reverse_end:ww`. These functions are documented on page 100.)

`\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 101.)
`\clist_gsort:cn`

13.6 Comma list conditionals

`\clist_if_empty_p:N` Simple copies from the token list variable material.
`\clist_if_empty_p:c`
`\clist_if_empty:NTF`
`\clist_if_empty:cTF`
`\clist_if_empty:p` 6288 `\prg_new_eq_conditional:Nnn \clist_if_empty:N \tl_if_empty:N`
`\clist_if_empty:p:c` 6289 { p , T , F , TF }
`\clist_if_empty:cTF` 6290 `\prg_new_eq_conditional:Nnn \clist_if_empty:c \tl_if_empty:c`
`\clist_if_empty:p` 6291 { p , T , F , TF }

(End definition for `\clist_if_empty:NTF`. This function is documented on page 101.)

`\clist_if_empty_p:n` As usual, we insert a token (here `?`) before grabbing any argument: this avoids losing braces. The argument of `\tl_if_empty:cTF` is empty if #1 is `?` followed by blank spaces (besides, this particular variant of the emptiness test is optimized). If the item of the comma list is blank, grab the next one. As soon as one item is non-blank, exit: the second auxiliary will grab `\prg_return_false:` as #2, unless every item in the comma list was blank and the loop actually got broken by the trailing `\q_mark \prg_return_false:` item.
`\clist_if_empty:nTF`
`_clist_if_empty_n:w`
`_clist_if_empty_n:wNw`

```
6292 \prg_new_conditional:Npnn \clist_if_empty:n #1 { p , T , F , TF }
6293 {
6294     \_clist_if_empty_n:w ? #1
6295     , \q_mark \prg_return_false:
6296     , \q_mark \prg_return_true:
6297     \q_stop
6298 }
6299 \cs_new:Npn \_clist_if_empty_n:w #1 ,
6300 {
6301     \tl_if_empty:cTF { \use_none:nn #1 ? }
6302     { \_clist_if_empty_n:w ? }
6303     { \_clist_if_empty_n:wNw }
6304 }
6305 \cs_new:Npn \_clist_if_empty_n:wNw #1 \q_mark #2#3 \q_stop {#2}
```

(End definition for `\clist_if_empty:nTF`, `_clist_if_empty_n:w`, and `_clist_if_empty_n:wNw`. These functions are documented on page 101.)

`\clist_if_in:NnTF` See description of the `\tl_if_in:Nn` function for details. We simply surround the comma list, and the item, with commas.
`\clist_if_in:NVTF`
`\clist_if_in:NoteTF`
`\clist_if_in:cNTF`
`\clist_if_in:cVTF`
`\clist_if_in:cOTF`
`\clist_if_in:nnTF`
`\clist_if_in:nVTF`
`\clist_if_in:nOTF`
`_clist_if_in_return:nn` 6306 `\prg_new_protected_conditional:Npnn \clist_if_in:Nn #1#2 { T , F , TF }`
`_clist_if_in_return:nn` 6307 {
`_clist_if_in_return:nn` 6308 {
`_clist_if_in_return:nn` 6309 }
`_clist_if_in_return:nn` 6310 `\prg_new_protected_conditional:Npnn \clist_if_in:nn #1#2 { T , F , TF }`
`_clist_if_in_return:nn` 6311 {
`_clist_if_in_return:nn` 6312 {
`_clist_if_in_return:nn` 6313 {
`_clist_if_in_return:nn` 6314 }

```

6315 \cs_new_protected:Npn \__clist_if_in_return:nn #1#2
6316 {
6317     \cs_set:Npn \__clist_tmp:w ##1 ,#2, { }
6318     \tl_if_empty:cTF
6319         { \__clist_tmp:w ,#1, {} {} ,#2, }
6320         { \prg_return_false: } { \prg_return_true: }
6321     }
6322 \cs_generate_variant:Nn \clist_if_in:NnT { NV , No }
6323 \cs_generate_variant:Nn \clist_if_in:NnT { c , cV , co }
6324 \cs_generate_variant:Nn \clist_if_in:NnF { NV , No }
6325 \cs_generate_variant:Nn \clist_if_in:NnF { c , cV , co }
6326 \cs_generate_variant:Nn \clist_if_in:NnTF { NV , No }
6327 \cs_generate_variant:Nn \clist_if_in:NnTF { c , cV , co }
6328 \cs_generate_variant:Nn \clist_if_in:nnT { nV , no }
6329 \cs_generate_variant:Nn \clist_if_in:nnF { nV , no }
6330 \cs_generate_variant:Nn \clist_if_in:nnTF { nV , no }

```

(End definition for `\clist_if_in:NnTF`, `\clist_if_in:nnTF`, and `__clist_if_in_return:nn`. These functions are documented on page 101.)

13.7 Mapping to comma lists

`\clist_map_function:NN`
`\clist_map_function:cN`
`__clist_map_function:Nw`

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 one comma-delimited item at a time. The end is marked by `\q_recursion_tail`. The auxiliary function `__clist_map_function:Nw` is used directly in `\clist_map_inline:Nn`. Change with care.

```

6331 \cs_new:Npn \clist_map_function:NN #1#2
6332 {
6333     \clist_if_empty:NF #1
6334     {
6335         \exp_last_unbraced:NNo \__clist_map_function:Nw #2 #1
6336         , \q_recursion_tail ,
6337         \__prg_break_point:Nn \clist_map_break: { }
6338     }
6339 }
6340 \cs_new:Npn \__clist_map_function:Nw #1#2 ,
6341 {
6342     \__quark_if_recursion_tail_break:nN {#2} \clist_map_break:
6343     #1 {#2}
6344     \__clist_map_function:Nw #1
6345 }
6346 \cs_generate_variant:Nn \clist_map_function:NN { c }

```

(End definition for `\clist_map_function:NN` and `__clist_map_function:Nw`. These functions are documented on page 102.)

`\clist_map_function:nN`
`__clist_map_function_n:Nn`
`__clist_map_unbrace:Nw`

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_spaces_generic:nw`. The auxiliary `__clist_map_function_n:Nn` receives as arguments the function, and the result of removing leading and trailing spaces from the item which lies until the next comma. Empty items are ignored, then one level of braces is removed by `__clist_map_unbrace:Nw`.

```
6347 \cs_new:Npn \clist_map_function:nN #1#2
```

```

6348  {
6349    \__clist_trim_spaces_generic:nw { \__clist_map_function_n:Nn #2 }
6350    \q_mark #1, \q_recursion_tail,
6351    \__prg_break_point:Nn \clist_map_break: { }
6352  }
6353 \cs_new:Npn \__clist_map_function_n:Nn #1 #
6354  {
6355    \__quark_if_recursion_tail_break:nN {#2} \clist_map_break:
6356    \tl_if_empty:nF {#2} { \__clist_map_unbrace:Nw #1 #2, }
6357    \__clist_trim_spaces_generic:nw { \__clist_map_function_n:Nn #1 }
6358    \q_mark
6359  }
6360 \cs_new:Npn \__clist_map_unbrace:Nw #1 #2, { #1 {#2} }

```

(End definition for `\clist_map_function:nN`, `__clist_map_function_n:Nn`, and `__clist_map_unbrace:Nw`. These functions are documented on page 102.)

`\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 T_EX’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.

```

6361 \cs_new_protected:Npn \clist_map_inline:Nn #1#2
6362  {
6363    \clist_if_empty:NF #1
6364    {
6365      \int_gincr:N \g__prg_map_int
6366      \cs_gset_protected:cpn
6367      { __prg_map_ \int_use:N \g__prg_map_int :w } ##1 {#2}
6368      \exp_last_unbraced:Nco \__clist_map_function:Nw
6369      { __prg_map_ \int_use:N \g__prg_map_int :w }
6370      #1 , \q_recursion_tail ,
6371      \__prg_break_point:Nn \clist_map_break:
6372      { \int_gdecr:N \g__prg_map_int }
6373    }
6374  }
6375 \cs_new_protected:Npn \clist_map_inline:nn #1
6376  {
6377    \clist_set:Nn \l__clist_internal_clist {#1}
6378    \clist_map_inline:Nn \l__clist_internal_clist
6379  }
6380 \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 102.)

`\clist_map_variable:NNn`
`\clist_map_variable:cNn`
`\clist_map_variable:nNn`
`__clist_map_variable:Nnw`

As for other comma-list mappings, filter out the case of an empty list. Same approach as `\clist_map_function:Nn`, additionally we store each item in the given variable. As for inline mappings, space trimming for the `n` variant is done by storing the comma list in a variable.

```

6381 \cs_new_protected:Npn \clist_map_variable:NNn #1#2#3
6382  {
6383    \clist_if_empty:NF #1

```

```

6384   {
6385     \exp_args:Nno \use:nn
6386     { \__clist_map_variable:Nnw #2 {#3} }
6387     #1
6388     , \q_recursion_tail , \q_recursion_stop
6389     \__prg_break_point:Nn \clist_map_break: { }
6390   }
6391 }
6392 \cs_new_protected:Npn \clist_map_variable:nNn #1
6393 {
6394   \clist_set:Nn \l__clist_internal_clist {#1}
6395   \clist_map_variable:NNn \l__clist_internal_clist
6396 }
6397 \cs_new_protected:Npn \__clist_map_variable:Nnw #1#2#3,
6398 {
6399   \tl_set:Nn #1 {#3}
6400   \quark_if_recursion_tail_stop:N #1
6401   \use:n {#2}
6402   \__clist_map_variable:Nnw #1 {#2}
6403 }
6404 \cs_generate_variant:Nn \clist_map_variable:NNn { c }

```

(End definition for `\clist_map_variable:NNn`, `\clist_map_variable:nNn`, and `__clist_map_variable:Nnw`. These functions are documented on page [102](#).)

`\clist_map_break:` The break statements use the general `__prg_map_break:Nn` mechanism.

```

\clist_map_break:n
6405 \cs_new:Npn \clist_map_break:
6406   { \__prg_map_break:Nn \clist_map_break: { } }
6407 \clist_map_break:n
6408   { \__prg_map_break:Nn \clist_map_break: }

```

(End definition for `\clist_map_break:` and `\clist_map_break:n`. These functions are documented on page [102](#).)

`\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).

```

6409 \cs_new:Npn \clist_count:N #1
6410   {
6411     \int_eval:n
6412     {
6413       0
6414       \clist_map_function:NN #1 \__clist_count:n
6415     }
6416   }
6417 \cs_generate_variant:Nn \clist_count:N { c }
6418 \cs_new:Npx \clist_count:n #1
6419   {
6420     \exp_not:N \int_eval:n
6421     {
6422       0
6423       \exp_not:N \__clist_count:w \c_space_tl

```

```

6424         #1 \exp_not:n { , \q_recursion_tail , \q_recursion_stop }
6425     }
6426   }
6427 \cs_new:Npn \__clist_count:n #1 { + 1 }
6428 \cs_new:Npx \__clist_count:w #1 ,
6429   {
6430     \exp_not:n { \exp_args:Nf \quark_if_recursion_tail_stop:n } {#1}
6431     \exp_not:N \tl_if_blank:nF {#1} { + 1 }
6432     \exp_not:N \__clist_count:w \c_space_tl
6433   }

```

(End definition for `\clist_count:N` and others. These functions are documented on page 103.)

13.8 Using comma lists

```

\clist_use:Nnnn
\clist_use:cnnn
\__clist_use:wwn
\__clist_use:nwwwnw
\__clist_use:nwwn
\clist_use:Nn
\clist_use:cn

```

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.

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 `\q_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 `\q_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, `\q_mark` is taken as a third item, and now the second `\q_mark` serves as a delimiter to `use_ii`, switching to the other *<continuation>*, `use_iii`, which uses the *<separator between final two>*.

```

6434 \cs_new:Npn \clist_use:Nnnn #1#2#3#4
6435   {
6436     \clist_if_exist:NTF #1
6437       {
6438         \int_case:nnF { \clist_count:N #1 }
6439           {
6440             { 0 } { }
6441             { 1 } { \exp_after:wN \__clist_use:wwn #1 , , { } }
6442             { 2 } { \exp_after:wN \__clist_use:wwn #1 , {#2} }
6443           }
6444           {
6445             \exp_after:wN \__clist_use:nwwwnwn
6446             \exp_after:wN { \exp_after:wN } #1 ,
6447             \q_mark , { \__clist_use:nwwwnwn {#3} }
6448             \q_mark , { \__clist_use:nwwn {#4} }
6449             \q_stop { }
6450           }
6451       }
6452       {
6453         \__msg_kernel_expandable_error:nnn
6454           { kernel } { bad-variable } {#1}
6455       }
6456   }

```

```

6457 \cs_generate_variant:Nn \clist_use:Nnnn { c }
6458 \cs_new:Npn \__clist_use:wwn #1 , #2 , #3 { \exp_not:n { #1 #3 #2 } }
6459 \cs_new:Npn \__clist_use:nwwwnwn
6460     #1#2 , #3 , #4 , #5 \q_mark , #6#7 \q_stop #8
6461     { #6 {#3} , {#4} , #5 \q_mark , {#6} #7 \q_stop { #8 #1 #2 } }
6462 \cs_new:Npn \__clist_use:nwwn #1#2 , #3 \q_stop #4
6463     { \exp_not:n { #4 #1 #2 } }
6464 \cs_new:Npn \clist_use:Nn #1#2
6465     { \clist_use:Nnnn #1 {#2} {#2} {#2} }
6466 \cs_generate_variant:Nn \clist_use:Nn { c }

```

(End definition for `\clist_use:Nnnn` and others. These functions are documented on page 103.)

13.9 Using a single item

`\clist_item:Nn`

`\clist_item:cN`

`__clist_item:nnN`

`__clist_item:ffoN`

`__clist_item:ffnN`

`__clist_item_N_loop:nw`

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.

```

6467 \cs_new:Npn \clist_item:Nn #1#2
6468     {
6469         \__clist_item:ffoN
6470         { \clist_count:N #1 }
6471         { \int_eval:n {#2} }
6472         #1
6473         \__clist_item_N_loop:nw
6474     }
6475 \cs_new:Npn \__clist_item:nnnN #1#2#3#4
6476     {
6477         \int_compare:nNnTF {#2} < 0
6478         {
6479             \int_compare:nNnTF {#2} < { - #1 }
6480             { \use_none_delimit_by_q_stop:w }
6481             { \exp_args:Nf #4 { \int_eval:n { #2 + 1 + #1 } } }
6482         }
6483         {
6484             \int_compare:nNnTF {#2} > {#1}
6485             { \use_none_delimit_by_q_stop:w }
6486             { #4 {#2} }
6487         }
6488         { } , #3 , \q_stop
6489     }
6490 \cs_generate_variant:Nn \__clist_item:nnnN { ffo, ff }
6491 \cs_new:Npn \__clist_item_N_loop:nw #1 #2,
6492     {
6493         \int_compare:nNnTF {#1} = 0
6494             { \use_i_delimit_by_q_stop:nw { \exp_not:n {#2} } }
6495             { \exp_args:Nf \__clist_item_N_loop:nw { \int_eval:n { #1 - 1 } } }
6496     }
6497 \cs_generate_variant:Nn \clist_item:Nn { c }

```

(End definition for `\clist_item:Nn`, `__clist_item:nnnN`, and `__clist_item_N_loop:nw`. These functions are documented on page 105.)

```

\clist_item:nn
  \__clist_item_n:nw
\__clist_item_n_loop:nw
  \__clist_item_n_end:n
\__clist_item_n_strip:w

6498 \cs_new:Npn \clist_item:nn #1#2
6499  {
6500    \__clist_item:ffnN
6501      { \clist_count:n {#1} }
6502      { \int_eval:n {#2} }
6503      {#1}
6504      \__clist_item_n:nw
6505    }
6506 \cs_new:Npn \__clist_item_n:nw #1
6507  { \__clist_item_n_loop:nw {#1} \prg_do_nothing: }
6508 \cs_new:Npn \__clist_item_n_loop:nw #1 #2,
6509  {
6510    \exp_args:No \tl_if_blank:nTF {#2}
6511      { \__clist_item_n_loop:nw {#1} \prg_do_nothing: }
6512      {
6513        \int_compare:nNnTF {#1} = 0
6514          { \exp_args:No \__clist_item_n_end:n {#2} }
6515          {
6516            \exp_args:Nf \__clist_item_n_loop:nw
6517              { \int_eval:n { #1 - 1 } }
6518              \prg_do_nothing:
6519            }
6520          }
6521      }
6522 \cs_new:Npn \__clist_item_n_end:n #1 #2 \q_stop
6523  {
6524    \__tl_trim_spaces:nn { \q_mark #1 }
6525    { \exp_last_unbraced:No \__clist_item_n_strip:w } ,
6526  }
6527 \cs_new:Npn \__clist_item_n_strip:w #1 , { \exp_not:n {#1} }

(End definition for \clist_item:nn and others. These functions are documented on page 105.)

```

13.10 Viewing comma lists

\clist_show:N Apply the general __msg_show_variable:NNNnn. In the case of an n-type comma-list, we must do things by hand, using the same message show-clist as for an N-type comma-list but with an empty name (first argument).

```

6528 \cs_new_protected:Npn \clist_show:N #1
6529  {
6530    \__msg_show_variable:NNNnn #1
6531    \clist_if_exist:NTF \clist_if_empty:NTF { clist }
6532    { \clist_map_function:NN #1 \__msg_show_item:n }
6533  }
6534 \cs_new_protected:Npn \clist_show:n #1
6535  {
6536    \__msg_show_pre:nnxxxx { LaTeX / kernel } { show-clist }
6537    { } { \clist_if_empty:nF {#1} { ? } } { } { }
6538    \__msg_show_wrap:n
6539    { \clist_map_function:nN {#1} \__msg_show_item:n }

```

```

6540   }
6541 \cs_generate_variant:Nn \clist_show:N { c }

```

(End definition for `\clist_show:N` and `\clist_show:n`. These functions are documented on page 105.)

```

\clist_log:N Redirect output of \clist_show:N and \clist_show:n to the log.
\clist_log:c
\clist_log:n
6542 \cs_new_protected:Npn \clist_log:N
6543   { \__msg_log_next: \clist_show:N }
6544 \cs_new_protected:Npn \clist_log:n
6545   { \__msg_log_next: \clist_show:n }
6546 \cs_generate_variant:Nn \clist_log:N { c }

```

(End definition for `\clist_log:N` and `\clist_log:n`. These functions are documented on page 106.)

13.11 Scratch comma lists

```

\l_tmpa_clist Temporary comma list variables.
\l_tmpb_clist
\g_tmpa_clist
\g_tmpb_clist
6547 \clist_new:N \l_tmpa_clist
6548 \clist_new:N \l_tmpb_clist
6549 \clist_new:N \g_tmpa_clist
6550 \clist_new:N \g_tmpb_clist

```

(End definition for `\l_tmpa_clist` and others. These variables are documented on page 106.)

```

6551 </initex | package>

```

14 I3token implementation

```

6552 <*initex | package>
6553 <@=char>

```

14.1 Manipulating and interrogating character tokens

Simple wrappers around the primitives.

```

\char_set_catcode:nn
\char_value_catcode:n
\char_show_value_catcode:n
6554 \cs_new_protected:Npn \char_set_catcode:nn #1#2
6555   {
6556     \tex_catcode:D \__int_eval:w #1 \__int_eval_end:
6557     = \__int_eval:w #2 \__int_eval_end:
6558   }
6559 \cs_new:Npn \char_value_catcode:n #1
6560   { \tex_the:D \tex_catcode:D \__int_eval:w #1\__int_eval_end: }
6561 \cs_new_protected:Npn \char_show_value_catcode:n #1
6562   { \__msg_show_wrap: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 109.)

```

\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
6563 \cs_new_protected:Npn \char_set_catcode_escape:N #1
6564   { \char_set_catcode:nn { '#1 } { 0 } }
6565 \cs_new_protected:Npn \char_set_catcode_group_begin:N #1
6566   { \char_set_catcode:nn { '#1 } { 1 } }
6567 \cs_new_protected:Npn \char_set_catcode_group_end:N #1
6568   { \char_set_catcode:nn { '#1 } { 2 } }

```

```

6569 \cs_new_protected:Npn \char_set_catcode_math_toggle:N #1
6570   { \char_set_catcode:nn { '#1 } { 3 } }
6571 \cs_new_protected:Npn \char_set_catcode_alignment:N #1
6572   { \char_set_catcode:nn { '#1 } { 4 } }
6573 \cs_new_protected:Npn \char_set_catcode_end_line:N #1
6574   { \char_set_catcode:nn { '#1 } { 5 } }
6575 \cs_new_protected:Npn \char_set_catcode_parameter:N #1
6576   { \char_set_catcode:nn { '#1 } { 6 } }
6577 \cs_new_protected:Npn \char_set_catcode_math_superscript:N #1
6578   { \char_set_catcode:nn { '#1 } { 7 } }
6579 \cs_new_protected:Npn \char_set_catcode_math_subscript:N #1
6580   { \char_set_catcode:nn { '#1 } { 8 } }
6581 \cs_new_protected:Npn \char_set_catcode_ignore:N #1
6582   { \char_set_catcode:nn { '#1 } { 9 } }
6583 \cs_new_protected:Npn \char_set_catcode_space:N #1
6584   { \char_set_catcode:nn { '#1 } { 10 } }
6585 \cs_new_protected:Npn \char_set_catcode_letter:N #1
6586   { \char_set_catcode:nn { '#1 } { 11 } }
6587 \cs_new_protected:Npn \char_set_catcode_other:N #1
6588   { \char_set_catcode:nn { '#1 } { 12 } }
6589 \cs_new_protected:Npn \char_set_catcode_active:N #1
6590   { \char_set_catcode:nn { '#1 } { 13 } }
6591 \cs_new_protected:Npn \char_set_catcode_comment:N #1
6592   { \char_set_catcode:nn { '#1 } { 14 } }
6593 \cs_new_protected:Npn \char_set_catcode_invalid:N #1
6594   { \char_set_catcode:nn { '#1 } { 15 } }

```

(End definition for `\char_set_catcode_escape:N` and others. These functions are documented on page 108.)

```

\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

```

```

6595 \cs_new_protected:Npn \char_set_catcode_escape:n #1
6596   { \char_set_catcode:nn {#1} { 0 } }
6597 \cs_new_protected:Npn \char_set_catcode_group_begin:n #1
6598   { \char_set_catcode:nn {#1} { 1 } }
6599 \cs_new_protected:Npn \char_set_catcode_group_end:n #1
6600   { \char_set_catcode:nn {#1} { 2 } }
6601 \cs_new_protected:Npn \char_set_catcode_math_toggle:n #1
6602   { \char_set_catcode:nn {#1} { 3 } }
6603 \cs_new_protected:Npn \char_set_catcode_alignment:n #1
6604   { \char_set_catcode:nn {#1} { 4 } }
6605 \cs_new_protected:Npn \char_set_catcode_end_line:n #1
6606   { \char_set_catcode:nn {#1} { 5 } }
6607 \cs_new_protected:Npn \char_set_catcode_parameter:n #1
6608   { \char_set_catcode:nn {#1} { 6 } }
6609 \cs_new_protected:Npn \char_set_catcode_math_superscript:n #1
6610   { \char_set_catcode:nn {#1} { 7 } }
6611 \cs_new_protected:Npn \char_set_catcode_math_subscript:n #1
6612   { \char_set_catcode:nn {#1} { 8 } }
6613 \cs_new_protected:Npn \char_set_catcode_ignore:n #1
6614   { \char_set_catcode:nn {#1} { 9 } }
6615 \cs_new_protected:Npn \char_set_catcode_space:n #1
6616   { \char_set_catcode:nn {#1} { 10 } }
6617 \cs_new_protected:Npn \char_set_catcode_letter:n #1

```

```

6618 { \char_set_catcode:nn {#1} { 11 } }
6619 \cs_new_protected:Npn \char_set_catcode_other:n #1
6620 { \char_set_catcode:nn {#1} { 12 } }
6621 \cs_new_protected:Npn \char_set_catcode_active:n #1
6622 { \char_set_catcode:nn {#1} { 13 } }
6623 \cs_new_protected:Npn \char_set_catcode_comment:n #1
6624 { \char_set_catcode:nn {#1} { 14 } }
6625 \cs_new_protected:Npn \char_set_catcode_invalid:n #1
6626 { \char_set_catcode:nn {#1} { 15 } }

```

(End definition for `\char_set_catcode_escape:n` and others. These functions are documented on page 109.)

`\char_set_mathcode:nn`

`\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`

Pretty repetitive, but necessary!

```

6627 \cs_new_protected:Npn \char_set_mathcode:nn #1#2
6628 {
6629     \tex_mathcode:D \__int_eval:w #1 \__int_eval_end:
6630     = \__int_eval:w #2 \__int_eval_end:
6631 }
6632 \cs_new:Npn \char_value_mathcode:n #1
6633 { \tex_the:D \tex_mathcode:D \__int_eval:w #1\__int_eval_end: }
6634 \cs_new_protected:Npn \char_show_value_mathcode:n #1
6635 { \__msg_show_wrap:n { > ~ \char_value_mathcode:n {#1} } }
6636 \cs_new_protected:Npn \char_set_lccode:nn #1#2
6637 {
6638     \tex_lccode:D \__int_eval:w #1 \__int_eval_end:
6639     = \__int_eval:w #2 \__int_eval_end:
6640 }
6641 \cs_new:Npn \char_value_lccode:n #1
6642 { \tex_the:D \tex_lccode:D \__int_eval:w #1\__int_eval_end: }
6643 \cs_new_protected:Npn \char_show_value_lccode:n #1
6644 { \__msg_show_wrap:n { > ~ \char_value_lccode:n {#1} } }
6645 \cs_new_protected:Npn \char_set_uccode:nn #1#2
6646 {
6647     \tex_uccode:D \__int_eval:w #1 \__int_eval_end:
6648     = \__int_eval:w #2 \__int_eval_end:
6649 }
6650 \cs_new:Npn \char_value_uccode:n #1
6651 { \tex_the:D \tex_uccode:D \__int_eval:w #1\__int_eval_end: }
6652 \cs_new_protected:Npn \char_show_value_uccode:n #1
6653 { \__msg_show_wrap:n { > ~ \char_value_uccode:n {#1} } }
6654 \cs_new_protected:Npn \char_set_sfcode:nn #1#2
6655 {
6656     \tex_sfcode:D \__int_eval:w #1 \__int_eval_end:
6657     = \__int_eval:w #2 \__int_eval_end:
6658 }
6659 \cs_new:Npn \char_value_sfcode:n #1
6660 { \tex_the:D \tex_sfcode:D \__int_eval:w #1\__int_eval_end: }
6661 \cs_new_protected:Npn \char_show_value_sfcode:n #1
6662 { \__msg_show_wrap:n { > ~ \char_value_sfcode:n {#1} } }

```

(End definition for `\char_set_mathcode:nn` and others. These functions are documented on page 110.)

`\l_char_active_seq`

Two sequences for dealing with special characters. The first is characters which may be

`\l_char_special_seq`

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`).

```

6663 \seq_new:N \l_char_special_seq
6664 \seq_set_split:Nnn \l_char_special_seq { }
6665   { \ " \# \$ % & \^ _ \{ \} \~ }
6666 \seq_new:N \l_char_active_seq
6667 \seq_set_split:Nnn \l_char_active_seq { }
6668   { " \$ & \^ _ \~ }

```

(End definition for `\l_char_active_seq` and `\l_char_special_seq`. These variables are documented on page 111.)

14.2 Creating character tokens

`\char_set_active_eq:NN`
`\char_set_active_eq:Nc`

Four simple functions with very similar definitions, so set up using an auxiliary. These are similar to LuaTeX's `\letcharcode` primitive.

```

6669 \group_begin:
6670   \char_set_catcode_active:N \^^@
6671   \cs_set_protected:Npn \__char_tmp:nN #1#2
6672   {
6673     \cs_new_protected:cpn { #1 :nN } ##1
6674     {
6675       \group_begin:
6676         \char_set_lccode:nn { '\^^@ } { ##1 }
6677         \tex_lowercase:D { \group_end: #2 ^^@ }
6678     }
6679     \cs_new_protected:cpn { #1 :NN } ##1
6680     { \exp_not:c { #1 : nN } { '#1 } }
6681   }
6682   \__char_tmp:nN { char_set_active_eq } \cs_set_eq:NN
6683   \__char_tmp:nN { char_gset_active_eq } \cs_gset_eq:NN
6684 \group_end:
6685 \cs_generate_variant:Nn \char_set_active_eq:NN { Nc }
6686 \cs_generate_variant:Nn \char_gset_active_eq:NN { Nc }
6687 \cs_generate_variant:Nn \char_set_active_eq:nN { nc }
6688 \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 107.)

`\char_generate:nn`
`__char_generate:nn`
`__char_generate_aux:nn`
`__char_generate_aux:nw`

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.

```

6689 \cs_new:Npn \char_generate:nn #1#2
6690   {
6691     \exp:w \exp_after:wN \__char_generate_aux:w
6692       \__int_value:w \__int_eval:w #1 \exp_after:wN ;
6693       \__int_value:w \__int_eval:w #2 ;
6694   }
6695 \cs_new:Npn \__char_generate:nn #1#2
6696   {
6697     \exp:w \exp_after:wN
6698       \__char_generate_aux:nnw \exp_after:wN
6699         { \__int_value:w \__int_eval:w #1 \exp_after:wN }

```

```

6700           {#2} \exp_end:
6701       }

```

Before doing any actual conversion, first some special case filtering. The \Ucharcat primitive cannot make active chars, so that is turned off here: if the primitive gets altered then the code is already in place for 8-bit engines and will kick in for LuaTeX too. Spaces are also banned here as LuaTeX emulation only makes normal (charcode 32 spaces). However, $\sim\circlearrowright$ 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.

```

6702 \cs_new:Npn \__char_generate_aux:w #1 ; #2 ;
6703   {
6704     \if_int_compare:w #2 = 13 \exp_stop_f:
6705       \__msg_kernel_expandable_error:nn { kernel } { char-active }
6706     \else:
6707       \if_int_compare:w #2 = 10 \exp_stop_f:
6708         \if_int_compare:w #1 = 0 \exp_stop_f:
6709           \__msg_kernel_expandable_error:nn { kernel } { char-null-space }
6710         \else:
6711           \__msg_kernel_expandable_error:nn { kernel } { char-space }
6712         \fi:
6713       \else:
6714         \if_int_odd:w 0
6715           \if_int_compare:w #2 < 1 \exp_stop_f: 1 \fi:
6716           \if_int_compare:w #2 = 5 \exp_stop_f: 1 \fi:
6717           \if_int_compare:w #2 = 9 \exp_stop_f: 1 \fi:
6718           \if_int_compare:w #2 > 13 \exp_stop_f: 1 \fi: \exp_stop_f:
6719         \__msg_kernel_expandable_error:nn { kernel }
6720           { char-invalid-catcode }
6721       \else:
6722         \if_int_odd:w 0
6723           \if_int_compare:w #1 < 0 \exp_stop_f: 1 \fi:
6724           \if_int_compare:w #1 > \c_char_max_int 1 \fi: \exp_stop_f:
6725             \__msg_kernel_expandable_error:nn { kernel }
6726               { char-out-of-range }
6727             \else:
6728               \__char_generate_aux:nnw {#1} {#2}
6729             \fi:
6730             \fi:
6731           \fi:
6732         \fi:
6733       \exp_end:
6734   }
6735 \tl_new:N \l__char_tmp_tl

```

Engine-dependent definitions are now needed for the implementation. For LuaTeX and recent XeTeX releases 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 for speed. 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.

```

6736 \group_begin:
6737 (*package)
6738   \char_set_catcode_active:N \^\~L
6739   \cs_set:Npn \^\~L { }
6740 
```

```

6741   \char_set_catcode_other:n { 0 }
6742   \if_int_odd:w 0
6743     \cs_if_exist:NT \luatex_directlua:D { 1 }
6744     \cs_if_exist:NT \utex_charcat:D { 1 } \exp_stop_f:
6745     \int_const:Nn \c__char_max_int { 1114111 }
6746     \cs_if_exist:NTF \luatex_directlua:D
6747     {
6748       \cs_new:Npn \__char_generate_aux:nnw #1#2#3 \exp_end:
6749       {
6750         #3
6751         \exp_after:wN \exp_end:
6752         \luatex_directlua:D { l3kernel.charcat(#1, #2) }
6753       }
6754     }
6755   {
6756     \cs_new:Npn \__char_generate_aux:nnw #1#2#3 \exp_end:
6757     {
6758       #3
6759       \exp_after:wN \exp_end:
6760       \utex_charcat:D #1 ~ #2 ~
6761     }
6762   }
6763 \else:

```

For engines where `\Ucharcat` isn't available (or emulated) then 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.

```

6764   \int_const:Nn \c__char_max_int { 255 }
6765   \tl_set:Nn \l__char_tmp_tl { \exp_not:N \or: }
6766   \char_set_catcode_group_begin:n { 0 } % {
6767   \tl_put_right:Nn \l__char_tmp_tl { ^^@ \if_false: } }
6768   \char_set_catcode_group_end:n { 0 }
6769   \tl_put_right:Nn \l__char_tmp_tl { { \fi: \exp_not:N \or: ^^@ } % }
6770   \tl_set:Nx \l__char_tmp_tl { \l__char_tmp_tl }
6771   \char_set_catcode_math_toggle:n { 0 }
6772   \tl_put_right:Nn \l__char_tmp_tl { \or: ^^@ }

```

As `\TeX` will be very unhappy if it finds an alignment character inside a primitive `\halign` even when skipping false branches, some precautions are required. `\TeX` will be happy if the token is hidden inside `\unexpanded` (which needs to be the primitive). The expansion chain here is required so that the conditional gets cleaned up correctly (other code assumes there is exactly one token to skip during the clean-up).

```

6773   \char_set_catcode_alignment:n { 0 }
6774   \tl_put_right:Nn \l__char_tmp_tl
6775   {
6776     \or:
6777     \etex_unexpanded:D \exp_after:wN
6778     { \exp_after:wN ^^@ \exp_after:wN }
6779   }
6780   \tl_put_right:Nn \l__char_tmp_tl { \or: }

```

```

6781 \char_set_catcode_parameter:n { 0 }
6782 \tl_put_right:Nn \l__char_tmp_tl { \or: ^@ }
6783 \char_set_catcode_math_superscript:n { 0 }
6784 \tl_put_right:Nn \l__char_tmp_tl { \or: ^@ }
6785 \char_set_catcode_math_subscript:n { 0 }
6786 \tl_put_right:Nn \l__char_tmp_tl { \or: ^@ }
6787 \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. We also set up active tokens although they are (currently) filtered out by the interface layer (\Ucharcat cannot make active tokens).

```

6788 \char_set_catcode_space:n { 0 }
6789 \tl_put_right:No \l__char_tmp_tl { \use:n { \or: } ^@ }
6790 \char_set_catcode_letter:n { 0 }
6791 \tl_put_right:Nn \l__char_tmp_tl { \or: ^@ }
6792 \char_set_catcode_other:n { 0 }
6793 \tl_put_right:Nn \l__char_tmp_tl { \or: ^@ }
6794 \char_set_catcode_active:n { 0 }
6795 \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. In package mode, ^L is awkward hence this is done in three parts. Notice that at this stage ^@ is active.

```

6796 \cs_set_protected:Npn \__char_tmp:n #1
6797 {
6798     \char_set_lccode:nn { 0 } {#1}
6799     \char_set_lccode:nn { 32 } {#1}
6800     \exp_args:Nx \tex_lowercase:D
6801     {
6802         \tl_const:Nn
6803             \exp_not:c { c__char_ \__int_to_roman:w #1 _tl }
6804             { \exp_not:o \l__char_tmp_tl }
6805     }
6806 }
6807 (*package)
6808     \int_step_function:nnnN { 0 } { 1 } { 11 } \__char_tmp:n
6809     \group_begin:
6810         \tl_replace_once:Nnn \l__char_tmp_tl { ^@ } { \ERROR }
6811         \__char_tmp:n { 12 }
6812     \group_end:
6813         \int_step_function:nnnN { 13 } { 1 } { 255 } \__char_tmp:n
6814     
```

/package)

(*initex)

```

6815     \int_step_function:nnnN { 0 } { 1 } { 255 } \__char_tmp:n
6816     
```

/initex)

```

6817     \int_step_function:nnnN { 0 } { 1 } { 255 } \__char_tmp:n
6818     \cs_new:Npn \__char_generate_aux:nw #1#2#3 \exp_end:
6819     {
6820         #3
6821         \exp_after:wN \exp_after:wN
6822         \exp_after:wN \exp_end:
6823         \exp_after:wN \exp_after:wN
6824         \if_case:w #2
6825             \exp_last_unbraced:Nv \exp_stop_f:

```

```

6826           { c__char_ \__int_to_roman:w #1 _tl }
6827       \fi:
6828   }
6829   \fi:
6830 \group_end:

```

(End definition for `\char_generate:nn` and others. These functions are documented on page 108.)

14.3 Generic tokens

```
6831 <@@=token>
```

`\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`

`\token_new:Nn` Creates a new token.

```
6832 \cs_new_protected:Npn \token_new:Nn #1#2 { \cs_new_eq:NN #1 #2 }
```

(End definition for `\token_new:Nn`. This function is documented on page 111.)

`\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 `__chk_if_free_cs:N` check.

```

6833 \group_begin:
6834   \__chk_if_free_cs:N \c_group_begin_token
6835   \tex_global:D \tex_let:D \c_group_begin_token {
6836     \__chk_if_free_cs:N \c_group_end_token
6837     \tex_global:D \tex_let:D \c_group_end_token }
6838     \char_set_catcode_math_toggle:N \
6839     \cs_new_eq:NN \c_math_toggle_token *
6840     \char_set_catcode_alignment:N \
6841     \cs_new_eq:NN \c_alignment_token *
6842     \cs_new_eq:NN \c_parameter_token #
6843     \cs_new_eq:NN \c_math_superscript_token ^
6844     \char_set_catcode_math_subscript:N \
6845     \cs_new_eq:NN \c_math_subscript_token *
6846     \__chk_if_free_cs:N \c_space_token
6847     \use:n { \tex_global:D \tex_let:D \c_space_token = ~ } ~
6848     \cs_new_eq:NN \c_catcode_letter_token a
6849     \cs_new_eq:NN \c_catcode_other_token 1
6850 \group_end:

```

(End definition for `\c_group_begin_token` and others. These functions are documented on page 111.)

`\c_catcode_active_tl` Not an implicit token!

```

6851 \group_begin:
6852   \char_set_catcode_active:N \
6853   \tl_const:Nn \c_catcode_active_tl { \exp_not:N * }
6854 \group_end:

```

(End definition for `\c_catcode_active_tl`. This variable is documented on page 111.)

14.4 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.

```
6855 \prg_new_conditional:Npnn \token_if_group_begin:N #1 { p , T , F , TF }
6856 {
6857   \if_catcode:w \exp_not:N #1 \c_group_begin_token
6858     \prg_return_true: \else: \prg_return_false: \fi:
6859 }
```

(End definition for `\token_if_group_begin:NTF`. This function is documented on page 112.)

`\token_if_group_end_p:N` Check if token is a end group token. We use the constant `\c_group_end_token` for this.

```
6860 \prg_new_conditional:Npnn \token_if_group_end:N #1 { p , T , F , TF }
6861 {
6862   \if_catcode:w \exp_not:N #1 \c_group_end_token
6863     \prg_return_true: \else: \prg_return_false: \fi:
6864 }
```

(End definition for `\token_if_group_end:NTF`. This function is documented on page 112.)

`\token_if_math_toggle_p:N` Check if token is a math shift token. We use the constant `\c_math_toggle_token` for this.

```
6865 \prg_new_conditional:Npnn \token_if_math_toggle:N #1 { p , T , F , TF }
6866 {
6867   \if_catcode:w \exp_not:N #1 \c_math_toggle_token
6868     \prg_return_true: \else: \prg_return_false: \fi:
6869 }
```

(End definition for `\token_if_math_toggle:NTF`. This function is documented on page 112.)

`\token_if_alignment_p:N` Check if token is an alignment tab token. We use the constant `\c_alignment_token` for this.

```
6870 \prg_new_conditional:Npnn \token_if_alignment:N #1 { p , T , F , TF }
6871 {
6872   \if_catcode:w \exp_not:N #1 \c_alignment_token
6873     \prg_return_true: \else: \prg_return_false: \fi:
6874 }
```

(End definition for `\token_if_alignment:NTF`. This function is documented on page 112.)

`\token_if_parameter_p:N` Check if token is a parameter token. We use the constant `\c_parameter_token` for this.

`\token_if_parameter:NTF` 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 will remain after the group.

```
6875 \group_begin:
6876 \cs_set_eq:NN \c_parameter_token \scan_stop:
6877 \prg_new_conditional:Npnn \token_if_parameter:N #1 { p , T , F , TF }
6878 {
6879   \if_catcode:w \exp_not:N #1 \c_parameter_token
6880     \prg_return_true: \else: \prg_return_false: \fi:
6881 }
6882 \group_end:
```

(End definition for `\token_if_parameter:NTF`. This function is documented on page 113.)

\token_if_math_superscript_p:N Check if token is a math superscript token. We use the constant `\c_math_superscript_token` for this.

```
6883 \prg_new_conditional:Npnn \token_if_math_superscript:N #1
6884   { p , T , F , TF }
6885   {
6886     \if_catcode:w \exp_not:N #1 \c_math_superscript_token
6887     \prg_return_true: \else: \prg_return_false: \fi:
6888   }
```

(End definition for `\token_if_math_superscript:NTF`. This function is documented on page 113.)

\token_if_math_subscript_p:N Check if token is a math subscript token. We use the constant `\c_math_subscript_token` for this.

```
6889 \prg_new_conditional:Npnn \token_if_math_subscript:N #1 { p , T , F , TF }
6890   {
6891     \if_catcode:w \exp_not:N #1 \c_math_subscript_token
6892     \prg_return_true: \else: \prg_return_false: \fi:
6893   }
```

(End definition for `\token_if_math_subscript:NTF`. This function is documented on page 113.)

\token_if_space_p:N Check if token is a space token. We use the constant `\c_space_token` for this.

```
6894 \prg_new_conditional:Npnn \token_if_space:N #1 { p , T , F , TF }
6895   {
6896     \if_catcode:w \exp_not:N #1 \c_space_token
6897     \prg_return_true: \else: \prg_return_false: \fi:
6898   }
```

(End definition for `\token_if_space:NTF`. This function is documented on page 113.)

\token_if_letter_p:N Check if token is a letter token. We use the constant `\c_catcode_letter_token` for this.

```
6899 \prg_new_conditional:Npnn \token_if_letter:N #1 { p , T , F , TF }
6900   {
6901     \if_catcode:w \exp_not:N #1 \c_catcode_letter_token
6902     \prg_return_true: \else: \prg_return_false: \fi:
6903   }
```

(End definition for `\token_if_letter:NTF`. This function is documented on page 113.)

\token_if_other_p:N Check if token is an other char token. We use the constant `\c_catcode_other_token` for this.

```
6904 \prg_new_conditional:Npnn \token_if_other:N #1 { p , T , F , TF }
6905   {
6906     \if_catcode:w \exp_not:N #1 \c_catcode_other_token
6907     \prg_return_true: \else: \prg_return_false: \fi:
6908   }
```

(End definition for `\token_if_other:NTF`. This function is documented on page 113.)

\token_if_active_p:N Check if token is an active char token. We use the constant `\c_catcode_active_tl` for this. A technical point is that `\c_catcode_active_tl` is in fact a macro expanding to `\exp_not:N *`, where * is active.

```
6909 \prg_new_conditional:Npnn \token_if_active:N #1 { p , T , F , TF }
6910   {
6911     \if_catcode:w \exp_not:N #1 \c_catcode_active_tl
6912     \prg_return_true: \else: \prg_return_false: \fi:
6913   }
```

(End definition for `\token_if_active:N`. This function is documented on page 113.)

`\token_if_eq_meaning_p:NN` Check if the tokens #1 and #2 have same meaning.
`\token_if_eq_meaning:NNTF`

```
6914 \prg_new_conditional:Npnn \token_if_eq_meaning:NN #1#2 { p , T , F , TF }
6915 {
6916     \if_meaning:w #1 #2
6917         \prg_return_true: \else: \prg_return_false: \fi:
6918 }
```

(End definition for `\token_if_eq_meaning:NNTF`. This function is documented on page 113.)

`\token_if_eq_catcode_p:NN` Check if the tokens #1 and #2 have same category code.
`\token_if_eq_catcode:NNTF`

```
6919 \prg_new_conditional:Npnn \token_if_eq_catcode:NN #1#2 { p , T , F , TF }
6920 {
6921     \if_catcode:w \exp_not:N #1 \exp_not:N #2
6922         \prg_return_true: \else: \prg_return_false: \fi:
6923 }
```

(End definition for `\token_if_eq_catcode:NNTF`. This function is documented on page 113.)

`\token_if_eq_charcode_p:NN` Check if the tokens #1 and #2 have same character code.
`\token_if_eq_charcode:NNTF`

```
6924 \prg_new_conditional:Npnn \token_if_eq_charcode:NN #1#2 { p , T , F , TF }
6925 {
6926     \if_charcode:w \exp_not:N #1 \exp_not:N #2
6927         \prg_return_true: \else: \prg_return_false: \fi:
6928 }
```

(End definition for `\token_if_eq_charcode:NNTF`. This function is documented on page 113.)

`\token_if_macro_p:N` When a token is a macro, `\token_to_meaning:N` will always output something like
`\token_if_macro:NNTF` `\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.

```
6929 \use:x
6930 {
6931     \prg_new_conditional:Npnn \exp_not:N \token_if_macro:N ##1
6932         { p , T , F , TF }
6933     {
6934         \exp_not:N \exp_after:wN \exp_not:N \__token_if_macro_p:w
6935         \exp_not:N \token_to_meaning:N ##1 \tl_to_str:n { ma : }
6936             \exp_not:N \q_stop
6937     }
6938     \cs_new:Npn \exp_not:N \__token_if_macro_p:w
```

```

6939     ##1 \tl_to_str:n { ma } ##2 \c_colon_str ##3 \exp_not:N \q_stop
6940 }
6941 {
6942     \if_int_compare:w \__str_if_eq_x:nn { #2 } { cro } = 0 \exp_stop_f:
6943         \prg_return_true:
6944     \else:
6945         \prg_return_false:
6946     \fi:
6947 }

```

(End definition for `\token_if_macro:NTF` and `_token_if_macro_p:w`. These functions are documented on page 114.)

`\token_if_cs_p:N` Check if token has same catcode as a control sequence. This follows the same pattern as for `\token_if_letter:N` etc. We use `\scan_stop:` for this.

```

6948 \prg_new_conditional:Npnn \token_if_cs:N #1 { p , T , F , TF }
6949 {
6950     \if_catcode:w \exp_not:N #1 \scan_stop:
6951         \prg_return_true: \else: \prg_return_false: \fi:
6952 }

```

(End definition for `\token_if_cs:NTF`. This function is documented on page 114.)

`\token_if_expandable_p:N` Check if token is expandable. We use the fact that TeX will temporarily convert `\exp_not:N <token>` into `\scan_stop:` if `<token>` 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 TeX's conditional apparatus).

```

6953 \prg_new_conditional:Npnn \token_if_expandable:N #1 { p , T , F , TF }
6954 {
6955     \exp_after:wN \if_meaning:w \exp_not:N #1 #1
6956         \prg_return_false:
6957     \else:
6958         \if_cs_exist:N #1
6959             \prg_return_true:
6960         \else:
6961             \prg_return_false:
6962         \fi:
6963     \fi:
6964 }

```

(End definition for `\token_if_expandable:NTF`. This function is documented on page 114.)

`_token_delimit_by_char: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 `\q_stop`, and returns the first one and its delimiter. This result will eventually be compared to another string.

```

\_token_delimit_by_count:w
\_token_delimit_by_dimen:w
\_token_delimit_by_macro:w
\_token_delimit_by_muskip:w
\_token_delimit_by_skip:w
\_token_delimit_by_toks:w

```

```

6965 \group_begin:
6966 \cs_set_protected:Npn \_token_tmp:w #1
6967 {
6968     \use:x
6969     {
6970         \cs_new:Npn \exp_not:c { \_token_delimit_by_ #1 :w }
6971             #####1 \tl_to_str:n {#1} #####2 \exp_not:N \q_stop
6972             { #####1 \tl_to_str:n {#1} }

```

```

6973     }
6974   }
6975 \__token_tmp:w { char" }
6976 \__token_tmp:w { count }
6977 \__token_tmp:w { dimen }
6978 \__token_tmp:w { macro }
6979 \__token_tmp:w { muskip }
6980 \__token_tmp:w { skip }
6981 \__token_tmp:w { toks }
6982 \group_end:

```

(End definition for `__token_delimit_by_char":w` and others.)

`\token_if_chardef_p:N` 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_x_return:nn` 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.
`\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_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`

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 five conditionals, `\cs_if_exist:cT` turns out to be `false`, 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).

```

6983 \group_begin:
6984 \cs_set_protected:Npn \__token_tmp:w #1#2#3
6985 {
6986   \use:x
6987   {
6988     \prg_new_conditional:Npnn \exp_not:c { token_if_ #1 :N } ####1
6989       { p , T , F , TF }
6990       {
6991         \cs_if_exist:cT { tex_ #2 :D }
6992         {
6993           \exp_not:N \if_meaning:w ####1 \exp_not:c { tex_ #2 :D }

```

```

6994   \exp_not:N \prg_return_false:
6995   \exp_not:N \else:
6996   \exp_not:N \if_meaning:w #####1 \exp_not:c { tex_ #2 def:D }
6997   \exp_not:N \prg_return_false:
6998   \exp_not:N \else:
6999   }
7000   \exp_not:N \__str_if_eq_x_return:nn
7001   {
7002     \exp_not:N \exp_after:wN
7003     \exp_not:c { __token_delimit_by_ #2 :w }
7004     \exp_not:N \token_to_meaning:N #####1
7005     ? \tl_to_str:n {#2} \exp_not:N \q_stop
7006   }
7007   { \exp_not:n {#3} }
7008   \cs_if_exist:cT { tex_ #2 :D }
7009   {
7010     \exp_not:N \fi:
7011     \exp_not:N \fi:
7012   }
7013 }
7014 }
7015 }
7016 \__token_tmp:w { chardef } { char" } { \token_to_str:N \char" }
7017 \__token_tmp:w { mathchardef } { char" } { \token_to_str:N \mathchar" }
7018 \__token_tmp:w { long_macro } { macro } { \tl_to_str:n { \long } macro }
7019 \__token_tmp:w { protected_macro } { macro }
7020 { \tl_to_str:n { \protected } macro }
7021 \__token_tmp:w { protected_long_macro } { macro }
7022 { \token_to_str:N \protected \tl_to_str:n { \long } macro }
7023 \__token_tmp:w { dim_register } { dimen } { \token_to_str:N \dimen }
7024 \__token_tmp:w { int_register } { count } { \token_to_str:N \count }
7025 \__token_tmp:w { muskip_register } { muskip } { \token_to_str:N \muskip }
7026 \__token_tmp:w { skip_register } { skip } { \token_to_str:N \skip }
7027 \__token_tmp:w { toks_register } { toks } { \token_to_str:N \toks }
7028 \group_end:

```

(End definition for `\token_if_chardef:NTF` and others. These functions are documented on page 114.)

```

\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

```

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 `\q_stop`.

The meaning of each one of the five `\...mark` primitives has the form `<letters>:<user material>`. 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.

```

7029 \tex_chardef:D \c__token_A_int = 'A ~ %
7030 \use:x
7031 {
7032   \prg_new_conditional:Npnn \exp_not:N \token_if_primitive:N ##1
7033   { p , T , F , TF }
7034   {
7035     \exp_not:N \token_if_macro:NTF ##1
7036     \exp_not:N \prg_return_false:
7037     {
7038       \exp_not:N \exp_after:wn \exp_not:N \__token_if_primitive>NNw
7039       \exp_not:N \token_to_meaning:N ##1
7040       \tl_to_str:n { : : : } \exp_not:N \q_stop ##1
7041     }
7042   }
7043   \cs_new:Npn \exp_not:N \__token_if_primitive>NNw
7044   ##1##2 ##3 \c_colon_str ##4 \exp_not:N \q_stop
7045   {
7046     \exp_not:N \tl_if_empty:oTF
7047     { \exp_not:N \__token_if_primitive_space:w ##3 ~ }
7048     {
7049       \exp_not:N \__token_if_primitive_loop:N ##3
7050       \c_colon_str \exp_not:N \q_stop
7051     }
7052     { \exp_not:N \__token_if_primitive_nullfont:N }
7053   }
7054 }
7055 \cs_new:Npn \__token_if_primitive_space:w #1 ~ { }
7056 \cs_new:Npn \__token_if_primitive_nullfont:N #1
7057 {
7058   \if_meaning:w \tex_nullfont:D #1
7059   \prg_return_true:
7060   \else:
7061   \prg_return_false:
7062   \fi:
7063 }
7064 \cs_new:Npn \__token_if_primitive_loop:N #1
7065 {
7066   \if_int_compare:w '#1 < \c__token_A_int %
7067   \exp_after:wn \__token_if_primitive:Nw
7068   \exp_after:wn #1
7069   \else:
7070   \exp_after:wn \__token_if_primitive_loop:N
7071   \fi:
7072 }
```

```

7073 \cs_new:Npn \__token_if_primitive:Nw #1 #2 \q_stop
7074 {
7075     \if:w : #1
7076         \exp_after:wN \__token_if_primitive_undefined:N
7077     \else:
7078         \prg_return_false:
7079         \exp_after:wN \use_none:n
7080     \fi:
7081 }
7082 \cs_new:Npn \__token_if_primitive_undefined:N #1
7083 {
7084     \if_cs_exist:N #1
7085         \prg_return_true:
7086     \else:
7087         \prg_return_false:
7088     \fi:
7089 }

```

(End definition for `\token_if_primitive:NTF` and others. These functions are documented on page [115](#).)

14.5 Peeking ahead at the next token

`\@@=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`

```

7091 \cs_new_eq:NN \l_peek_token ?
7092 \cs_new_eq:NN \g_peek_token ?

```

(End definition for `\l_peek_token` and `\g_peek_token`. These variables are documented on page [116](#).)

`\l__peek_search_token` The token to search for as an implicit token: cf. `\l__peek_search_tl`.

`\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`.

`\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`

```

7095 \cs_new:Npn \__peek_true:w { }
7096 \cs_new:Npn \__peek_true_aux:w { }
7097 \cs_new:Npn \__peek_false:w { }
7098 \cs_new:Npn \__peek_tmp:w { }

```

(End definition for `_peek_true:w` and others.)

`\peek_after:Nw` Simple wrappers for `\futurelet`: no arguments absorbed here.

`\peek_gafter:Nw`

```
7099 \cs_new_protected:Npn \peek_after:Nw
7100   { \tex_futurelet:D \l_peek_token }
7101 \cs_new_protected:Npn \peek_gafter:Nw
7102   { \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 115.)

`_peek_true_remove:w` A function to remove the next token and then regain control.

```
7103 \cs_new_protected:Npn \_peek_true_remove:w
7104   {
7105     \group_align_safe_end:
7106     \tex_afterassignment:D \_peek_true_aux:w
7107     \cs_set_eq:NN \_peek_tmp:w
7108   }
```

(End definition for `_peek_true_remove:w`)

`_peek_token_generic:NNTF` The generic function stores 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.

```
7109 \cs_new_protected:Npn \_peek_token_generic:NNTF #1#2#3#4
7110   {
7111     \cs_set_eq:NN \l_peek_search_token #2
7112     \tl_set:Nn \l_peek_search_tl {\#2}
7113     \cs_set:Npx \_peek_true:w
7114       {
7115         \exp_not:N \group_align_safe_end:
7116         \exp_not:n {\#3}
7117       }
7118     \cs_set:Npx \_peek_false:w
7119       {
7120         \exp_not:N \group_align_safe_end:
7121         \exp_not:n {\#4}
7122       }
7123     \group_align_safe_begin:
7124       \peek_after:Nw #1
7125   }
7126 \cs_new_protected:Npn \_peek_token_generic:NNT #1#2#3
7127   { \_peek_token_generic:NNTF #1 #2 {\#3} {} }
7128 \cs_new_protected:Npn \_peek_token_generic:NNF #1#2#3
7129   { \_peek_token_generic:NNTF #1 #2 {} {\#3} }
```

(End definition for `_peek_token_generic:NNTF`)

`_peek_token_remove_generic:NNTF` For token removal there needs to be a call to the auxiliary function which does the work.

```
7130 \cs_new_protected:Npn \_peek_token_remove_generic:NNTF #1#2#3#4
7131   {
7132     \cs_set_eq:NN \l_peek_search_token #2
7133     \tl_set:Nn \l_peek_search_tl {\#2}
7134     \cs_set_eq:NN \_peek_true:w \_peek_true_remove:w
7135     \cs_set:Npx \_peek_true_aux:w { \exp_not:n {\#3} }
```

```

7136     \cs_set:Npx \__peek_false:w
7137     {
7138         \exp_not:N \group_align_safe_end:
7139         \exp_not:n {#4}
7140     }
7141     \group_align_safe_begin:
7142         \peek_after:Nw #1
7143     }
7144 \cs_new_protected:Npn \__peek_token_remove_generic:NNT #1#2#3
7145     { \__peek_token_remove_generic:NNTF #1 #2 {#3} { } }
7146 \cs_new_protected:Npn \__peek_token_remove_generic:NNF #1#2#3
7147     { \__peek_token_remove_generic:NNTF #1 #2 { } {#3} }

```

(End definition for `__peek_token_remove_generic:NNTF`.)

`__peek_execute_branches_meaning`: The meaning test is straight forward.

```

7148 \cs_new:Npn \__peek_execute_branches_meaning:
7149     {
7150         \if_meaning:w \l_peek_token \l__peek_search_token
7151             \exp_after:wN \__peek_true:w
7152         \else:
7153             \exp_after:wN \__peek_false:w
7154         \fi:
7155     }

```

(End definition for `__peek_execute_branches_meaning`.)

```

\__peek_execute_branches_catcode:
\__peek_execute_branches_charcode:
\__peek_execute_branches_catcode_aux:
\__peek_execute_branches_catcode_auxii:N
\__peek_execute_branches_catcode_auxiii:

```

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 will only be given in the `auxii:N` and `auxiii:` auxiliaries. For our purposes, three kinds of tokens may follow the peeking function:

- 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_t1`. The `\exp_not:N` prevents outer macros (coming from non-L^AT_EX3 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.

```

7156 \cs_new:Npn \__peek_execute_branches_catcode:
7157     { \if_catcode:w \__peek_execute_branches_catcode_aux: }
7158 \cs_new:Npn \__peek_execute_branches_charcode:
7159     { \if_charcode:w \__peek_execute_branches_catcode_aux: }
7160 \cs_new:Npn \__peek_execute_branches_catcode_aux:

```

```

7161   {
7162     \if_catcode:w \exp_not:N \l_peek_token \scan_stop:
7163       \exp_after:wN \exp_after:wN
7164         \exp_after:wN \__peek_execute_branches_catcode_auxii:N
7165           \exp_after:wN \exp_not:N
7166         \else:
7167           \exp_after:wN \__peek_execute_branches_catcode_auxii:
7168         \fi:
7169     }
7170   \cs_new:Npn \__peek_execute_branches_catcode_auxii:N #1
7171   {
7172     \exp_not:N #1
7173     \exp_after:wN \exp_not:N \l__peek_search_tl
7174       \exp_after:wN \__peek_true:w
7175     \else:
7176       \exp_after:wN \__peek_false:w
7177     \fi:
7178   #1
7179 }
7180 \cs_new:Npn \__peek_execute_branches_catcode_auxiii:
7181 {
7182   \exp_not:N \l_peek_token
7183   \exp_after:wN \exp_not:N \l__peek_search_tl
7184     \exp_after:wN \__peek_true:w
7185   \else:
7186     \exp_after:wN \__peek_false:w
7187   \fi:
7188 }

```

(End definition for `__peek_execute_branches_catcode:` and others.)

`__peek_ignore_spaces_execute_branches:`

This function removes one space token at a time, and calls `__peek_execute_branches:` when encountering the first non-space token. We directly use the primitive meaning test rather than `\token_if_eq_meaning:NNTF` because `\l_peek_token` may be an outer macro (coming from non-L^AT_EX3 packages). Spaces are removed using a side-effect of f-expansion: `\exp:w \exp_end_continue_f:w` removes one space.

```

7189 \cs_new_protected:Npn \__peek_ignore_spaces_execute_branches:
7190   {
7191     \if_meaning:w \l_peek_token \c_space_token
7192       \exp_after:wN \peek_after:Nw
7193       \exp_after:wN \__peek_ignore_spaces_execute_branches:
7194         \exp:w \exp_end_continue_f:w
7195     \else:
7196       \exp_after:wN \__peek_execute_branches:
7197     \fi:
7198   }

```

(End definition for `__peek_ignore_spaces_execute_branches:..`)

`__peek_def:nnnn`
`__peek_def:nnnn`

The public functions themselves cannot be defined using `\prg_new_conditional:Npnn` and so a couple of auxiliary functions are used. As a result, everything is done inside a group. As a result things are a bit complicated.

```

7199 \group_begin:
7200   \cs_set:Npn \__peek_def:nnnn #1#2#3#4

```

```

7201   {
7202     \__peek_def:nnnn {#1} {#2} {#3} {#4} { TF }
7203     \__peek_def:nnnn {#1} {#2} {#3} {#4} { T }
7204     \__peek_def:nnnn {#1} {#2} {#3} {#4} { F }
7205   }
7206 \cs_set:Npn \__peek_def:nnnn #1#2#3#4#5
7207   {
7208     \cs_new_protected:cpx { #1 #5 }
7209   {
7210     \tl_if_empty:nF {#2}
7211       { \exp_not:n { \cs_set_eq:NN \__peek_execute_branches: #2 } }
7212     \exp_not:c { #3 #5 }
7213     \exp_not:n {#4}
7214   }
7215 }

```

(End definition for `__peek_def:nnnn` and `__peek_def:nnnnn`.)

\peek_catcode:NTF With everything in place the definitions can take place. First for category codes.

```

7216 \__peek_def:nnnn { peek_catcode:N }
7217   { }
7218   { __peek_token_generic:NN }
7219   { \__peek_execute_branches_catcode: }
7220 \__peek_def:nnnn { peek_catcode_ignore_spaces:N }
7221   { \__peek_execute_branches_catcode: }
7222   { __peek_token_generic:NN }
7223   { \__peek_ignore_spaces_execute_branches: }
7224 \__peek_def:nnnn { peek_catcode_remove:N }
7225   { }
7226   { __peek_token_remove_generic:NN }
7227   { \__peek_execute_branches_catcode: }
7228 \__peek_def:nnnn { peek_catcode_remove_ignore_spaces:N }
7229   { \__peek_execute_branches_catcode: }
7230   { __peek_token_remove_generic:NN }
7231   { \__peek_ignore_spaces_execute_branches: }

```

(End definition for `\peek_catcode:NTF` and others. These functions are documented on page 116.)

\peek_charcode:NTF Then for character codes.

```

7232 \__peek_def:nnnn { peek_charcode:N }
7233   { }
7234   { __peek_token_generic:NN }
7235   { \__peek_execute_branches_charcode: }
7236 \__peek_def:nnnn { peek_charcode_ignore_spaces:N }
7237   { \__peek_execute_branches_charcode: }
7238   { __peek_token_generic:NN }
7239   { \__peek_ignore_spaces_execute_branches: }
7240 \__peek_def:nnnn { peek_charcode_remove:N }
7241   { }
7242   { __peek_token_remove_generic:NN }
7243   { \__peek_execute_branches_charcode: }
7244 \__peek_def:nnnn { peek_charcode_remove_ignore_spaces:N }
7245   { \__peek_execute_branches_charcode: }
7246   { __peek_token_remove_generic:NN }
7247   { \__peek_ignore_spaces_execute_branches: }

```

(End definition for `\peek_charcode:NTF` and others. These functions are documented on page 116.)

`\peek_meaning:NTF`

Finally for meaning, with the group closed to remove the temporary definition functions.

```
7248  \__peek_def:nmm { peek_meaning:N }
7249  {
7250  { __peek_token_generic:NN }
7251  { \__peek_execute_branches_meaning: }
7252  \__peek_def:nmm { peek_meaning_ignore_spaces:N }
7253  { \__peek_execute_branches_meaning: }
7254  { __peek_token_generic:NN }
7255  { \__peek_ignore_spaces_execute_branches: }
7256  \__peek_def:nnnn { peek_meaning_remove:N }
7257  {
7258  { __peek_token_remove_generic:NN }
7259  { \__peek_execute_branches_meaning: }
7260  \__peek_def:nnnn { peek_meaning_remove_ignore_spaces:N }
7261  { \__peek_execute_branches_meaning: }
7262  { __peek_token_remove_generic:NN }
7263  { \__peek_ignore_spaces_execute_branches: }
7264 \group_end:
```

(End definition for `\peek_meaning:NTF` and others. These functions are documented on page 117.)

14.6 Decomposing a macro definition

`\token_get_prefix_spec:N`

`\token_get_arg_spec:N`

`\token_get_replacement_spec:N`

`__peek_get_prefix_arg_replacement:wN`

We sometimes want to test if a control sequence can be expanded to reveal a hidden value. 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 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 `\scan_stop:` is returned instead.

```
7265 \exp_args:Nno \use:nn
7266 { \cs_new:Npn \__peek_get_prefix_arg_replacement:wN #1 }
7267 { \tl_to_str:n { macro : } #2 -> #3 \q_stop #4 }
7268 { #4 {#1} {#2} {#3} }
7269 \cs_new:Npn \token_get_prefix_spec:N #1
7270 {
7271     \token_if_macro:NTF #1
7272     {
7273         \exp_after:wN \__peek_get_prefix_arg_replacement:wN
7274         \token_to_meaning:N #1 \q_stop \use_i:nnn
7275     }
7276     { \scan_stop: }
7277 }
7278 \cs_new:Npn \token_get_arg_spec:N #1
7279 {
7280     \token_if_macro:NTF #1
7281     {
7282         \exp_after:wN \__peek_get_prefix_arg_replacement:wN
7283         \token_to_meaning:N #1 \q_stop \use_i:nnn
7284     }
7285     { \scan_stop: }
7286 }
```

```

7287 \cs_new:Npn \token_get_replacement_spec:N #1
7288   {
7289     \token_if_macro:NTF #1
7290     {
7291       \exp_after:wN \__peek_get_prefix_arg_replacement:wN
7292         \token_to_meaning:N #1 \q_stop \use_iii:nnn
7293     }
7294   { \scan_stop: }
7295 }
```

(End definition for `\token_get_prefix_spec:N` and others. These functions are documented on page [119](#).)

7296 ⟨/initex | package⟩

15 l3prop implementation

The following test files are used for this code: `m3prop001`, `m3prop002`, `m3prop003`, `m3prop004`, `m3show001`.

```

7297 {*initex | package}
7298 (@@=prop)
```

A property list is a macro whose top-level expansion is of the form

```

\__prop \__prop_pair:wn <key1> \__prop {<value1>}
...
\__prop \__prop_pair:wn <keyn> \__prop {<valuen>}
```

where `__prop` is a scan mark (equal to `\scan_stop:`), and `__prop_pair:wn` can be used to map through the property list.

__prop A private scan mark is used as a marker after each key, and at the very beginning of the property list.

```
7299 \__scan_new:N \__prop
```

(End definition for `__prop`.)

__prop_pair:wn The delimiter is always defined, but when misused simply triggers an error and removes its argument.

```

7300 \cs_new:Npn \__prop_pair:wn #1 \__prop #2
7301   { \__msg_kernel_expandable_error:nn { kernel } { misused-prop } }
```

(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.

```
7302 \tl_new:N \l__prop_internal_tl
```

(End definition for `\l__prop_internal_tl`.)

\c_empty_prop An empty prop.

```
7303 \tl_const:Nn \c_empty_prop { \__prop }
```

(End definition for `\c_empty_prop`. This variable is documented on page [127](#).)

15.1 Allocation and initialisation

\prop_new:N Property lists are initialized with the value \c_empty_prop.

```

\prop_new:c 7304 \cs_new_protected:Npn \prop_new:N #1
             {
               __chk_if_free_cs:N #1
               \cs_gset_eq:NN #1 \c_empty_prop
             }
7309 \cs_generate_variant:Nn \prop_new:N { c }
```

(End definition for \prop_new:N. This function is documented on page 122.)

\prop_clear:N The same idea for clearing.

```

\prop_clear:c 7310 \cs_new_protected:Npn \prop_clear:N #1
               { \prop_set_eq:NN #1 \c_empty_prop }
\prop_gclear:N 7311 \cs_generate_variant:Nn \prop_clear:N { c }
\prop_gclear:c 7312 \cs_new_protected:Npn \prop_gclear:N #1
               { \prop_gset_eq:NN #1 \c_empty_prop }
7313 \cs_generate_variant:Nn \prop_gclear:N { c }
```

(End definition for \prop_clear:N and \prop_gclear:N. These functions are documented on page 122.)

\prop_clear_new:N Once again a simple variation of the token list functions.

```

\prop_clear_new:c 7316 \cs_new_protected:Npn \prop_clear_new:N #1
                  { \prop_if_exist:NTF #1 { \prop_clear:N #1 } { \prop_new:N #1 } }
\prop_gclear_new:N 7317 \cs_generate_variant:Nn \prop_clear_new:N { c }
\prop_gclear_new:c 7318 \cs_new_protected:Npn \prop_gclear_new:N #1
                  { \prop_if_exist:NTF #1 { \prop_gclear:N #1 } { \prop_new:N #1 } }
7319 \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 122.)

\prop_set_eq:NN These are simply copies from the token list functions.

```

\prop_set_eq:cN 7322 \cs_new_eq:NN \prop_set_eq:NN \tl_set_eq:NN
\prop_set_eq:Nc 7323 \cs_new_eq:NN \prop_set_eq:Nc \tl_set_eq:Nc
\prop_set_eq:cc 7324 \cs_new_eq:NN \prop_set_eq:cN \tl_set_eq:cN
\prop_gset_eq:NN 7325 \cs_new_eq:NN \prop_set_eq:cc \tl_set_eq:cc
\prop_gset_eq:cN 7326 \cs_new_eq:NN \prop_gset_eq:NN \tl_gset_eq:NN
\prop_gset_eq:Nc 7327 \cs_new_eq:NN \prop_gset_eq:Nc \tl_gset_eq:Nc
\prop_gset_eq:cc 7328 \cs_new_eq:NN \prop_gset_eq:cN \tl_gset_eq:cN
7329 \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 122.)

\l_tmpa_prop We can now initialize the scratch variables.

```

\l_tmpb_prop 7330 \prop_new:N \l_tmpa_prop
\g_tmpa_prop 7331 \prop_new:N \l_tmpb_prop
\g_tmpb_prop 7332 \prop_new:N \g_tmpa_prop
7333 \prop_new:N \g_tmpb_prop
```

(End definition for \l_tmpa_prop and others. These variables are documented on page 127.)

15.2 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 \q_mark #4 #5 \q_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*.

```
7334 \cs_new_protected:Npn \__prop_split:NnTF #1#2
7335   { \exp_args:NNo \__prop_split_aux:NnTF #1 { \tl_to_str:n {#2} } }
7336 \cs_new_protected:Npn \__prop_split_aux:NnTF #1#2#3#4
7337   {
7338     \cs_set:Npn \__prop_split_aux:w ##1
7339       \__prop_pair:wn #2 \s__prop ##2 ##3 \q_mark ##4 ##5 \q_stop
7340       { ##4 {#3} {#4} }
7341     \exp_after:wN \__prop_split_aux:w #1 \q_mark \use_i:nn
7342       \__prop_pair:wn #2 \s__prop { } \q_mark \use_ii:nn \q_stop
7343   }
7344 \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
\prop_remove:NV
\prop_remove:cN
\prop_remove:cV
\prop_gremove:Nn
\prop_gremove:NV
\prop_gremove:cN
\prop_gremove:cV
```

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.

```
7345 \cs_new_protected:Npn \prop_remove:Nn #1#2
7346   {
7347     \__prop_split:NnTF #1 {#2}
7348     { \tl_set:Nn #1 { ##1 ##3 } }
7349     { }
7350   }
7351 \cs_new_protected:Npn \prop_gremove:Nn #1#2
7352   {
7353     \__prop_split:NnTF #1 {#2}
7354     { \tl_gset:Nn #1 { ##1 ##3 } }
7355     { }
7356   }
7357 \cs_generate_variant:Nn \prop_remove:Nn { NV }
7358 \cs_generate_variant:Nn \prop_remove:Nn { c , cV }
7359 \cs_generate_variant:Nn \prop_gremove:Nn { NV }
7360 \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 124.)

`\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`.

```
7361 \cs_new_protected:Npn \prop_get:NnN #1#2#3
7362   {
7363     \__prop_split:NnTF #1 {#2}
7364     {
7365       \tl_set:Nn #3 {##2}
7366       \tl_set:Nn #3 { \q_no_value }
7367     }
7368   \cs_generate_variant:Nn \prop_get:NnN { NV , No }
7369   \cs_generate_variant:Nn \prop_get:NnN { c , cV , co }
```

(End definition for `\prop_get:NnN`. This function is documented on page 123.)

`\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.

```
7369 \cs_new_protected:Npn \prop_pop:NnN #1#2#3
7370   {
7371     \__prop_split:NnTF #1 {#2}
7372     {
7373       \tl_set:Nn #3 {##2}
7374       \tl_set:Nn #1 { ##1 ##3 }
7375     }
7376     { \tl_set:Nn #3 { \q_no_value } }
7377   }
7378 \cs_new_protected:Npn \prop_gpop:NnN #1#2#3
7379   {
7380     \__prop_split:NnTF #1 {#2}
7381     {
7382       \tl_set:Nn #3 {##2}
7383       \tl_gset:Nn #1 { ##1 ##3 }
7384     }
7385     { \tl_set:Nn #3 { \q_no_value } }
7386   }
7387 \cs_generate_variant:Nn \prop_pop:NnN { No }
7388 \cs_generate_variant:Nn \prop_pop:NnN { c , co }
7389 \cs_generate_variant:Nn \prop_gpop:NnN { No }
7390 \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 123.)

`\prop_item:Nn` Getting the value corresponding to a key in a property list in an expandable fashion is similar to mapping some tokens. Go through the property list one $\langle key \rangle$ - $\langle value \rangle$ pair at a time: the arguments of `__prop_item_Nn:nwn` are the $\langle key \rangle$ we are looking for, a $\langle key \rangle$ of the property list, and its associated value. The $\langle keys \rangle$ are compared (as strings). If they match, the $\langle value \rangle$ is returned, within `\exp_not:n`. The loop terminates even if the $\langle key \rangle$ is missing, and yields an empty value, because we have appended the appropriate $\langle key \rangle$ - $\langle empty value \rangle$ pair to the property list.

```
7391 \cs_new:Npn \prop_item:Nn #1#2
7392   {
7393     \exp_last_unbraced:Noo \__prop_item_Nn:nwn { \tl_to_str:n {#2} } #1
```

```

7394      \__prop_pair:wn \tl_to_str:n {#2} \s__prop { }
7395      \__prg_break_point:
7396    }
7397  \cs_new:Npn \__prop_item_Nn:nwwn #1#2 \__prop_pair:wn #3 \s__prop #4
7398  {
7399    \str_if_eq_x:nnTF {#1} {#3}
7400    { \__prg_break:n { \exp_not:n {#4} } }
7401    { \__prop_item_Nn:nwwn {#1} }
7402  }
7403 \cs_generate_variant:Nn \prop_item:Nn { c }

```

(End definition for `\prop_item:Nn` and `__prop_item_Nn:nwwn`. These functions are documented on page 124.)

\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:cnNTF
7404 \prg_new_protected_conditional:Npnn \prop_pop:NnN #1#2#3 { T , F , TF }
7405  {
7406    \__prop_split:NnTF #1 {#2}
7407    {
7408      \tl_set:Nn #3 {##2}
7409      \tl_set:Nn #1 { ##1 ##3 }
7410      \prg_return_true:
7411    }
7412    { \prg_return_false: }
7413  }
7414 \prg_new_protected_conditional:Npnn \prop_gpop:NnN #1#2#3 { T , F , TF }
7415  {
7416    \__prop_split:NnTF #1 {#2}
7417    {
7418      \tl_set:Nn #3 {##2}
7419      \tl_gset:Nn #1 { ##1 ##3 }
7420      \prg_return_true:
7421    }
7422    { \prg_return_false: }
7423  }
7424 \cs_generate_variant:Nn \prop_pop:NnNT { c }
7425 \cs_generate_variant:Nn \prop_pop:NnNF { c }
7426 \cs_generate_variant:Nn \prop_pop:NnTF { c }
7427 \cs_generate_variant:Nn \prop_gpop:NnNT { c }
7428 \cs_generate_variant:Nn \prop_gpop:NnNF { c }
7429 \cs_generate_variant:Nn \prop_gpop:NnTF { c }

```

(End definition for `\prop_pop:NnNTF` and `\prop_gpop:NnNTF`. These functions are documented on page 125.)

\prop_put:Nnn Since the branches of `__prop_split:NnTF` are used as the replacement text of an internal macro, and since the `\langle key\rangle` and new `\langle value\rangle` may contain arbitrary tokens, it is not safe to include them in the argument of `__prop_split:NnTF`. We thus start by storing in `\l1___prop_internal_t1` tokens which (after x-expansion) encode the key–value pair. This variable can safely be used in `__prop_split:NnTF`. If the `\langle key\rangle` was absent, append the new key–value to the list. Otherwise concatenate the extracts `##1` and `##3` with the new

`\prop_put:NnV`
`\prop_put:Nno`
`\prop_put:Nnx`
`\prop_put:Nvx`
`\prop_put:Nvv`
`\prop_put:Nvv`
`\prop_put:NnV`
`\prop_put:Noo`
`\prop_put:cnn`
`\prop_put:cNv`
`\prop_put:cno`
`\prop_put:cnx`
`\prop_put:cVn`
`\prop_put:cVV`
`\prop_put:con`
`\prop_put:coo`
`\prop_gput:Nnn`

key–value pair `\l__prop_internal_t1`. The updated entry is placed at the same spot as the original `<key>` in the property list, preserving the order of entries.

```

7430 \cs_new_protected:Npn \prop_put:Nnn { \__prop_put>NNnn \tl_set:Nx }
7431 \cs_new_protected:Npn \prop_gput:Nnn { \__prop_put>NNnn \tl_gset:Nx }
7432 \cs_new_protected:Npn \__prop_put:NNnn #1#2#3#4
7433 {
7434     \tl_set:Nn \l__prop_internal_t1
7435     {
7436         \exp_not:N \__prop_pair:wn \tl_to_str:n {#3}
7437         \s__prop { \exp_not:n {#4} }
7438     }
7439     \__prop_split:NnTF #2 {#3}
7440     { #1 #2 { \exp_not:n {##1} \l__prop_internal_t1 \exp_not:n {##3} } }
7441     { #1 #2 { \exp_not:o {#2} \l__prop_internal_t1 } }
7442 }
7443 \cs_generate_variant:Nn \prop_put:Nnn
7444     { NnV , Nno , Nnx , NV , NVV , No , Noo }
7445 \cs_generate_variant:Nn \prop_put:Nnn
7446     { c , cnV , cno , cnx , cV , cVV , co , coo }
7447 \cs_generate_variant:Nn \prop_gput:Nnn
7448     { NnV , Nno , Nnx , NV , NVV , No , Noo }
7449 \cs_generate_variant:Nn \prop_gput:Nnn
7450     { c , cnV , cno , cnx , cV , cVV , co , coo }
```

(End definition for `\prop_put:Nnn`, `\prop_gput:Nnn`, and `__prop_put:NNnn`. These functions are documented on page 123.)

\prop_put_if_new:Nnn 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`.

```

451 \cs_new_protected:Npn \prop_put_if_new:Nnn
452     { \__prop_put_if_new>NNnn \tl_set:Nx }
453 \cs_new_protected:Npn \prop_gput_if_new:Nnn
454     { \__prop_put_if_new>NNnn \tl_gset:Nx }
455 \cs_new_protected:Npn \__prop_put_if_new:NNnn #1#2#3#4
456 {
457     \tl_set:Nn \l__prop_internal_t1
458     {
459         \exp_not:N \__prop_pair:wn \tl_to_str:n {#3}
460         \s__prop \exp_not:n { {#4} }
461     }
462     \__prop_split:NnTF #2 {#3}
463     { }
464     { #1 #2 { \exp_not:o {#2} \l__prop_internal_t1 } }
465 }
466 \cs_generate_variant:Nn \prop_put_if_new:Nnn { c }
467 \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 123.)

15.3 Property list conditionals

\prop_if_exist_p:N Copies of the `cs` functions defined in l3basics.
\prop_if_exist_p:c
\prop_if_exist:NTF
\prop_if_exist:cTF

```

7468 \prg_new_eq_conditional:NNN \prop_if_exist:N \cs_if_exist:N
7469   { TF , T , F , p }
7470 \prg_new_eq_conditional:NNN \prop_if_exist:c \cs_if_exist:c
7471   { TF , T , F , p }

```

(End definition for `\prop_if_exist:NTF`. This function is documented on page 124.)

`\prop_if_empty_p:N` Same test as for token lists.

```

7472 \prg_new_conditional:Npnn \prop_if_empty:N #1 { p , T , F , TF }
7473   {
7474     \tl_if_eq:NNTF #1 \c_empty_prop
7475       \prg_return_true: \prg_return_false:
7476     }
7477 \cs_generate_variant:Nn \prop_if_empty_p:N { c }
7478 \cs_generate_variant:Nn \prop_if_empty:NT { c }
7479 \cs_generate_variant:Nn \prop_if_empty:NF { c }
7480 \cs_generate_variant:Nn \prop_if_empty:NTF { c }

```

(End definition for `\prop_if_empty:NTF`. This function is documented on page 124.)

`\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

```

\prg_new_protected_conditional:Npnn \prop_if_in:Nn #1 #2
  {
    \__prop_split:NnTF #1 {#2}
    { \prg_return_true: }
    { \prg_return_false: }
  }

```

but `__prop_split:NnTF` is non-expandable.

Instead, the key is compared to each key in turn using `\str_if_eq_x:nn`, which is expandable. To terminate the mapping, we append to the property list the key that is searched for. This second `\tl_to_str:n` is not expanded at the start, but only when included in the `\str_if_eq_x:nn`. It cannot make the breaking mechanism choke, because the arbitrary token list material is enclosed in braces. The second argument of `__prop_if_in:nwwn` is most often empty. When the `\langle key` is found in the list, `__prop_if_in:N` receives `__prop_pair:wn`, and if it is found as the extra item, the function receives `\q_recursion_tail`, easily recognizable.

Here, `\prop_map_function>NN` is not sufficient for the mapping, since it can only map a single token, and cannot carry the key that is searched for.

```

7481 \prg_new_conditional:Npnn \prop_if_in:Nn #1#2 { p , T , F , TF }
7482   {
7483     \exp_last_unbraced:Noo \__prop_if_in:nwwn { \tl_to_str:n {#2} } #1
7484     \__prop_pair:wn \tl_to_str:n {#2} \s__prop { }
7485     \q_recursion_tail
7486     \__prg_break_point:
7487   }
7488 \cs_new:Npn \__prop_if_in:nwwn #1#2 \__prop_pair:wn #3 \s__prop #4
7489   {
7490     \str_if_eq_x:nnTF {#1} {#3}
7491     { \__prop_if_in:N }
7492     { \__prop_if_in:nwwn {#1} }

```

```

7493     }
7494   \cs_new:Npn \__prop_if_in:N #1
7495   {
7496     \if_meaning:w \q_recursion_tail #1
7497       \prg_return_false:
7498     \else:
7499       \prg_return_true:
7500     \fi:
7501     \__prg_break:
7502   }
7503 \cs_generate_variant:Nn \prop_if_in_p:Nn { NV , No }
7504 \cs_generate_variant:Nn \prop_if_in_p:Nn { c , cV , co }
7505 \cs_generate_variant:Nn \prop_if_in:NnT { NV , No }
7506 \cs_generate_variant:Nn \prop_if_in:NnT { c , cV , co }
7507 \cs_generate_variant:Nn \prop_if_in:NnF { NV , No }
7508 \cs_generate_variant:Nn \prop_if_in:NnF { c , cV , co }
7509 \cs_generate_variant:Nn \prop_if_in:NnTF { NV , No }
7510 \cs_generate_variant:Nn \prop_if_in:NnTF { c , cV , co }

```

(End definition for `\prop_if_in:NnTF`, `__prop_if_in:nwn`, and `__prop_if_in:N`. These functions are documented on page 124.)

15.4 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.

```

7511 \prg_new_protected_conditional:Npnn \prop_get:NnN #1#2#3 { T , F , TF }
7512   {
7513     \__prop_split:NnTF #1 {#2}
7514     {
7515       \tl_set:Nn #3 {##2}
7516       \prg_return_true:
7517     }
7518     { \prg_return_false: }
7519   }
7520 \cs_generate_variant:Nn \prop_get:NnNT { NV , No }
7521 \cs_generate_variant:Nn \prop_get:NnNF { NV , No }
7522 \cs_generate_variant:Nn \prop_get:NnNTF { NV , No }
7523 \cs_generate_variant:Nn \prop_get:NnNT { c , cV , co }
7524 \cs_generate_variant:Nn \prop_get:NnNF { c , cV , co }
7525 \cs_generate_variant:Nn \prop_get:NnNTF { c , cV , co }

```

(End definition for `\prop_get:NnNTF`. This function is documented on page 125.)

15.5 Mapping to property lists

The fastest way to do a recursion here is to use an `\if_meaning:w` test: the keys are strings, and thus cannot match the marker `\q_recursion_tail`. A special case to note is when the key #3 is empty: then `\q_recursion_tail` is compared to `\exp_after:wN`, also different. Note that #2 is empty, except at the first iteration, where it is `\s__prop`.

```

7526 \cs_new:Npn \prop_map_function>NN #1#2
7527   {
7528     \exp_last_unbraced:NNo \__prop_map_function:Nwwn #2 #1

```

```

7529      \_prop_pair:wn \q_recursion_tail \s__prop { }
7530      \_prg_break_point:Nn \prop_map_break: { }
7531  }
7532 \cs_new:Npn \_prop_map_function:Nwwn #1#2 \_prop_pair:wn #3 \s__prop #4
7533  {
7534      \if_meaning:w \q_recursion_tail #3
7535      \exp_after:wN \prop_map_break:
7536      \fi:
7537      #1 {#3} {#4}
7538      \_prop_map_function:Nwwn #1
7539  }
7540 \cs_generate_variant:Nn \prop_map_function:NN { Nc }
7541 \cs_generate_variant:Nn \prop_map_function:NN { c , cc }

```

(End definition for `\prop_map_function:NN` and `_prop_map_function:Nwwn`. These functions are documented on page 125.)

\prop_map_inline:Nn
\prop_map_inline:cn

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.

```

7542 \cs_new_protected:Npn \prop_map_inline:Nn #1#2
7543  {
7544      \cs_gset_eq:cN
7545      { \_prg_map_ \int_use:N \g__prg_map_int :wn } \_prop_pair:wn
7546      \int_gincr:N \g__prg_map_int
7547      \cs_gset_protected:Npn \_prop_pair:wn ##1 \s__prop ##2 {#2}
7548      #1
7549      \_prg_break_point:Nn \prop_map_break:
7550      {
7551          \int_gdecr:N \g__prg_map_int
7552          \cs_gset_eq:Nc \_prop_pair:wn
7553          { \_prg_map_ \int_use:N \g__prg_map_int :wn }
7554      }
7555  }
7556 \cs_generate_variant:Nn \prop_map_inline:Nn { c }

```

(End definition for `\prop_map_inline:Nn`. This function is documented on page 125.)

\prop_map_break: The break statements are based on the general `_prg_map_break:Nn`.

\prop_map_break:n

```

7557 \cs_new:Npn \prop_map_break:
7558  { \_prg_map_break:Nn \prop_map_break: { } }
7559 \cs_new:Npn \prop_map_break:n
7560  { \_prg_map_break:Nn \prop_map_break: }

```

(End definition for `\prop_map_break:` and `\prop_map_break:n`. These functions are documented on page 126.)

15.6 Viewing property lists

\prop_show:N Apply the general `_msg_show_variable:NNNnn`. Contrarily to sequences and comma lists, we use `_msg_show_item:nn` to format both the key and the value for each pair.

```

7561 \cs_new_protected:Npn \prop_show:N #1
7562 {
7563     \__msg_show_variable>NNNnn #1
7564     \prop_if_exist:NTF \prop_if_empty:NTF { prop }
7565     { \prop_map_function:NN #1 \__msg_show_item:nn }
7566 }
7567 \cs_generate_variant:Nn \prop_show:N { c }

```

(End definition for `\prop_show:N`. This function is documented on page 126.)

`\prop_log:N` Redirect output of `\prop_show:N` to the log.

```

\prop_log:c
7568 \cs_new_protected:Npn \prop_log:N
7569 { \__msg_log_next: \prop_show:N }
7570 \cs_generate_variant:Nn \prop_log:N { c }

```

(End definition for `\prop_log:N`. This function is documented on page 126.)

7571 ⟨/initex | package⟩

16 I3msg implementation

7572 ⟨*initex | package⟩

7573 ⟨@=msg⟩

`\l__msg_internal_tl` A general scratch for the module.

```

7574 \tl_new:N \l__msg_internal_tl

```

(End definition for `\l__msg_internal_tl`.)

`\l__msg_line_context_bool` Controls whether the line context is shown as part of the decoration of all (non-expandable) messages.

```

7575 \bool_new:N \l__msg_line_context_bool

```

(End definition for `\l__msg_line_context_bool`.)

16.1 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
7576 \tl_const:Nn \c__msg_text_prefix_tl { msg-text->~ }
7577 \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
7578 \prg_new_conditional:Npnn \msg_if_exist:nn #1#2 { p , T , F , TF }
7579 {
7580     \cs_if_exist:cTF { \c__msg_text_prefix_tl #1 / #2 }
7581     { \prg_return_true: } { \prg_return_false: }
7582 }

```

(End definition for `\msg_if_exist:nnTF`. This function is documented on page 129.)

__chk_if_free_msg:nn This auxiliary is similar to __chk_if_free_cs:N, and is used when defining messages with \msg_new:nnnn. It could be inlined in \msg_new:nnnn, but the experimental l3trace module needs to disable this check when reloading a package with the extra tracing information.

```

7583 \cs_new_protected:Npn \_\_chk_if_free_msg:nn #1#2
7584 {
7585     \msg_if_exist:nnT {#1} {#2}
7586     {
7587         \_\_msg_kernel_error:nnxx { kernel } { message-already-defined }
7588         {#1} {#2}
7589     }
7590 }
7591 (*package)
7592 \if_bool:N \l@expl@log@functions@bool
7593     \cs_gset_protected:Npn \_\_chk_if_free_msg:nn #1#2
7594     {
7595         \msg_if_exist:nnT {#1} {#2}
7596         {
7597             \_\_msg_kernel_error:nnxx { kernel } { message-already-defined }
7598             {#1} {#2}
7599         }
7600         \_\_chk_log:x { Defining~message~#1 / #2 ~\msg_line_context: }
7601     }
7602 \fi:
7603 
```

(End definition for __chk_if_free_msg:nn.)

\msg_new:nnnn Setting a message simply means saving the appropriate text into two functions. A sanity check first.

\msg_gset:nnnn

```

7604 \cs_new_protected:Npn \msg_new:nnnn #1#2
7605 {
7606     \_\_chk_if_free_msg:nn {#1} {#2}
7607     \msg_gset:nnnn {#1} {#2}
7608 }
7609 \cs_new_protected:Npn \msg_new:nnn #1#2#3
7610 { \msg_new:nnnn {#1} {#2} {#3} { } }
7611 \cs_new_protected:Npn \msg_set:nnnn #1#2#3#4
7612 {
7613     \cs_set:cpn { \c_\msg_text_prefix_tl #1 / #2 }
7614     ##1##2##3##4 {#3}
7615     \cs_set:cpn { \c_\msg_more_text_prefix_tl #1 / #2 }
7616     ##1##2##3##4 {#4}
7617 }
7618 \cs_new_protected:Npn \msg_set:nnn #1#2#3
7619 { \msg_set:nnnn {#1} {#2} {#3} { } }
7620 \cs_new_protected:Npn \msg_gset:nnnn #1#2#3#4
7621 {
7622     \cs_gset:cpn { \c_\msg_text_prefix_tl #1 / #2 }
7623     ##1##2##3##4 {#3}
7624     \cs_gset:cpn { \c_\msg_more_text_prefix_tl #1 / #2 }
7625     ##1##2##3##4 {#4}
7626 }
7627 \cs_new_protected:Npn \msg_gset:nnn #1#2#3

```

```
7628 { \msg_gset:nnn {#1} {#2} {#3} { } }
```

(End definition for `\msg_new:nnn` and others. These functions are documented on page 128.)

16.2 Messages: support functions and text

Simple pieces of text for messages.

```
\c__msg_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

7629 \tl_const:Nn \c__msg_error_text_tl
7630 {
7631   This-is-a-coding-error.
7632   \\ \\
7633 }
7634 \tl_const:Nn \c__msg_continue_text_tl
7635 { Type-<return>-to-continue }
7636 \tl_const:Nn \c__msg_critical_text_tl
7637 { Reading-the-current-file-'g_file_current_name_tl'-will-stop. }
7638 \tl_const:Nn \c__msg_fatal_text_tl
7639 { This-is-a-fatal-error:-LaTeX-will-abort. }
7640 \tl_const:Nn \c__msg_help_text_tl
7641 { For-immediate-help-type-H-<return> }
7642 \tl_const:Nn \c__msg_no_info_text_tl
7643 {
7644   LaTeX-does-not-know-anything-more-about-this-error,-sorry.
7645   \c__msg_return_text_tl
7646 }
7647 \tl_const:Nn \c__msg_on_line_text_tl { on-line }
7648 \tl_const:Nn \c__msg_return_text_tl
7649 {
7650   \\ \\
7651   Try-typing-<return>-to-proceed.
7652   \\
7653   If-that-doesn't-work,-type-X-<return>-to-quit.
7654 }
7655 \tl_const:Nn \c__msg_trouble_text_tl
7656 {
7657   \\ \\
7658   More-errors-will-almost-certainly-follow: \\
7659   the-LaTeX-run-should-be-aborted.
7660 }
```

(End definition for `\c__msg_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.

```
7661 \cs_new:Npn \msg_line_number: { \int_use:N \tex_inlineno:D }
7662 \cs_gset:Npn \msg_line_context:
7663 {
7664   \c__msg_on_line_text_tl
7665   \c_space_tl
7666   \msg_line_number:
7667 }
```

(End definition for `\msg_line_number:` and `\msg_line_context:`. These functions are documented on page 129.)

16.3 Showing messages: low level mechanism

`\msg_interrupt:nnn`

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 TeX's `\errhelp` register before issuing an `\errmessage`.

```

7668 \cs_new_protected:Npn \msg_interrupt:nnn #1#2#3
7669   {
7670     \tl_if_empty:nTF {#3}
7671       {
7672         \__msg_interrupt_wrap:nn { \\ \c__msg_no_info_text_tl }
7673           {#1 \\ \\ \\ #2 \\ \\ \\ \c__msg_continue_text_tl }
7674       }
7675       {
7676         \__msg_interrupt_wrap:nn { \\ #3 }
7677           {#1 \\ \\ \\ #2 \\ \\ \\ \c__msg_help_text_tl }
7678       }
7679   }

```

(End definition for `\msg_interrupt:nnn`. This function is documented on page 133.)

`__msg_interrupt_wrap:nn`

First setup TeX'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.

```

7680 \cs_new_protected:Npn \__msg_interrupt_wrap:nn #1#2
7681   {
7682     \iow_wrap:nnnN {#1} { | ~ } { } \__msg_interrupt_more_text:n
7683     \iow_wrap:nnnN {#2} { ! ~ } { } \__msg_interrupt_text:n
7684   }
7685 \cs_new_protected:Npn \__msg_interrupt_more_text:n #1
7686   {
7687     \exp_args:Nx \tex_errhelp:D
7688     {
7689       | , , , , , , , , , , , , , , , , , , , , ,
7690       #1 \iow_newline:
7691       |
7692     }
7693   }

```

(End definition for `__msg_interrupt_wrap:nn` and `__msg_interrupt_more_text:n`.)

`__msg_interrupt_text: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”: TeX'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 {<dots>}` which fills the output with dots. 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 will be inserted after the message is entirely cleaned up.

The `__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.

```

7694 \group_begin:
7695   \char_set_lccode:nn {'\{} {'\}
7696   \char_set_lccode:nn {'\}} {'\}
7697   \char_set_lccode:nn {'\&} {'\!}
7698   \char_set_catcode_active:N \&
7699 \tex_lowercase:D
7700 {
7701   \group_end:
7702   \cs_new_protected:Npn \__msg_interrupt_text:n #1
7703   {
7704     \iow_term:x
7705     {
7706       \iow_newline:
7707       !!!!!!!!!!!!!!!!!!!!!!!!
7708       \iow_newline:
7709       !
7710     }
7711   \__iow_with:Nnn \tex_newlinechar:D { '\^J }
7712   {
7713     \__iow_with:Nnn \tex_errorcontextlines:D { -1 }
7714   {
7715     \group_begin:
7716     \cs_set_protected:Npn &
7717     {
7718       \tex_errmessage:D
7719       {
7720         #1
7721         \use_none:n
7722         { ..... }
7723       }
7724     }
7725     \exp_after:wN
7726     \group_end:
7727     &
7728   }
7729 }
7730 }
7731 }
```

(End definition for `__msg_interrupt_text:n`.)

\msg_log:n Printing to the log or terminal without a stop is rather easier. A bit of simple visual work sets things off nicely.

```

7732 \cs_new_protected:Npn \msg_log:n #1
7733 {
7734   \iow_log:n { ..... }
7735   \iow_wrap:nnnN { . ~ #1} { . ~ } { } \iow_log:n
7736   \iow_log:n { ..... }
```

```

7737   }
7738 \cs_new_protected:Npn \msg_term:n #1
7739   {
7740     \iow_term:n { *****
7741     \iow_wrap:nnnN { * ~ #1} { * ~ } { } \iow_term:n
7742     \iow_term:n { ***** }
7743   }

```

(End definition for `\msg_log:n` and `\msg_term:n`. These functions are documented on page 134.)

16.4 Displaying messages

\LaTeX is handling error messages and so the \TeX ones are disabled. This is already done by the $\text{\TeX}_2\epsilon$ kernel, so to avoid messing up any deliberate change by a user this is only set in format mode.

```

7744 <*initex>
7745 \int_gset:Nn \tex_errorcontextlines:D { -1 }
7746 </initex>

```

`\msg_fatal_text:n`

`\msg_critical_text:n`

`\msg_error_text:n`

`\msg_warning_text:n`

`\msg_info_text:n`

A function for issuing messages: both the text and order could in principle vary.

```

7747 \cs_new:Npn \msg_fatal_text:n #1
7748   {
7749     Fatal~#1~error
7750     \bool_if:NT \l__msg_line_context_bool { ~ \msg_line_context: }
7751   }
7752 \cs_new:Npn \msg_critical_text:n #1
7753   {
7754     Critical~#1~error
7755     \bool_if:NT \l__msg_line_context_bool { ~ \msg_line_context: }
7756   }
7757 \cs_new:Npn \msg_error_text:n #1
7758   {
7759     #1~error
7760     \bool_if:NT \l__msg_line_context_bool { ~ \msg_line_context: }
7761   }
7762 \cs_new:Npn \msg_warning_text:n #1
7763   {
7764     #1~warning
7765     \bool_if:NT \l__msg_line_context_bool { ~ \msg_line_context: }
7766   }
7767 \cs_new:Npn \msg_info_text:n #1
7768   {
7769     #1~info
7770     \bool_if:NT \l__msg_line_context_bool { ~ \msg_line_context: }
7771   }

```

(End definition for `\msg_fatal_text:n` and others. These functions are documented on page 129.)

`\msg_see_documentation_text:n`

Contextual footer information. The \LaTeX module only comprises \LaTeX_3 code, so we refer to the \LaTeX_3 documentation rather than simply “ \LaTeX ”.

```

7772 \cs_new:Npn \msg_see_documentation_text:n #1
7773   {
7774     \\ \\ See-the-
7775     \str_if_eq:nnTF {#1} { \LaTeX } { \LaTeX_3 } {#1} ~

```

```

7776     documentation~for~further~information.
7777 }

```

(End definition for `\msg_see_documentation_text:n`. This function is documented on page 130.)

```

\__msg_class_new:nn
7778 \group_begin:
7779   \cs_set_protected:Npn \__msg_class_new:nn #1#2
7780   {
7781     \prop_new:c { l__msg_redirect_ #1 _prop }
7782     \cs_new_protected:cpx { __msg_ #1 _code:nnnnnn }
7783     ##1##2##3##4##5##6 {#2}
7784     \cs_new_protected:cpx { msg_ #1 :nnnnnn } ##1##2##3##4##5##6
7785     {
7786       \use:x
7787       {
7788         \exp_not:n { \__msg_use:nnnnnnn {#1} {##1} {##2} }
7789         { \tl_to_str:n {##3} } { \tl_to_str:n {##4} }
7790         { \tl_to_str:n {##5} } { \tl_to_str:n {##6} }
7791       }
7792     }
7793     \cs_new_protected:cpx { msg_ #1 :nnnn } ##1##2##3##4##5
7794     { \exp_not:c { msg_ #1 :nnnnnn } {##1} {##2} {##3} {##4} {##5} { } }
7795     \cs_new_protected:cpx { msg_ #1 :nnnn } ##1##2##3##4
7796     { \exp_not:c { msg_ #1 :nnnnnn } {##1} {##2} {##3} {##4} { } { } }
7797     \cs_new_protected:cpx { msg_ #1 :nnn } ##1##2##3
7798     { \exp_not:c { msg_ #1 :nnnnnn } {##1} {##2} {##3} { } { } { } }
7799     \cs_new_protected:cpx { msg_ #1 :nn } ##1##2
7800     { \exp_not:c { msg_ #1 :nnnnnn } {##1} {##2} { } { } { } { } }
7801     \cs_new_protected:cpx { msg_ #1 :nnxxxx } ##1##2##3##4##5##6
7802     {
7803       \use:x
7804       {
7805         \exp_not:N \exp_not:n
7806         { \exp_not:c { msg_ #1 :nnnnnn } {##1} {##2} }
7807         {##3} {##4} {##5} {##6}
7808       }
7809     }
7810     \cs_new_protected:cpx { msg_ #1 :nnxxx } ##1##2##3##4##5
7811     { \exp_not:c { msg_ #1 :nnxxxx } {##1} {##2} {##3} {##4} {##5} { } }
7812     \cs_new_protected:cpx { msg_ #1 :nnxx } ##1##2##3##4
7813     { \exp_not:c { msg_ #1 :nnxxxx } {##1} {##2} {##3} {##4} { } { } }
7814     \cs_new_protected:cpx { msg_ #1 :nnx } ##1##2##3
7815     { \exp_not:c { msg_ #1 :nnxxxx } {##1} {##2} {##3} { } { } { } }
7816   }

```

(End definition for `__msg_class_new:nn`.)

`\msg_fatal:nnnnnn` For fatal errors, after the error message TeX bails out.

```

\msg_fatal:nnxxxx
\msg_fatal:nnnnnn
\msg_fatal:nnxxx
\msg_fatal:nnnn
\msg_fatal:nnxx
\msg_fatal:nnn
\msg_fatal:nnx
\msg_fatal:nn
7817   \__msg_class_new:nn { fatal }
7818   {
7819     \msg_interrupt:nnn
7820     { \msg_fatal_text:n {#1} : ~ "#2" }
7821     {
7822       \use:c { \c__msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6}

```

```

7823     \msg_see_documentation_text:n {#1}
7824   }
7825   { \c__msg_fatal_text_tl }
7826 \tex_end:D
7827 }
```

(End definition for `\msg_fatal:nnnnnn` and others. These functions are documented on page 130.)

`\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
\msg_critical:nn
```

```

7828   \__msg_class_new:nn { critical }
7829   {
7830     \msg_interrupt:nnn
7831     { \msg_critical_text:n {#1} : ~ "#2" }
7832     {
7833       \use:c { \c__msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6}
7834       \msg_see_documentation_text:n {#1}
7835     }
7836     { \c__msg_critical_text_tl }
7837   \tex_endininput:D
7838 }
```

(End definition for `\msg_critical:nnnnnn` and others. These functions are documented on page 130.)

`\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`

`__msg_error:cnnnnn`

`__msg_no_more_text:nnnn`

For an error, the interrupt routine is called. We check if there is a “more text” by comparing that control sequence with a permanently empty text.

```

7839   \__msg_class_new:nn { error }
7840   {
7841     \__msg_error:cnnnnn
7842     { \c__msg_more_text_prefix_tl #1 / #2 }
7843     {#3} {#4} {#5} {#6}
7844     {
7845       \msg_interrupt:nnn
7846         { \msg_error_text:n {#1} : ~ "#2" }
7847         {
7848           \use:c { \c__msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6}
7849           \msg_see_documentation_text:n {#1}
7850         }
7851       }
7852     }
7853   \cs_new_protected:Npn \__msg_error:cnnnnn #1#2#3#4#5#6
7854   {
7855     \cs_if_eq:cNTF {#1} \__msg_no_more_text:nnnn
7856     { #6 { } }
7857     { #6 { \use:c {#1} {#2} {#3} {#4} {#5} } }
7858   }
7859   \cs_new:Npn \__msg_no_more_text:nnnn #1#2#3#4 { }
```

(End definition for `\msg_error:nnnnnn` and others. These functions are documented on page 131.)

`\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`

Warnings are printed to the terminal.

```

7860   \__msg_class_new:nn { warning }
7861   {
7862     \msg_term:n
7863     {
7864       \msg_warning_text:n {#1} : ~ "#2" \\ \\
```

```

7865           \use:c { \c__msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6}
7866       }
7867   }

```

(End definition for `\msg_warning:nnnnnn` and others. These functions are documented on page 131.)

`\msg_info:nnnnnn` Information only goes into the log.

```

\msg_info:nnxxxx 7868  \__msg_class_new:nn { info }
\msg_info:nnnnn 7869  {
\msg_info:nnxxx 7870  \msg_log:n
\msg_info:nnnn 7871  {
\msg_info:nnxx 7872  \msg_info_text:n {#1} : ~ "#2" \\ \\
\msg_info:nnn 7873  \use:c { \c__msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6}
\msg_info:nnx 7874  }
\msg_info:nn 7875  }

```

(End definition for `\msg_info:nnnnnn` and others. These functions are documented on page 131.)

`\msg_log:nnnnnn` “Log” data is very similar to information, but with no extras added.

```

\msg_log:nnxxxx 7876  \__msg_class_new:nn { log }
\msg_log:nnnnn 7877  {
\msg_log:nnxxx 7878  \iow_wrap:nnnN
\msg_log:nnnn 7879  { \use:c { \c__msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6} }
\msg_log:nnxx 7880  { } { } \iow_log:n
\msg_log:nnn 7881  }

```

(End definition for `\msg_log:nnnnnn` and others. These functions are documented on page 131.)

`\msg_none:nnnnnn` The none message type is needed so that input can be gobbled.

```

\msg_none:nnxxxx 7882  \__msg_class_new:nn { none } { }
\msg_none:nnnnn
\msg_none:nnxxx
\msg_none:nnnn
\msg_none:nnxx
\msg_none:nnn
\msg_log:nnx
\msg_log:nn

```

(End definition for `\msg_none:nnnnnn` and others. These functions are documented on page 132.)

End the group to eliminate `__msg_class_new:nn`.

```

7883 \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.

```

7884 \cs_new:Npn \__msg_class_chk_exist:nT #1
7885  {
7886      \cs_if_free:cTF { __msg_ #1 _code:nnnnnn }
7887      { \__msg_kernel_error:nx { kernel } { message-class-unknown } {#1} }
7888  }

```

(End definition for `__msg_class_chk_exist:nT`.)

`\l__msg_class_tl` Support variables needed for the redirection system.

```

\l__msg_current_class_tl 7889 \tl_new:N \l__msg_class_tl
7890 \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

```

7891 \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 with items {/module/submodule}, {/module}, and {}.

7892 \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.

7893 \seq_new:N \l__msg_class_loop_seq

(End definition for \l__msg_class_loop_seq.)

__msg_use:nnnnnn
__msg_use_redirect_name:n
__msg_use_hierarchy:nwwN
__msg_use_redirect_module:n
__msg_use_code:
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_t1 is when __msg_use_code: is called.

7894 \cs_new_protected:Npn __msg_use:nnnnnnn #1#2#3#4#5#6#7
7895 {
7896 \msg_if_exist:nnTF {#2} {#3}
7897 {
7898 __msg_class_chk_exist:nT {#1}
7899 {
7900 \tl_set:Nn \l__msg_current_class_t1 {#1}
7901 \cs_set_protected:Npx __msg_use_code:
7902 {
7903 \exp_not:n
7904 {
7905 \use:c { __msg_ \l__msg_class_t1 _code:nnnnnn }
7906 {#2} {#3} {#4} {#5} {#6} {#7}
7907 }
7908 }
7909 __msg_use_redirect_name:n { #2 / #3 }
7910 }
7911 }
7912 { __msg_kernel_error:nxxx { kernel } { message-unknown } {#2} {#3} }
7913 }
7914 \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 module/submodule/message (with an arbitrary number of slashes), and store {/module/submodule}, {/module} and {} into \l__msg_hierarchy_seq. We will then map through this sequence, applying the most specific redirection.

7915 \cs_new_protected:Npn __msg_use_redirect_name:n #1
7916 {
7917 \prop_get:NnNTF \l__msg_redirect_prop { / #1 } \l__msg_class_t1
7918 { __msg_use_code: }
7919 {
7920 \seq_clear:N \l__msg_hierarchy_seq
7921 __msg_use_hierarchy:nwwN { }
7922 #1 \q_mark __msg_use_hierarchy:nwwN
7923 / \q_mark \use:none_delimit_by_q_stop:w

```

7924         \q_stop
7925     \__msg_use_redirect_module:n { }
7926   }
7927 }
7928 \cs_new_protected:Npn \__msg_use_hierarchy:nwwN #1#2 / #3 \q_mark #4
7929 {
7930     \seq_put_left:Nn \l__msg_hierarchy_seq {#1}
7931     #4 { #1 / #2 } #3 \q_mark #4
7932 }

```

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`.

```

7933 \cs_new_protected:Npn \__msg_use_redirect_module:n #1
7934 {
7935     \seq_map_inline:Nn \l__msg_hierarchy_seq
7936     {
7937         \prop_get:cnNTF { l__msg_redirect_ \l__msg_current_class_tl _prop }
7938         {##1} \l__msg_class_tl
7939     {
7940         \seq_map_break:n
7941         {
7942             \tl_if_eq:NNTF \l__msg_current_class_tl \l__msg_class_tl
7943             { \__msg_use_code: }
7944             {
7945                 \tl_set_eq:NN \l__msg_current_class_tl \l__msg_class_tl
7946                 \__msg_use_redirect_module:n {##1}
7947             }
7948         }
7949     }
7950     {
7951         \str_if_eq:nnT {##1} {#1}
7952         {
7953             \tl_set_eq:NN \l__msg_class_tl \l__msg_current_class_tl
7954             \seq_map_break:n { \__msg_use_code: }
7955         }
7956     }
7957 }
7958 }

```

(End definition for `__msg_use:nnnnnnn` and others.)

`\msg_redirect_name:nnn` Named message will always use the given class even if that class is redirected further. An empty target class cancels any existing redirection for that message.

```

7959 \cs_new_protected:Npn \msg_redirect_name:nnn #1#2#3
7960 {
7961     \tl_if_empty:nTF {#3}

```

```

7962 { \prop_remove:Nn \l__msg_redirect_prop { / #1 / #2 } }
7963 {
7964     \__msg_class_chk_exist:nT {#3}
7965     { \prop_put:Nnn \l__msg_redirect_prop { / #1 / #2 } {#3} }
7966 }
7967 }

```

(End definition for `\msg_redirect_name:nnn`. This function is documented on page 133.)

`\msg_redirect_class:nn` 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`.

```

7968 \cs_new_protected:Npn \msg_redirect_class:nn
7969     { \__msg_redirect:nnn { } }
7970 \cs_new_protected:Npn \msg_redirect_module:nnn #1
7971     { \__msg_redirect:nnn { / #1 } }
7972 \cs_new_protected:Npn \__msg_redirect:nnn #1#2#3
7973     {
7974         \__msg_class_chk_exist:nT {#2}
7975     {
7976         \tl_if_empty:nTF {#3}
7977             { \prop_remove:cn { l__msg_redirect_ #2 _prop } {#1} }
7978         {
7979             \__msg_class_chk_exist:nT {#3}
7980             {
7981                 \prop_put:cnn { l__msg_redirect_ #2 _prop } {#1} {#3}
7982                 \tl_set:Nn \l__msg_current_class_tl {#2}
7983                 \seq_clear:N \l__msg_class_loop_seq
7984                 \__msg_redirect_loop_chk:nnn {#2} {#3} {#1}
7985             }
7986         }
7987     }
7988 }

```

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.

```

7989 \cs_new_protected:Npn \__msg_redirect_loop_chk:nnn #1#2#3
7990     {
7991         \seq_put_right:Nn \l__msg_class_loop_seq {#1}
7992         \prop_get:cnNT { l__msg_redirect_ #1 _prop } {#3} \l__msg_class_tl
7993         {
7994             \str_if_eq_x:nnF { \l__msg_class_tl } {#1}
7995             {
7996                 \tl_if_eq:NNTF \l__msg_class_tl \l__msg_current_class_tl
7997             }

```

```

7998   \prop_put:cnn { l__msg_redirect_ #2 _prop } {#3} {#2}
7999   \_msg_kernel_warning:nxxxxx
8000     { kernel } { message-redirect-loop }
8001     { \seq_item:Nn \l__msg_class_loop_seq { 1 } }
8002     { \seq_item:Nn \l__msg_class_loop_seq { 2 } }
8003     {#3}
8004   {
8005     \seq_map_function:NN \l__msg_class_loop_seq
8006       \_msg_redirect_loop_list:n
8007       { \seq_item:Nn \l__msg_class_loop_seq { 1 } }
8008   }
8009   { \_msg_redirect_loop_chk:onn \l__msg_class_tl {#2} {#3} }
8010 }
8011 }
8012 }
8013 }
8014 \cs_generate_variant:Nn \_msg_redirect_loop_chk:nnn { o }
8015 \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 132.)

16.5 Kernel-specific functions

`_msg_kernel_new:nnnn`
`_msg_kernel_new:nnn`
`_msg_kernel_set:nnnn`
`_msg_kernel_set:nnn`

```

8016 \cs_new_protected:Npn \_msg_kernel_new:nnnn #1#2
8017   { \msg_new:nnnn { LaTeX } { #1 / #2 } }
8018 \cs_new_protected:Npn \_msg_kernel_new:nnn #1#2
8019   { \msg_new:nnn { LaTeX } { #1 / #2 } }
8020 \cs_new_protected:Npn \_msg_kernel_set:nnnn #1#2
8021   { \msg_set:nnnn { LaTeX } { #1 / #2 } }
8022 \cs_new_protected:Npn \_msg_kernel_set:nnn #1#2
8023   { \msg_set:nnn { LaTeX } { #1 / #2 } }
```

(End definition for `_msg_kernel_new:nnnn` and others.)

`_msg_kernel_class_new:nN`
`_msg_kernel_class_new_aux:nN`

All the functions for kernel messages come in variants ranging from 0 to 4 arguments. Those with less than 4 arguments are defined in terms of the 4-argument variant, in a way very similar to `_msg_class_new:nn`. This auxiliary is destroyed at the end of the group.

```

8024 \group_begin:
8025   \cs_set_protected:Npn \_msg_kernel_class_new:nN #1
8026     { \_msg_kernel_class_new_aux:nN { kernel_ #1 } }
8027   \cs_set_protected:Npn \_msg_kernel_class_new:nN #1#2
8028   {
8029     \cs_new_protected:cpn { __msg_ #1 :nnnnnn } ##1##2##3##4##5##6
8030     {
8031       \use:x
8032       {
8033         \exp_not:n { #2 { LaTeX } { ##1 / ##2 } }
8034         { \tl_to_str:n {##3} } { \tl_to_str:n {##4} }
8035         { \tl_to_str:n {##5} } { \tl_to_str:n {##6} }
8036     }
```

```

8037     }
8038 \cs_new_protected:cp { __msg_ #1 :nnnnn } ##1##2##3##4##5
8039     { \exp_not:c { __msg_ #1 :nnnnnn } {##1} {##2} {##3} {##4} {##5} { } }
8040 \cs_new_protected:cp { __msg_ #1 :nnnn } ##1##2##3##4
8041     { \exp_not:c { __msg_ #1 :nnnnnn } {##1} {##2} {##3} {##4} { } { } }
8042 \cs_new_protected:cp { __msg_ #1 :nnn } ##1##2##3##4
8043     { \exp_not:c { __msg_ #1 :nnnnnn } {##1} {##2} {##3} { } { } { } }
8044 \cs_new_protected:cp { __msg_ #1 :nn } ##1##2
8045     { \exp_not:c { __msg_ #1 :nnnnnn } {##1} {##2} { } { } { } { } }
8046 \cs_new_protected:cp { __msg_ #1 :nnxxxx } ##1##2##3##4##5##6
8047     {
8048     \use:x
8049     {
8050         \exp_not:N \exp_not:n
8051             { \exp_not:c { __msg_ #1 :nnnnnn } {##1} {##2} }
8052             {##3} {##4} {##5} {##6}
8053     }
8054     }
8055 \cs_new_protected:cp { __msg_ #1 :nnxxx } ##1##2##3##4##5
8056     { \exp_not:c { __msg_ #1 :nnxxxx } {##1} {##2} {##3} {##4} {##5} { } }
8057 \cs_new_protected:cp { __msg_ #1 :nnxx } ##1##2##3##4
8058     { \exp_not:c { __msg_ #1 :nnxxxx } {##1} {##2} {##3} {##4} { } { } }
8059 \cs_new_protected:cp { __msg_ #1 :nnx } ##1##2##3
8060     { \exp_not:c { __msg_ #1 :nnxxxx } {##1} {##2} {##3} { } { } { } }
8061 }

```

(End definition for `__msg_kernel_class_new:nN` and `__msg_kernel_class_new_aux:nN`.)

`__msg_kernel_fatal:nnnnn`
`__msg_kernel_fatal:nnxxxx`
`__msg_kernel_fatal:nnnnn`
`__msg_kernel_fatal:nnxxxx`
`__msg_kernel_fatal:nnnn`
`__msg_kernel_fatal:nnxx`
`__msg_kernel_fatal:nnx`
`__msg_kernel_fatal:nn`
`__msg_kernel_error:nnnnn`
`__msg_kernel_error:nnnnnn`

Neither fatal kernel errors nor kernel errors can be redirected. We directly use the code for (non-kernel) fatal errors and errors, adding the “`LATEX`” module name. Three functions are already defined by `l3basics`; we need to undefine them to avoid errors.

```

8062 \__msg_kernel_class_new:nN { fatal } \__msg_fatal_code:nnnnnn
8063 \cs_undefine:N \__msg_kernel_error:nnxx
8064 \cs_undefine:N \__msg_kernel_error:nnx
8065 \cs_undefine:N \__msg_kernel_error:nn
8066 \__msg_kernel_class_new:nN { error } \__msg_error_code:nnnnnn

```

(End definition for `__msg_kernel_fatal:nnnnnn` and others.)

`__msg_kernel_warning:nnnnn`
`__msg_kernel_warning:nnnnnn`
`__msg_kernel_warning:nnnn`
`__msg_kernel_warning:nnnnnn`
`__msg_kernel_warning:nnnn`
`__msg_kernel_warning:nnxx`
`__msg_kernel_warning:nnx`
`__msg_kernel_warning:nn`
`__msg_kernel_info:nnnnn`
`__msg_kernel_info:nnnnnn`
`__msg_kernel_info:nnxxxx`
`__msg_kernel_info:nnnn`
`__msg_kernel_info:nnxx`
`__msg_kernel_info:nnx`
`__msg_kernel_info:nn`
`__msg_kernel_info:nnnn`

Kernel messages which can be redirected simply use the machinery for normal messages, with the module name “`LATEX`”.

```

8067 \__msg_kernel_class_new:nN { warning } \msg_warning:nnxxxx
8068 \__msg_kernel_class_new:nN { info } \msg_info:nnxxxx

```

(End definition for `__msg_kernel_warning:nnnnnn` and others.)

End the group to eliminate `__msg_kernel_class_new:nN`.

```
8069 \group_end:
```

Error messages needed to actually implement the message system itself.

```

8070 \__msg_kernel_new:nnnn { kernel } { message-already-defined }
8071     { Message~'#2'~for~module~'#1'~already~defined. }
8072     {
8073         \c__msg_coding_error_text_tl
8074         LaTeX-was-asked-to-define-a-new-message-called-'#2'\\
8075         by-the-module-'#1':~this~message~already~exists.

```

```

8076      \c__msg_return_text_tl
8077  }
8078 \__msg_kernel_new:nnnn { kernel } { message-unknown }
8079  { Unknown~message~'#2'~for~module~'#1'. }
8080  {
8081      \c__msg_coding_error_text_tl
8082      LaTeX~was~asked~to~display~a~message~called~'#2'\\
8083      by~the~module~'#1':~this~message~does~not~exist.
8084      \c__msg_return_text_tl
8085  }
8086 \__msg_kernel_new:nnnn { kernel } { message-class-unknown }
8087  { Unknown~message~class~'#1'. }
8088  {
8089      LaTeX~has~been~asked~to~redirect~messages~to~a~class~'#1':\\
8090      this~was~never~defined.
8091      \c__msg_return_text_tl
8092  }
8093 \__msg_kernel_new:nnnn { kernel } { message-redirect-loop }
8094  {
8095      Message~redirection~loop~caused~by~ {#1} ~=>~ {#2}
8096      \tl_if_empty:nF {#3} { ~for~module~' \use_none:n #3 ' } .
8097  }
8098  {
8099      Adding~the~message~redirection~ {#1} ~=>~ {#2}
8100      \tl_if_empty:nF {#3} { ~for~the~module~' \use_none:n #3 ' } ~
8101      created~an~infinite~loop\\\\
8102      \iow_indent:n { #4 \\\\ }
8103  }

     Messages for earlier kernel modules.

8104 \__msg_kernel_new:nnnn { kernel } { bad-number-of-arguments }
8105  { Function~'#1'~cannot~be~defined~with~#2~arguments. }
8106  {
8107      \c__msg_coding_error_text_tl
8108      LaTeX~has~been~asked~to~define~a~function~'#1'~with~
8109      #2~arguments.~
8110      TeX~allows~between~0~and~9~arguments~for~a~single~function.
8111  }
8112 \__msg_kernel_new:nnn { kernel } { char-active }
8113  { Cannot~generate~active~chars. }
8114 \__msg_kernel_new:nnn { kernel } { char-invalid-catcode }
8115  { Invalid~catcode~for~char~generation. }
8116 \__msg_kernel_new:nnn { kernel } { char-null-space }
8117  { Cannot~generate~null~char~as~a~space. }
8118 \__msg_kernel_new:nnn { kernel } { char-out-of-range }
8119  { Charcode~requested~out~of~engine~range. }
8120 \__msg_kernel_new:nnn { kernel } { char-space }
8121  { Cannot~generate~space~chars. }
8122 \__msg_kernel_new:nnnn { kernel } { command-already-defined }
8123  { Control~sequence~'#1~already~defined. }
8124  {
8125      \c__msg_coding_error_text_tl
8126      LaTeX~has~been~asked~to~create~a~new~control~sequence~'#1'~
8127      but~this~name~has~already~been~used~elsewhere. \\ \\
8128      The~current~meaning~is:\\

```

```

8129      \\ #2
8130    }
8131 \__msg_kernel_new:nnnn { kernel } { command-not-defined }
8132   { Control-sequence~#1~undefined. }
8133   {
8134     \c__msg_error_text_tl
8135     LaTeX~has~been~asked~to~use~a~control~sequence~'#1':\\\
8136     this~has~not~been~defined~yet.
8137   }
8138 \__msg_kernel_new:nnn { kernel } { deprecated-command }
8139   {
8140     The~deprecated~command~'#2'~has~been~or~will~be~removed~on~#1.
8141     \tl_if_empty:nF {#3} { ~Use~instead~'#3'. }
8142   }
8143 \__msg_kernel_new:nnnn { kernel } { empty-search-pattern }
8144   { Empty~search~pattern. }
8145   {
8146     \c__msg_error_text_tl
8147     LaTeX~has~been~asked~to~replace~an~empty~pattern~by~'#1':~that~
8148     would~lead~to~an~infinite~loop!
8149   }
8150 \__msg_kernel_new:nnnn { kernel } { out-of-registers }
8151   { No~room~for~a~new~#1. }
8152   {
8153     TeX~only~supports~\int_use:N \c_max_register_int \\ %
8154     of~each~type.~All~the~#1~registers~have~been~used.~
8155     This~run~will~be~aborted~now.
8156   }
8157 \__msg_kernel_new:nnnn { kernel } { non-base-function }
8158   { Function~'#1'~is~not~a~base~function }
8159   {
8160     \c__msg_error_text_tl
8161     Functions~defined~through~\iow_char:N\cs_new:Nn~must~have~
8162     a~signature~consisting~of~only~normal~arguments~'N'~and~'n'.~
8163     To~define~variants~use~\iow_char:N\cs_generate_variant:Nn~
8164     and~to~define~other~functions~use~\iow_char:N\cs_new:Npn.
8165   }
8166 \__msg_kernel_new:nnnn { kernel } { missing-colon }
8167   { Function~'#1'~contains~no~':'. }
8168   {
8169     \c__msg_error_text_tl
8170     Code-level~functions~must~contain~':~to~separate~the~
8171     argument~specification~from~the~function~name.~This~is~
8172     needed~when~defining~conditionals~or~variants,~or~when~building~a~
8173     parameter~text~from~the~number~of~arguments~of~the~function.
8174   }
8175 \__msg_kernel_new:nnnn { kernel } { overflow }
8176   { Integers~larger~than~2^{30}-1~cannot~be~stored~in~arrays. }
8177   {
8178     An~attempt~was~made~to~store~#3~at~position~#2~in~the~array~'#1'.~
8179     The~largest~allowed~value~#4~will~be~used~instead.
8180   }
8181 \__msg_kernel_new:nnnn { kernel } { out-of-bounds }
8182   { Access~to~an~entry~beyond~an~array's~bounds. }

```

```

8183  {
8184      An~attempt~was~made~to~access~or~store~data~at~position~#2~of~the~
8185      array~'#1',~but~this~array~has~entries~at~positions~from~1~to~#3.
8186  }
8187 \__msg_kernel_new:nnnn { kernel } { protected-predicate }
8188 { Predicate~'#1'~must~be~expandable. }
8189 {
8190     \c__msg_error_text_tl
8191     LaTeX~has~been~asked~to~define~'#1'~as~a~protected~predicate.~
8192     Only~expandable~tests~can~have~a~predicate~version.
8193 }
8194 \__msg_kernel_new:nnnn { kernel } { conditional-form-unknown }
8195 { Conditional~form~'#1'~for~function~'#2'~unknown. }
8196 {
8197     \c__msg_error_text_tl
8198     LaTeX~has~been~asked~to~define~the~conditional~form~'#1'~of~
8199     the~function~'#2',~but~only~'TF',~'T',~'F',~and~'p'~forms~exist.
8200 }
8201 (*package)
8202 \bool_if:NT \l@expl@check@declarations@bool
8203 {
8204     \__msg_kernel_new:nnnn { check } { non-declared-variable }
8205     { The~variable~'#1~has~not~been~declared~\msg_line_context:. }
8206     {
8207         Checking~is~active,~and~you~have~tried~do~so~something~like: \\ \\
8208         \\ \tl_set:Nn ~ #1 ~ \{ ~ ... ~ \} \\ \\
8209         without~first~having: \\ \\
8210         \\ \tl_new:N ~ #1 \\ \\
8211         \\ \\
8212         LaTeX~will~create~the~variable~and~continue.
8213     }
8214 }
8215 
```

```

8216 \__msg_kernel_new:nnnn { kernel } { scanmark-already-defined }
8217 { Scan~mark~'#1~already~defined. }
8218 {
8219     \c__msg_error_text_tl
8220     LaTeX~has~been~asked~to~create~a~new~scan~mark~'#1'~
8221     but~this~name~has~already~been~used~for~a~scan~mark.
8222 }
8223 \__msg_kernel_new:nnnn { kernel } { variable-not-defined }
8224 { Variable~'#1~undefined. }
8225 {
8226     \c__msg_error_text_tl
8227     LaTeX~has~been~asked~to~show~a~variable~'#1,~but~this~has~not~
8228     been~defined~yet.
8229 }
8230 \__msg_kernel_new:nnnn { kernel } { variant-too-long }
8231 { Variant~form~'#1'~longer~than~base~signature~of~'#2'. }
8232 {
8233     \c__msg_error_text_tl
8234     LaTeX~has~been~asked~to~create~a~variant~of~the~function~'#2'~
8235     with~a~signature~starting~with~'#1',~but~that~is~longer~than~
8236     the~signature~(part~after~the~colon)~of~'#2'.

```

```

8237     }
8238 \__msg_kernel_new:nnn { kernel } { invalid-variant }
8239   { Variant~form~'#1'~invalid~for~base~form~'#2'. }
8240   {
8241     \c__msg_coding_error_text_tl
8242     LaTeX~has~been~asked~to~create~a~variant~of~the~function~'#2'~
8243     with~a~signature~starting~with~'#1',~but~cannot~change~an~argument~
8244     from~type~'#3'~to~type~'#4'.
8245   }

```

Some errors only appear in expandable settings, hence don't need a "more-text" argument.

```

8246 \__msg_kernel_new:nnn { kernel } { bad-variable }
8247   { Erroneous~variable~'#1 used! }
8248 \__msg_kernel_new:nnn { kernel } { misused-sequence }
8249   { A~sequence~was~misused. }
8250 \__msg_kernel_new:nnn { kernel } { misused-prop }
8251   { A~property~list~was~misused. }
8252 \__msg_kernel_new:nnn { kernel } { negative-replication }
8253   { Negative~argument~for~\prg_replicate:nn. }
8254 \__msg_kernel_new:nnn { kernel } { unknown-comparison }
8255   { Relation~'#1'~unknown:~use~=,~<,~>,~==,~!=,~<=,~>=. }
8256 \__msg_kernel_new:nnn { kernel } { zero-step }
8257   { Zero~step~size~for~step~function~#1. }

```

Messages used by the "show" functions.

```

8258 \__msg_kernel_new:nnn { kernel } { show-clist }
8259   {
8260     The~comma~list~ \tl_if_empty:nF {#1} { #1 ~ }
8261     \tl_if_empty:nTF {#2}
8262       { is~empty }
8263       { contains~the~items~(without~outer~braces): }
8264   }
8265 \__msg_kernel_new:nnn { kernel } { show-prop }
8266   {
8267     The~property~list~#1~
8268     \tl_if_empty:nTF {#2}
8269       { is~empty }
8270       { contains~the~pairs~(without~outer~braces): }
8271   }
8272 \__msg_kernel_new:nnn { kernel } { show-seq }
8273   {
8274     The~sequence~#1~
8275     \tl_if_empty:nTF {#2}
8276       { is~empty }
8277       { contains~the~items~(without~outer~braces): }
8278   }
8279 \__msg_kernel_new:nnn { kernel } { show-streams }
8280   {
8281     \tl_if_empty:nTF {#2} { No~ } { The~following~ }
8282     \str_case:nn {#1}
8283     {
8284       { ior } { input ~ }
8285       { iow } { output ~ }
8286     }

```

```

8287     streams~are~
8288     \tl_if_empty:nTF {#2} { open } { in-use: }
8289 }

```

16.6 Expandable errors

`__msg_expandable_error:n`
`__msg_expandable_error:w`

In expansion only context, we cannot use the normal means of reporting errors. Instead, we feed TeX an undefined control sequence, `\LaTeX3 error:`. It is thus interrupted, and shows the context, which thanks to the odd-looking `\use:n` is

```

<argument> \LaTeX3 error:
The error message.

```

In other words, TeX is processing the argument of `\use:n`, which is `\LaTeX3 error: <error message>`. Then `__msg_expandable_error:w` cleans up. In fact, there is an extra subtlety: if the user inserts tokens for error recovery, they should be kept. Thus we also use an odd space character (with category code 7) and keep tokens until that space character, dropping everything else until `\q_stop`. The `\exp_end:` prevents losing braces around the user-inserted text if any, and stops the expansion of `\exp:w`. The group is used to prevent `\LaTeX3~error:` from being globally equal to `\scan_stop:`.

```

8290 \group_begin:
8291   \cs_set_protected:Npn \__msg_tmp:w #1#2
8292   {
8293     \cs_new:Npn \__msg_expandable_error:n ##1
8294     {
8295       \exp:w
8296       \exp_after:wN \exp_after:wN
8297       \exp_after:wN \__msg_expandable_error:w
8298       \exp_after:wN \exp_after:wN
8299       \exp_after:wN \exp_end:
8300       \use:n { #1 #2 ##1 } #2
8301     }
8302     \cs_new:Npn \__msg_expandable_error:w ##1 #2 ##2 #2 {##1}
8303   }
8304   \exp_args:Nc \__msg_tmp:w { \LaTeX3-error: }
8305   { \char_generate:nn {`} {7} }
8306 \group_end:

```

(End definition for `__msg_expandable_error:n` and `__msg_expandable_error:w`.)

`__msg_kernel_expandable_error:nnnnn`
`__msg_kernel_expandable_error:nnnn`

The command built from the csname `\c_@@_text_prefix_tl` LaTeX / #1 / #2 takes four arguments and builds the error text, which is fed to `__msg_expandable_error:n`.

```

8307 \cs_new:Npn \__msg_kernel_expandable_error:nnnnnn #1#2#3#4#5#6
8308   {
8309     \exp_args:Nf \__msg_expandable_error:n
8310     {
8311       \exp_args:NNc \exp_after:wN \exp_stop_f:
8312       { \c_@@_text_prefix_tl LaTeX / #1 / #2 }
8313       {#3} {#4} {#5} {#6}
8314     }
8315   }
8316 \cs_new:Npn \__msg_kernel_expandable_error:nnnnn #1#2#3#4#5
8317   {

```

```

8318     \_\_msg_kernel_expandable_error:nnnnnn
8319         {#1} {#2} {#3} {#4} {#5} { }
8320     }
8321 \cs_new:Npn \_\_msg_kernel_expandable_error:nnnn #1#2#3#4
8322     {
8323         \_\_msg_kernel_expandable_error:nnnnnn
8324             {#1} {#2} {#3} {#4} { } { }
8325     }
8326 \cs_new:Npn \_\_msg_kernel_expandable_error:nnn #1#2#3
8327     {
8328         \_\_msg_kernel_expandable_error:nnnnnn
8329             {#1} {#2} {#3} { } { } { }
8330     }
8331 \cs_new:Npn \_\_msg_kernel_expandable_error:nn #1#2
8332     {
8333         \_\_msg_kernel_expandable_error:nnnnnn
8334             {#1} {#2} { } { } { } { }
8335     }

```

(End definition for `__msg_kernel_expandable_error:nnnnnn` and others.)

16.7 Showing variables

Functions defined in this section are used for diagnostic functions in `\clist`, `\file`, `\prop`, `\seq`, `\xtemplate`

```

\g__msg_log_next_bool
  \_\_msg_log_next:
    8336 \bool_new:N \g__msg_log_next_bool
    8337 \cs_new_protected:Npn \_\_msg_log_next:
    8338   { \bool_gset_true:N \g__msg_log_next_bool }

```

(End definition for `\g__msg_log_next_bool` and `__msg_log_next:`)

`__msg_show_pre:nnnnnn`
`__msg_show_pre:nnxxxx`
`__msg_show_pre:nnnnnV`
`__msg_show_pre_aux:n`

```

  8339 \cs_new_protected:Npn \_\_msg_show_pre:nnnnnn #1#2#3#4#5#6
  8340   {
  8341     \exp_args:Nx \iow_wrap:nnN
  8342     {
  8343       \exp_not:c { \c__msg_text_prefix_tl #1 / #2 }
  8344           { \tl_to_str:n {#3} }
  8345           { \tl_to_str:n {#4} }
  8346           { \tl_to_str:n {#5} }
  8347           { \tl_to_str:n {#6} }
  8348     }
  8349     { } { } \_\_msg_show_pre_aux:n
  8350   }
  8351 \cs_new_protected:Npn \_\_msg_show_pre:nnxxxx #1#2#3#4#5#6
  8352   {
  8353     \use:x
  8354       { \exp_not:n { \_\_msg_show_pre:nnnnnn {#1} {#2} } {#3} {#4} {#5} {#6} }
  8355   }
  8356 \cs_generate_variant:Nn \_\_msg_show_pre:nnnnnn { nnnnV }

```

```

8357 \cs_new_protected:Npn \__msg_show_pre_aux:n
8358   { \bool_if:NTF \g__msg_log_next_bool { \iow_log:n } { \iow_term:n } }

```

(End definition for `__msg_show_pre:nnnnn` and `__msg_show_pre_aux:n`.)

__msg_show_variable:NNNnn The arguments of `__msg_show_variable:NNNnn` are

- The `<variable>` to be shown as #1.
- An `<if-exist>` conditional #2 with NTF signature.
- An `<if-empty>` conditional #3 or other function with NTF signature (sometimes `\use_if:nnn`).
- The `<message>` #4 to use.
- A construction #5 which produces the formatted string eventually passed to the `\showtokens` primitive. Typically this is a mapping of the form `\seq_map_function:NN <variable> __msg_show_item:n`.

If `<if-exist>` `<variable>` is `false`, throw an error and remember to reset `\g__msg_log_next_bool`, which is otherwise reset by `__msg_show_wrap:n`. If `<message>` is not empty, output the message `\TeX/kernel/show-<message>` with as its arguments the `<variable>`, and either an empty second argument or `?` depending on the result of `<if-empty>` `<variable>`. Afterwards, show the contents of #5 using `__msg_show_wrap:n` or `__msg_log_wrap:n`.

```

8359 \cs_new_protected:Npn \__msg_show_variable:NNNnn #1#2#3#4#5
8360   {
8361     #2 #1
8362     {
8363       \tl_if_empty:nF {#4}
8364       {
8365         \__msg_show_pre:nnxxxx { \TeX / kernel } { show- #4 }
8366         { \token_to_str:N #1 } { #3 #1 { } { ? } } { } { }
8367       }
8368       \__msg_show_wrap:n {#5}
8369     }
8370     {
8371       \__msg_kernel_error:nnx { kernel } { variable-not-defined }
8372       { \token_to_str:N #1 }
8373       \bool_gset_false:N \g__msg_log_next_bool
8374     }
8375   }

```

(End definition for `__msg_show_variable:NNNnn`.)

__msg_show_wrap:Nn A short-hand used for `\int_show:n` and many other functions that passes to `__msg_show_wrap:n` the result of applying #1 (a function such as `\int_eval:n`) to the expression #2. The leading `>~` is needed by `__msg_show_wrap:n`. The use of x-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:nnN`. 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.

```

8376 \cs_new_protected:Npn \__msg_show_wrap:Nn #1#2
8377 {
8378   \exp_args:Nx \__msg_show_wrap:n
8379   {
8380     > ~ \exp_not:n { \tl_to_str:n {#2} } =
8381     \exp_not:N \tl_to_str:n { #1 {#2} }
8382   }
8383 }

```

(End definition for `__msg_show_wrap:Nn`.)

`__msg_show_wrap:n`

`__msg_show_wrap_aux:n`
`__msg_show_wrap_aux:w`

The argument of `__msg_show_wrap:n` is line-wrapped using `\iow_wrap:nnN`. Everything before the first `>` in the wrapped text is removed, as well as an optional space following it (because of f-expansion). In order for line-wrapping to give the correct result, the first `>` must in fact appear at the beginning of a line and be followed by a space (or a line-break), so in practice, the argument of `__msg_show_wrap:n` begins with `>~` or `\>~`.

The line-wrapped text is then either sent to the log file through `\iow_log:x`, or shown in the terminal using the ε -TeX primitive `\showtokens` after removing a leading `>~` and trailing dot since those are added automatically by `\showtokens`. The trailing dot was included in the first place because its presence can affect line-wrapping. Note that the space after `>` is removed through f-expansion rather than by using an argument delimited by `>~` because the space may have been replaced by a line-break when line-wrapping.

A special case is that if the line-wrapped text is a single dot (in other words if the argument of `__msg_show_wrap:n` x-expands to nothing) then no `>~` should be removed. This makes it unnecessary to check explicitly for emptiness when using for instance `\seq_map_function>NN <seq var> __msg_show_item:n` as the argument of `__msg_show_wrap:n`.

Finally, the token list `\l__msg_internal_tl` containing the result of all these manipulations is displayed to the terminal using `\etex_showtokens:D` and odd `\exp_after:wN` which expand the closing brace to improve the output slightly. The calls to `__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.

Note also that `\g__msg_log_next_bool` is only reset if that is necessary. This allows the user of an interactive prompt to insert tokens as a response to ε -TeX's `\showtokens`.

```

8384 \cs_new_protected:Npn \__msg_show_wrap:n #1
8385   { \iow_wrap:nnN { #1 . } { } { } \__msg_show_wrap_aux:n }
8386 \cs_new_protected:Npn \__msg_show_wrap_aux:n #1
8387   {
8388     \tl_if_single:nTF {#1}
8389     { \tl_clear:N \l__msg_internal_tl }
8390     { \tl_set:Nf \l__msg_internal_tl { \__msg_show_wrap_aux:w #1 \q_stop } }
8391   \bool_if:NTF \g__msg_log_next_bool
8392   {
8393     \iow_log:x { > ~ \l__msg_internal_tl . }
8394     \bool_gset_false:N \g__msg_log_next_bool
8395   }
8396   {
8397     \__iow_with:Nnn \tex_newlinechar:D { 10 }
8398   }
8399   \__iow_with:Nnn \tex_errorcontextlines:D { -1 }

```

```

8400      {
8401        \etex_showtokens:D \exp_after:wN \exp_after:wN \exp_after:wN
8402          { \exp_after:wN \l__msg_internal_tl }
8403      }
8404    }
8405  }
8406 }
8407 \cs_new:Npn \__msg_show_wrap_aux:w #1 > #2 . \q_stop {#2}
(End definition for \__msg_show_wrap:n, \__msg_show_wrap_aux:n, and \__msg_show_wrap_aux:w.)

```

Each item in the variable is formatted using one of the following functions.

```

8408 \cs_new:Npn \__msg_show_item:n #1
8409  {
8410    \\ > \\ \\{ \tl_to_str:n {#1} \\
8411  }
8412 \cs_new:Npn \__msg_show_item:nn #1#2
8413  {
8414    \\ > \\ \\{ \tl_to_str:n {#1} \\
8415    \\ \\ => \\ \\{ \tl_to_str:n {#2} \\
8416  }
8417 \cs_new:Npn \__msg_show_item_unbraced:nn #1#2
8418  {
8419    \\ > \\ \\{ \tl_to_str:n {#1}
8420    \\ \\ => \\ \\{ \tl_to_str:n {#2}
8421  }

```

(End definition for __msg_show_item:n, __msg_show_item:nn, and __msg_show_item_unbraced:nn.)
8422 </initex | package>

17 l3file implementation

The following test files are used for this code: m3file001.

```

8423 <*initex | package>
8424 <@C=file>

```

17.1 File operations

\g_file_current_name_tl The name of the current file should be available at all times. For the format the file name needs to be picked up at the start of the file. In L^AT_EX 2 _{ε} package mode the current file name is collected from \currname.

```

8425 \tl_new:N \g_file_current_name_tl
8426 <*initex>
8427 \tex_everyjob:D \exp_after:wN
8428  {
8429    \tex_the:D \tex_everyjob:D
8430    \tl_gset:Nx \g_file_current_name_tl { \tex_jobname:D }
8431  }
8432 </initex>
8433 <*package>
8434 \cs_if_exist:NT \currname
8435  { \tl_gset_eq:NN \g_file_current_name_tl \currname }
8436 </package>

```

(End definition for `\g_file_current_name_tl`. This variable is documented on page 138.)

`\g_file_stack_seq` The input list of files is stored as a sequence stack.

```
8437 \seq_new:N \g_file_stack_seq
```

(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! In format mode, this is done at the very start of the TeX run. In package mode we will eventually copy the contents of `\@filelist`.

```
8438 \seq_new:N \g_file_record_seq
8439 {*initex}
8440 \tex_everyjob:D \exp_after:wN
8441 {
8442   \tex_the:D \tex_everyjob:D
8443   \seq_gput_right:NV \g_file_record_seq \g_file_current_name_tl
8444 }
8445 
```

(End definition for `\g_file_record_seq`.)

`\l_file_internal_tl` Used as a short-term scratch variable. It may be possible to reuse `\l_file_internal_name_tl` there.

```
8446 \tl_new:N \l_file_internal_tl
```

(End definition for `\l_file_internal_tl`.)

`\l_file_internal_name_tl` Used to return the fully-qualified name of a file.

```
8447 \tl_new:N \l_file_internal_name_tl
```

(End definition for `\l_file_internal_name_tl`.)

`\l_file_search_path_seq` The current search path.

```
8448 \seq_new:N \l_file_search_path_seq
```

(End definition for `\l_file_search_path_seq`.)

`\l_file_saved_search_path_seq` The current search path has to be saved for package use.

```
8449 {*package}
8450 \seq_new:N \l_file_saved_search_path_seq
8451 
```

(End definition for `\l_file_saved_search_path_seq`.)

`\l_file_internal_seq` Scratch space for comma list conversion in package mode.

```
8452 {*package}
8453 \seq_new:N \l_file_internal_seq
8454 
```

(End definition for `\l_file_internal_seq`.)

`__file_name_sanitize:nn`

For converting a token list to a string where active characters are treated as strings from the start. The logic to the quoting normalisation is the same as used by `lualatexquotejobname`: check for balanced ", and assuming they balance strip all of them out before quoting the entire name if it contains spaces.

```

8455 \cs_new_protected:Npn \__file_name_sanitize:nn #1#2
8456   {
8457     \group_begin:
8458     \seq_map_inline:Nn \l_char_active_seq
8459     {
8460       \tl_set:Nx \l_file_internal_tl { \iow_char:N ##1 }
8461       \char_set_active_eq:NN ##1 \l_file_internal_tl
8462     }
8463     \tl_set:Nx \l_file_internal_name_tl {#1}
8464     \tl_set:Nx \l_file_internal_name_tl
8465     { \tl_to_str:N \l_file_internal_name_tl }
8466     \int_compare:nNnTF
8467     {
8468       \int_mod:nn
8469       {
8470         0 \tl_map_function:NN \l_file_internal_name_tl
8471         \__file_name_sanitize_aux:n
8472       }
8473       { 2 }
8474     }
8475     = 0
8476   {
8477     \tl_remove_all:Nn \l_file_internal_name_tl { " }
8478     \tl_if_in:NnT \l_file_internal_name_tl { ~ }
8479     {
8480       \tl_set:Nx \l_file_internal_name_tl
8481       { " \exp_not:V \l_file_internal_name_tl " }
8482     }
8483   }
8484   {
8485     \__msg_kernel_error:nnx
8486     { kernel } { unbalanced-quote-in-filename }
8487     { \l_file_internal_name_tl }
8488   }
8489   \use:x
8490   {
8491     \group_end:
8492     \exp_not:n {#2} { \l_file_internal_name_tl }
8493   }
8494 }
8495 \cs_new:Npn \__file_name_sanitize_aux:n #1
8496   { \token_if_eq_charcode:NNT #1 " { + 1 } }
```

(End definition for `__file_name_sanitize:nn` and `__file_name_sanitize_aux:n`.)

`\file_add_path:nN`

`__file_add_path:nN`

`__file_add_path_search:nN`

The way to test if a file exists is to try to open it: if it does not exist then TeX will report end-of-file. For files which are in the current directory, this is straight-forward. For other locations, a search has to be made looking at each potential path in turn. The first location is of course treated as the correct one. If nothing is found, #2 is returned empty.

```

8497 \cs_new_protected:Npn \file_add_path:nN #1
8498   { \__file_name_sanitize:nn {#1} { \__file_add_path:nN } }
8499 \cs_new_protected:Npn \__file_add_path:nN #1#2
8500   {
8501     \__ior_open:Nn \g__file_internal_ior {#1}
8502     \ior_if_eof:NTF \g__file_internal_ior
8503       { \__file_add_path_search:nN {#1} #2 }
8504       { \tl_set:Nn #2 {#1} }
8505     \ior_close:N \g__file_internal_ior
8506   }
8507 \cs_new_protected:Npn \__file_add_path_search:nN #1#2
8508   {
8509     \tl_set:Nn #2 { \q_no_value }
8510 (*package)
8511   \cs_if_exist:NT \input@path
8512   {
8513     \seq_set_eq:NN \l__file_saved_search_path_seq
8514       \l__file_search_path_seq
8515     \seq_set_split:NnV \l__file_internal_seq { , } \input@path
8516     \seq_concat:NNN \l__file_search_path_seq
8517       \l__file_search_path_seq \l__file_internal_seq
8518   }
8519 
8520   \seq_map_inline:Nn \l__file_search_path_seq
8521   {
8522     \__ior_open:Nn \g__file_internal_ior { ##1 #1 }
8523     \ior_if_eof:NF \g__file_internal_ior
8524     {
8525       \tl_set:Nx #2 { ##1 #1 }
8526       \seq_map_break:
8527     }
8528   }
8529 
8530   \cs_if_exist:NT \input@path
8531   {
8532     \seq_set_eq:NN \l__file_search_path_seq
8533       \l__file_saved_search_path_seq
8534   }
8535 
8536 }

```

(End definition for `\file_add_path:nN`, `__file_add_path:nN`, and `__file_add_path_search:nN`. These functions are documented on page [138](#).)

\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 will contain something, whereas if the file was not located then the return value will be `\q_no_value`.

```

8537 \prg_new_protected_conditional:Npnn \file_if_exist:n #1 { T , F , TF }
8538   {
8539     \file_add_path:nN {#1} \l__file_internal_name_tl
8540     \quark_if_no_value:NTF \l__file_internal_name_tl
8541       { \prg_return_false: }
8542       { \prg_return_true: }
8543   }

```

(End definition for `\file_if_exist:nTF`. This function is documented on page 138.)

```
\file_input:n  
\_file_if_exist:nT  
\_file_input:n  
\_file_input:V  
\_file_input_aux:n  
\_file_input_aux:o
```

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.

```
8544 \cs_new_protected:Npn \file_input:n #1  
8545 {  
8546     \_file_if_exist:nT {#1}  
8547     { \_file_input:V \l_file_internal_name_tl }  
8548 }
```

This code is spun out as a separate function to encapsulate the error message into a easy-to-reuse form.

```
8549 \cs_new_protected:Npn \_file_if_exist:nT #1#2  
8550 {  
8551     \file_if_exist:nTF {#1}  
8552     {#2}  
8553     {  
8554         \_file_name_sanitize:nn {#1}  
8555         { \_msg_kernel_error:nnx { kernel } { file-not-found } }  
8556     }  
8557 }  
8558 \cs_new_protected:Npn \_file_input:n #1  
8559 {  
8560     \tl_if_in:nnTF {#1} { . }  
8561     { \_file_input_aux:n {#1} }  
8562     { \_file_input_aux:o { \tl_to_str:n { #1 . tex } } }  
8563 }  
8564 \cs_generate_variant:Nn \_file_input:n { V }  
8565 \cs_new_protected:Npn \_file_input_aux:n #1  
8566 {  
8567     <initex>  
8568     \seq_gput_right:Nn \g_file_record_seq {#1}  
8569     </initex>  
8570     <package>  
8571         \clist_if_exist:NTF \@filelist  
8572         { \addtofilelist {#1} }  
8573         { \seq_gput_right:Nn \g_file_record_seq {#1} }  
8574     </package>  
8575     \seq_gpush:No \g_file_stack_seq \g_file_current_name_tl  
8576     \tl_gset:Nn \g_file_current_name_tl {#1}  
8577     \tex_input:D #1 \c_space_tl  
8578     \seq_gpop>NN \g_file_stack_seq \l_file_internal_tl  
8579     \tl_gset_eq>NN \g_file_current_name_tl \l_file_internal_tl  
8580 }  
8581 \cs_generate_variant:Nn \_file_input_aux:n { o }
```

(End definition for `\file_input:n` and others. These functions are documented on page 138.)

```
\file_path_include:n  
\file_path_remove:n  
\_file_path_include:n
```

Wrapper functions to manage the search path.

```
8582 \cs_new_protected:Npn \file_path_include:n #1  
8583 { \_file_name_sanitize:nn {#1} { \_file_path_include:n } }  
8584 \cs_new_protected:Npn \_file_path_include:n #1  
8585 {
```

```

8586      \seq_if_in:NnF \l_file_search_path_seq {#1}
8587      { \seq_put_right:Nn \l_file_search_path_seq {#1} }
8588  }
8589 \cs_new_protected:Npn \file_path_remove:n #1
8590  {
8591      \__file_name_sanitize:nn {#1}
8592      { \seq_remove_all:Nn \l_file_search_path_seq }
8593  }

```

(End definition for `\file_path_include:n`, `\file_path_remove:n`, and `__file_path_include:n`. These functions are documented on page 138.)

\file_list: A function to list all files used to the log, without duplicates. In package mode, if `\@filelist` is still defined, we need to take this list of file names into account (we capture it `\AtBeginDocument` into `\g_file_record_seq`), turning each file name into a string.

```

8594 \cs_new_protected:Npn \file_list:
8595  {
8596      \seq_set_eq:NN \l_file_internal_seq \g_file_record_seq
8597  (*package)
8598      \clist_if_exist:NT \@filelist
8599  {
8600      \clist_map_inline:Nn \@filelist
8601  {
8602          \seq_put_right:No \l_file_internal_seq
8603          { \tl_to_str:n {##1} }
8604      }
8605  }
8606  (/package)
8607      \seq_remove_duplicates:N \l_file_internal_seq
8608      \iow_log:n { *~File-List~* }
8609      \seq_map_inline:Nn \l_file_internal_seq { \iow_log:n {##1} }
8610      \iow_log:n { ***** }
8611  }

```

(End definition for `\file_list:`. This function is documented on page 139.)

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`.

```

8612 (*package)
8613 \AtBeginDocument
8614  {
8615      \clist_map_inline:Nn \@filelist
8616      { \seq_gput_right:No \g_file_record_seq { \tl_to_str:n {#1} } }
8617  }
8618 (/package)

```

17.2 Input operations

8619 `(@@=ior)`

17.2.1 Variables and constants

\c_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.

```
8620 \int_const:Nn \c_term_ior { 16 }
```

(End definition for `\c_term_ior`. This variable is documented on page 145.)

- `\g__ior_streams_seq` A list of the currently-available input streams to be used as a stack. In format mode, all streams (from 0 to 15) are available, while the package requests streams to L^AT_EX 2_ε as they are needed (initially none are needed), so the starting point varies!

```
8621 \seq_new:N \g__ior_streams_seq
8622 {*initex}
8623 \seq_gset_split:Nnn \g__ior_streams_seq { , }
8624 { 0 , 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 , 10 , 11 , 12 , 13 , 14 , 15 }
8625 
```

(End definition for `\g__ior_streams_seq`.)

- `\l__ior_stream_t1` Used to recover the raw stream number from the stack.

```
8626 \tl_new:N \l__ior_stream_t1
```

(End definition for `\l__ior_stream_t1`.)

- `\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.

```
8627 \prop_new:N \g__ior_streams_prop
8628 {*package}
8629 \int_step_inline:nnnn
8630 { 0 }
8631 { 1 }
8632 {
8633   \cs_if_exist:NTF \normalend
8634   { \tex_count:D 38 \scan_stop: }
8635   {
8636     \tex_count:D 16 \scan_stop:
8637     \cs_if_exist:NT \loccount { - 1 }
8638   }
8639 }
8640 {
8641   \prop_gput:Nnn \g__ior_streams_prop {#1} { Reserved~by~format }
8642 }
8643 
```

(End definition for `\g__ior_streams_prop`.)

17.2.2 Stream management

- `\ior_new:N` Reserving a new stream is done by defining the name as equal to using the terminal.

```
8644 \cs_new_protected:Npn \ior_new:N #1 { \cs_new_eq:NN #1 \c_term_ior }
8645 \cs_generate_variant:Nn \ior_new:N { c }
```

(End definition for `\ior_new:N`. This function is documented on page 139.)

\ior_open:Nn
\ior_open:cn
_ior_open_aux:Nn

Opening an input stream requires a bit of pre-processing. The file name is sanitized to deal with active characters, before an auxiliary adds a path and checks that the file really exists. If those two tests pass, then pass the information on to the lower-level function which deals with streams.

```

8646 \cs_new_protected:Npn \ior_open:Nn #1#2
8647   { \__file_name_sanitize:nn {#2} { \_ior_open_aux:Nn #1 } }
8648 \cs_generate_variant:Nn \ior_open:Nn { c }
8649 \cs_new_protected:Npn \_ior_open_aux:Nn #1#2
8650   {
8651     \file_add_path:nN {#2} \l__file_internal_name_tl
8652     \quark_if_no_value:NTF \l__file_internal_name_tl
8653       { \__msg_kernel_error:nnx { kernel } { file-not-found } {#2} }
8654       { \__ior_open:No #1 \l__file_internal_name_tl }
8655   }

```

(End definition for `\ior_open:Nn` and `_ior_open_aux:Nn`. These functions are documented on page 139.)

\ior_open:NnTF
\ior_open:cNTF

Much the same idea for opening a read with a conditional, except the auxiliary function does not issue an error if the file is not found.

_ior_open_aux:NnTF

```

8656 \prg_new_protected_conditional:Npnn \ior_open:Nn #1#2 { T , F , TF }
8657   { \__file_name_sanitize:nn {#2} { \_ior_open_aux:NnTF #1 } }
8658 \cs_generate_variant:Nn \ior_open:NnT { c }
8659 \cs_generate_variant:Nn \ior_open:NnF { c }
8660 \cs_generate_variant:Nn \ior_open:NnTF { c }
8661 \cs_new_protected:Npn \_ior_open_aux:NnTF #1#2
8662   {
8663     \file_add_path:nN {#2} \l__file_internal_name_tl
8664     \quark_if_no_value:NTF \l__file_internal_name_tl
8665       { \prg_return_false: }
8666       {
8667         \__ior_open:No #1 \l__file_internal_name_tl
8668         \prg_return_true:
8669       }
8670   }

```

(End definition for `\ior_open:NnTF` and `_ior_open_aux:NnTF`. These functions are documented on page 139.)

_ior_new:N

In package mode, 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`.

```

8671 (*package)
8672 \exp_args:NNf \cs_new_protected:Npn \_ior_new:N
8673   { \exp_args:NNc \exp_after:wN \exp_stop_f: { newread } }
8674 
```

(End definition for `_ior_new:N`.)

_ior_open:Nn
_ior_open:No
_ior_open_stream:Nn

The stream allocation itself uses the fact that there is a list of all of those available, so allocation is simply a question of using the number at the top of the list. In package mode, life gets more complex as it's important to keep things in sync. That is done using

a 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 L^AT_EX 2 _{ε} for a new stream and use that number (after a bit of conversion).

```

8675 \cs_new_protected:Npn \__ior_open:Nn #1#2
8676 {
8677     \ior_close:N #1
8678     \seq_gpop:NNTF \g__ior_streams_seq \l__ior_stream_tl
8679     { \__ior_open_stream:Nn #1 {#2} }
8680     {*initex}
8681     { \__msg_kernel_fatal:nn { kernel } { input-streams-exhausted } }
8682     {/initex}
8683     {*package}
8684     {
8685         \__ior_new:N #1
8686         \tl_set:Nx \l__ior_stream_tl { \int_eval:n {#1} }
8687         \__ior_open_stream:Nn #1 {#2}
8688     }
8689     {/package}
8690 }
8691 \cs_generate_variant:Nn \__ior_open:Nn { No }
8692 \cs_new_protected:Npn \__ior_open_stream:Nn #1#2
8693 {
8694     \tex_global:D \tex_chardef:D #1 = \l__ior_stream_tl \scan_stop:
8695     \prop_gput:NVn \g__ior_streams_prop #1 {#2}
8696     \tex_openin:D #1 #2 \scan_stop:
8697 }

```

(End definition for `__ior_open:Nn` and `__ior_open_stream:Nn`.)

\ior_close:N Closing a stream means getting rid of it at the TeX 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.

```

8698 \cs_new_protected:Npn \ior_close:N #1
8699 {
8700     \int_compare:nT { -1 < #1 < \c_term_ior }
8701     {
8702         \tex_closein:D #1
8703         \prop_gremove:NV \g__ior_streams_prop #1
8704         \seq_if_in:NVF \g__ior_streams_seq #1
8705         { \seq_gpush:NV \g__ior_streams_seq #1 }
8706         \cs_gset_eq:NN #1 \c_term_ior
8707     }
8708 }
8709 \cs_generate_variant:Nn \ior_close:N { c }

```

(End definition for `\ior_close:N`. This function is documented on page 140.)

\ior_list_streams: `__ior_list_streams:Nn` 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 (in fact a question mark) otherwise. The code of the message `show-streams` takes care of translating `ior/iow` to English. The list of streams is formatted using `__msg_show_item_unbraced:nn`.

```

8710 \cs_new_protected:Npn \ior_list_streams:
8711   { \__ior_list_streams:Nn \g__ior_streams_prop { ior } }
8712 \cs_new_protected:Npn \__ior_list_streams:Nn #1#2
8713   {
8714     \__msg_show_pre:nxxxxx { LaTeX / kernel } { show-streams }
8715     {#2} { \prop_if_empty:NF #1 { ? } } { } { }
8716     \__msg_show_wrap:n
8717     { \prop_map_function:NN #1 \__msg_show_item_unbraced:nn }
8718   }

```

(End definition for `\ior_list_streams:` and `__ior_list_streams:Nn`. These functions are documented on page 140.)

17.2.3 Reading input

`\if_eof:w` The primitive conditional

```
8719 \cs_new_eq:NN \if_eof:w \tex_ifeof:D
```

(End definition for `\if_eof:w.`)

`\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).

```

8720 \prg_new_conditional:Npnn \ior_if_eof:N #1 { p , T , F , TF }
8721   {
8722     \cs_if_exist:NTF #1
8723       {
8724         \int_compare:nTF { -1 < #1 < \c_term_ior }
8725           {
8726             \if_eof:w #1
8727               \prg_return_true:
8728             \else:
8729               \prg_return_false:
8730             \fi:
8731           }
8732         { \prg_return_true: }
8733       }
8734     { \prg_return_true: }
8735   }

```

(End definition for `\ior_if_eof:NTF`. This function is documented on page 142.)

`\ior_get:NN` And here we read from files.

```
8736 \cs_new_protected:Npn \ior_get:NN #1#2
8737   { \tex_read:D #1 to #2 }
```

(End definition for `\ior_get:NN`. This function is documented on page 140.)

`\ior_str_get:NN` Reading as strings is a more complicated wrapper, as we wish to remove the endline character.

```

8738 \cs_new_protected:Npn \ior_str_get:NN #1#2
8739   {
8740     \use:x
8741       {
8742         \int_set:Nn \tex_endlinechar:D { -1 }
```

```

8743     \exp_not:n { \etex_readline:D #1 to #2 }
8744     \int_set:Nn \tex_endlinechar:D { \int_use:N \tex_endlinechar:D }
8745   }
8746 }

```

(End definition for `\ior_str_get:NN`. This function is documented on page 141.)

`\ior_map_break:` Usual map breaking functions.

```

\ior_map_break:n
8747 \cs_new:Npn \ior_map_break:
8748   { \__prg_map_break:Nn \ior_map_break: { } }
8749 \cs_new:Npn \ior_map_break:n
8750   { \__prg_map_break:Nn \ior_map_break: }

```

(End definition for `\ior_map_break:` and `\ior_map_break:n`. These functions are documented on page 141.)

`\ior_map_inline:Nn`

`\ior_str_map_inline:Nn`

`__ior_map_inline>NNn`

`__ior_map_inline>NNNn`

`__ior_map_inline_loop:NNN`

`\l__ior_internal_tl`

Mapping to an input stream can be done on either a token or a string basis, hence the set up. Within that, there is a check to avoid reading past the end of a file, hence the two applications of `\ior_if_eof:N`. This mapping cannot be nested with twice the same stream, as the stream has only one “current line”.

```

8751 \cs_new_protected:Npn \ior_map_inline:Nn
8752   { \__ior_map_inline>NNn \ior_get>NN }
8753 \cs_new_protected:Npn \ior_str_map_inline:Nn
8754   { \__ior_map_inline>NNn \ior_str_get>NN }
8755 \cs_new_protected:Npn \__ior_map_inline>NNn
8756   {
8757     \int_gincr:N \g__prg_map_int
8758     \exp_args:Nc \__ior_map_inline>NNNn
8759       { __prg_map_ \int_use:N \g__prg_map_int :n }
8760   }
8761 \cs_new_protected:Npn \__ior_map_inline>NNNn #1#2#3#4
8762   {
8763     \cs_gset_protected:Npn #1 ##1 {#4}
8764     \ior_if_eof:NF #3 { \__ior_map_inline_loop:NNN #1#2#3 }
8765     \__prg_break_point:Nn \ior_map_break:
8766       { \int_gdecr:N \g__prg_map_int }
8767   }
8768 \cs_new_protected:Npn \__ior_map_inline_loop:NNN #1#2#3
8769   {
8770     #2 #3 \l__ior_internal_tl
8771     \ior_if_eof:NF #3
8772     {
8773       \exp_args:No #1 \l__ior_internal_tl
8774       \__ior_map_inline_loop:NNN #1#2#3
8775     }
8776   }
8777 \tl_new:N \l__ior_internal_tl

```

(End definition for `\ior_map_inline:Nn` and others. These functions are documented on page 141.)

`\g_file_internal_ior` Needed by the higher-level code, but cannot be created until here.

```

8778 \ior_new:N \g_file_internal_ior

```

(End definition for `\g_file_internal_ior`.)

17.3 Output operations

8779 ⟨@@=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.

17.3.1 Variables and constants

\c_log_iow Here we allocate two output streams for writing to the transcript file only (**\c_log_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.

```
8780 \int_const:Nn \c_log_iow { -1 }
8781 \int_const:Nn \c_term_iow
8782 {
8783   \cs_if_exist:NTF \luatex_directlua:D
8784   {
8785     \int_compare:nNnTF \luatex_luatexversion:D > { 80 }
8786     { 128 }
8787     { 16 }
8788   }
8789   { 16 }
8790 }
```

(End definition for **\c_log_iow** and **\c_term_iow**. These variables are documented on page 145.)

\g__iow_streams_seq A list of the currently-available output streams to be used as a stack.

```
8791 \seq_new:N \g__iow_streams_seq
8792 (*initex)
8793 \use:x
8794 {
8795   \exp_not:n { \seq_gset_split:Nnn \g__iow_streams_seq { } }
8796   {
8797     \int_step_function:nnnN { 0 } { 1 } { \c_term_iow }
8798     \prg_do_nothing:
8799   }
8800 }
```

(End definition for **\g__iow_streams_seq**.)

\l__iow_stream_t1 Used to recover the raw stream number from the stack.

```
8802 \tl_new:N \l__iow_stream_t1
```

(End definition for **\l__iow_stream_t1**.)

\g__iow_streams_prop As for reads with the appropriate adjustment of the register numbers to check on.

```
8803 \prop_new:N \g__iow_streams_prop
8804 (*package)
8805 \int_step_inline:nnnn
8806   { 0 }
8807   { 1 }
8808   {
8809     \cs_if_exist:NTF \normalend
8810     { \tex_count:D 39 \scan_stop: }
```

```

8811   {
8812     \tex_count:D 17 \scan_stop:
8813     \cs_if_exist:NT \loccount { - 1 }
8814   }
8815 }
8816 {
8817   \prop_gput:Nnn \g__iow_streams_prop {#1} { Reserved~by~format }
8818 }
8819 
```

(End definition for `\g__iow_streams_prop`.)

17.4 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.

```

8820 \cs_new_protected:Npn \iow_new:N #1 { \cs_new_eq:NN #1 \c_term_iow }
8821 \cs_generate_variant:Nn \iow_new:N { c }

```

(End definition for `\iow_new:N`. This function is documented on page 139.)

`__iow_new:N` As for read streams, copy `\newwrite` in package mode, making sure that it is not `\outer`.

```

8822 
```

`(*package)`

```

8823 \exp_args:NNF \cs_new_protected:Npn \__iow_new:N
8824   { \exp_args:NNC \exp_after:wN \exp_stop_f: { newwrite } }
8825 
```

(End definition for `__iow_new:N`.)

`\iow_open:Nn` The same idea as for reading, but without the path and without the need to allow for a conditional version.

```

8826 \cs_new_protected:Npn \iow_open:Nn #1#2
8827   { \__file_name_sanitize:nn {#2} { \__iow_open:Nn #1 } }
8828 \cs_generate_variant:Nn \iow_open:Nn { c }
8829 \cs_new_protected:Npn \__iow_open:Nn #1#2
8830   {
8831     \iow_close:N #1
8832     \seq_gpop:NNTF \g__iow_streams_seq \l__iow_stream_tl
8833       { \__iow_open_stream:Nn #1 {#2} }
8834 
```

`(*initex)`

```

8835   { \__msg_kernel_fatal:nn { kernel } { output-streams-exhausted } }
8836 
```

`(/initex)`

```

8837 
```

`(*package)`

```

8838   {
8839     \__iow_new:N #1
8840     \tl_set:Nx \l__iow_stream_tl { \int_eval:n {#1} }
8841     \__iow_open_stream:Nn #1 {#2}
8842   }
8843 
```

`(/package)`

```

8844   }
8845 \cs_generate_variant:Nn \__iow_open:Nn { No }
8846 \cs_new_protected:Npn \__iow_open_stream:Nn #1#2
8847   {
8848     \tex_global:D \tex_chardef:D #1 = \l__iow_stream_tl \scan_stop:

```

```

8849     \prop_gput:NVn \g_iow_streams_prop #1 {#2}
8850     \tex_immediate:D \tex_openout:D #1 #2 \scan_stop:
8851 }

```

(End definition for `\iow_open:Nn`, `_iow_open:Nn`, and `_iow_open_stream:Nn`. These functions are documented on page 139.)

\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.

```

8852 \cs_new_protected:Npn \iow_close:N #1
8853 {
8854     \int_compare:nT { - \c_log_iow < #1 < \c_term_iow }
8855     {
8856         \tex_immediate:D \tex_closeout:D #1
8857         \prop_gremove:NV \g_iow_streams_prop #1
8858         \seq_if_in:NVF \g_iow_streams_seq #1
8859         { \seq_gpush:NV \g_iow_streams_seq #1 }
8860         \cs_gset_eq:NN #1 \c_term_iow
8861     }
8862 }
8863 \cs_generate_variant:Nn \iow_close:N { c }

```

(End definition for `\iow_close:N`. This function is documented on page 140.)

\iow_list_streams: Done as for input, but with a copy of the auxiliary so the name is correct.

```

8864 \cs_new_protected:Npn \iow_list_streams:
8865     { \_iow_list_streams:Nn \g_iow_streams_prop { iow } }
8866 \cs_new_eq:NN \_iow_list_streams:Nn \_ior_list_streams:Nn

```

(End definition for `\iow_list_streams:` and `_iow_list_streams:Nn`. These functions are documented on page 140.)

17.4.1 Deferred writing

\iow_shipout_x:Nn First the easy part, this is the primitive, which expects its argument to be braced.

```

8867 \cs_new_protected:Npn \iow_shipout_x:Nn #1#
8868     { \tex_write:D #1 {#2} }
8869 \cs_generate_variant:Nn \iow_shipout_x:Nn { c, Nx, cx }

```

(End definition for `\iow_shipout_x:Nn`. This function is documented on page 143.)

\iow_shipout:Nn With ε - \TeX available deferred writing without expansion is easy.

```

8870 \cs_new_protected:Npn \iow_shipout:Nn #1#
8871     { \tex_write:D #1 { \exp_not:n {#2} } }
8872 \cs_generate_variant:Nn \iow_shipout:Nn { c, Nx, cx }

```

(End definition for `\iow_shipout:Nn`. This function is documented on page 143.)

17.4.2 Immediate writing

__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.

```

8873 \cs_new_protected:Npn \_\_iow_with:Nnn #1#2
8874 {
8875     \int_compare:nNnTF {#1} = {#2}
8876     { \use:n }
8877     { \exp_args:No \_\_iow_with_aux:nNnn { \int_use:N #1 } #1 {#2} }
8878 }
8879 \cs_new_protected:Npn \_\_iow_with_aux:nNnn #1#2#3#4
8880 {
8881     \int_set:Nn #2 {#3}
8882     #4
8883     \int_set:Nn #2 {#1}
8884 }
```

(End definition for `__iow_with:Nnn` and `__iow_with_aux:nNnn`.)

\iow_now:Nn This routine writes the second argument onto the output stream without expansion. If this stream isn't open, the output goes to the terminal instead. If the first argument is no output stream at all, we get an internal error. We don't use the expansion done by `\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 `__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.

```

8885 \cs_new_protected:Npn \iow_now:Nn #1#2
8886 {
8887     \_\_iow_with:Nnn \tex_newlinechar:D { '\^J }
8888     { \tex_immediate:D \tex_write:D #1 { \exp_not:n {#2} } }
8889 }
8890 \cs_generate_variant:Nn \iow_now:Nn { c, Nx, cx }
```

(End definition for `\iow_now:Nn`. This function is documented on page 142.)

\iow_log:n Writing to the log and the terminal directly are relatively easy.

```

8891 \cs_set_protected:Npn \iow_log:x { \iow_now:Nx \c_log_iow }
8892 \cs_new_protected:Npn \iow_log:n { \iow_now:Nn \c_log_iow }
8893 \cs_set_protected:Npn \iow_term:x { \iow_now:Nx \c_term_iow }
8894 \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 142.)

17.4.3 Special characters for writing

\iow_newline: Global variable holding the character that forces a new line when something is written to an output stream.

```
8895 \cs_new:Npn \iow_newline: { ^J }
```

(End definition for `\iow_newline:`. This function is documented on page 143.)

\iow_char:N Function to write any escaped char to an output stream.

```
8896 \cs_new_eq:NN \iow_char:N \cs_to_str:N
```

(End definition for `\iow_char:N`. This function is documented on page 143.)

17.4.4 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.

`\c_catcode_other_space_tl` Create a space with category code 12: an “other” space.

```
8897 \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 144.*)

`\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 TeXLive and MikTeX.

```
8898 \int_new:N \l_iow_line_count_int
8899 \int_set:Nn \l_iow_line_count_int { 78 }
```

(*End definition for \l_iow_line_count_int. This variable is documented on page 144.*)

`\l__iow_newline_tl` The token list inserted to produce a new line, with the $\langle run-on\ text \rangle$.

```
8900 \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.

```
8901 \int_new:N \l__iow_line_target_int
```

(*End definition for \l__iow_line_target_int.*)

`__iow_set_indent:n
 __iow_unindent:w
\l__iow_one_indent_tl
\l__iow_one_indent_int` 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.

```
8902 \tl_new:N \l__iow_one_indent_tl
8903 \int_new:N \l__iow_one_indent_int
8904 \cs_new:Npn \__iow_unindent:w { }
8905 \cs_new_protected:Npn \__iow_set_indent:n #1
8906   {
8907     \tl_set:Nx \l__iow_one_indent_tl
8908       { \exp_args:No \__str_to_other_fast:n { \tl_to_str:n {#1} } }
8909     \int_set:Nn \l__iow_one_indent_int { \str_count:N \l__iow_one_indent_tl }
8910     \exp_last_unbraced:NNo
8911       \cs_set:Npn \__iow_unindent:w \l__iow_one_indent_int { }
8912   }
8913 \exp_args:Nx \__iow_set_indent:n { \prg_replicate:nn { 4 } { ~ } }
```

(*End definition for __iow_set_indent:n and others.*)

`\l__iow_indent_tl
\l__iow_indent_int` The current indentation (some copies of `\l__iow_one_indent_tl`) and its number of characters.

```
8914 \tl_new:N \l__iow_indent_tl
8915 \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.
\l_iow_line_part_tl
8916 \tl_new:N \l_iow_line_tl
8917 \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.
8918 \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.
8919 \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 starts with the same recognizable string, \c_iow_wrap_end_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.
\c_iow_wrap_newline_marker_tl
\c_iow_wrap_indent_marker_tl
\c_iow_wrap_unindent_marker_tl

```

8920 \group_begin:
8921   \int_set:Nn \tex_escapechar:D { -1 }
8922   \tl_const:Nx \c_iow_wrap_marker_tl
8923   { \tl_to_str:n { \^I \^O \^W \^_ \^W \^R \^A \^P } }
8924 \group_end:
8925 \tl_map_inline:nn
8926 { { end } { newline } { indent } { unindent } }
8927 {
8928   \tl_const:c { c_iow_wrap_#1 _marker_tl }
8929   {
8930     \c_iow_wrap_marker_tl
8931     #1
8932     \c_catcode_other_space_tl
8933   }
8934 }
```

(End definition for \c_iow_wrap_marker_tl and others.)

\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 the instruction for unindenting afterwards. The second produces an error expandably. Note that there will be no forced line-break, so the indentation only changes when the next line is started.
__iow_indent_error:n

```

8935 \cs_new_protected:Npn \iow_indent:n #1
8936 {
8937   \__msg_kernel_error:nnnn { kernel } { iow-indent }
8938   { \iow_wrap:nnN } { \iow_indent:n } { #1 }
8939   #1
8940 }
8941 \cs_new:Npx \__iow_indent:n #1
8942 {
8943   \c_iow_wrap_indent_marker_tl
```

```

8944      #1
8945      \c__iow_wrap_unindent_marker_tl
8946    }
8947 \cs_new:Npn \__iow_indent_error:n #1
8948  {
8949    \__msg_kernel_expandable_error:nnnn { kernel } { iow-indent }
8950    { \iow_wrap:nnN } { \iow_indent:n } {#1}
8951    #1
8952  }

```

(End definition for `\iow_indent:n`, `__iow_indent:n`, and `__iow_indent_error:n`. These functions are documented on page 144.)

\iow_wrap:nnN
`__iow_wrap_set:Nx` The main wrapping function works as follows. First give `\\"`, `_` and other formatting commands the correct definition for messages and perform the given setup #3. 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.

```

8953 \cs_new_protected:Npn \iow_wrap:nnN #1#2#3#4
8954  {
8955    \group_begin:
8956      \int_set:Nn \tex_escapechar:D { -1 }
8957      \cs_set:Npx \{ { \token_to_str:N \{ }
8958      \cs_set:Npx \# { \token_to_str:N \# }
8959      \cs_set:Npx \} { \token_to_str:N \} }
8960      \cs_set:Npx \% { \token_to_str:N \% }
8961      \cs_set:Npx \~ { \token_to_str:N \~ }
8962      \int_set:Nn \tex_escapechar:D { 92 }
8963      \cs_set_eq:NN \\ \c__iow_newline_marker_tl
8964      \cs_set_eq:NN \ \ \c_catcode_other_space_tl
8965      \cs_set_eq:NN \iow_indent:n \__iow_indent:n
8966      #3

```

Then fully-expand the input: in package mode, the expansion uses L^AT_EX 2 _{ϵ} ’s `\protected` mechanism. 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:nnN`.

```

8967 (*initex)
8968   \tl_set:Nx \l__iow_wrap_tl {#1}
8969  (/initex)
8970  (*package)
8971   \__iow_wrap_set:Nx \l__iow_wrap_tl {#1}
8972  (/package)
8973   \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).

```

8974   \tl_set:Nx \l__iow_newline_tl { \iow_newline: #2 }
8975   \tl_set:Nx \l__iow_newline_tl { \tl_to_str:N \l__iow_newline_tl }
8976   \int_set:Nn \l__iow_line_target_int
8977     { \l_iow_line_count_int - \str_count:N \l__iow_newline_tl + 1 }

```

There is then a loop over the input, which will store 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`.

```

8978      \__iow_wrap_do:
8979      \exp_args:Nnf \group_end:
8980      #4 { \tl_to_str:N \l__iow_wrap_tl }
8981  }

```

As using the generic loader will mean that `\protected@edef` is not available, it's not placed directly in the wrap function but is set up as an auxiliary. In the generic loader this can then be redefined.

```

8982 (*package)
8983 \cs_new_eq:NN \__iow_wrap_set:Nx \protected@edef
8984 (/package)

```

(End definition for `\iow_wrap:nnN` and `__iow_wrap_set:Nx`. These functions are documented on page 144.)

`__iow_wrap_do:` Escape spaces. Set up a few variables, in particular the initial value of `\l__iow_wrap_tl`: the space will stop the f-expansion of the main wrapping function and `\use_none:n` will remove a newline marker inserted by later code. The main loop consists of repeatedly calling the `chunk` auxiliary to wrap chunks delimited by (newline or indentation) markers.

```

8985 \cs_new_protected:Npn \__iow_wrap_do:
8986  {
8987  \tl_set:Nx \l__iow_wrap_tl
8988  {
8989  \exp_args:No \__str_to_other_fast:n \l__iow_wrap_tl
8990  \c__iow_end_marker_tl
8991  }
8992  \exp_after:wN \__iow_wrap_start:w \l__iow_wrap_tl
8993  }
8994 \cs_new_protected:Npn \__iow_wrap_start:w
8995  {
8996  \bool_set_false:N \l__iow_line_break_bool
8997  \tl_clear:N \l__iow_line_tl
8998  \tl_clear:N \l__iow_line_part_tl
8999  \tl_set:Nn \l__iow_wrap_tl { ~ \use_none:n }
9000  \int_zero:N \l__iow_indent_int
9001  \tl_clear:N \l__iow_indent_tl
9002  \__iow_wrap_chunk:nw { \l__iow_line_count_int }
9003  }

```

(End definition for `__iow_wrap_do:` 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_end_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.

```

9004 \cs_set_protected:Npn \__iow_tmp:w #1#2
9005  {
9006  \cs_new_protected:Npn \__iow_wrap_chunk:nw ##1##2 #2
9007  {
9008  \tl_if_empty:nTF {##2}

```

```

9009      {
9010          \tl_clear:N \l__iow_line_part_tl
9011          \__iow_wrap_next:nw {##1}
9012      }
9013      {
9014          \tl_if_empty:NTF \l__iow_line_tl
9015          {
9016              \__iow_wrap_line:nw
9017              { \l__iow_indent_t1 }
9018              ##1 - \l__iow_indent_int ;
9019          }
9020          { \__iow_wrap_line:nw { } ##1 ; }
9021          ##2 #1
9022          \__iow_wrap_end_chunk:w 7 6 5 4 3 2 1 0 \q_stop
9023      }
9024  }
9025  \cs_new_protected:Npn \__iow_wrap_next:nw ##1##2 #1
9026  { \use:c { __iow_wrap_##2:n } {##1} }
9027 }
9028 \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_end:Nnnnnnnn
\__iow_wrap_line_end:nw
\__iow_wrap_end_chunk:w

```

This is followed by `{<string>} <intexpr>`. It stores the `<string>` and up to `<intexpr>` characters from the current chunk into `\l__iow_line_part_tl`. Characters are grabbed 8 at a time and left in `\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. The weird `\use_none:nnnnn` ensures that the required data is in the right place.

```

9029 \cs_new_protected:Npn \__iow_wrap_line:nw #
9030  {
9031      \tex_eodef:D \l__iow_line_part_tl { \if_false: } \fi:
9032      #1
9033      \exp_after:wN \__iow_wrap_line_loop:w
9034      \__int_value:w \__int_eval:w
9035  }
9036 \cs_new:Npn \__iow_wrap_line_loop:w #1 ; #2#3#4#5#6#7#8#9
9037  {
9038      \if_int_compare:w #1 < 8 \exp_stop_f:
9039          \__iow_wrap_line_aux:Nw #1
9040      \fi:
9041      #2 #3 #4 #5 #6 #7 #8 #9
9042      \exp_after:wN \__iow_wrap_line_loop:w
9043      \__int_value:w \__int_eval:w #1 - 8 ;

```

```

9044     }
9045 \cs_new:Npn \__iow_wrap_line_aux:Nw #1#2#3 \exp_after:wN #4 ;
9046 {
9047     #2
9048     \exp_after:wN \__iow_wrap_line_end:NnnnnnnnN
9049     \exp_after:wN #1
9050     \exp:w \exp_end_continue_f:w
9051     \exp_after:wN \exp_after:wN
9052     \if_case:w #1 \exp_stop_f:
9053         \prg_do_nothing:
9054     \or: \use_none:n
9055     \or: \use_none:nn
9056     \or: \use_none:nnn
9057     \or: \use_none:nnnn
9058     \or: \use_none:nnnnn
9059     \or: \use_none:nnnnnn
9060     \or: \use_none:nnnnnnn
9061     \fi:
9062     { } { } { } { } { } { } { } { } #3
9063 }
9064 \cs_new:Npn \__iow_wrap_line_end:NnnnnnnnN #1#2#3#4#5#6#7#8#9
9065 {
9066     #2 #3 #4 #5 #6 #7 #8
9067     \use_none:nnnnn \__int_eval:w 8 - ; #9
9068     \token_if_eq_charcode:NNTF \c_space_token #9
9069     { \__iow_wrap_line_end:nw { } }
9070     { \if_false: { \fi: } \__iow_wrap_break:w #9 }
9071 }
9072 \cs_new:Npn \__iow_wrap_line_end:nw #1
9073 {
9074     \if_false: { \fi: }
9075     \__iow_wrap_store_do:n {#1}
9076     \__iow_wrap_next_line:w
9077 }
9078 \cs_new:Npn \__iow_wrap_end_chunk:w
9079     #1 \__int_eval:w #2 - #3 ; #4#5 \q_stop
9080 {
9081     \if_false: { \fi: }
9082     \exp_args:Nf \__iow_wrap_next:nw { \int_eval:n { #2 - #4 } }
9083 }

```

(End definition for `__iow_wrap_line:nw` and others.)

`__iow_wrap_break:w`
`__iow_wrap_break_first:w`
`__iow_wrap_break_none:w`
`__iow_wrap_break_loop:w`
`__iow_wrap_break_end:w`

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 `break_loop` auxiliary, which leaves “words” (delimited by spaces) until it hits the trailing space: then its argument `##3` is ? `__iow_wrap_break_end:w` instead of a single token, and that `break_end` 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 `break_first` auxiliary calls the `break_none` auxiliary. In that case, if the current line is empty, the complete word (including `##4`, characters beyond what we had grabbed) is added to the line, making it over-long.

Otherwise, the word will be 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).

```

9084 \cs_set_protected:Npn \__iow_tmp:w #1
9085   {
9086     \cs_new:Npn \__iow_wrap_break:w
9087       {
9088         \tex_eodef:D \l__iow_line_part_tl
9089         { \if_false: } \fi:
9090         \exp_after:wN \__iow_wrap_break_first:w
9091         \l__iow_line_part_tl
9092         #1
9093         { ? \__iow_wrap_break_end:w }
9094         \q_mark
9095       }
9096     \cs_new:Npn \__iow_wrap_break_first:w ##1 #1 ##2
9097       {
9098         \use_none:nn ##2 \__iow_wrap_break_none:w
9099         \__iow_wrap_break_loop:w ##1 #1 ##2
9100       }
9101     \cs_new:Npn \__iow_wrap_break_none:w ##1##2 #1 ##3 \q_mark ##4 #1
9102       {
9103         \tl_if_empty:NTF \l__iow_line_tl
9104         { ##2 ##4 \__iow_wrap_line_end:nw { } }
9105         { \__iow_wrap_line_end:nw { \__iow_wrap_trim:N } ##2 ##4 #1 }
9106       }
9107     \cs_new:Npn \__iow_wrap_break_loop:w ##1 #1 ##2 #1 ##3
9108       {
9109         \use_none:n ##3
9110         ##1 #1
9111         \__iow_wrap_break_loop:w ##2 #1 ##3
9112       }
9113     \cs_new:Npn \__iow_wrap_break_end:w ##1 #1 ##2 ##3 #1 ##4 \q_mark
9114       { ##1 \__iow_wrap_line_end:nw { } ##3 }
9115     }
9116 \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).

```

9117 \cs_new_protected:Npn \__iow_wrap_next_line:w #1#2 \q_stop
9118   {
9119     \tl_clear:N \l__iow_line_tl
9120     \token_if_eq_meaning:NNTF #1 \__iow_wrap_end_chunk:w
9121       {
9122         \tl_clear:N \l__iow_line_part_tl
9123         \bool_set_true:N \l__iow_line_break_bool
9124         \__iow_wrap_next:nw { \l__iow_line_target_int }
9125       }
9126       {
9127         \__iow_wrap_line:nw
9128           { \l__iow_indent_tl }
9129           \l__iow_line_target_int - \l__iow_indent_int ;

```

```

9130          #1 #2 \q_stop
9131      }
9132  }

```

(End definition for `_iow_wrap_next_line:w.`)

`_iow_wrap_indent:` 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.

```

9133 \cs_new_protected:Npn \_iow_wrap_indent:n #1
9134 {
9135     \tl_put_right:Nx \l_iow_line_tl { \l_iow_line_part_tl }
9136     \bool_set_false:N \l_iow_line_break_bool
9137     \int_add:Nn \l_iow_indent_int { \l_iow_one_indent_int }
9138     \tl_put_right:No \l_iow_indent_tl { \l_iow_one_indent_tl }
9139     \_iow_wrap_chunk:nw {#1}
9140 }
9141 \cs_new_protected:Npn \_iow_wrap_unindent:n #1
9142 {
9143     \tl_put_right:Nx \l_iow_line_tl { \l_iow_line_part_tl }
9144     \bool_set_false:N \l_iow_line_break_bool
9145     \int_sub:Nn \l_iow_indent_int { \l_iow_one_indent_int }
9146     \tl_set:Nx \l_iow_indent_tl
9147         { \exp_after:wN \_iow_unindent:w \l_iow_indent_tl }
9148     \_iow_wrap_chunk:nw {#1}
9149 }

```

(End definition for `_iow_wrap_indent: and _iow_wrap_unindent:.`)

`_iow_wrap_newline:` 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.

```

9150 \cs_new_protected:Npn \_iow_wrap_newline:n #1
9151 {
9152     \bool_if:NF \l_iow_line_break_bool
9153         { \_iow_wrap_store_do:n { \_iow_wrap_trim:N } }
9154     \bool_set_false:N \l_iow_line_break_bool
9155     \_iow_wrap_chunk:nw { \l_iow_line_target_int }
9156 }
9157 \cs_new_protected:Npn \_iow_wrap_end:n #1
9158 {
9159     \bool_if:NF \l_iow_line_break_bool
9160         { \_iow_wrap_store_do:n { \_iow_wrap_trim:N } }
9161     \bool_set_false:N \l_iow_line_break_bool
9162 }

```

(End definition for `_iow_wrap_newline: and _iow_wrap_end:.`)

`_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`).

```

9163 \cs_new_protected:Npn \__iow_wrap_store_do:n #1
9164 {
9165     \tl_set:Nx \l__iow_line_tl
9166     { \l__iow_line_tl \l__iow_line_part_tl }
9167     \tl_set:Nx \l__iow_wrap_tl
9168     {
9169         \l__iow_wrap_tl
9170         \l__iow_newline_tl
9171         #1 \l__iow_line_tl
9172     }
9173     \tl_clear:N \l__iow_line_tl
9174 }

```

(End definition for `__iow_wrap_store_do:n`.)

`__iow_wrap_trim:N` Remove one trailing “other” space from the argument.

```

\__iow_wrap_trim:w
9175 \cs_set_protected:Npn \__iow_tmp:w #1
9176 {
9177     \cs_new:Npn \__iow_wrap_trim:N ##1
9178     { \tl_if_empty:NF ##1 { \exp_after:wN \__iow_wrap_trim:w ##1 \q_stop } }
9179     \cs_new:Npn \__iow_wrap_trim:w ##1 #1 \q_stop {##1}
9180 }
9181 \exp_args:NV \__iow_tmp:w \c_catcode_other_space_tl

```

(End definition for `__iow_wrap_trim:N` and `__iow_wrap_trim:w`.)

17.5 Messages

```

\__msg_kernel_new:nnnn { kernel } { file-not-found }
9182 { File~'#1'~not~found. }
9183 {
9184     The~requested~file~could~not~be~found~in~the~current~directory,~
9185     in~the~TeX~search~path~or~in~the~LaTeX~search~path.
9186 }
9187
\__msg_kernel_new:nnnn { kernel } { input-streams-exhausted }
9188 { Input~streams~exhausted }
9189 {
9190     TeX~can~only~open~up~to~16~input~streams~at~one~time.\\
9191     All~16~are~currently~in~use,~and~something~wanted~to~open~
9192     another~one.
9193 }
9194
\__msg_kernel_new:nnnn { kernel } { output-streams-exhausted }
9195 { Output~streams~exhausted }
9196 {
9197     TeX~can~only~open~up~to~16~output~streams~at~one~time.\\
9198     All~16~are~currently~in~use,~and~something~wanted~to~open~
9199     another~one.
9200 }
9201
\__msg_kernel_new:nnnn { kernel } { unbalanced-quote-in-filename }
9202 { Unbalanced~quotes~in~file~name~'#1'. }
9203 {
9204     File~names~must~contain~balanced~numbers~of~quotes~(").
9205 }
9206
\__msg_kernel_new:nnnn { kernel } { iow-indent }
9207 { Only~#1~(arg~1)~allows~#2 }
9208

```

```

9209     {
9210     The~command~#2 can~only~be~used~in~messages~
9211     which~will~be~wrapped~using~#1.~
9212     It~was~called~with~argument~'#3'.
9213 }
```

17.6 Deprecated functions

\ior_get_str:NN For removal after 2017-12-31.

```

9214 \cs_new_protected:Npn \ior_get_str:NN
9215   {
9216     \__msg_kernel_warning:nxxxx { kernel } { deprecated-command }
9217     { 2017-12-31 }
9218     { \token_to_str:N \ior_get_str:NN }
9219     { \token_to_str:N \ior_str_get:NN }
9220     \cs_gset_eq:NN \ior_get_str:NN \ior_str_get:NN
9221     \ior_str_get:NN
9222 }
```

(End definition for \ior_get_str:NN.)

9223 ⟨/initex | package⟩

18 l3skip implementation

```

9224 ⟨*initex | package⟩
9225 ⟨@@=dim⟩
```

18.1 Length primitives renamed

\if_dim:w Primitives renamed.

```

\__dim_eval:w 9226 \cs_new_eq:NN \if_dim:w      \tex_ifdim:D
\__dim_eval_end: 9227 \cs_new_eq:NN \__dim_eval:w      \etex_dimexpr:D
                  9228 \cs_new_eq:NN \__dim_eval_end: \tex_relax:D
```

(End definition for \if_dim:w, __dim_eval:w, and __dim_eval_end:. These functions are documented on page 160.)

18.2 Creating and initialising dim variables

\dim_new:N Allocating ⟨dim⟩ registers ...

```

\dim_new:c 9229 ⟨*package⟩
            9230 \cs_new_protected:Npn \dim_new:N #1
            9231   {
            9232     \__chk_if_free_cs:N #1
            9233     \cs:w newdimen \cs_end: #1
            9234   }
            9235 ⟨/package⟩
            9236 \cs_generate_variant:Nn \dim_new:N { c }
```

(End definition for \dim_new:N. This function is documented on page 147.)

\dim_const:Nn Contrarily to integer constants, we cannot avoid using a register, even for constants.

```
\dim_const:cn
 9237 \cs_new_protected:Npn \dim_const:Nn #1
 9238   {
 9239     \dim_new:N #1
 9240     \dim_gset:Nn #1
 9241   }
 9242 \cs_generate_variant:Nn \dim_const:Nn { c }
```

(End definition for `\dim_const:Nn`. This function is documented on page 147.)

\dim_zero:N Reset the register to zero.

```
\dim_zero:c
 9243 \cs_new_protected:Npn \dim_zero:N #1 { #1 \c_zero_dim }
\dim_gzero:N
 9244 \cs_new_protected:Npn \dim_gzero:N { \tex_global:D \dim_zero:N }
\dim_gzero:c
 9245 \cs_generate_variant:Nn \dim_zero:N { c }
 9246 \cs_generate_variant:Nn \dim_gzero:N { c }
```

(End definition for `\dim_zero:N` and `\dim_gzero:N`. These functions are documented on page 147.)

\dim_zero_new:N Create a register if needed, otherwise clear it.

```
\dim_zero_new:c
 9247 \cs_new_protected:Npn \dim_zero_new:N #1
 9248   { \dim_if_exist:NTF #1 { \dim_zero:N #1 } { \dim_new:N #1 } }
\dim_gzero_new:N
 9249 \cs_new_protected:Npn \dim_gzero_new:N #1
 9250   { \dim_if_exist:NTF #1 { \dim_gzero:N #1 } { \dim_new:N #1 } }
 9251 \cs_generate_variant:Nn \dim_zero_new:N { c }
 9252 \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 147.)

\dim_if_exist_p:N Copies of the `cs` functions defined in `l3basics`.

```
\dim_if_exist_p:c
 9253 \prg_new_eq_conditional:NNn \dim_if_exist:N \cs_if_exist:N
\dim_if_exist:NTF
 9254   { TF , T , F , p }
\dim_if_exist:cTF
 9255 \prg_new_eq_conditional:NNn \dim_if_exist:c \cs_if_exist:c
 9256   { TF , T , F , p }
```

(End definition for `\dim_if_exist:NTF`. This function is documented on page 147.)

18.3 Setting dim variables

\dim_set:Nn Setting dimensions is easy enough.

```
\dim_set:cn
 9257 \cs_new_protected:Npn \dim_set:Nn #1#2
 9258   { #1 ~ \_dim_eval:w #2 \_dim_eval_end: }
\dim_gset:cn
 9259 \cs_new_protected:Npn \dim_gset:Nn { \tex_global:D \dim_set:Nn }
 9260 \cs_generate_variant:Nn \dim_set:Nn { c }
 9261 \cs_generate_variant:Nn \dim_gset:Nn { c }
```

(End definition for `\dim_set:Nn` and `\dim_gset:Nn`. These functions are documented on page 148.)

\dim_set_eq:NN All straightforward.

```
\dim_set_eq:cN
 9262 \cs_new_protected:Npn \dim_set_eq:NN #1#2 { #1 = #2 }
\dim_set_eq:Nc
 9263 \cs_generate_variant:Nn \dim_set_eq:NN { c }
\dim_set_eq:cc
 9264 \cs_generate_variant:Nn \dim_set_eq:NN { Nc , cc }
\dim_gset_eq:NN
 9265 \cs_new_protected:Npn \dim_gset_eq:NN #1#2 { \tex_global:D #1 = #2 }
\dim_gset_eq:cN
 9266 \cs_generate_variant:Nn \dim_gset_eq:NN { c }
\dim_gset_eq:Nc
 9267 \cs_generate_variant:Nn \dim_gset_eq:NN { Nc , cc }
\dim_gset_eq:cc
```

(End definition for `\dim_set_eq:NN` and `\dim_gset_eq:NN`. These functions are documented on page 148.)

`\dim_add:Nn` Using `by` here deals with the (incorrect) case `\dimen123`.

```

\dim_add:cn 9268 \cs_new_protected:Npn \dim_add:Nn #1#2
\dim_gadd:Nn 9269 { \tex_advance:D #1 by \_dim_eval:w #2 \_dim_eval_end: }
\dim_gadd:cn 9270 \cs_new_protected:Npn \dim_gadd:Nn { \tex_global:D \dim_add:Nn }
\dim_sub:Nn 9271 \cs_generate_variant:Nn \dim_add:Nn { c }
\dim_sub:cn 9272 \cs_generate_variant:Nn \dim_gadd:Nn { c }
\dim_gsub:Nn 9273 \cs_new_protected:Npn \dim_sub:Nn #1#2
\dim_gsub:cn 9274 { \tex_advance:D #1 by - \_dim_eval:w #2 \_dim_eval_end: }
9275 \cs_new_protected:Npn \dim_gsub:Nn { \tex_global:D \dim_sub:Nn }
9276 \cs_generate_variant:Nn \dim_sub:Nn { c }
9277 \cs_generate_variant:Nn \dim_gsub:Nn { c }
```

(End definition for `\dim_add:Nn` and others. These functions are documented on page 148.)

18.4 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 9278 \cs_new:Npn \dim_abs:n #1
\__dim_max:nn 9279 {
\__dim_min:nn 9280   \exp_after:wN \__dim_abs:N
\__dim_maxmin:wwN 9281   \dim_use:N \_dim_eval:w #1 \_dim_eval_end:
9282 }
\__dim_max:nn 9283 \cs_new:Npn \__dim_abs:N #1
9284 { \if_meaning:w - #1 \else: \exp_after:wN #1 \fi: }
\__dim_max:nn 9285 \cs_new:Npn \dim_max:nn #1#2
9286 {
9287   \dim_use:N \_dim_eval:w \exp_after:wN \__dim_maxmin:wwN
9288   \dim_use:N \_dim_eval:w #1 \exp_after:wN ;
9289   \dim_use:N \_dim_eval:w #2 ;
9290   >
9291   \_dim_eval_end:
9292 }
\__dim_min:nn 9293 \cs_new:Npn \dim_min:nn #1#2
9294 {
9295   \dim_use:N \_dim_eval:w \exp_after:wN \__dim_maxmin:wwN
9296   \dim_use:N \_dim_eval:w #1 \exp_after:wN ;
9297   \dim_use:N \_dim_eval:w #2 ;
9298   <
9299   \_dim_eval_end:
9300 }
\__dim_maxmin:wwN 9301 \cs_new:Npn \__dim_maxmin:wwN #1 ; #2 ; #3
9302 {
9303   \if_dim:w #1 #3 #2 ~
9304   #1
9305   \else:
9306   #2
9307   \fi:
9308 }
```

(End definition for `\dim_abs:n` and others. These functions are documented on page 148.)

`\dim_ratio:nn` With dimension expressions, something like `10 pt * (5 pt / 10 pt)` will not work.
`_dim_ratio:n` Instead, the ratio part needs to be converted to an integer expression. Using `_int_value:w` forces everything into `sp`, avoiding any decimal parts.

```

9309 \cs_new:Npn \dim_ratio:nn #1#2
9310   { \_dim_ratio:n {#1} / \_dim_ratio:n {#2} }
9311 \cs_new:Npn \_dim_ratio:n #1
9312   { \_int_value:w \_dim_eval:w #1 \_dim_eval_end: }
```

(End definition for `\dim_ratio:nn` and `_dim_ratio:n`. These functions are documented on page 149.)

18.5 Dimension expression conditionals

`\dim_compare_p:nNn` Simple comparison.

`\dim_compare:nNnTF`

```

9313 \prg_new_conditional:Npnn \dim_compare:nNn #1#2#3 { p , T , F , TF }
9314   {
9315     \if_dim:w \_dim_eval:w #1 #2 \_dim_eval:w #3 \_dim_eval_end:
9316       \prg_return_true: \else: \prg_return_false: \fi:
9317   }
```

(End definition for `\dim_compare:nNnTF`. This function is documented on page 149.)

`\dim_compare_p:n`
`\dim_compare:nTF`
`_dim_compare:w`
`_dim_compare:wNN`
`_dim_compare_=:w`
`_dim_compare_!:w`
`_dim_compare_<:w`
`_dim_compare_>:w` 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 `_prg_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 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 `>`.

```

9318 \prg_new_conditional:Npnn \dim_compare:n { p , T , F , TF }
9319   {
9320     \exp_after:wN \_dim_compare:w
9321     \dim_use:N \_dim_eval:w #1 \_prg_compare_error:
9322   }
9323 \cs_new:Npn \_dim_compare:w #1 \_prg_compare_error:
9324   {
9325     \exp_after:wN \if_false: \exp:w \exp_end_continue_f:w
9326       \_dim_compare:wNN #1 ? { = \_dim_compare_end:w \else: } \q_stop
9327     }
9328 \exp_args:Nno \use:nn
9329   { \cs_new:Npn \_dim_compare:wNN #1 }
9330   { \tl_to_str:n {pt} }
9331 #2#3
9332   {
9333     \if_meaning:w = #3
9334       \use:c { \_dim_compare_#2:w }
9335     \fi:
9336       #1 pt \exp_stop_f:
9337     \prg_return_false:
9338     \exp_after:wN \use_none_delimit_by_q_stop:w
9339   \fi:
9340   \reverse_if:N \if_dim:w #1 pt #2
9341     \exp_after:wN \_dim_compare:wNN
9342     \dim_use:N \_dim_eval:w #3
```

```

9343   }
9344 \cs_new:cpn { __dim_compare_ ! :w }
9345   #1 \reverse_if:N #2 ! #3 = { #1 #2 = #3 }
9346 \cs_new:cpn { __dim_compare_ = :w }
9347   #1 \__dim_eval:w = { #1 \__dim_eval:w }
9348 \cs_new:cpn { __dim_compare_ < :w }
9349   #1 \reverse_if:N #2 < #3 = { #1 #2 > #3 }
9350 \cs_new:cpn { __dim_compare_ > :w }
9351   #1 \reverse_if:N #2 > #3 = { #1 #2 < #3 }
9352 \cs_new:Npn \__dim_compare_end:w #1 \prg_return_false: #2 \q_stop
9353   { #1 \prg_return_false: \else: \prg_return_true: \fi: }

```

(End definition for `\dim_compare:nTF` and others. These functions are documented on page 150.)

`\dim_case:nn` `\dim_case:nnTF`

For dimension cases, the first task to fully expand the check condition. The over all idea is then much the same as for `\str_case:nn(TF)` as described in l3basics.

```

9354 \cs_new:Npn \dim_case:nnTF #1
9355   {
9356     \exp:w
9357     \exp_args:Nf \__dim_case:nnTF { \dim_eval:n {#1} }
9358   }
9359 \cs_new:Npn \dim_case:nnT #1#2#3
9360   {
9361     \exp:w
9362     \exp_args:Nf \__dim_case:nnTF { \dim_eval:n {#1} } {#2} {#3} { }
9363   }
9364 \cs_new:Npn \dim_case:nnF #1#2
9365   {
9366     \exp:w
9367     \exp_args:Nf \__dim_case:nnTF { \dim_eval:n {#1} } {#2} { }
9368   }
9369 \cs_new:Npn \dim_case:nn #1#2
9370   {
9371     \exp:w
9372     \exp_args:Nf \__dim_case:nnTF { \dim_eval:n {#1} } {#2} { } { }
9373   }
9374 \cs_new:Npn \__dim_case:nnTF #1#2#3#4
9375   { \__dim_case:nw {#1} #2 {#1} { } \q_mark {#3} \q_mark {#4} \q_stop }
9376 \cs_new:Npn \__dim_case:nw #1#2#3
9377   {
9378     \dim_compare:nNnTF {#1} = {#2}
9379     { \__dim_case_end:nw {#3} }
9380     { \__dim_case:nw {#1} }
9381   }
9382 \cs_new_eq:NN \__dim_case_end:nw \__prg_case_end:nw

```

(End definition for `\dim_case:nnTF` and others. These functions are documented on page 151.)

18.6 Dimension expression loops

`\dim_while_do:nn`

`while_do` and `do_while` functions for dimensions. Same as for the `int` type only the names have changed.

`\dim_do_while:nn` `\dim_do_until:nn`

```

9383 \cs_new:Npn \dim_while_do:nn #1#2
9384   {

```

```

9385     \dim_compare:nT {#1}
9386     {
9387         #2
9388         \dim_while_do:nn {#1} {#2}
9389     }
9390 }
9391 \cs_new:Npn \dim_until_do:nn #1#2
9392 {
9393     \dim_compare:nF {#1}
9394     {
9395         #2
9396         \dim_until_do:nn {#1} {#2}
9397     }
9398 }
9399 \cs_new:Npn \dim_do_while:nn #1#2
9400 {
9401     #2
9402     \dim_compare:nT {#1}
9403     { \dim_do_while:nn {#1} {#2} }
9404 }
9405 \cs_new:Npn \dim_do_until:nn #1#2
9406 {
9407     #2
9408     \dim_compare:nF {#1}
9409     { \dim_do_until:nn {#1} {#2} }
9410 }

```

(End definition for `\dim_while_do:nn` and others. These functions are documented on page 152.)

`\dim_while_do:nNnn` `while_do` and `do_while` functions for dimensions. Same as for the `int` type only the names have changed.

```

9411 \cs_new:Npn \dim_while_do:nNnn #1#2#3#4
9412 {
9413     \dim_compare:nNnT {#1} #2 {#3}
9414     {
9415         #4
9416         \dim_while_do:nNnn {#1} #2 {#3} {#4}
9417     }
9418 }
9419 \cs_new:Npn \dim_until_do:nNnn #1#2#3#4
9420 {
9421     \dim_compare:nNnF {#1} #2 {#3}
9422     {
9423         #4
9424         \dim_until_do:nNnn {#1} #2 {#3} {#4}
9425     }
9426 }
9427 \cs_new:Npn \dim_do_while:nNnn #1#2#3#4
9428 {
9429     #4
9430     \dim_compare:nNnT {#1} #2 {#3}
9431     { \dim_do_while:nNnn {#1} #2 {#3} {#4} }
9432 }
9433 \cs_new:Npn \dim_do_until:nNnn #1#2#3#4

```

```

9434   {
9435     #4
9436     \dim_compare:nNnF {#1} #2 {#3}
9437     { \dim_do_until:nNnn {#1} #2 {#3} {#4} }
9438   }

```

(End definition for `\dim_while_do:nNnn` and others. These functions are documented on page 152.)

18.7 Using dim expressions and variables

`\dim_eval:n` Evaluating a dimension expression expandably.

```

9439 \cs_new:Npn \dim_eval:n #1
9440   { \dim_use:N \__dim_eval:w #1 \__dim_eval_end: }

```

(End definition for `\dim_eval:n`. This function is documented on page 152.)

`\dim_use:N` Accessing a $\langle\text{dim}\rangle$.

`\dim_use:c` $\dim_use:N$

We hand-code this for some speed gain:

```

9442 \%cs_generate_variant:Nn \dim_use:N { c }
9443 \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 152.)

`\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. 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.

```

9444 \cs_new:Npn \dim_to_decimal:n #1
9445   {
9446     \exp_after:wN
9447       \__dim_to_decimal:w \dim_use:N \__dim_eval:w (#1) \__dim_eval_end:
9448   }
9449 \use:x
9450   {
9451     \cs_new:Npn \exp_not:N \__dim_to_decimal:w
9452       ##1 . ##2 \tl_to_str:n { pt }
9453   }
9454   {
9455     \int_compare:nNnTF {#2} > { 0 }
9456       { #1 . #2 }
9457       { #1 }
9458   }

```

(End definition for `\dim_to_decimal:n` and `__dim_to_decimal:w`. These functions are documented on page 153.)

`\dim_to_decimal_in_bp:n` Conversion to big points is done using a scaling inside `__dim_eval:w` as ε-T_EX 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).

```

9459 \cs_new:Npn \dim_to_decimal_in_bp:n #1
9460   { \dim_to_decimal:n { ( #1 ) * 800 / 803 } }

```

(End definition for `\dim_to_decimal_in_bp:n`. This function is documented on page 153.)

\dim_to_decimal_in_sp:n Another hard-coded conversion: this one is necessary to avoid things going off-scale.

```
9461 \cs_new:Npn \dim_to_decimal_in_sp:n #1
9462   { \int_eval:n { \__dim_eval:w #1 \__dim_eval_end: } }
```

(End definition for `\dim_to_decimal_in_sp:n`. This function is documented on page 153.)

\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.

```
9463 \cs_new:Npn \dim_to_decimal_in_unit:nn #1#2
9464   {
9465     \dim_to_decimal:n
9466     {
9467       1pt *
9468       \dim_ratio:nn {#1} {#2}
9469     }
9470   }
```

(End definition for `\dim_to_decimal_in_unit:nn`. This function is documented on page 153.)

\dim_to_fp:n Defined in `l3fp-convert`, documented here.

(End definition for `\dim_to_fp:n`. This function is documented on page 154.)

18.8 Viewing dim variables

\dim_show:N Diagnostics.

```
9471 \cs_new_eq:NN \dim_show:N \__kernel_register_show:N
9472 \cs_generate_variant:Nn \dim_show:N { c }
```

(End definition for `\dim_show:N`. This function is documented on page 154.)

\dim_show:c Diagnostics. We don't use the TeX primitive `\showthe` to show dimension expressions: this gives a more unified output.

```
9473 \cs_new_protected:Npn \dim_show:n
9474   { \__msg_show_wrap:Nn \dim_eval:n }
```

(End definition for `\dim_show:n`. This function is documented on page 154.)

\dim_log:N Diagnostics. Redirect output of `\dim_show:n` to the log.

```
9475 \cs_new_eq:NN \dim_log:N \__kernel_register_log:N
9476 \cs_new_eq:NN \dim_log:c \__kernel_register_log:c
9477 \cs_new_protected:Npn \dim_log:n
9478   { \__msg_log_next: \dim_show:n }
```

(End definition for `\dim_log:N` and `\dim_log:n`. These functions are documented on page 154.)

18.9 Constant dimensions

\c_zero_dim Constant dimensions.

```
9479 \dim_const:Nn \c_zero_dim { 0 pt }
9480 \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 154.)

18.10 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

```
9481 \dim_new:N \l_tmpa_dim
9482 \dim_new:N \l_tmpb_dim
9483 \dim_new:N \g_tmpa_dim
9484 \dim_new:N \g_tmpb_dim
```

(End definition for `\l_tmpa_dim` and others. These variables are documented on page 154.)

18.11 Creating and initialising skip variables

\skip_new:N Allocation of a new internal registers.

```
9485 {*package}
9486 \cs_new_protected:Npn \skip_new:N #1
9487 {
9488     \__chk_if_free_cs:N #1
9489     \cs:w newskip \cs_end: #1
9490 }
9491 
```

`\cs_generate_variant:Nn \skip_new:N { c }`

(End definition for `\skip_new:N`. This function is documented on page 155.)

\skip_const:Nn Contrarily to integer constants, we cannot avoid using a register, even for constants.

```
9493 \cs_new_protected:Npn \skip_const:Nn #1
9494 {
9495     \skip_new:N #1
9496     \skip_gset:Nn #1
9497 }
9498 
```

`\cs_generate_variant:Nn \skip_const:Nn { c }`

(End definition for `\skip_const:Nn`. This function is documented on page 155.)

\skip_zero:N Reset the register to zero.

```
9499 \cs_new_protected:Npn \skip_zero:N #1 { #1 \c_zero_skip }
9500 \cs_new_protected:Npn \skip_gzero:N { \tex_global:D \skip_zero:N }
9501 \cs_generate_variant:Nn \skip_zero:N { c }
9502 \cs_generate_variant:Nn \skip_gzero:N { c }
```

(End definition for `\skip_zero:N` and `\skip_gzero:N`. These functions are documented on page 155.)

\skip_zero_new:N Create a register if needed, otherwise clear it.

```
9503 \cs_new_protected:Npn \skip_zero_new:N #1
9504     { \skip_if_exist:NTF #1 { \skip_zero:N #1 } { \skip_new:N #1 } }
9505 \cs_new_protected:Npn \skip_gzero_new:N #1
9506     { \skip_if_exist:NTF #1 { \skip_gzero:N #1 } { \skip_new:N #1 } }
9507 \cs_generate_variant:Nn \skip_zero_new:N { c }
9508 \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 155.)

```

\skip_if_exist_p:N Copies of the cs functions defined in l3basics.
\skip_if_exist_p:c 9509 \prg_new_eq_conditional:NNN \skip_if_exist:N \cs_if_exist:N
\skip_if_exist:NTF 9510 { TF , T , F , p }
\skip_if_exist:cTF 9511 \prg_new_eq_conditional:NNN \skip_if_exist:c \cs_if_exist:c
9512 { TF , T , F , p }

```

(End definition for `\skip_if_exist:NTF`. This function is documented on page 155.)

18.12 Setting skip variables

`\skip_set:Nn` Much the same as for dimensions.

```

\skip_set:cn 9513 \cs_new_protected:Npn \skip_set:Nn #1#2
\skip_gset:Nn 9514 { #1 ~ \etex_glueexpr:D #2 \scan_stop: }
\skip_gset:cn 9515 \cs_new_protected:Npn \skip_gset:Nn { \tex_global:D \skip_set:Nn }
9516 \cs_generate_variant:Nn \skip_set:Nn { c }
9517 \cs_generate_variant:Nn \skip_gset:Nn { c }

```

(End definition for `\skip_set:Nn` and `\skip_gset:Nn`. These functions are documented on page 155.)

`\skip_set_eq:NN` All straightforward.

```

\skip_set_eq:cN 9518 \cs_new_protected:Npn \skip_set_eq:NN #1#2 { #1 = #2 }
\skip_set_eq:Nc 9519 \cs_generate_variant:Nn \skip_set_eq:NN { c }
\skip_set_eq:cc 9520 \cs_generate_variant:Nn \skip_set_eq:NN { Nc , cc }
\skip_gset_eq:NN 9521 \cs_new_protected:Npn \skip_gset_eq:NN #1#2 { \tex_global:D #1 = #2 }
\skip_gset_eq:cN 9522 \cs_generate_variant:Nn \skip_gset_eq:NN { c }
\skip_gset_eq:Nc 9523 \cs_generate_variant:Nn \skip_gset_eq:NN { Nc , cc }

```

(End definition for `\skip_set_eq:NN` and `\skip_gset_eq:NN`. These functions are documented on page 155.)

`\skip_add:Nn` Using by here deals with the (incorrect) case `\skip123`.

```

\skip_add:cn 9524 \cs_new_protected:Npn \skip_add:Nn #1#2
\skip_gadd:Nn 9525 { \tex_advance:D #1 by \etex_glueexpr:D #2 \scan_stop: }
\skip_gadd:cn 9526 \cs_new_protected:Npn \skip_gadd:Nn { \tex_global:D \skip_add:Nn }
\skip_sub:Nn 9527 \cs_generate_variant:Nn \skip_add:Nn { c }
\skip_sub:cn 9528 \cs_generate_variant:Nn \skip_gadd:Nn { c }
\skip_gsub:Nn 9529 \cs_new_protected:Npn \skip_sub:Nn #1#2
\skip_gsub:cn 9530 { \tex_advance:D #1 by - \etex_glueexpr:D #2 \scan_stop: }
9531 \cs_new_protected:Npn \skip_gsub:Nn { \tex_global:D \skip_sub:Nn }
9532 \cs_generate_variant:Nn \skip_sub:Nn { c }
9533 \cs_generate_variant:Nn \skip_gsub:Nn { c }

```

(End definition for `\skip_add:Nn` and others. These functions are documented on page 155.)

18.13 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.

```

9534 \prg_new_conditional:Npnn \skip_if_eq:nn #1#2 { p , T , F , TF }
9535 {
9536   \if_int_compare:w
9537     \_str_if_eq_x:nn { \skip_eval:n { #1 } } { \skip_eval:n { #2 } }
9538     = 0 \exp_stop_f:
9539       \prg_return_true:

```

```

9540     \else:
9541         \prg_return_false:
9542     \fi:
9543 }

```

(End definition for `\skip_if_eq:nnTF`. This function is documented on page 156.)

`\skip_if_finite:p:n` `\skip_if_finite:nTF`
`_skip_if_finite:wwNw`

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.

```

9544 \cs_set_protected:Npn \__cs_tmp:w #1
9545 {
9546     \prg_new_conditional:Npnn \skip_if_finite:n ##1 { p , T , F , TF }
9547     {
9548         \exp_after:wN \__skip_if_finite:wwNw
9549         \skip_use:N \etex_glueexpr:D ##1 ; \prg_return_false:
9550         #1 ; \prg_return_true: \q_stop
9551     }
9552     \cs_new:Npn \__skip_if_finite:wwNw ##1 #1 ##2 ; ##3 ##4 \q_stop {##3}
9553 }
9554 \exp_args:No \__cs_tmp:w { \tl_to_str:n { fil } }

```

(End definition for `\skip_if_finite:nTF` and `_skip_if_finite:wwNw`. These functions are documented on page 156.)

18.14 Using skip expressions and variables

`\skip_eval:n` Evaluating a skip expression expandably.

```

9555 \cs_new:Npn \skip_eval:n #1
9556     { \skip_use:N \etex_glueexpr:D #1 \scan_stop: }

```

(End definition for `\skip_eval:n`. This function is documented on page 156.)

`\skip_use:N` Accessing a `\langle skip\rangle`.

```

9557 \cs_new_eq:NN \skip_use:N \tex_the:D
9558 \%cs_generate_variant:Nn \skip_use:N { c }
9559 \cs_new:Npn \skip_use:c #1 { \tex_the:D \cs:w #1 \cs_end: }

```

(End definition for `\skip_use:N`. This function is documented on page 156.)

18.15 Inserting skips into the output

`\skip_horizontal:N` Inserting skips.

```

9560 \cs_new_eq:NN \skip_horizontal:N \tex_hskip:D
9561 \cs_new:Npn \skip_horizontal:n #1
9562     { \skip_horizontal:N \etex_glueexpr:D #1 \scan_stop: }
9563 \cs_new_eq:NN \skip_vertical:N \tex_vskip:D
9564 \cs_new:Npn \skip_vertical:n #1
9565     { \skip_vertical:N \etex_glueexpr:D #1 \scan_stop: }
9566 \%cs_generate_variant:Nn \skip_horizontal:N { c }
9567 \%cs_generate_variant:Nn \skip_vertical:N { c }

```

(End definition for `\skip_horizontal:N` and others. These functions are documented on page 157.)

18.16 Viewing skip variables

\skip_show:N Diagnostics.
9568 \cs_new_eq:NN \skip_show:N _kernel_register_show:N
9569 \cs_generate_variant:Nn \skip_show:N { c }
(End definition for `\skip_show:N`. This function is documented on page 156.)

\skip_show:c Diagnostics. We don't use the TeX primitive `\showthe` to show skip expressions: this gives a more unified output.
9570 \cs_new_protected:Npn \skip_show:n
9571 { _msg_show_wrap:Nn \skip_eval:n }
(End definition for `\skip_show:n`. This function is documented on page 157.)

\skip_log:N Diagnostics. Redirect output of `\skip_show:n` to the log.
9572 \cs_new_eq:NN \skip_log:N _kernel_register_log:N
9573 \cs_new_eq:NN \skip_log:c _kernel_register_log:c
9574 \cs_new_protected:Npn \skip_log:n
9575 { _msg_log_next: \skip_show:n }
(End definition for `\skip_log:N` and `\skip_log:n`. These functions are documented on page 157.)

18.17 Constant skips

\c_zero_skip Skips with no rubber component are just dimensions but need to terminate correctly.
9576 \skip_const:Nn \c_zero_skip { \c_zero_dim }
9577 \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 157.)

18.18 Scratch skips

\l_tmpa_skip We provide two local and two global scratch registers, maybe we need more or less.
\l_tmpb_skip
9578 \skip_new:N \l_tmpa_skip
\g_tmpa_skip
9579 \skip_new:N \l_tmpb_skip
\g_tmpb_skip
9580 \skip_new:N \g_tmpa_skip
9581 \skip_new:N \g_tmpb_skip
(End definition for `\l_tmpa_skip` and others. These variables are documented on page 157.)

18.19 Creating and initialising muskip variables

\muskip_new:N And then we add muskips.
9582 (*package)
9583 \cs_new_protected:Npn \muskip_new:N #1
9584 {
9585 _chk_if_free_cs:N #1
9586 \cs:w newmuskip \cs_end: #1
9587 }
9588

9589 \cs_generate_variant:Nn \muskip_new:N { c }

(End definition for `\muskip_new:N`. This function is documented on page 158.)

\muskip_const:Nn Contrarily to integer constants, we cannot avoid using a register, even for constants.

```
9590 \cs_new_protected:Npn \muskip_const:Nn #1
9591 {
9592     \muskip_new:N #1
9593     \muskip_gset:Nn #1
9594 }
9595 \cs_generate_variant:Nn \muskip_const:Nn { c }
```

(End definition for **\muskip_const:Nn**. This function is documented on page 158.)

\muskip_zero:N Reset the register to zero.

```
9596 \cs_new_protected:Npn \muskip_zero:N #1
9597 { #1 \c_zero_muskip }
9598 \cs_new_protected:Npn \muskip_gzero:N { \tex_global:D \muskip_zero:N }
9599 \cs_generate_variant:Nn \muskip_zero:N { c }
9600 \cs_generate_variant:Nn \muskip_gzero:N { c }
```

(End definition for **\muskip_zero:N** and **\muskip_gzero:N**. These functions are documented on page 158.)

\muskip_zero_new:N Create a register if needed, otherwise clear it.

```
9601 \cs_new_protected:Npn \muskip_zero_new:N #1
9602 { \muskip_if_exist:NTF #1 { \muskip_zero:N #1 } { \muskip_new:N #1 } }
9603 \cs_new_protected:Npn \muskip_gzero_new:N #1
9604 { \muskip_if_exist:NTF #1 { \muskip_gzero:N #1 } { \muskip_new:N #1 } }
9605 \cs_generate_variant:Nn \muskip_zero_new:N { c }
9606 \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 158.)

\muskip_if_exist_p:N Copies of the **cs** functions defined in l3basics.

```
9607 \prg_new_eq_conditional:NNN \muskip_if_exist:N \cs_if_exist:N
9608 { TF , T , F , p }
9609 \prg_new_eq_conditional:NNN \muskip_if_exist:c \cs_if_exist:c
9610 { TF , T , F , p }
```

(End definition for **\muskip_if_exist:NTF**. This function is documented on page 158.)

18.20 Setting muskip variables

\muskip_set:Nn This should be pretty familiar.

```
9611 \cs_new_protected:Npn \muskip_set:Nn #1#2
9612 { #1 ~ \etex_muexpr:D #2 \scan_stop: }
9613 \cs_new_protected:Npn \muskip_gset:Nn { \tex_global:D \muskip_set:Nn }
9614 \cs_generate_variant:Nn \muskip_set:Nn { c }
9615 \cs_generate_variant:Nn \muskip_gset:Nn { c }
```

(End definition for **\muskip_set:Nn** and **\muskip_gset:Nn**. These functions are documented on page 159.)

\muskip_set_eq:NN All straightforward.

```
9616 \cs_new_protected:Npn \muskip_set_eq:NN #1#2 { #1 = #2 }
9617   \cs_generate_variant:Nn \muskip_set_eq:NN { c }
9618   \cs_generate_variant:Nn \muskip_set_eq:NN { Nc , cc }
9619 \cs_new_protected:Npn \muskip_gset_eq:NN #1#2 { \tex_global:D #1 = #2 }
9620   \cs_generate_variant:Nn \muskip_gset_eq:NN { c }
9621   \cs_generate_variant:Nn \muskip_gset_eq:NN { Nc , cc }
```

(End definition for `\muskip_set_eq:NN` and `\muskip_gset_eq:NN`. These functions are documented on page 159.)

\muskip_add:Nn Using by here deals with the (incorrect) case `\muskip123`.

```
9622 \cs_new_protected:Npn \muskip_add:Nn #1#2
9623   { \tex_advance:D #1 by \etex_muexpr:D #2 \scan_stop: }
9624 \cs_new_protected:Npn \muskip_gadd:Nn { \tex_global:D \muskip_add:Nn }
9625   \cs_generate_variant:Nn \muskip_add:Nn { c }
9626 \cs_generate_variant:Nn \muskip_gadd:Nn { c }
9627 \cs_new_protected:Npn \muskip_sub:Nn #1#2
9628   { \tex_advance:D #1 by - \etex_muexpr:D #2 \scan_stop: }
9629 \cs_new_protected:Npn \muskip_gsub:Nn { \tex_global:D \muskip_sub:Nn }
9630   \cs_generate_variant:Nn \muskip_sub:Nn { c }
9631 \cs_generate_variant:Nn \muskip_gsub:Nn { c }
```

(End definition for `\muskip_add:Nn` and others. These functions are documented on page 158.)

18.21 Using muskip expressions and variables

\muskip_eval:n Evaluating a muskip expression expandably.

```
9632 \cs_new:Npn \muskip_eval:n #1
9633   { \muskip_use:N \etex_muexpr:D #1 \scan_stop: }
```

(End definition for `\muskip_eval:n`. This function is documented on page 159.)

\muskip_use:N Accessing a `\{muskip\}`.

```
9634 \cs_new_eq:NN \muskip_use:N \tex_the:D
9635   \cs_generate_variant:Nn \muskip_use:N { c }
```

(End definition for `\muskip_use:N`. This function is documented on page 159.)

18.22 Viewing muskip variables

\muskip_show:N Diagnostics.

```
9636 \cs_new_eq:NN \muskip_show:N \__kernel_register_show:N
9637   \cs_generate_variant:Nn \muskip_show:N { c }
```

(End definition for `\muskip_show:N`. This function is documented on page 159.)

\muskip_show:n Diagnostics. We don't use the TeX primitive `\showthe` to show muskip expressions: this gives a more unified output.

```
9638 \cs_new_protected:Npn \muskip_show:n
9639   { \__msg_show_wrap:Nn \muskip_eval:n }
```

(End definition for `\muskip_show:n`. This function is documented on page 159.)

```

\muskip_log:N  Diagnostics. Redirect output of \muskip_show:n to the log.
\muskip_log:c
\muskip_log:n  9640 \cs_new_eq:NN \muskip_log:N \__kernel_register_log:N
                9641 \cs_new_eq:NN \muskip_log:c \__kernel_register_log:c
                9642 \cs_new_protected:Npn \muskip_log:n
                9643 { \__msg_log_next: \muskip_show:n }

```

(End definition for `\muskip_log:N` and `\muskip_log:n`. These functions are documented on page 160.)

18.23 Constant muskips

```

\c_zero_muskip  Constant muskips given by their value.
\c_max_muskip   9644 \muskip_const:Nn \c_zero_muskip { 0 mu }
                  9645 \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 160.)

18.24 Scratch muskips

```

\l_tmpa_muskip
\l_tmpb_muskip
\g_tmpa_muskip
\g_tmpb_muskip  9646 \muskip_new:N \l_tmpa_muskip
                  9647 \muskip_new:N \l_tmpb_muskip
                  9648 \muskip_new:N \g_tmpa_muskip
                  9649 \muskip_new:N \g_tmpb_muskip

```

(End definition for `\l_tmpa_muskip` and others. These variables are documented on page 160.)

9650 ⟨/initex | package⟩

19 I3keys Implementation

9651 ⟨*initex | package⟩

19.1 Low-level interface

The low-level key parser is based heavily on `keyval`, but with a number of additional “safety” requirements and with the idea that the parsed list of key–value pairs can be processed in a variety of ways. The net result is that this code needs around twice 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.

9652 ⟨@=keyval⟩

```

\l__keyval_key_tl  The current key name and value.
\l__keyval_value_tl  9653 \tl_new:N \l__keyval_key_tl
                     9654 \tl_new:N \l__keyval_value_tl

```

(End definition for `\l__keyval_key_tl` and `\l__keyval_value_tl`.)

```

\l__keyval_sanitise_tl  A token list variable for dealing with awkward category codes in the input.
                         9655 \tl_new:N \l__keyval_sanitise_tl

```

(End definition for `\l__keyval_sanitise_tl`.)

\keyval_parse:NNn

The main function starts off by normalising category codes in package mode. That's relatively "expensive" so is skipped (hopefully) in format mode. We then hand off to the parser. The use of \q_mark here prevents loss of braces from the key argument. This particular quark is chosen as it fits in with _tl_trim_spaces:nn and allows a performance enhancement as the token can be carried through. Notice that by passing the two processor commands along the input stack we avoid the need to track these at all.

```
9656 \cs_new_protected:Npn \keyval_parse:NNn #1#2#3
9657 {
9658 (*initex)
9659   \__keyval_loop>NNw #1#2 \q_mark #3 , \q_recursion_tail ,
9660 
```

```
9661 (*package)
9662   \tl_set:Nn \l__keyval_sanitise_tl {#3}
9663   \__keyval_sanitise_equals:
9664   \__keyval_sanitise_comma:
9665   \exp_after:wN \__keyval_loop>NNw \exp_after:wN #1 \exp_after:wN #2
9666   \exp_after:wN \q_mark \l__keyval_sanitise_tl , \q_recursion_tail ,
9667 
```

```
9668 }
```

(End definition for \keyval_parse:NNn. This function is documented on page 174.)

__keyval_sanitise_equals:

A reasonably fast search and replace set up specifically for the active tokens. The nature of the input is known so everything is hard-coded. With only two tokens to cover, the speed gain from using dedicated functions is worth it.

```
9669 (*package)
9670 \group_begin:
9671   \char_set_catcode_active:n { '= }
9672   \char_set_catcode_active:n { '\, }
9673 \cs_new_protected:Npn \__keyval_sanitise_equals:
9674 {
9675   \exp_after:wN \__keyval_sanitise_equals_auxi:w \l__keyval_sanitise_tl
9676   \q_mark = \q_nil =
9677   \exp_after:wN \__keyval_sanitise_aux:w \l__keyval_sanitise_tl
9678 }
9679 \cs_new_protected:Npn \__keyval_sanitise_equals_auxi:w #1 =
9680 {
9681   \tl_set:Nn \l__keyval_sanitise_tl {#1}
9682   \__keyval_sanitise_equals_auxii:w
9683 }
9684 \cs_new_protected:Npn \__keyval_sanitise_equals_auxii:w #1 =
9685 {
9686   \if_meaning:w \q_nil #1 \scan_stop:
9687   \else:
9688     \tl_set:Nx \l__keyval_sanitise_tl
9689     {
9690       \exp_not:o \l__keyval_sanitise_tl
9691       \token_to_str:N =
9692       \exp_not:n {#1}
9693     }
9694   \exp_after:wN \__keyval_sanitise_equals_auxii:w
9695 }
```

```

9696     }
9697   \cs_new_protected:Npn \__keyval_sanitise_comma:
9698   {
9699     \exp_after:wN \__keyval_sanitise_comma_auxi:w \l__keyval_sanitise_tl
9700     \q_mark , \q_nil ,
9701     \exp_after:wN \__keyval_sanitise_aux:w \l__keyval_sanitise_tl
9702   }
9703   \cs_new_protected:Npn \__keyval_sanitise_comma_auxi:w #1 ,
9704   {
9705     \tl_set:Nn \l__keyval_sanitise_tl {#1}
9706     \__keyval_sanitise_comma_auxii:w
9707   }
9708   \cs_new_protected:Npn \__keyval_sanitise_comma_auxii:w #1 ,
9709   {
9710     \if_meaning:w \q_nil #1 \scan_stop:
9711     \else:
9712       \tl_set:Nx \l__keyval_sanitise_tl
9713       {
9714         \exp_not:o \l__keyval_sanitise_tl
9715         \token_to_str:N ,
9716         \exp_not:n {#1}
9717       }
9718       \exp_after:wN \__keyval_sanitise_comma_auxii:w
9719       \fi:
9720   }
9721 \group_end:
9722 \cs_new_protected:Npn \__keyval_sanitise_aux:w #1 \q_mark
9723 { \tl_set:Nn \l__keyval_sanitise_tl {#1} }
9724 
```

(End definition for `__keyval_sanitise_equals:` and others.)

`__keyval_loop>NNw`

A fast test for the end of the loop, remembering to remove the leading quark first. Assuming that is not the case, look for a key and value then loop around, re-inserting a leading quark in front of the next position.

```

9725 \cs_new_protected:Npn \__keyval_loop>NNw #1#2#3 ,
9726   {
9727     \exp_after:wN \if_meaning:w \exp_after:wN \q_recursion_tail
9728     \use_none:n #3 \prg_do_nothing:
9729     \else:
9730       \__keyval_split>NNw #1#2#3 == \q_stop
9731       \exp_after:wN \__keyval_loop>NNw \exp_after:wN #1 \exp_after:wN #2
9732         \exp_after:wN \q_mark
9733     \fi:
9734   }

```

(End definition for `__keyval_loop>NNw`.)

`__keyval_split>NNw`

`__keyval_split_value>NNw`

`__keyval_split_tidy:w`
`__keyval_action:`

The value is picked up separately from the key so there can be another quark inserted at the front, keeping braces and allowing both parts to share the same code paths. The key is found first then there's a check that there is something there: this is biased to the common case of there actually being a key. For the value, we first need to see if there is anything to do: if there is, extract it. The appropriate action is then inserted in front

of the key and value. Doing this using an assignment is marginally faster than an an expansion chain.

```

9735 \cs_new_protected:Npn \__keyval_split>NNw #1#2#3 =
9736   {
9737     \__keyval_def:Nn \l__keyval_key_tl {#3}
9738     \if_meaning:w \l__keyval_key_tl \c_empty_tl
9739       \exp_after:wN \__keyval_split_tidy:w
9740     \else:
9741       \exp_after:wN \__keyval_split_value>NNw \exp_after:wN #1 \exp_after:wN #2
9742       \exp_after:wN \q_mark
9743     \fi:
9744   }
9745 \cs_new_protected:Npn \__keyval_split_value>NNw #1#2#3 = #4 \q_stop
9746   {
9747     \if:w \scan_stop: \tl_to_str:n {#4} \scan_stop:
9748     \cs_set:Npx \__keyval_action:
9749       { \exp_not:N #1 { \exp_not:o \l__keyval_key_tl } }
9750   \else:
9751     \if:w \scan_stop: \etex_detokenize:D \exp_after:wN { \use_none:n #4 }
9752       \scan_stop:
9753       \__keyval_def:Nn \l__keyval_value_tl {#3}
9754       \cs_set:Npx \__keyval_action:
9755         {
9756           \exp_not:N #2
9757             { \exp_not:o \l__keyval_key_tl }
9758             { \exp_not:o \l__keyval_value_tl }
9759         }
9760       \else:
9761         \cs_set:Npn \__keyval_action:
9762           { \__msg_kernel_error:nn { kernel } { misplaced-equals-sign } }
9763         \fi:
9764       \fi:
9765       \__keyval_action:
9766   }
9767 \cs_new_protected:Npn \__keyval_split_tidy:w #1 \q_stop
9768   {
9769     \if:w \scan_stop: \etex_detokenize:D \exp_after:wN { \use_none:n #1 }
9770       \scan_stop:
9771     \else:
9772       \exp_after:wN \__keyval_empty_key:
9773     \fi:
9774   }
9775 \cs_new:Npn \__keyval_action: { }
9776 \cs_new_protected:Npn \__keyval_empty_key:
9777   { \__msg_kernel_error:nn { kernel } { misplaced-equals-sign } }

(End definition for \__keyval_split>NNw and others.)

```

__keyval_def:Nn
__keyval_def_aux:n
__keyval_def_aux:w

First trim spaces off, then potentially remove a set of braces. By using the internal interface __tl_trim_spaces:nn we can take advantage of the fact it needs a leading \q_mark in this process. The \exp_after:wN removes the quark, the delimited argument deals with any braces.

```

9778 \cs_new_protected:Npn \__keyval_def:Nn #1#2
9779   { \tl_set:Nx #1 { \__tl_trim_spaces:nn {#2} \__keyval_def_aux:n } }

```

```

9780 \cs_new:Npn \__keyval_def_aux:n #1
9781   { \exp_after:wN \__keyval_def_aux:w #1 \q_stop }
9782 \cs_new:Npn \__keyval_def_aux:w #1 \q_stop { \exp_not:n {#1} }

(End definition for \__keyval_def:Nn, \__keyval_def_aux:n, and \__keyval_def_aux:w.)
```

One message for the low level parsing system.

```

9783 \__msg_kernel_new:nnnn { kernel } { misplaced-equals-sign }
9784   { Misplaced~equals~sign~in~key-value~input~\msg_line_number: }
9785   {
9786     LaTeX~is~attempting~to~parse~some~key-value~input~but~found~
9787     two~equals~signs~not~separated~by~a~comma.
9788 }
```

19.2 Constants and variables

9789 ⟨@@=keys⟩

\c_keys_code_root_tl Various storage areas for the different data which make up keys.
\c_keys_default_root_tl
\c_keys_groups_root_tl
\c_keys_inherit_root_tl
\c_keys_type_root_tl
\c_keys_validate_root_tl

```

9790 \tl_const:Nn \c_keys_code_root_tl { key~code~>~ }
9791 \tl_const:Nn \c_keys_default_root_tl { key~default~>~ }
9792 \tl_const:Nn \c_keys_groups_root_tl { key~groups~>~ }
9793 \tl_const:Nn \c_keys_inherit_root_tl { key~inherit~>~ }
9794 \tl_const:Nn \c_keys_type_root_tl { key~type~>~ }
9795 \tl_const:Nn \c_keys_validate_root_tl { key~validate~>~ }
```

(End definition for \c_keys_code_root_tl and others.)

\c_keys_props_root_tl The prefix for storing properties.

```
9796 \tl_const:Nn \c_keys_props_root_tl { key~prop~>~ }
```

(End definition for \c_keys_props_root_tl.)

\l_keys_choice_int Publicly accessible data on which choice is being used when several are generated as a set.
\l_keys_choice_tl

```

9797 \int_new:N \l_keys_choice_int
9798 \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 168.)

\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.

```
9799 \clist_new:N \l_keys_groups_clist
```

(End definition for \l_keys_groups_clist.)

\l_keys_key_tl The name of a key itself: needed when setting keys.

```
9800 \tl_new:N \l_keys_key_tl
```

(End definition for \l_keys_key_tl. This variable is documented on page 170.)

\l_keys_module_tl The module for an entire set of keys.

```
9801 \tl_new:N \l_keys_module_tl
```

(End definition for \l_keys_module_tl.)

`\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.

9802 `\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.

9803 `\bool_new:N \l__keys_only_known_bool`

(*End definition for \l__keys_only_known_bool.*)

\l_keys_path_t1 The “path” of the current key is stored here: this is available to the programmer and so is public.

9804 `\tl_new:N \l_keys_path_t1`

(*End definition for \l_keys_path_t1. This variable is documented on page 170.*)

`\l__keys_property_t1` The “property” begin set for a key at definition time is stored here.

9805 `\tl_new:N \l__keys_property_t1`

(*End definition for \l__keys_property_t1.*)

`\l__keys_selective_bool` Two flags for using key groups: one to indicate that “selective” setting is active, a second `\l__keys_filtered_bool` to specify which type (“opt-in” or “opt-out”).

9806 `\bool_new:N \l__keys_selective_bool`

9807 `\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.

9808 `\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.

9809 `\tl_new:N \l__keys_unused_clist`

(*End definition for \l__keys_unused_clist.*)

\l_keys_value_t1 The value given for a key: may be empty if no value was given.

9810 `\tl_new:N \l_keys_value_t1`

(*End definition for \l_keys_value_t1. This variable is documented on page 170.*)

`\l__keys_tmp_bool` Scratch space.

9811 `\bool_new:N \l__keys_tmp_bool`

(*End definition for \l__keys_tmp_bool.*)

19.3 The key defining mechanism

\keys_define:nn
_keys_define:nnn
_keys_define:onn

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).

```

9812 \cs_new_protected:Npn \keys_define:nn
9813   { \_keys_define:onn \l__keys_module_tl }
9814 \cs_new_protected:Npn \_keys_define:nnn #1#2#3
9815   {
9816     \tl_set:Nx \l__keys_module_tl { \_keys_remove_spaces:n {#2} }
9817     \keyval_parse:NNn \_keys_define:n \_keys_define:nn {#3}
9818     \tl_set:Nn \l__keys_module_tl {#1}
9819   }
9820 \cs_generate_variant:Nn \_keys_define:nnn { o }
```

(End definition for \keys_define:nn and _keys_define:nnn. These functions are documented on page 163.)

_keys_define:n
_keys_define:nn
_keys_define_aux:nn

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.

```

9821 \cs_new_protected:Npn \_keys_define:n #1
9822   {
9823     \bool_set_true:N \l__keys_no_value_bool
9824     \_keys_define_aux:nn {#1} { }
9825   }
9826 \cs_new_protected:Npn \_keys_define:nn #1#2
9827   {
9828     \bool_set_false:N \l__keys_no_value_bool
9829     \_keys_define_aux:nn {#1} {#2}
9830   }
9831 \cs_new_protected:Npn \_keys_define_aux:nn #1#2
9832   {
9833     \_keys_property_find:n {#1}
9834     \cs_if_exist:cTF { \c__keys_props_root_tl } \l__keys_property_tl
9835     { \_keys_define_code:n {#2} }
9836     {
9837       \tl_if_empty:NF \l__keys_property_tl
9838       {
9839         \__msg_kernel_error:nnxx { kernel } { property-unknown }
9840         { \l__keys_property_tl } { \l__keys_path_tl }
9841       }
9842     }
9843   }
```

(End definition for _keys_define:n, _keys_define:nn, and _keys_define_aux:nn.)

_keys_property_find:n
_keys_property_find:w

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.

```

9844 \cs_new_protected:Npn \_keys_property_find:n #1
9845   {
9846     \tl_set:Nx \l__keys_property_tl { \_keys_remove_spaces:n {#1} }
9847     \exp_after:wN \_keys_property_find:w \l__keys_property_tl . . \q_stop {#1}
```

```

9848     }
9849 \cs_new_protected:Npn \__keys_property_find:w #1 . #2 . #3 \q_stop #4
9850 {
9851     \tl_if_blank:nTF {#3}
9852     {
9853         \tl_clear:N \l__keys_property_tl
9854         \__msg_kernel_error:nnn { kernel } { key-no-property } {#4}
9855     }
9856     {
9857         \str_if_eq:nnTF {#3} { . }
9858         {
9859             \tl_set:Nx \l__keys_path_tl
9860             {
9861                 \tl_if_empty:NF \l__keys_module_tl
9862                     { \l__keys_module_tl / }
9863                     #1
9864             }
9865             \tl_set:Nn \l__keys_property_tl { . #2 }
9866         }
9867         {
9868             \tl_set:Nx \l__keys_path_tl { \l__keys_module_tl / #1 . #2 }
9869             \__keys_property_search:w #3 \q_stop
9870         }
9871     }
9872 }
9873 \cs_new_protected:Npn \__keys_property_search:w #1 . #2 \q_stop
9874 {
9875     \str_if_eq:nnTF {#2} { . }
9876     {
9877         \tl_set:Nx \l__keys_path_tl { \l__keys_path_tl }
9878         \tl_set:Nn \l__keys_property_tl { . #1 }
9879     }
9880     {
9881         \tl_set:Nx \l__keys_path_tl { \l__keys_path_tl . #1 }
9882         \__keys_property_search:w #2 \q_stop
9883     }
9884 }

```

(End definition for `__keys_property_find:n` and `__keys_property_find:w`.)

`__keys_define_code:n` Two possible cases. If there is a value for the key, then just use the function. If not, then 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 is missing the earlier tests will have failed.

```

9885 \cs_new_protected:Npn \__keys_define_code:n #1
9886 {
9887     \bool_if:NTF \l__keys_no_value_bool
9888     {
9889         \exp_after:wN \__keys_define_code:w
9890             \l__keys_property_tl \q_stop
9891             { \use:c { \c__keys_props_root_tl \l__keys_property_tl } }
9892     }
9893         \__msg_kernel_error:nnxx { kernel }
9894             { property-requires-value } { \l__keys_property_tl }

```

```

9895             { \l_keys_path_t1 }
9896         }
9897     }
9898     { \use:c { \c__keys_props_root_t1 \l__keys_property_t1 } {#1} }
9899   }
9900 \use:x
9901   {
9902     \cs_new:Npn \exp_not:N \__keys_define_code:w
9903       ##1 \c_colon_str ##2 \exp_not:N \q_stop
9904   }
9905   { \tl_if_empty:nTF {#2} }

```

(End definition for `__keys_define_code:n` and `__keys_define_code:w`.)

19.4 Turning properties into actions

`__keys_bool_set:Nn` Boolean keys are really just choices, but all done by hand. The second argument here is the scope: either empty or `g` for global.

```

9906 \cs_new_protected:Npn \__keys_bool_set:Nn #1#2
9907   {
9908     \bool_if_exist:NF #1 { \bool_new:N #1 }
9909     \__keys_choice_make:
9910     \__keys_cmd_set:nx { \l_keys_path_t1 / true }
9911       { \exp_not:c { bool_ #2 set_true:N } \exp_not:N #1 }
9912     \__keys_cmd_set:nx { \l_keys_path_t1 / false }
9913       { \exp_not:c { bool_ #2 set_false:N } \exp_not:N #1 }
9914     \__keys_cmd_set:nn { \l_keys_path_t1 / unknown }
9915       {
9916         \__msg_kernel_error:nnx { kernel } { boolean-values-only }
9917           { \l_keys_key_t1 }
9918       }
9919     \__keys_default_set:n { true }
9920   }
9921 \cs_generate_variant:Nn \__keys_bool_set:Nn { c }

```

(End definition for `__keys_bool_set:Nn`.)

`__keys_bool_set_inverse:Nn` Inverse boolean setting is much the same.

```

9922 \cs_new_protected:Npn \__keys_bool_set_inverse:Nn #1#2
9923   {
9924     \bool_if_exist:NF #1 { \bool_new:N #1 }
9925     \__keys_choice_make:
9926     \__keys_cmd_set:nx { \l_keys_path_t1 / true }
9927       { \exp_not:c { bool_ #2 set_false:N } \exp_not:N #1 }
9928     \__keys_cmd_set:nx { \l_keys_path_t1 / false }
9929       { \exp_not:c { bool_ #2 set_true:N } \exp_not:N #1 }
9930     \__keys_cmd_set:nn { \l_keys_path_t1 / unknown }
9931       {
9932         \__msg_kernel_error:nnx { kernel } { boolean-values-only }
9933           { \l_keys_key_t1 }
9934       }
9935     \__keys_default_set:n { true }
9936   }
9937 \cs_generate_variant:Nn \__keys_bool_set_inverse:Nn { c }

```

(End definition for `_keys_bool_set_inverse:Nn`.)

`_keys_choice_make:` To make a choice from a key, two steps: set the code, and set the unknown key. As multichoices and choices are essentially the same bar one function, the code is given together.

```

9938 \cs_new_protected:Npn \_keys_choice_make:
9939   { \_keys_choice_make:N \_keys_choice_find:n }
9940 \cs_new_protected:Npn \_keys_multichoice_make:
9941   { \_keys_choice_make:N \_keys_multichoice_find:n }
9942 \cs_new_protected:Npn \_keys_choice_make:N #1
9943   {
9944     \cs_if_exist:cTF
9945       { \c_keys_type_root_tl \_keys_parent:o \l_keys_path_tl }
9946       {
9947         \str_if_eq_x:nnTF
9948           { \exp_not:v { \c_keys_type_root_tl \_keys_parent:o \l_keys_path_tl } }
9949           { choice }
9950           {
9951             \msg_kernel_error:nnxx { kernel } { nested-choice-key }
9952               { \l_keys_path_tl } { \_keys_parent:o \l_keys_path_tl }
9953           }
9954           { \_keys_choice_make_aux:N #1 }
9955       }
9956       { \_keys_choice_make_aux:N #1 }
9957   }
9958 \cs_new_protected:Npn \_keys_choice_make_aux:N #1
9959   {
9960     \cs_set_nopar:cpn { \c_keys_type_root_tl \l_keys_path_tl } { choice }
9961     \_keys_cmd_set:nn { \l_keys_path_tl } { #1 {##1} }
9962     \_keys_cmd_set:nn { \l_keys_path_tl / unknown }
9963     {
9964       \msg_kernel_error:nnxx { kernel } { key-choice-unknown }
9965         { \l_keys_path_tl } {##1}
9966     }
9967   }

```

(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.

```

9968 \cs_new_protected:Npn \_keys_choices_make:nn
9969   { \_keys_choices_make:Nnn \_keys_choice_make: }
9970 \cs_new_protected:Npn \_keys_multichoices_make:nn
9971   { \_keys_choices_make:Nnn \_keys_multichoice_make: }
9972 \cs_new_protected:Npn \_keys_choices_make:Nnn #1#2#3
9973   {
9974     #1
9975     \int_zero:N \l_keys_choice_int
9976     \clist_map_inline:nn {#2}
9977     {
9978       \int_incr:N \l_keys_choice_int
9979       \_keys_cmd_set:nx { \l_keys_path_tl / \_keys_remove_spaces:n {##1} }
9980       {
9981         \tl_set:Nn \exp_not:N \l_keys_choice_tl {##1}

```

```

9982     \int_set:Nn \exp_not:N \l_keys_choice_int
9983         { \int_use:N \l_keys_choice_int }
9984     \exp_not:n {#3}
9985 }
9986 }
9987 }
```

(End definition for `__keys_choices_make:nn`, `__keys_multichoices_make:nn`, and `__keys_choices-make:Nnn`.)

`__keys_cmd_set:nn` Setting the code for a key first checks that the basic data structures exist, then saves the code.

```

9988 \cs_new_protected:Npn \__keys_cmd_set:nn #1#2
9989 {
9990     \cs_if_exist:cF { \c_keys_code_root_tl #1 }
9991         { \__chk_log:x { Defining-key-#1~\msg_line_context: } }
9992     \cs_set_protected:cpn { \c_keys_code_root_tl #1 } ##1 {#2}
9993 }
9994 \cs_generate_variant:Nn \__keys_cmd_set:nn { nx , Vn , Vo }
```

(End definition for `__keys_cmd_set:nn`.)

`__keys_default_set:n` Setting a default value is easy. These are stored using `\cs_set:cpx` as this avoids any worries about whether a token list exists.

```

9995 \cs_new_protected:Npn \__keys_default_set:n #1
9996 {
9997     \tl_if_empty:nTF {#1}
9998     {
9999         \cs_set_eq:cN
10000             { \c_keys_default_root_tl \l_keys_path_tl }
10001             \tex_undefined:D
10002     }
10003     {
10004         \cs_set:cpx
10005             { \c_keys_default_root_tl \l_keys_path_tl }
10006             { \exp_not:n {#1} }
10007     }
10008 }
```

(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.

```

10009 \cs_new_protected:Npn \__keys_groups_set:n #1
10010 {
10011     \clist_set:Nn \l__keys_groups_clist {#1}
10012     \clist_if_empty:NTF \l__keys_groups_clist
10013     {
10014         \cs_set_eq:cN { \c_keys_groups_root_tl \l_keys_path_tl }
10015             \tex_undefined:D
10016     }
10017     {
10018         \clist_set_eq:cN { \c_keys_groups_root_tl \l_keys_path_tl }
```

```

10019         \l__keys_groups_clist
10020     }
10021 }
```

(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.

```

10022 \cs_new_protected:Npn \_keys_inherit:n #1
10023 {
10024     \_keys_undefine:
10025     \cs_set_nopar:cpn { \c__keys_inherit_root_tl \l_keys_path_tl } {#1}
10026 }
```

(End definition for `_keys_inherit:n`.)

`_keys_initialise:n` A set up for initialisation: just run the code if it exists.

```

10027 \cs_new_protected:Npn \_keys_initialise:n #1
10028 {
10029     \cs_if_exist_use:cT { \c__keys_code_root_tl \l_keys_path_tl } { {#1} }
10030 }
```

(End definition for `_keys_initialise:n`.)

`_keys_meta_make:n` To create a meta-key, simply set up to pass data through.

```

10031 \cs_new_protected:Npn \_keys_meta_make:n #1
10032 {
10033     \_keys_cmd_set:Vo \l_keys_path_tl
10034     {
10035         \exp_after:wN \keys_set:nn
10036         \exp_after:wN { \l__keys_module_tl } {#1}
10037     }
10038 }
10039 \cs_new_protected:Npn \_keys_meta_make:nn #1#2
10040 { \_keys_cmd_set:Vn \l_keys_path_tl { \keys_set:nn {#1} {#2} } }
```

(End definition for `_keys_meta_make:n` and `_keys_meta_make:nn`.)

`_keys_undefine:` Undefining a key has to be done without `\cs_undefine:c` as that function acts globally.

```

10041 \cs_new_protected:Npn \_keys_undefine:
10042 {
10043     \clist_map_inline:nn
10044     { code , default , groups , inherit , type , validate }
10045     {
10046         \cs_set_eq:cN
10047         { \tl_use:c { c__keys_ ##1 _root_tl } \l_keys_path_tl }
10048         \tex_undefined:D
10049     }
10050 }
```

(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.

```

10051 \cs_new_protected:Npn \__keys_value_requirement:nn #1#2
10052   {
10053     \str_case:nnF {#2}
10054     {
10055       { true }
10056       {
10057         \cs_set_eq:cc
10058         { \c__keys_validate_root_tl \l__keys_path_tl }
10059         { __keys_validate_ #1 : }
10060       }
10061     { false }
10062     {
10063       \cs_if_eq:ccT
10064         { \c__keys_validate_root_tl \l__keys_path_tl }
10065         { __keys_validate_ #1 : }
10066         {
10067           \cs_set_eq:cN
10068             { \c__keys_validate_root_tl \l__keys_path_tl }
10069             \tex_undefined:D
10070           }
10071         }
10072     }
10073   {
10074     \__msg_kernel_error:nnx { kernel } { property-boolean-values-only }
10075     { .value_ #1 :n }
10076   }
10077 }
10078 \cs_new_protected:Npn \__keys_validate_forbidden:
10079   {
10080     \bool_if:NF \l__keys_no_value_bool
10081     {
10082       \__msg_kernel_error:nnxx { kernel } { value-forbidden }
10083       { \l__keys_path_tl } { \l__keys_value_tl }
10084       \__keys_validate_cleanup:w
10085     }
10086   }
10087 \cs_new_protected:Npn \__keys_validate_required:
10088   {
10089     \bool_if:NT \l__keys_no_value_bool
10090     {
10091       \__msg_kernel_error:nnx { kernel } { value-required }
10092       { \l__keys_path_tl }
10093       \__keys_validate_cleanup:w
10094     }
10095   }
10096 \cs_new_protected:Npn \__keys_validate_cleanup:w #1 \cs_end: #2#3 { }
  
```

(End definition for `__keys_value_requirement:nn` and others.)

`__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.

```
10097 \cs_new_protected:Npn \__keys_variable_set:NnnN #1#2#3#4
10098  {
10099    \use:c { #2_if_exist:NF } #1 { \use:c { #2 _new:N } #1 }
10100    \__keys_cmd_set:nx { \l_keys_path_tl }
10101    {
10102      \exp_not:c { #2 _ #3 set:N #4 }
10103      \exp_not:N #1
10104      \exp_not:n { {##1} }
10105    }
10106  }
10107 \cs_generate_variant:Nn \__keys_variable_set:NnnN { c }
```

(End definition for `__keys_variable_set:NnnN`.)

19.5 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.

```
10108 \cs_new_protected:cpn { \c__keys_props_root_tl .bool_set:N } #1
10109  { \__keys_bool_set:Nn #1 { } }
10110 \cs_new_protected:cpn { \c__keys_props_root_tl .bool_set:c } #1
10111  { \__keys_bool_set:cn {#1} { } }
10112 \cs_new_protected:cpn { \c__keys_props_root_tl .bool_gset:N } #1
10113  { \__keys_bool_set:Nn #1 { g } }
10114 \cs_new_protected:cpn { \c__keys_props_root_tl .bool_gset:c } #1
10115  { \__keys_bool_set:cn {#1} { g } }
```

(End definition for `.bool_set:N` and `.bool_gset:N`. These functions are documented on page 164.)

`.bool_set_inverse:N` One function for this.

```
10116 \cs_new_protected:cpn { \c__keys_props_root_tl .bool_set_inverse:N } #1
10117  { \__keys_bool_set_inverse:Nn #1 { } }
10118 \cs_new_protected:cpn { \c__keys_props_root_tl .bool_set_inverse:c } #1
10119  { \__keys_bool_set_inverse:cn {#1} { } }
10120 \cs_new_protected:cpn { \c__keys_props_root_tl .bool_gset_inverse:N } #1
10121  { \__keys_bool_set_inverse:Nn #1 { g } }
10122 \cs_new_protected:cpn { \c__keys_props_root_tl .bool_gset_inverse:c } #1
10123  { \__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 164.)

`.choice:` Making a choice is handled internally, as it is also needed by `.generate_choices:n`.

```
10124 \cs_new_protected:cpn { \c__keys_props_root_tl .choice: }
10125  { \__keys_choice_make: }
```

(End definition for `.choice:`. This function is documented on page 164.)

.choices:nn For auto-generation of a series of mutually-exclusive choices. Here, #1 will consist of two separate arguments, hence the slightly odd-looking implementation.

```

10126 \cs_new_protected:cpn { \c__keys_props_root_tl .choices:nn } #1
10127   { \__keys_choices_make:nn #1 }
10128 \cs_new_protected:cpn { \c__keys_props_root_tl .choices:Vn } #1
10129   { \exp_args:NV \__keys_choices_make:nn #1 }
10130 \cs_new_protected:cpn { \c__keys_props_root_tl .choices:on } #1
10131   { \exp_args:No \__keys_choices_make:nn #1 }
10132 \cs_new_protected:cpn { \c__keys_props_root_tl .choices:xn } #1
10133   { \exp_args:Nx \__keys_choices_make:nn #1 }

```

(End definition for `.choices:nn`. This function is documented on page 164.)

.code:n Creating code is simply a case of passing through to the underlying `set` function.

```

10134 \cs_new_protected:cpn { \c__keys_props_root_tl .code:n } #1
10135   { \__keys_cmd_set:nn { \l_keys_path_tl } {#1} }

```

(End definition for `.code:n`. This function is documented on page 164.)

.clist_set:N

```

10136 \cs_new_protected:cpn { \c__keys_props_root_tl .clist_set:N } #1
10137   { \__keys_variable_set:NnnN #1 { clist } { } n }
10138 \cs_new_protected:cpn { \c__keys_props_root_tl .clist_set:c } #1
10139   { \__keys_variable_set:cnnN {#1} { clist } { } n }
10140 \cs_new_protected:cpn { \c__keys_props_root_tl .clist_gset:N } #1
10141   { \__keys_variable_set:NnnN #1 { clist } { g } n }
10142 \cs_new_protected:cpn { \c__keys_props_root_tl .clist_gset:c } #1
10143   { \__keys_variable_set:cnnN {#1} { clist } { g } n }

```

(End definition for `.clist_set:N` and `.clist_gset:N`. These functions are documented on page 164.)

.default:n Expansion is left to the internal functions.

```

10144 \cs_new_protected:cpn { \c__keys_props_root_tl .default:n } #1
10145   { \__keys_default_set:n {#1} }
10146 \cs_new_protected:cpn { \c__keys_props_root_tl .default:V } #1
10147   { \exp_args:NV \__keys_default_set:n #1 }
10148 \cs_new_protected:cpn { \c__keys_props_root_tl .default:o } #1
10149   { \exp_args:No \__keys_default_set:n {#1} }
10150 \cs_new_protected:cpn { \c__keys_props_root_tl .default:x } #1
10151   { \exp_args:Nx \__keys_default_set:n {#1} }

```

(End definition for `.default:n`. This function is documented on page 165.)

.dim_set:N Setting a variable is very easy: just pass the data along.

```

10152 \cs_new_protected:cpn { \c__keys_props_root_tl .dim_set:N } #1
10153   { \__keys_variable_set:NnnN #1 { dim } { } n }
10154 \cs_new_protected:cpn { \c__keys_props_root_tl .dim_set:c } #1
10155   { \__keys_variable_set:cnnN {#1} { dim } { } n }
10156 \cs_new_protected:cpn { \c__keys_props_root_tl .dim_gset:N } #1
10157   { \__keys_variable_set:NnnN #1 { dim } { g } n }
10158 \cs_new_protected:cpn { \c__keys_props_root_tl .dim_gset:c } #1
10159   { \__keys_variable_set:cnnN {#1} { dim } { g } n }

```

(End definition for `.dim_set:N` and `.dim_gset:N`. These functions are documented on page 165.)

```

.fp_set:N Setting a variable is very easy: just pass the data along.
.fp_set:c 10160 \cs_new_protected:cpn { \c__keys_props_root_tl .fp_set:N } #1
.fp_gset:N 10161   { \__keys_variable_set:NnnN #1 { fp } { } n }
.fp_gset:c 10162 \cs_new_protected:cpn { \c__keys_props_root_tl .fp_set:c } #1
10163   { \__keys_variable_set:cnnN {#1} { fp } { } n }
10164 \cs_new_protected:cpn { \c__keys_props_root_tl .fp_gset:N } #1
10165   { \__keys_variable_set:NnnN #1 { fp } { g } n }
10166 \cs_new_protected:cpn { \c__keys_props_root_tl .fp_gset:c } #1
10167   { \__keys_variable_set:cnnN {#1} { fp } { g } n }

```

(End definition for `.fp_set:N` and `.fp_gset:N`. These functions are documented on page 165.)

`.groups:n` A single property to create groups of keys.

```

10168 \cs_new_protected:cpn { \c__keys_props_root_tl .groups:n } #1
10169   { \__keys_groups_set:n {#1} }

```

(End definition for `.groups:n`. This function is documented on page 165.)

`.inherit:n` Nothing complex: only one variant at the moment!

```

10170 \cs_new_protected:cpn { \c__keys_props_root_tl .inherit:n } #1
10171   { \__keys_inherit:n {#1} }

```

(End definition for `.inherit:n`. This function is documented on page 165.)

`.initial:n` The standard hand-off approach.

```

.initial:V 10172 \cs_new_protected:cpn { \c__keys_props_root_tl .initial:n } #1
.initial:o 10173   { \__keys_initialise:n {#1} }
.initial:x 10174 \cs_new_protected:cpn { \c__keys_props_root_tl .initial:V } #1
10175   { \exp_args:NV \__keys_initialise:n {#1} }
10176 \cs_new_protected:cpn { \c__keys_props_root_tl .initial:o } #1
10177   { \exp_args:No \__keys_initialise:n {#1} }
10178 \cs_new_protected:cpn { \c__keys_props_root_tl .initial:x } #1
10179   { \exp_args:Nx \__keys_initialise:n {#1} }

```

(End definition for `.initial:n`. This function is documented on page 166.)

`.int_set:N` Setting a variable is very easy: just pass the data along.

```

.int_set:c 10180 \cs_new_protected:cpn { \c__keys_props_root_tl .int_set:N } #1
.int_gset:N 10181   { \__keys_variable_set:NnnN #1 { int } { } n }
.int_gset:c 10182 \cs_new_protected:cpn { \c__keys_props_root_tl .int_set:c } #1
10183   { \__keys_variable_set:cnnN {#1} { int } { } n }
10184 \cs_new_protected:cpn { \c__keys_props_root_tl .int_gset:N } #1
10185   { \__keys_variable_set:NnnN #1 { int } { g } n }
10186 \cs_new_protected:cpn { \c__keys_props_root_tl .int_gset:c } #1
10187   { \__keys_variable_set:cnnN {#1} { int } { g } n }

```

(End definition for `.int_set:N` and `.int_gset:N`. These functions are documented on page 166.)

`.meta:n` Making a meta is handled internally.

```

10188 \cs_new_protected:cpn { \c__keys_props_root_tl .meta:n } #1
10189   { \__keys_meta_make:n {#1} }

```

(End definition for `.meta:n`. This function is documented on page 166.)

.meta:nn Meta with path: potentially lots of variants, but for the moment no so many defined.

```
10190 \cs_new_protected:cpn { \c_keys_props_root_tl .meta:nn } #1
10191   { \_keys_meta:nn #1 }
```

(End definition for **.meta:nn**. This function is documented on page [166](#).)

.multichoice: The same idea as **.choice:** and **.choices:nn**, but where more than one choice is allowed.

```
10192 \cs_new_protected:cpn { \c_keys_props_root_tl .multichoice: }
10193   { \_keys_multichoice_make: }
10194 \cs_new_protected:cpn { \c_keys_props_root_tl .multichoice:nn } #1
10195   { \_keys_multichoice:nn #1 }
10196 \cs_new_protected:cpn { \c_keys_props_root_tl .multichoice:Vn } #1
10197   { \exp_args:N \_keys_multichoice:nn #1 }
10198 \cs_new_protected:cpn { \c_keys_props_root_tl .multichoice:on } #1
10199   { \exp_args:No \_keys_multichoice:nn #1 }
10200 \cs_new_protected:cpn { \c_keys_props_root_tl .multichoice:xn } #1
10201   { \exp_args:Nx \_keys_multichoice:nn #1 }
```

(End definition for **.multichoice:** and **.multichoice:nn**. These functions are documented on page [166](#).)

.skip_set:N Setting a variable is very easy: just pass the data along.

```
10202 \cs_new_protected:cpn { \c_keys_props_root_tl .skip_set:N } #1
10203   { \_keys_variable_set:NnnN #1 { skip } { } n }
10204 \cs_new_protected:cpn { \c_keys_props_root_tl .skip_set:c } #1
10205   { \_keys_variable_set:cnnN {#1} { skip } { } n }
10206 \cs_new_protected:cpn { \c_keys_props_root_tl .skip_gset:N } #1
10207   { \_keys_variable_set:NnnN #1 { skip } { g } n }
10208 \cs_new_protected:cpn { \c_keys_props_root_tl .skip_gset:c } #1
10209   { \_keys_variable_set:cnnN {#1} { skip } { g } n }
```

(End definition for **.skip_set:N** and **.skip_gset:N**. These functions are documented on page [166](#).)

.tl_set:N Setting a variable is very easy: just pass the data along.

```
10210 \cs_new_protected:cpn { \c_keys_props_root_tl .tl_set:N } #1
10211   { \_keys_variable_set:NnnN #1 { tl } { } n }
10212 \cs_new_protected:cpn { \c_keys_props_root_tl .tl_set:c } #1
10213   { \_keys_variable_set:cnnN {#1} { tl } { } n }
10214 \cs_new_protected:cpn { \c_keys_props_root_tl .tl_set_x:N } #1
10215   { \_keys_variable_set:NnnN #1 { tl } { } x }
10216 \cs_new_protected:cpn { \c_keys_props_root_tl .tl_set_x:c } #1
10217   { \_keys_variable_set:cnnN {#1} { tl } { } x }
10218 \cs_new_protected:cpn { \c_keys_props_root_tl .tl_gset:N } #1
10219   { \_keys_variable_set:NnnN #1 { tl } { g } n }
10220 \cs_new_protected:cpn { \c_keys_props_root_tl .tl_gset:c } #1
10221   { \_keys_variable_set:cnnN {#1} { tl } { g } n }
10222 \cs_new_protected:cpn { \c_keys_props_root_tl .tl_gset_x:N } #1
10223   { \_keys_variable_set:NnnN #1 { tl } { g } x }
10224 \cs_new_protected:cpn { \c_keys_props_root_tl .tl_gset_x:c } #1
10225   { \_keys_variable_set:cnnN {#1} { tl } { g } x }
```

(End definition for **.tl_set:N** and others. These functions are documented on page [166](#).)

.undefine: Another simple wrapper.

```
10226 \cs_new_protected:cpn { \c_keys_props_root_tl .undefine: }
10227   { \_keys_undefine: }
```

(End definition for `.undefined`. This function is documented on page 167.)

`.value_forbidden:n` These are very similar, so both call the same function.

```
.value_required:n 10228 \cs_new_protected:cpn { \c_keys_props_root_tl .value_forbidden:n } #1
                   { \_keys_value_requirement:nn { forbidden } {#1} }
10230 \cs_new_protected:cpn { \c_keys_props_root_tl .value_required:n } #1
                   { \_keys_value_requirement:nn { required } {#1} }
```

(End definition for `.value_forbidden:n` and `.value_required:n`. These functions are documented on page 167.)

19.6 Setting keys

`\keys_set:nn` A simple wrapper again.

```
\keys_set:nV 10232 \cs_new_protected:Npn \keys_set:nn
\keys_set:nv 10233 { \_keys_set:onn { \l_keys_module_tl } }
\keys_set:no 10234 \cs_new_protected:Npn \_keys_set:nnn #1#2#3
\__keys_set:nnn 10235 {
\__keys_set:onn 10236 \tl_set:Nx \l_keys_module_tl { \_keys_remove_spaces:n {#2} }
10237 \keyval_parse:NNn \_keys_set:n \_keys_set:nn {#3}
10238 \tl_set:Nn \l_keys_module_tl {#1}
}
10239 }
10240 \cs_generate_variant:Nn \keys_set:nn { nV , nv , no }
10241 \cs_generate_variant:Nn \_keys_set:nnn { o }
```

(End definition for `\keys_set:nn` and `_keys_set:nnn`. These functions are documented on page 170.)

`\keys_set_known:nnN`

`\keys_set_known:nVN`

`\keys_set_known:nvN`

`\keys_set_known:noN`

`__keys_set_known:nnnN`

`__keys_set_known:onnN`

`\keys_set_known:nn`

`\keys_set_known:nV`

`\keys_set_known:nv`

`\keys_set_known:no`

`__keys_keys_set_known:nn`

```
10242 \cs_new_protected:Npn \keys_set_known:nnN
10243 { \_keys_set_known:onnN \l_keys_unused_clist }
10244 \cs_generate_variant:Nn \keys_set_known:nnN { nV , nv , no }
10245 \cs_new_protected:Npn \_keys_set_known:nnnN #1#2#3#4
10246 {
10247 \clist_clear:N \l_keys_unused_clist
10248 \keys_set_known:nn {#2} {#3}
10249 \tl_set:Nx #4 { \exp_not:o { \l_keys_unused_clist } }
10250 \tl_set:Nn \l_keys_unused_clist {#1}
}
10251 }
10252 \cs_generate_variant:Nn \_keys_set_known:nnnN { o }
10253 \cs_new_protected:Npn \keys_set_known:nn #1#2
10254 {
10255 \bool_if:NTF \l_keys_only_known_bool
10256 { \keys_set:nn }
10257 { \_keys_set_known:nn }
10258 {#1} {#2}
}
10259 }
10260 \cs_generate_variant:Nn \keys_set_known:nn { nV , nv , no }
10261 \cs_new_protected:Npn \_keys_set_known:nn #1#2
10262 {
10263 \bool_set_true:N \l_keys_only_known_bool
10264 \keys_set:nn {#1} {#2}
```

```

10265     \bool_set_false:N \l__keys_only_known_bool
10266 }

```

(End definition for `\keys_set_known:nnN` and others. These functions are documented on page 171.)

```

\keys_set_filter:nnN
\keys_set_filter:nnVN
\keys_set_filter:nnvN
\keys_set_filter:nnnN
\__keys_set_filter:nnnnN
\__keys_set_filter:onnnN
\keys_set_filter:nnN
\keys_set_filter:nnV
\keys_set_filter:nnV
\keys_set_filter:nnnO
\__keys_set_filter:nnnN
\keys_set_groups:nnN
\keys_set_groups:nnV
\keys_set_groups:nnV
\keys_set_groups:nnnO
\__keys_set_groups:nnnN
\__keys_set_selective:nnN
\__keys_set_selective:nnnnN
\__keys_set_selective:onnN
\__keys_set_selective:nnN

```

The idea of setting keys in a selective manner again uses flags wrapped around the basic code. The comments on `\keys_set_known:nnN` also apply here. We have a bit more shuffling to do to keep everything nestable.

```

10267 \cs_new_protected:Npn \keys_set_filter:nnnN
10268 {
10269     \__keys_set_filter:onnnN \l__keys_unused_clist
10270 \cs_generate_variant:Nn \keys_set_filter:nnnN { nnV , nnv , nno }
10271 \cs_new_protected:Npn \__keys_set_filter:nnnnN #1#2#3#4#5
10272 {
10273     \clist_clear:N \l__keys_unused_clist
10274     \keys_set_filter:nnn {#2} {#3} {#4}
10275     \tl_set:Nx #5 { \exp_not:o { \l__keys_unused_clist } }
10276     \tl_set:Nn \l__keys_unused_clist {#1}
10277 \cs_generate_variant:Nn \__keys_set_filter:nnnnN { o }
10278 \cs_new_protected:Npn \keys_set_filter:nnn #1#2#3
10279 {
10280     \bool_if:NTF \l__keys_filtered_bool
10281     { \__keys_set_selective:nnN }
10282     { \__keys_set_filter:nnn }
10283     {#1} {#2} {#3}
10284 }
10285 \cs_generate_variant:Nn \keys_set_filter:nnn { nnV , nnv , nno }
10286 \cs_new_protected:Npn \__keys_set_filter:nnn #1#2#3
10287 {
10288     \bool_set_true:N \l__keys_filtered_bool
10289     \__keys_set_selective:nn {#1} {#2} {#3}
10290     \bool_set_false:N \l__keys_filtered_bool
10291 }
10292 \cs_new_protected:Npn \keys_set_groups:nnn #1#2#3
10293 {
10294     \bool_if:NTF \l__keys_filtered_bool
10295     { \__keys_set_groups:nnN }
10296     { \__keys_set_selective:nnN }
10297     {#1} {#2} {#3}
10298 }
10299 \cs_generate_variant:Nn \keys_set_groups:nnn { nnV , nnv , nno }
10300 \cs_new_protected:Npn \__keys_set_groups:nnn #1#2#3
10301 {
10302     \bool_set_false:N \l__keys_filtered_bool
10303     \__keys_set_selective:nnn {#1} {#2} {#3}
10304     \bool_set_true:N \l__keys_filtered_bool
10305 }
10306 \cs_new_protected:Npn \__keys_set_selective:nnN
10307 {
10308     \__keys_set_selective:onnN \l__keys_selective_seq
10309 \cs_new_protected:Npn \__keys_set_selective:nnnn #1#2#3#4
10310 {
10311     \seq_set_from_clist:Nn \l__keys_selective_seq {#3}
10312     \bool_if:NTF \l__keys_selective_bool
10313     { \keys_set:nn }

```

```

10313     { \__keys_set_selective:nn }
10314     {#2} {#4}
10315     \tl_set:Nn \l__keys_selective_seq {#1}
10316   }
10317 \cs_generate_variant:Nn \__keys_set_selective:nnnn { o }
10318 \cs_new_protected:Npn \__keys_set_selective:nn #1#2
10319   {
10320     \bool_set_true:N \l__keys_selective_bool
10321     \keys_set:nn {#1} {#2}
10322     \bool_set_false:N \l__keys_selective_bool
10323   }

```

(End definition for `\keys_set_filter:nnN` and others. These functions are documented on page 172.)

`__keys_set:n`
`__keys_set:nn`
`__keys_set_aux:nnn`
`__keys_set_aux:onn`
`__keys_find_key_module:w`
`__keys_set_aux:`
`__keys_set_selective:`

A shared system once again. First, set the current path and add a default if needed. There are then checks to see if the a value is required or forbidden. If everything passes, move on to execute the code.

```

10324 \cs_new_protected:Npn \__keys_set:n #1
10325   {
10326     \bool_set_true:N \l__keys_no_value_bool
10327     \__keys_set_aux:onn \l__keys_module_tl {#1} { }
10328   }
10329 \cs_new_protected:Npn \__keys_set:nn #1#2
10330   {
10331     \bool_set_false:N \l__keys_no_value_bool
10332     \__keys_set_aux:onn \l__keys_module_tl {#1} {#2}
10333   }

```

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.

```

10334 \cs_new_protected:Npn \__keys_set_aux:nnn #1#2#3
10335   {
10336     \tl_set:Nx \l_keys_path_tl
10337     {
10338       \tl_if_blank:nF {#1}
10339       { #1 / }
10340       \__keys_remove_spaces:n {#2}
10341     }
10342     \tl_clear:N \l__keys_module_tl
10343     \exp_after:wN \__keys_find_key_module:w \l_keys_path_tl / \q_stop
10344     \__keys_value_or_default:n {#3}
10345     \bool_if:NTF \l__keys_selective_bool
10346       { \__keys_set_selective: }
10347       { \__keys_execute: }
10348     \tl_set:Nn \l__keys_module_tl {#1}
10349   }
10350 \cs_generate_variant:Nn \__keys_set_aux:nnn { o }
10351 \cs_new_protected:Npn \__keys_find_key_module:w #1 / #2 \q_stop
10352   {
10353     \tl_if_blank:nTF {#2}
10354       { \tl_set:Nn \l_keys_key_tl {#1} }
10355       {

```

```

10356     \tl_put_right:Nx \l__keys_module_tl
10357     {
10358         \tl_if_empty:NF \l__keys_module_tl { / }
10359         #1
10360     }
10361     \__keys_find_key_module:w #2 \q_stop
10362 }
10363 }
```

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.

```

10364 \cs_new_protected:Npn \__keys_set_selective:
10365 {
10366     \cs_if_exist:cTF { \c__keys_groups_root_tl \l__keys_path_tl }
10367     {
10368         \clist_set_eq:Nc \l__keys_groups_clist
10369         { \c__keys_groups_root_tl \l__keys_path_tl }
10370         \__keys_check_groups:
10371     }
10372     {
10373         \bool_if:NTF \l__keys_filtered_bool
10374             { \__keys_execute: }
10375             { \__keys_store_unused: }
10376     }
10377 }
```

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.

```

10378 \cs_new_protected:Npn \__keys_check_groups:
10379 {
10380     \bool_set_false:N \l__keys_tmp_bool
10381     \seq_map_inline:Nn \l__keys_selective_seq
10382     {
10383         \clist_map_inline:Nn \l__keys_groups_clist
10384         {
10385             \str_if_eq:nnT {##1} {####1}
10386             {
10387                 \bool_set_true:N \l__keys_tmp_bool
10388                 \clist_map_break:n { \seq_map_break: }
10389             }
10390         }
10391     }
10392     \bool_if:NTF \l__keys_tmp_bool
10393     {
10394         \bool_if:NTF \l__keys_filtered_bool
10395             { \__keys_store_unused: }
10396             { \__keys_execute: }
10397     }
10398     {
10399         \bool_if:NTF \l__keys_filtered_bool
10400             { \__keys_execute: }
10401             { \__keys_store_unused: }
10402     }
```

```
10403     }
```

(End definition for `_keys_set:n` and others.)

`_keys_value_or_default:n` If a value is given, return it as #1, otherwise send a default if available.

```
10404 \cs_new_protected:Npn \_keys_value_or_default:n #1
10405 {
10406     \bool_if:NTF \l__keys_no_value_bool
10407     {
10408         \cs_if_exist:cTF { \c__keys_default_root_tl \l_keys_path_tl }
10409         {
10410             \tl_set_eq:Nc
10411                 \l_keys_value_tl
10412                 { \c__keys_default_root_tl \l_keys_path_tl }
10413         }
10414         { \tl_clear:N \l_keys_value_tl }
10415     }
10416     { \tl_set:Nn \l_keys_value_tl {#1} }
10417 }
```

(End definition for `_keys_value_or_default:n`.)

`_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.

```
10418 \cs_new_protected:Npn \_keys_execute:
10419 {
10420     \cs_if_exist:cTF { \c__keys_code_root_tl \l_keys_path_tl }
10421     {
10422         \cs_if_exist_use:c { \c__keys_validate_root_tl \l_keys_path_tl }
10423         \cs:w \c__keys_code_root_tl \l_keys_path_tl \exp_after:wN \cs_end:
10424             \exp_after:wN { \l_keys_value_tl }
10425     }
10426     { \_keys_execute_unknown: }
10427 }
10428 \cs_new_protected:Npn \_keys_execute_unknown:
10429 {
10430     \bool_if:NTF \l__keys_only_known_bool
10431     { \_keys_store_unused: }
10432     {
10433         \cs_if_exist:cTF
10434             { \c__keys_inherit_root_tl \_keys_parent:o \l_keys_path_tl }
10435             {
10436                 \clist_map_inline:cn
10437                     { \c__keys_inherit_root_tl \_keys_parent:o \l_keys_path_tl }
10438                     {
10439                         \cs_if_exist:cT
10440                             { \c__keys_code_root_tl ##1 / \l_keys_key_tl }
10441                             {
10442                                 \cs:w \c__keys_code_root_tl ##1 / \l_keys_key_tl
10443                                     \exp_after:wN \cs_end: \exp_after:wN
10444                                         { \l_keys_value_tl }
10445                                 \clist_map_break:
```

```

10446          }
10447      }
10448  }
10449  {
10450      \cs_if_exist:cTF { \c_keys_code_root_tl \l_keys_module_tl / unknown }
10451      {
10452          \cs:w \c_keys_code_root_tl \l_keys_module_tl / unknown
10453          \exp_after:wN \cs_end: \exp_after:wn { \l_keys_value_tl }
10454      }
10455      {
10456          \msg_kernel_error:nxx { kernel } { key-unknown }
10457          { \l_keys_path_tl } { \l_keys_module_tl }
10458      }
10459  }
10460 }
10461 }
10462 \cs_new:Npn \keys_execute:nn #1#2
10463 {
10464     \cs_if_exist:cTF { \c_keys_code_root_tl #1 }
10465     {
10466         \cs:w \c_keys_code_root_tl #1 \exp_after:wN \cs_end:
10467         \exp_after:wn { \l_keys_value_tl }
10468     }
10469     {#2}
10470 }
10471 \cs_new_protected:Npn \keys_store_unused:
10472 {
10473     \clist_put_right:Nx \l_keys_unused_clist
10474     {
10475         \exp_not:o \l_keys_key_tl
10476         \bool_if:NF \l_keys_no_value_bool
10477             { = { \exp_not:o \l_keys_value_tl } }
10478     }
10479 }

```

(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 unknown key. That will exist, as it is created when a choice is first made. So there is no need for any escape code. For multiple choices, the same code ends up used in a mapping.

```

10480 \cs_new:Npn \keys_choice_find:n #1
10481 {
10482     \keys_execute:nn { \l_keys_path_tl / \keys_remove_spaces:n {#1} }
10483     { \keys_execute:nn { \l_keys_path_tl / unknown } { } }
10484 }
10485 \cs_new:Npn \keys_multichoice_find:n #1
10486 { \clist_map_function:nN {#1} \keys_choice_find:n }

```

(End definition for `\keys_choice_find:n` and `\keys_multichoice_find:n`.)

19.7 Utilities

`\keys_parent:n` Used to strip off the ending part of the key path after the last `/`.

`\keys_parent:o` `\cs_new:Npn \keys_parent:n #1`

`\keys_parent:w`

```

10488 { \__keys_parent:w #1 / / \q_stop { } }
10489 \cs_generate_variant:Nn \__keys_parent:n { o }
10490 \cs_new:Npn \__keys_parent:w #1 / #2 / #3 \q_stop #4
10491 {
10492     \tl_if_blank:nTF {#2}
10493         { \use_none:n #4 }
10494         {
10495             \__keys_parent:w #2 / #3 \q_stop { #4 / #1 }
10496         }
10497 }

```

(End definition for `__keys_parent:n` and `__keys_parent:w`.)

`__keys_remove_spaces:n`
`__keys_remove_spaces:w`

Removes all spaces from the input which is detokenized as a result. This function has the same effect as `\zap@space` in L^AT_EX 2_< after applying `\tl_to_str:n`. It is set up to be fast as the use case here is tightly defined. The ? is only there to allow for a space after `\use_none:nn` responsible for ending the loop.

```

10498 \cs_new:Npn \__keys_remove_spaces:n #1
10499 {
10500     \exp_after:wN \__keys_remove_spaces:w \tl_to_str:n {#1}
10501     \use_none:nn ? ~
10502 }
10503 \cs_new:Npn \__keys_remove_spaces:w #1 ~
10504 { #1 \__keys_remove_spaces:w }

```

(End definition for `__keys_remove_spaces:n` and `__keys_remove_spaces:w`.)

`\keys_if_exist_p:nn` A utility for others to see if a key exists.

`\keys_if_exist:nnTF`

```

10505 \prg_new_conditional:Npnn \keys_if_exist:nn #1#2 { p , T , F , TF }
10506 {
10507     \cs_if_exist:cTF
10508         { \c__keys_code_root_tl \__keys_remove_spaces:n { #1 / #2 } }
10509         { \prg_return_true: }
10510         { \prg_return_false: }
10511 }

```

(End definition for `\keys_if_exist:nnTF`. This function is documented on page 172.)

`\keys_if_choice_exist_p:nnn`
`\keys_if_choice_exist:nnnTF`

Just an alternative view on `\keys_if_exist:nnTF`.

```

10512 \prg_new_conditional:Npnn \keys_if_choice_exist:nnn #1#2#3
10513 { p , T , F , TF }
10514 {
10515     \cs_if_exist:cTF
10516         { \c__keys_code_root_tl \__keys_remove_spaces:n { #1 / #2 / #3 } }
10517         { \prg_return_true: }
10518         { \prg_return_false: }
10519 }

```

(End definition for `\keys_if_choice_exist:nnnTF`. This function is documented on page 172.)

`\keys_show:nn`
`__keys_show:N`

To show a key, test for its existence to issue the correct message (same message, but with a t or f argument, then build the control sequences which contain the code and other information about the key, call an intermediate auxiliary which constructs the code that will be displayed to the terminal, and finally conclude with `__msg_show_wrap:n`.

```

10520 \cs_new_protected:Npn \keys_show:nn #1#2
10521 {
10522     \keys_if_exist:nnTF {#1} {#2}
10523     {
10524         \__msg_show_pre:nxxxxx { LaTeX / kernel } { show-key }
10525         { \__keys_remove_spaces:n { #1 / #2 } } { t } { } { }
10526         \exp_args:Nc \__keys_show:N
10527             { \c__keys_code_root_tl \__keys_remove_spaces:n { #1 / #2 } }
10528     }
10529     {
10530         \__msg_show_pre:nxxxxx { LaTeX / kernel } { show-key }
10531         { \__keys_remove_spaces:n { #1 / #2 } } { f } { } { }
10532         \__msg_show_wrap:n { }
10533     }
10534 }
10535 \cs_new_protected:Npn \__keys_show:N #1
10536 {
10537     \use:x
10538     {
10539         \__msg_show_wrap:n
10540         {
10541             \exp_not:N \__msg_show_item_unbraced:nn { code }
10542             { \token_get_replacement_spec:N #1 }
10543         }
10544     }
10545 }

```

(End definition for `\keys_show:nn` and `__keys_show:N`. These functions are documented on page 172.)

`\keys_log:nn` Redirect output of `\keys_show:nn` to the log.

```

10546 \cs_new_protected:Npn \keys_log:nn
10547     { \__msg_log_next: \keys_show:nn }

```

(End definition for `\keys_log:nn`. This function is documented on page 172.)

19.8 Messages

For when there is a need to complain.

```

10548 \__msg_kernel_new:nnnn { kernel } { boolean-values-only }
10549     { Key~'#1'~accepts~boolean~values~only. }
10550     { The~key~'#1'~only~accepts~the~values~'true'~and~'false'. }
10551 \__msg_kernel_new:nnnn { kernel } { key-choice-unknown }
10552     { Key~'#1'~accepts~only~a~fixed~set~of~choices. }
10553     {
10554         The~key~'#1'~only~accepts~predefined~values,~
10555         and~'#2'~is~not~one~of~these.
10556     }
10557 \__msg_kernel_new:nnnn { kernel } { key-no-property }
10558     { No~property~given~in~definition~of~key~'#1'. }
10559     {
10560         \c__msg_coding_error_text_tl
10561         Inside~\keys_define:nn each~key~name~
10562         needs~a~property: \\ \\
10563         \iow_indent:n { #1 .<property> } \\ \\

```

```

10564     LaTeX~did~not~find~a~'.~'~to~indicate~the~start~of~a~property.
10565 }
10566 \__msg_kernel_new:nnnn { kernel } { key-unknown }
10567   { The~key~'#1'~is~unknown~and~is~being~ignored. }
10568   {
10569     The~module~'#2'~does~not~have~a~key~called~'#1'.\\
10570     Check~that~you~have~spelled~the~key~name~correctly.
10571   }
10572 \__msg_kernel_new:nnnn { kernel } { nested-choice-key }
10573   { Attempt~to~define~'#1'~as~a~nested~choice~key. }
10574   {
10575     The~key~'#1'~cannot~be~defined~as~a~choice~as~the~parent~key~'#2'~is~
10576     itself~a~choice.
10577   }
10578 \__msg_kernel_new:nnnn { kernel } { property-boolean-values-only }
10579   { The~property~'#1'~accepts~boolean~values~only. }
10580   {
10581     \c__msg_error_text_tl
10582       The~property~'#1'~only~accepts~the~values~'true'~and~'false'.
10583   }
10584 \__msg_kernel_new:nnnn { kernel } { property-requires-value }
10585   { The~property~'#1'~requires~a~value. }
10586   {
10587     \c__msg_error_text_tl
10588       LaTeX~was~asked~to~set~property~'#1'~for~key~'#2'.\\
10589       No~value~was~given~for~the~property,~and~one~is~required.
10590   }
10591 \__msg_kernel_new:nnnn { kernel } { property-unknown }
10592   { The~key~property~'#1'~is~unknown. }
10593   {
10594     \c__msg_error_text_tl
10595       LaTeX~has~been~asked~to~set~the~property~'#1'~for~key~'#2':~
10596       this~property~is~not~defined.
10597   }
10598 \__msg_kernel_new:nmmn { kernel } { value-forbidden }
10599   { The~key~'#1'~does~not~take~a~value. }
10600   {
10601     The~key~'#1'~should~be~given~without~a~value.\\
10602     The~value~'#2'~was~present:~the~key~will~be~ignored.
10603   }
10604 \__msg_kernel_new:nnnn { kernel } { value-required }
10605   { The~key~'#1'~requires~a~value. }
10606   {
10607     The~key~'#1'~must~have~a~value.\\
10608     No~value~was~present:~the~key~will~be~ignored.
10609   }
10610 \__msg_kernel_new:nnn { kernel } { show-key }
10611   {
10612     The~key~'#1'
10613     \str_if_eq:nnTF {#2} { t }
10614       { has~the~properties: }
10615       { is~undefined. }
10616   }
10617 </initex | package>

```

20 l3fp implementation

Nothing to see here: everything is in the subfiles!

21 l3fp-aux implementation

10618 `(*initex | package)`

10619 `\s_fp \s_fp_chk:w <case> <sign> <body>` ;

21.1 Internal representation

Internally, a floating point number $\langle X \rangle$ is a token list containing

`\s_fp \s_fp_chk:w <case> <sign> <body>` ;

Let us explain each piece separately.

Internal floating point numbers will be used in expressions, and in this context will be subject to **f**-expansion. They must leave a recognizable mark after **f**-expansion, to prevent the floating point number from being re-parsed. Thus, `\s_fp` is simply another name for `\relax`.

When used directly without an accessor function, floating points should produce an error: this is the role of `\s_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 `<case>`, which is a single digit:

- 0 zeros: `+0` and `-0`,
- 1 “normal” numbers (positive and negative),
- 2 infinities: `+inf` and `-inf`,
- 3 quiet and signalling `nan`.

The `<sign>` is 0 (positive) or 2 (negative), except in the case of `nan`, which have `<sign>` = 1. This ensures that changing the `<sign>` digit to 2 – `<sign>` is exactly equivalent to changing the sign of the number.

Special floating point numbers have the form

`\s_fp \s_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 (`<case>` = 1) have the form

`\s_fp \s_fp_chk:w 1 <sign> {\<exponent>} {\<X1>} {\<X2>} {\<X3>} {\<X4>} ;`

Here, the `<exponent>` 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 `<exponent>` is minimal, in other words, $1000 \leq \langle X_1 \rangle \leq 9999$.

Calculations are done in base 10000, *i.e.* one myriad.

Table 1: Internal representation of floating point numbers.

Representation	Meaning
0 0 \s_fp_... ;	Positive zero.
0 2 \s_fp_... ;	Negative zero.
1 0 {\langle exponent\rangle} {\langle X_1\rangle} {\langle X_2\rangle} {\langle X_3\rangle} {\langle X_4\rangle} ;	Positive floating point.
1 2 {\langle exponent\rangle} {\langle X_1\rangle} {\langle X_2\rangle} {\langle X_3\rangle} {\langle X_4\rangle} ;	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.

21.2 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.
	<pre>10620 \cs_new:Npn _fp_use_none_stop_f:n #1 { \exp_stop_f: }</pre> <i>(End definition for _fp_use_none_stop_f:n.)</i>
_fp_use_s:n	Those functions place a semicolon after one or two arguments (typically digits).
_fp_use_s:nn	<pre>10621 \cs_new:Npn _fp_use_s:n #1 { #1; }</pre> <pre>10622 \cs_new:Npn _fp_use_s:nn #1#2 { #1#2; }</pre> <i>(End definition for _fp_use_s:n and _fp_use_s:nn.)</i>
_fp_use_none_until_s:w	Those functions select specific arguments among a set of arguments delimited by a semi-colon.
_fp_use_i_until_s:nw	<pre>10623 \cs_new:Npn _fp_use_none_until_s:w #1; { }</pre>
_fp_use_ii_until_s:nnw	<pre>10624 \cs_new:Npn _fp_use_i_until_s:nw #1#2; {#1}</pre> <pre>10625 \cs_new:Npn _fp_use_ii_until_s:nnw #1#2#3; {#2}</pre> <i>(End definition for _fp_use_none_until_s:w, _fp_use_i_until_s:nw, and _fp_use_ii_until_s:nnw.)</i>
_fp_reverse_args:Nww	Many internal functions take arguments delimited by semicolons, and it is occasionally useful to swap two such arguments.
	<pre>10626 \cs_new:Npn _fp_reverse_args:Nww #1 #2; #3; { #1 #3; #2; }</pre> <i>(End definition for _fp_reverse_args:Nww.)</i>
_fp_rrrot:www	Rotate three arguments delimited by semicolons. This is the inverse (or the square) of the Forth primitive ROT, hence the name.
	<pre>10627 \cs_new:Npn _fp_rrrot:www #1; #2; #3; { #2; #3; #1; }</pre> <i>(End definition for _fp_rrrot:www.)</i>
_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	<pre>10628 \cs_new:Npn _fp_use_i:ww #1; #2; { #1; }</pre> <pre>10629 \cs_new:Npn _fp_use_i:www #1; #2; #3; { #1; }</pre> <i>(End definition for _fp_use_i:ww and _fp_use_i:www.)</i>

21.3 Constants, and structure of floating points

\s_fp Floating points numbers all start with `\s_fp __fp_chk:w`, where `\s_fp` is equal to the TeX 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.

```
10630 \__scan_new:N \s_fp
10631 \cs_new_protected:Npn \__fp_chk:w #1 ;
10632 {
10633     \__msg_kernel_error:n { kernel } { misused-fp }
10634     { \fp_to_tl:n { \s_fp \__fp_chk:w #1 ; } }
10635 }
```

(End definition for `\s_fp` and `__fp_chk:w`.)

\s_fp_mark Aliases of `\tex_relax:D`, used to terminate expressions.

\s_fp_stop

```
10636 \__scan_new:N \s_fp_mark
10637 \__scan_new:N \s_fp_stop
```

(End definition for `\s_fp_mark` and `\s_fp_stop`.)

\s_fp_invalid A couple of scan marks used to indicate where special floating point numbers come from.

\s_fp_underflow

```
10638 \__scan_new:N \s_fp_invalid
\s_fp_overflow
```

\s_fp_division

```
10639 \__scan_new:N \s_fp_underflow
10640 \__scan_new:N \s_fp_overflow
\b_fp_exact
```

\s_fp_exact

```
10641 \__scan_new:N \s_fp_division
10642 \__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

```
10643 \tl_const:Nn \c_zero_fp { \s_fp \__fp_chk:w 0 \s_fp_exact ; }
```

\c_inf_fp

```
10644 \tl_const:Nn \c_minus_zero_fp { \s_fp \__fp_chk:w 0 2 \s_fp_exact ; }
```

\c_minus_inf_fp

```
10645 \tl_const:Nn \c_inf_fp { \s_fp \__fp_chk:w 2 0 \s_fp_exact ; }
```

\c_nan_fp

```
10646 \tl_const:Nn \c_minus_inf_fp { \s_fp \__fp_chk:w 2 2 \s_fp_exact ; }
```

```
10647 \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 181.)

\c_fp_prec_int The number of digits of floating points.

\c_fp_half_prec_int

```
10648 \int_const:Nn \c_fp_prec_int { 16 }
\c_fp_block_int
```

\c_fp_block_int

```
10649 \int_const:Nn \c_fp_half_prec_int { 8 }
10650 \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.

```
10651 \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 `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 TeX count.

```
10652 \int_const:Nn \c_fp_minus_min_exponent_int { 10000 }
10653 \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.

```
10654 \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.

```
10655 \tl_const:Nx \c_fp_overflowing_fp
10656 {
10657   \s_fp \fp_chk:w 1 0
10658   { \int_eval:n { \c_fp_max_exponent_int + 1 } }
10659   {1000} {0000} {0000} {0000} ;
10660 }
```

(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.

```
10661 \cs_new:Npn \_fp_zero_fp:N #1
10662   { \s_fp \fp_chk:w 0 #1 \s_fp_underflow ; }
10663 \cs_new:Npn \_fp_inf_fp:N #1
10664   { \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.

```
10665 \cs_new:Npn \_fp_exponent:w \s_fp \fp_chk:w #1
10666 {
10667   \if_meaning:w 1 #1
10668     \exp_after:wN \fp_use_i_until_s:nnw
10669   \else:
10670     \exp_after:wN \fp_use_i_until_s:nw
10671     \exp_after:wN 0
10672   \fi:
10673 }
```

(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.

```
10674 \cs_new:Npn \_fp_neg_sign:N #1
10675   { \int_eval:w 2 - #1 \int_eval_end: }
```

(End definition for `_fp_neg_sign:N`.)

21.4 Overflow, underflow, and exact zero

`__fp_sanitize:Nw` 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_underflow:w` and `__fp_overflow:w` are defined in `l3fp-traps`.

```

10676 \cs_new:Npn \__fp_sanitize:Nw #1 #2;
10677 {
10678   \if_case:w
10679     \if_int_compare:w #2 > \c_fp_max_exponent_int 1 ~ \else:
10680     \if_int_compare:w #2 < - \c_fp_minus_min_exponent_int 2 ~ \else:
10681     \if_meaning:w 1 #1 3 ~ \fi: \fi: \fi: 0 ~
10682     \or: \exp_after:wN \__fp_overflow:w
10683     \or: \exp_after:wN \__fp_underflow:w
10684     \or: \exp_after:wN \__fp_sanitize_zero:w
10685     \fi:
10686     \s_fp \__fp_chk:w 1 #1 {#2}
10687   }
10688 \cs_new:Npn \__fp_sanitize:wN #1; #2 { \__fp_sanitize:Nw #2 #1; }
10689 \cs_new:Npn \__fp_sanitize_zero:w \s_fp \__fp_chk:w #1 #2 #3;
10690   { \c_zero_fp }

```

(End definition for `__fp_sanitize:Nw`, `__fp_sanitize:wN`, and `__fp_sanitize_zero:w`.)

21.5 Expanding after a floating point number

`__fp_exp_after_o:w` Places $\langle tokens \rangle$ (empty in the case of `__fp_exp_after_o:w`) between the $\langle floating\ point \rangle$ and the $\langle more\ tokens \rangle$, then hits those tokens with either o-expansion (one `\exp-after:wN`) 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.

```

10691 \cs_new:Npn \__fp_exp_after_o:w \s_fp \__fp_chk:w #1
10692 {
10693   \if_meaning:w 1 #1
10694     \exp_after:wN \__fp_exp_after_normal:nNNw
10695   \else:
10696     \exp_after:wN \__fp_exp_after_special:nNNw
10697     \fi:
10698   { }
10699   #1
10700 }
10701 \cs_new:Npn \__fp_exp_after_f:nw #1 \s_fp \__fp_chk:w #2
10702 {
10703   \if_meaning:w 1 #2
10704     \exp_after:wN \__fp_exp_after_normal:nNNw
10705   \else:
10706     \exp_after:wN \__fp_exp_after_special:nNNw
10707     \fi:
10708   { \exp:w \exp_end_continue_f:w #1 }
10709   #2
10710 }

```

(End definition for `__fp_exp_after_o:w` and `__fp_exp_after_f:nw`.)

```
\__fp_exp_after_special:nNw Special floating point numbers are easy to jump over since they contain few tokens.
```

```
10711 \cs_new:Npn \__fp_exp_after_special:nNw #1#2#3#4;
10712 {
10713     \exp_after:wN \s_fp
10714     \exp_after:wN \__fp_chk:w
10715     \exp_after:wN #2
10716     \exp_after:wN #3
10717     \exp_after:wN #4
10718     \exp_after:wN ;
10719     #1
10720 }
```

(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.
```

```
10721 \cs_new:Npn \__fp_exp_after_normal:nNw #1 1 #2 #3 #4#5#6#7;
10722 {
10723     \exp_after:wN \__fp_exp_after_normal:Nwwww
10724     \exp_after:wN #2
10725     \__int_value:w #3 \exp_after:wN ;
10726     \__int_value:w 1 #4 \exp_after:wN ;
10727     \__int_value:w 1 #5 \exp_after:wN ;
10728     \__int_value:w 1 #6 \exp_after:wN ;
10729     \__int_value:w 1 #7 \exp_after:wN ; #1
10730 }
10731 \cs_new:Npn \__fp_exp_after_normal:Nwwww
10732     #1 #2; 1 #3 ; 1 #4 ; 1 #5 ; 1 #6 ;
10733 { \s_fp \__fp_chk:w 1 #1 {#2} {#3} {#4} {#5} {#6} ; }
```

(End definition for `__fp_exp_after_normal:nNw`.)

```
\__fp_exp_after_array_f:w
```

```
\__fp_after_stop_f:nw
10734 \cs_new:Npn \__fp_exp_after_array_f:w #1
10735 {
10736     \cs:w __fp_exp_after \__fp_type_from_scan:N #1 _f:nw \cs_end:
10737     { \__fp_exp_after_array_f:w }
10738     #1
10739 }
10740 \cs_new_eq:NN \__fp_exp_after_stop_f:nw \use_none:nn
```

(End definition for `__fp_exp_after_array_f:w` and `__fp_exp_after_stop_f:nw`.)

21.6 Packing digits

When a positive integer #1 is known to be less than 10^8 , the following trick will split 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
    \__int_value:w \__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 TeX'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
\__int_value:w \__int_eval:w - 5 0000
\exp_after:wN \pack:NNNNNw
\__int_value:w \__int_eval:w 4 9995 0000
+ 12345 * 6677
\exp_after:wN \pack:NNNNNw
\__int_value:w \__int_eval:w 5 0000 0000
+ 12345 * 8899 ;
```

The `\exp_after:wN` triggers `__int_value:w __int_eval:w`, which starts a first computation, whose initial value is -50000 (the “leading shift”). In that computation appears an `\exp_after:wN`, which triggers the nested computation `__int_value:w __int_eval:w` with starting value 499950000 (the “middle shift”). That, in turn, expands `\exp_after:wN` which triggers the third computation. The third computation's value is $500000000 + 12345 \times 8899$, 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 will also work 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 $500000000/10^4 + 499950000 = 500000000$. As long as the operands are in some range, the result of this second computation will have 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.

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.

```
10741 \int_const:Nn \c_fp_leading_shift_int { - 5 0000 }
10742 \int_const:Nn \c_fp_middle_shift_int { 5 0000 * 9999 }
10743 \int_const:Nn \c_fp_trailing_shift_int { 5 0000 * 10000 }
10744 \cs_new:Npn \__fp_pack:NNNNNw #1 #2#3#4#5 #6; { + #1#2#3#4#5 ; {#6} }
```

(End definition for `__fp_pack:NNNNNw` and others.)

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.

```
10745 \int_const:Nn \c_fp_big_leading_shift_int { - 15 2374 }
```

```

10746 \int_const:Nn \c_fp_big_middle_shift_int { 15 2374 * 9999 }
10747 \int_const:Nn \c_fp_big_trailing_shift_int { 15 2374 * 10000 }
10748 \cs_new:Npn \_fp_pack_big:NNNNNw #1#2 #3#4#5#6 #7;
10749 { + #1#2#3#4#5#6 ; {#7} }

```

(End definition for `_fp_pack_big:NNNNNw` and others.)

`_fp_pack_Bigg:NNNNNw` This set of shifts allows for computations with results in the range $[-1 \cdot 10^9, 147483647]$; the end-point is $2^{31} - 1 - 2 \cdot 10^9 \simeq 1.47 \cdot 10^8$. Shifted values all have exactly 10 digits.

```

10750 \int_const:Nn \c_fp_Bigg_leading_shift_int { - 20 0000 }
10751 \int_const:Nn \c_fp_Bigg_middle_shift_int { 20 0000 * 9999 }
10752 \int_const:Nn \c_fp_Bigg_trailing_shift_int { 20 0000 * 10000 }
10753 \cs_new:Npn \_fp_pack_Bigg:NNNNNw #1#2 #3#4#5#6 #7;
10754 { + #1#2#3#4#5#6 ; {#7} }

```

(End definition for `_fp_pack_Bigg:NNNNNw` and others.)

`_fp_pack_twice_four:wNNNNNNNN` Grabs two sets of 4 digits and places them before the semi-colon delimiter. Putting several copies of this function before a semicolon will pack more digits since each will take the digits packed by the others in its first argument.

```

10755 \cs_new:Npn \_fp_pack_twice_four:wNNNNNNNN #1; #2#3#4#5 #6#7#8#9
10756 { #1 {#2#3#4#5} {#6#7#8#9} ; }

```

(End definition for `_fp_pack_twice_four:wNNNNNNNN`.)

`_fp_pack_eight:wNNNNNNNN` 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 will pack more digits since each will take the digits packed by the others in its first argument.

```

10757 \cs_new:Npn \_fp_pack_eight:wNNNNNNNN #1; #2#3#4#5 #6#7#8#9
10758 { #1 {#2#3#4#5#6#7#8#9} ; }

```

(End definition for `_fp_pack_eight:wNNNNNNNN`.)

`_fp_basics_pack_low:NNNNW` 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.

```

10759 \cs_new:Npn \_fp_basics_pack_low:NNNNw #1 #2#3#4#5 #6;
10760 { + #1 - 1 ; {#2#3#4#5} {#6} ; }
10761 \cs_new:Npn \_fp_basics_pack_high:NNNNw #1 #2#3#4#5 #6;
10762 {
10763     \if_meaning:w 2 #1
10764         \_fp_basics_pack_high_carry:w
10765     \fi:
10766     ; {#2#3#4#5} {#6}
10767 }
10768 \cs_new:Npn \_fp_basics_pack_high_carry:w \fi: ; #1
10769 { \fi: + 1 ; {1000} }

```

(End definition for `_fp_basics_pack_low:NNNNw`, `_fp_basics_pack_high:NNNNw`, and `_fp_basics_pack_high_carry:w`.)

`_fp_basics_pack_weird_low:NNNNw` This is used in `|3fp-basics` for additions and divisions. Their syntax is confusing, hence the name.
`_fp_basics_pack_weird_high:NNNNNNNNw`

```

10770 \cs_new:Npn \_fp_basics_pack_weird_low:NNNNw #1 #2#3#4 #5;
10771 {
10772     \if_meaning:w 2 #1
10773         + 1
10774     \fi:
10775     \_int_eval_end:
10776     #2#3#4; {#5} ;
10777 }
10778 \cs_new:Npn \_fp_basics_pack_weird_high:NNNNNNNNw
10779     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`.)

21.7 Decimate (dividing by a power of 10)

`_fp_decimate:nNnnn` Each $\langle X_i \rangle$ consists in 4 digits exactly, and $1000 \leq \langle X_1 \rangle < 9999$. The first argument determines by how much we shift the digits. $\langle f_1 \rangle$ is called as follows: where $0 \leq \langle X'_i \rangle < 10^8 - 1$ are 8 digit integers, forming the truncation of our number. In other words,

$$\left(\sum_{i=1}^4 \langle X_i \rangle \cdot 10^{-4i} \cdot 10^{-\langle shift \rangle} - \langle X'_1 \rangle \cdot 10^{-8} + \langle X'_2 \rangle \cdot 10^{-16} \right) \in [0, 10^{-16}).$$

To round properly later, we need to remember some information about the difference. The $\langle rounding \rangle$ 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, $\langle rounding \rangle$ is 1 (not 0), and $\langle X'_1 \rangle$ and $\langle X'_2 \rangle$ are both zero.

If the shift is 1, the $\langle rounding \rangle$ 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 $\langle rounding \rangle$ digit to be placed after the $\langle X'_i \rangle$, but the choice we make involves less reshuffling.

Note that this function treats negative $\langle shift \rangle$ as 0.

```

10780 \cs_new:Npn \_fp_decimate:nNnnn #1
10781 {
10782     \cs:w
10783         \_fp_decimate_
10784         \if_int_compare:w \_int_eval:w #1 > \c_fp_prec_int
10785             tiny
10786         \else:
10787             \_int_to_roman:w \_int_eval:w #1
10788         \fi:
10789         :Nnnn
10790     \cs_end:
10791 }
```

Each of the auxiliaries see the function $\langle f_1 \rangle$, followed by 4 blocks of 4 digits.

(End definition for `_fp_decimate:nNnnn`.)

```
\_fp_decimate_:Nnnnn
```

```
\_fp_decimate_tiny:Nnnnn
```

If the $\langle shift \rangle$ is zero, or too big, life is very easy.

```
10792 \cs_new:Npn \_fp_decimate_:Nnnnn #1 #2#3#4#5
  { #1 0 {#2#3} {#4#5} ; }
10794 \cs_new:Npn \_fp_decimate_tiny:Nnnnn #1 #2#3#4#5
  { #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_auxii:Nnnnn
\_fp_decimate_auxiii:Nnnnn
\_fp_decimate_auxiv:Nnnnn
\_fp_decimate_auxv:Nnnnn
\_fp_decimate_auxvi:Nnnnn
\_fp_decimate_auxvii:Nnnnn
\_fp_decimate_auxviii:Nnnnn
\_fp_decimate_auxix:Nnnnn
\_fp_decimate_auxx: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
```

Shifting happens in two steps: compute the $\langle rounding \rangle$ digit, and repack digits into 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; after one step of expansion, #2 yields the “extra digits” which are then converted by $_fp_round_digit:Nw$ to the $\langle rounding \rangle$ digit (note the + separating blocks of digits to 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 rest. For this to work, #3 alternates between braced and unbraced blocks of 4 digits, in such a way that the 5 first and 5 next token groups yield the correct blocks of 8 digits.

```
10796 \cs_new:Npn \_fp_tmp:w #1 #2 #3
10797  {
10798    \cs_new:cpn { \_fp_decimate_ #1 :Nnnnn } ##1 ##2##3##4##5
10799    {
10800      \exp_after:wN ##1
10801      \_int_value:w
10802      \exp_after:wN \_fp_round_digit:Nw #2 ;
10803      \_fp_decimate_pack:nnnnnnnnnw #3 ;
10804    }
10805  }
10806 \_fp_tmp:w {i}  {\use_none:nnn      #50}{ 0{#2}#3{#4}#5      }
10807 \_fp_tmp:w {ii} {\use_none:nn      #5 }{ 00{#2}#3{#4}#5      }
10808 \_fp_tmp:w {iii} {\use_none:n      #5 }{ 000{#2}#3{#4}#5      }
10809 \_fp_tmp:w {iv}  {           #5 }{ {0000}#2{#3}#4 #5      }
10810 \_fp_tmp:w {v}   {\use_none:nnn     #4#5 }{ 0{0000}#2{#3}#4 #5      }
10811 \_fp_tmp:w {vi}  {\use_none:nn      #4#5 }{ 00{0000}#2{#3}#4 #5      }
10812 \_fp_tmp:w {vii} {\use_none:n      #4#5 }{ 000{0000}#2{#3}#4 #5      }
10813 \_fp_tmp:w {viii}{           #4#5 }{ {0000}0000{#2}#3 #4 #5      }
10814 \_fp_tmp:w {ix}  {\use_none:nnn    #3#4+#5}{ 0{0000}0000{#2}#3 #4 #5      }
10815 \_fp_tmp:w {x}   {\use_none:nn      #3#4+#5}{ 00{0000}0000{#2}#3 #4 #5      }
10816 \_fp_tmp:w {xi}  {\use_none:n      #3#4+#5}{ 000{0000}0000{#2}#3 #4 #5      }
10817 \_fp_tmp:w {xii} {           #3#4+#5}{ {0000}0000{0000}#2 #3 #4 #5      }
10818 \_fp_tmp:w {xiii}{\use_none:nnn#2#3+#4#5}{ 0{0000}0000{0000}#2 #3 #4 #5      }
10819 \_fp_tmp:w {xiv} {\use_none:nn #2#3+#4#5}{ 00{0000}0000{0000}#2 #3 #4 #5      }
10820 \_fp_tmp:w {xv}  {\use_none:n #2#3+#4#5}{ 000{0000}0000{0000}#2 #3 #4 #5      }
10821 \_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.)

```
\_fp_decimate_pack:nnnnnnnnnw
```

The computation of the $\langle rounding \rangle$ 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.

¹⁰No, the argument spec is not a mistake: the function calls an auxiliary to do half of the job.

```

10822 \cs_new:Npn \__fp_decimate_pack:nnnnnnnnnw #1#2#3#4#5
10823   { \__fp_decimate_pack:nnnnnnw { #1#2#3#4#5 } }
10824 \cs_new:Npn \__fp_decimate_pack:nnnnnnw #1 #2#3#4#5#6
10825   { {#1} {#2#3#4#5#6} }

```

(End definition for `__fp_decimate_pack:nnnnnnnnnw`.)

21.8 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 In this example, the case 0 will return the floating point $\langle fp\ var \rangle$, expanding once after that floating point. Case 1 will do $\langle some\ computation \rangle$ using the $\langle floating\ point \rangle$ (presumably compute the operation requested by the user in that non-trivial case). Case 2 will return the $\langle floating\ point \rangle$ without modifying it, removing the $\langle junk \rangle$ and expanding once after. Case 3 will close the conditional, remove the $\langle junk \rangle$ and the $\langle floating\ point \rangle$, and expand $\langle something \rangle$ next. In other cases, the “ $\langle junk \rangle$ ” is expanded, performing some other operation on the $\langle floating\ point \rangle$. We provide similar functions with two trailing $\langle floating\ points \rangle$.

- `__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.

```
10826 \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 $\langle junk \rangle$ may not contain semicolons.

```
10827 \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 $\langle fp\ var \rangle$) then expands once after it.

```
10828 \cs_new:Npn \__fp_case_return_o:Nw #1#2 \fi: #3 \s_fp #4 ;
10829   { \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.

```
10830 \cs_new:Npn \__fp_case_return_same_o:w #1 \fi: #2 \s_fp
10831   { \fi: \__fp_exp_after_o:w \s_fp }
```

(End definition for `__fp_case_return_same_o:w`.)

<code>_fp_case_return_o:Nww</code>	Same as <code>_fp_case_return_o:Nw</code> but with two trailing floating points. 10832 \cs_new:Npn _fp_case_return_o:Nww #1#2 \fi: #3 \s_fp #4 ; #5 ; 10833 { \fi: \exp_after:wN #1 }
	<i>(End definition for _fp_case_return_o:Nww.)</i>

<code>_fp_case_return_i_o:ww</code>	Similar to <code>_fp_case_return_same_o:w</code> , but this returns the first or second of two trailing floating point numbers, expanding once after the result. 10834 \cs_new:Npn _fp_case_return_i_o:ww #1 \fi: #2 \s_fp #3 ; \s_fp #4 ; 10835 { \fi: _fp_exp_after_o:w \s_fp #3 ; } 10836 \cs_new:Npn _fp_case_return_i_o:ww #1 \fi: #2 \s_fp #3 ; 10837 { \fi: _fp_exp_after_o:w }
	<i>(End definition for _fp_case_return_i_o:ww and _fp_case_return_i_o:ww.)</i>

21.9 Integer floating points

`_fp_int_p:w` Tests if the floating point argument is an integer. For normal floating point numbers, this holds if the rounding digit resulting from `_fp_decimate:nNnnnn` is 0.

```

10838 \prg_new_conditional:Npnn \_fp_int:w \s_fp \_fp_chk:w #1 #2 #3 #4;
10839   { TF , T , F , p }
10840   {
10841     \if_case:w #1 \exp_stop_f:
10842       \prg_return_true:
10843     \or:
10844       \if_charcode:w 0
10845         \_fp_decimate:nNnnnn { \c_fp_prec_int - #3 }
10846         \_fp_use_i_until_s:nw #4
10847         \prg_return_true:
10848       \else:
10849         \prg_return_false:
10850       \fi:
10851     \else: \prg_return_false:
10852     \fi:
10853   }
```

(End definition for _fp_int:wTF.)

21.10 Small integer floating points

`_fp_small_int:wTF` Tests if the floating point argument is an integer or $\pm\infty$. If so, it is converted 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 .

```

10854 \cs_new:Npn \_fp_small_int:wTF \s_fp \_fp_chk:w #1#2
10855   {
10856     \if_case:w #1 \exp_stop_f:
10857       \_fp_case_return:nw { \_fp_small_int_true:wTF 0 ; }
10858     \or: \exp_after:wN \_fp_small_int_normal:NnwTF
10859     \or:
10860       \_fp_case_return:nw
```

```

10861      {
10862          \exp_after:wN \__fp_small_int_true:wTF \__int_value:w
10863              \if_meaning:w 2 #2 - \fi: 1 0000 0000 ;
10864      }
10865      \else: \__fp_case_return:nw \use_ii:nn
10866      \fi:
10867      #2
10868  }
10869 \cs_new:Npn \__fp_small_int_true:wTF #1; #2#3 { #2 {#1} }
10870 \cs_new:Npn \__fp_small_int_normal:NnwTF #1#2#3;
10871  {
10872      \__fp_decimate:nNnnn { \c__fp_prec_int - #2 }
10873          \__fp_small_int_test:NnnwNw
10874          #3 #1
10875  }
10876 \cs_new:Npn \__fp_small_int_test:NnnwNw #1#2#3#4; #5
10877  {
10878      \if_meaning:w 0 #1
10879          \exp_after:wN \__fp_small_int_true:wTF
10880              \__int_value:w \if_meaning:w 2 #5 - \fi:
10881                  \if_int_compare:w #2 > 0 \exp_stop_f:
10882                      1 0000 0000
10883                  \else:
10884                      #3
10885                  \fi:
10886          \exp_after:wN ;
10887      \else:
10888          \exp_after:wN \use_ii:nn
10889      \fi:
10890  }

```

(End definition for `__fp_small_int:wTF` and others.)

21.11 Length of a floating point array

`__fp_array_count:n`
`__fp_array_count_loop:Nw`

Count the number of items in an array of floating points. 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.

```

10891 \cs_new:Npn \__fp_array_count:n #1
10892  {
10893      \__int_value:w \__int_eval:w 0
10894          \__fp_array_count_loop:Nw #1 { ? \__prg_break: } ;
10895          \__prg_break_point:
10896          \__int_eval_end:
10897  }
10898 \cs_new:Npn \__fp_array_count_loop:Nw #1#2;
10899  { \use_none:n #1 + 1 \__fp_array_count_loop:Nw }

```

(End definition for `__fp_array_count:n` and `__fp_array_count_loop:Nw`.)

21.12 x-like expansion expandably

`__fp_expand:n`
`__fp_expand_loop:nwnN`

This expandable function behaves in a way somewhat similar to `\use:x`, but much less robust. The argument is f-expanded, then the leading item (often a single character

token) is moved to a storage area after `\s_fp_mark`, and f-expansion is applied again, repeating until the argument is empty. The result built one piece at a time is then inserted in the input stream. Note that spaces are ignored by this procedure, unless surrounded with braces. Multiple tokens which do not need expansion can be inserted within braces.

```

10900 \cs_new:Npn \__fp_expand:n #1
10901 {
10902     \__fp_expand_loop:nwnN { }
10903     #1 \prg_do_nothing:
10904     \s_fp_mark { } \__fp_expand_loop:nwnN
10905     \s_fp_mark { } \__fp_use_i_until_s:nw ;
10906 }
10907 \cs_new:Npn \__fp_expand_loop:nwnN #1#2 \s_fp_mark #3 #4
10908 {
10909     \exp_after:wN #4 \exp:w \exp_end_continue_f:w
10910     #2
10911     \s_fp_mark { #3 #1 } #4
10912 }
```

(End definition for `__fp_expand:n` and `__fp_expand_loop:nwnN`.)

21.13 Messages

Using a floating point directly is an error.

```

10913 \__msg_kernel_new:nnnn { kernel } { misused-fp }
10914 { A-floating-point-with-value-'#1'-was-misused. }
10915 {
10916     To-obtain-the-value-of-a-floating-point-variable,-use-
10917     '\token_to_str:N \fp_to_decimal:N',-
10918     '\token_to_str:N \fp_to_scientific:N',-or-other-
10919     conversion-functions.
10920 }
10921 </initex | package>
```

22 l3fp-traps Implementation

```

10922 <*initex | package>
10923 <@=fp>
```

Exceptions should be accessed by an n-type argument, among

- `invalid_operation`
- `division_by_zero`
- `overflow`
- `underflow`
- `inexact` (actually never used).

22.1 Flags

`flag_fp_invalid_operation` Flags to denote exceptions.

```
10924 \flag_new:n { fp_invalid_operation }
10925 \flag_new:n { fp_division_by_zero }
10926 \flag_new:n { fp_overflow }
10927 \flag_new:n { fp_underflow }
```

(End definition for flag `fp_invalid_operation` and others. These variables are documented on page 183.)

22.2 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_t1_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
10928 \cs_new_protected:Npn \fp_trap:nn #1#2
10929 {
10930     \cs_if_exist_use:cF { __fp_trap_#1_set_#2: }
10931     {
10932         \clist_if_in:nnTF
10933             { invalid_operation , division_by_zero , overflow , underflow }
10934             {#1}
10935             {
10936                 \__msg_kernel_error:nnxx { kernel }
10937                 { unknown-fpu-trap-type } {#1} {#2}
10938             }
10939     }
```

```

10940           \__msg_kernel_error:nnx
10941             { kernel } { unknown-fpu-exception } {#1}
10942         }
10943     }
10944 }
```

(End definition for `\fp_trap:nn`. This function is documented on page 183.)

```
\__fp_trap_invalid_operation_set_error:
\__fp_trap_invalid_operation_set_flag:
\__fp_trap_invalid_operation_set_none:
  \__fp_trap_invalid_operation_set:N
```

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.

```

10945   \cs_new_protected:Npn \__fp_trap_invalid_operation_set_error:
10946     { \__fp_trap_invalid_operation_set:N \prg_do_nothing: }
10947   \cs_new_protected:Npn \__fp_trap_invalid_operation_set_flag:
10948     { \__fp_trap_invalid_operation_set:N \use_none:nnnnn }
10949   \cs_new_protected:Npn \__fp_trap_invalid_operation_set_none:
10950     { \__fp_trap_invalid_operation_set:N \use_none:nnnnnnn }
10951   \cs_new_protected:Npn \__fp_trap_invalid_operation_set:N #1
10952     {
10953       \exp_args:Nno \use:n
10954         { \cs_set:Npn \__fp_invalid_operation:nw ##1##2##3; }
10955       {
10956         #1
10957         \__fp_error:nfn { fp-invalid } {##2} { \fp_to_tl:n {##3}; } { }
10958         \flag_raise:n { fp_invalid_operation }
10959         ##1
10960       }
10961     \exp_args:Nno \use:n
10962       { \cs_set:Npn \__fp_invalid_operation_o:Nww ##1##2; ##3; }
10963     {
10964       #1
10965       \__fp_error:nffn { fp-invalid-ii }
10966         { \fp_to_tl:n {##2}; } { \fp_to_tl:n {##3}; } {##1}
10967         \flag_raise:n { fp_invalid_operation }
10968         \exp_after:wN \c_nan_fp
10969     }
10970   \exp_args:Nno \use:n
10971     { \cs_set:Npn \__fp_invalid_operation_tl_o:ff ##1##2 }
10972   {
10973     #1
10974     \__fp_error:nffn { fp-invalid } {##1} {##2} { }
10975     \flag_raise:n { fp_invalid_operation }
10976     \exp_after:wN \c_nan_fp
10977   }
10978 }
```

(End definition for `__fp_trap_invalid_operation_set_error:` and others.)

```
\__fp_trap_division_by_zero_set_error:
\__fp_trap_division_by_zero_set_flag:
\__fp_trap_division_by_zero_set_none:
  \__fp_trap_division_by_zero_set:N
```

We provide three types of trapping for invalid operations and division by zero: either produce an error and raise the relevant flag; or only raise the flag; or don't even raise the flag. In all cases, the function must produce a result, namely its first argument, $\pm\infty$ or NaN .

```

10979   \cs_new_protected:Npn \__fp_trap_division_by_zero_set_error:
10980     { \__fp_trap_division_by_zero_set:N \prg_do_nothing: }
```

```

10981 \cs_new_protected:Npn \__fp_trap_division_by_zero_set_flag:
10982   { \__fp_trap_division_by_zero_set:N \use_none:nnnnn }
10983 \cs_new_protected:Npn \__fp_trap_division_by_zero_set_none:
10984   { \__fp_trap_division_by_zero_set:N \use_none:nnnnnnn }
10985 \cs_new_protected:Npn \__fp_trap_division_by_zero_set:N #1
10986   {
10987     \exp_args:Nno \use:n
10988     { \cs_set:Npn \__fp_division_by_zero_o:Nnw ##1##2##3; }
10989     {
10990       #1
10991       \__fp_error:nnfn { fp-zero-div } {##2} { \fp_to_tl:n { ##3; } } { }
10992       \flag_raise:n { fp_division_by_zero }
10993       \exp_after:wN ##1
10994     }
10995     \exp_args:Nno \use:n
10996     { \cs_set:Npn \__fp_division_by_zero_o:NNww ##1##2##3; ##4; }
10997     {
10998       #1
10999       \__fp_error:nffn { fp-zero-div-ii }
11000       { \fp_to_tl:n { ##3; } } { \fp_to_tl:n { ##4; } } {##2}
11001       \flag_raise:n { fp_division_by_zero }
11002       \exp_after:wN ##1
11003     }
11004   }

```

(End definition for `__fp_trap_division_by_zero_set_error`: and others.)

```

\__fp_trap_overflow_set_error:
\__fp_trap_overflow_set_flag:
\__fp_trap_overflow_set_none:
\__fp_trap_overflow_set:N
  \__fp_trap_underflow_set_error:
  \__fp_trap_underflow_set_flag:
  \__fp_trap_underflow_set_none:
\__fp_trap_underflow_set:N
\__fp_trap_overflow_set:NnNn

```

Just as for invalid operations and division by zero, the three different behaviours are obtained by feeding `\prg_do_nothing:`, `\use_none:nnnnn` or `\use_none:nnnnnnn` to an auxiliary, with a further auxiliary common to overflow and underflow functions. In most cases, the argument of the `__fp_overflow:w` and `__fp_underflow:w` functions will be an (almost) normal number (with an exponent outside the allowed range), and the error message thus displays that number together with the result to which it overflowed or underflowed. For extreme cases such as `10 ** 1e9999`, the exponent would be too large for TeX, and `__fp_overflow:w` receives $\pm\infty$ (`__fp_underflow:w` would receive ± 0); then we cannot do better than simply say an overflow or underflow occurred.

```

11005 \cs_new_protected:Npn \__fp_trap_overflow_set_error:
11006   { \__fp_trap_overflow_set:N \prg_do_nothing: }
11007 \cs_new_protected:Npn \__fp_trap_overflow_set_flag:
11008   { \__fp_trap_overflow_set:N \use_none:nnnnn }
11009 \cs_new_protected:Npn \__fp_trap_overflow_set_none:
11010   { \__fp_trap_overflow_set:N \use_none:nnnnnnn }
11011 \cs_new_protected:Npn \__fp_trap_overflow_set:N #1
11012   { \__fp_trap_overflow_set:NnNn #1 { overflow } \__fp_inf_fp:N { inf } }
11013 \cs_new_protected:Npn \__fp_trap_underflow_set_error:
11014   { \__fp_trap_underflow_set:N \prg_do_nothing: }
11015 \cs_new_protected:Npn \__fp_trap_underflow_set_flag:
11016   { \__fp_trap_underflow_set:N \use_none:nnnnn }
11017 \cs_new_protected:Npn \__fp_trap_underflow_set_none:
11018   { \__fp_trap_underflow_set:N \use_none:nnnnnnn }
11019 \cs_new_protected:Npn \__fp_trap_underflow_set:N #1
11020   { \__fp_trap_overflow_set:NnNn #1 { underflow } \__fp_zero_fp:N { 0 } }
11021 \cs_new_protected:Npn \__fp_trap_overflow_set:NnNn #1#2#3#4
11022   {

```

```

11023   \exp_args:Nno \use:n
11024     { \cs_set:cpn { __fp_ #2 :w } \s__fp \__fp_chk:w ##1##2##3; }
11025   {
11026     #1
11027     \__fp_error:nffn
11028       { fp-flow \if_meaning:w 1 ##1 -to \fi: }
11029       { \fp_to_tl:n { \s__fp \__fp_chk:w ##1##2##3; } }
11030       { \token_if_eq_meaning:NNF 0 ##2 { - } #4 }
11031       {#2}
11032     \flag_raise:n { fp_#2 }
11033     #3 ##2
11034   }
11035 }
```

(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 operations to trigger an error, and division by zero, overflow, and underflow to act silently on their flag.
`__fp_division_by_zero_o:Nww`
`__fp_division_by_zero_o:Nww`
`__fp_overflow:w`
`__fp_underflow:w`

```

11036 \cs_new:Npn \__fp_invalid_operation:nnw #1#2#3; { }
11037 \cs_new:Npn \__fp_invalid_operation_o:Nww #1#2; #3; { }
11038 \cs_new:Npn \__fp_invalid_operation_tl_o:ff #1 #2 { }
11039 \cs_new:Npn \__fp_division_by_zero_o:Nnw #1#2#3; { }
11040 \cs_new:Npn \__fp_division_by_zero_o:NNww #1#2#3; #4; { }
11041 \cs_new:Npn \__fp_overflow:w { }
11042 \cs_new:Npn \__fp_underflow:w { }
11043 \fp_trap:nn { invalid_operation } { error }
11044 \fp_trap:nn { division_by_zero } { flag }
11045 \fp_trap:nn { overflow } { flag }
11046 \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 expanding after.
`__fp_invalid_operation_o:fw`

```

11047 \cs_new:Npn \__fp_invalid_operation_o:nw
11048   { \__fp_invalid_operation:nnw { \exp_after:wN \c_nan_fp } }
11049 \cs_generate_variant:Nn \__fp_invalid_operation_o:nw { f }
```

(End definition for `__fp_invalid_operation_o:nw`.)

22.3 Errors

`__fp_error:nnnn`
`__fp_error:nfnf`
`__fp_error:nffn`

```

11050 \cs_new:Npn \__fp_error:nnnn
11051   { \msg_kernel_expandable_error:nnnnn { kernel } }
11052 \cs_generate_variant:Nn \__fp_error:nnnn { nnf, nff }
```

(End definition for `__fp_error:nnnn`.)

22.4 Messages

Some messages.

```
11053 \__msg_kernel_new:nnn { kernel } { unknown-fpu-exception }
11054 {
11055     The~FPU~exception~'#1'~is~not~known:~
11056     that~trap~will~never~be~triggered.
11057 }
11058 {
11059     The~only~exceptions~to~which~traps~can~be~attached~are \\
11060     \iow_indent:n
11061     {
11062         * ~ invalid_operation \\
11063         * ~ division_by_zero \\
11064         * ~ overflow \\
11065         * ~ underflow
11066     }
11067 }
11068 \__msg_kernel_new:nnn { kernel } { unknown-fpu-trap-type }
11069 {
11070     The~FPU~trap~type~'#2'~is~not~known. }
11071 {
11072     The~trap~type~must~be~one~of \\
11073     \iow_indent:n
11074     {
11075         * ~ error \\
11076         * ~ flag \\
11077         * ~ none
11078     }
11079 \__msg_kernel_new:nnn { kernel } { fp-flow }
11080 {
11081     An ~ #3 ~ occurred. }
11082 \__msg_kernel_new:nnn { kernel } { fp-flow-to }
11083 {
11084     #1 ~ #3 ed ~ to ~ #2 .
11085 \__msg_kernel_new:nnn { kernel } { fp-zero-div }
11086 {
11087     Division~by~zero~in~ #1 (#2)
11088 \__msg_kernel_new:nnn { kernel } { fp-zero-div-ii }
11089 {
11090     Division~by~zero~in~ (#1) #3 (#2)
11091 \__msg_kernel_new:nnn { kernel } { fp-invalid }
11092 {
11093     Invalid~operation~ #1 (#2)
11094 \__msg_kernel_new:nnn { kernel } { fp-invalid-ii }
11095 {
11096     Invalid~operation~ (#1) #3 (#2)
11097 
```

23 I3fp-round implementation

```
11092 <*initex | package>
11093 <@@=fp>

\__fp_parse_word_trunc:N
\__fp_parse_word_floor:N
\__fp_parse_word.ceil:N
11094 \cs_new:Npn \__fp_parse_word_trunc:N
11095 {
11096     \__fp_parse_function>NNN \__fp_round_o:Nw \__fp_round_to_zero>NNN
11097 \cs_new:Npn \__fp_parse_word_floor:N
11098 {
11099     \__fp_parse_function>NNN \__fp_round_o:Nw \__fp_round_to_ninf>NNN }
```

```

11098 \cs_new:Npn \__fp_parse_word_ceil:N
11099   { \__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 This looks for +, -, 0 after round. That syntax is deprecated.

```

11100 \cs_new:Npn \__fp_parse_word_round:N #1#2
11101   {
11102     \if_meaning:w + #2
11103       \__fp_parse_round:Nw \__fp_round_to_pinf>NNN
11104     \else:
11105       \if_meaning:w 0 #2
11106         \__fp_parse_round:Nw \__fp_round_to_zero>NNN
11107       \else:
11108         \if_meaning:w - #2
11109           \__fp_parse_round:Nw \__fp_round_to_ninf>NNN
11110         \fi:
11111       \fi:
11112     \fi:
11113   \__fp_parse_function>NNN
11114   \__fp_round_o:Nw \__fp_round_to_nearest>NNN #1
11115   #2
11116 }
11117 \cs_new:Npn \__fp_parse_round:Nw
11118   #1 #2 \__fp_round_to_nearest>NNN #3#4 { #2 #1 #3 }
11119 \cs_new:Npn \__fp_parse_round_deprecation_error:Nw
11120   #1 #2 \__fp_round_to_nearest>NNN #3#4
11121 {
11122   \__fp_error:nfn { fp-deprecated } { round#4() }
11123   {
11124     \str_case:nn {#2}
11125       { { + } { ceil } { 0 } { trunc } { - } { floor } }
11126     } { }
11127   #2 #1 #3
11128 }

```

(End definition for __fp_parse_word_round:N and others.)

23.1 Rounding tools

\c_fp_five_int This is used as the half-point for which numbers are rounded up/down.

```
11129 \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`

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 `__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 will 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 $+\infty$.

```

11130 \cs_new:Npn \__fp_round_return_one:
11131   { \exp_after:wN 1 \exp_after:wN \exp_stop_f: \exp:w }
11132 \cs_new:Npn \__fp_round_to_ninf:NNN #1 #2 #3
11133   {
11134     \if_meaning:w 2 #1
11135       \if_int_compare:w #3 > 0 \exp_stop_f:
11136         \__fp_round_return_one:
11137       \fi:

```

```

11138      \fi:
11139      0 \exp_stop_f:
11140    }
11141  \cs_new:Npn \__fp_round_to_zero:NNN #1 #2 #3 { 0 \exp_stop_f: }
11142  \cs_new:Npn \__fp_round_to_pinf:NNN #1 #2 #3
11143  {
11144    \if_meaning:w 0 #1
11145      \if_int_compare:w #3 > 0 \exp_stop_f:
11146        \__fp_round_return_one:
11147      \fi:
11148    \fi:
11149    0 \exp_stop_f:
11150  }
11151  \cs_new:Npn \__fp_round_to_nearest:NNN #1 #2 #3
11152  {
11153    \if_int_compare:w #3 > \c__fp_five_int
11154      \__fp_round_return_one:
11155    \else:
11156      \if_meaning:w 5 #3
11157        \if_int_odd:w #2 \exp_stop_f:
11158          \__fp_round_return_one:
11159        \fi:
11160      \fi:
11161    \fi:
11162    0 \exp_stop_f:
11163  }
11164  \cs_new:Npn \__fp_round_to_nearest_ninf:NNN #1 #2 #3
11165  {
11166    \if_int_compare:w #3 > \c__fp_five_int
11167      \__fp_round_return_one:
11168    \else:
11169      \if_meaning:w 5 #3
11170        \if_meaning:w 2 #1
11171          \__fp_round_return_one:
11172        \fi:
11173      \fi:
11174    \fi:
11175    0 \exp_stop_f:
11176  }
11177  \cs_new:Npn \__fp_round_to_nearest_zero:NNN #1 #2 #3
11178  {
11179    \if_int_compare:w #3 > \c__fp_five_int
11180      \__fp_round_return_one:
11181    \fi:
11182    0 \exp_stop_f:
11183  }
11184  \cs_new:Npn \__fp_round_to_nearest_pinf:NNN #1 #2 #3
11185  {
11186    \if_int_compare:w #3 > \c__fp_five_int
11187      \__fp_round_return_one:
11188    \else:
11189      \if_meaning:w 5 #3
11190        \if_meaning:w 0 #1
11191          \__fp_round_return_one:

```

```

11192     \fi:
11193     \fi:
11194     \fi:
11195     0 \exp_stop_f:
11196   }
11197 \cs_new_eq:NN \_fp_round:NNN \_fp_round_to_nearest:NNN

```

(End definition for `_fp_round:NNN` and others.)

`_fp_round_s:NNNw` Similar to `_fp_round:NNN`, but with an extra semicolon, this function expands to `0\exp_stop_f:`; if rounding $\langle final\ sign\rangle\langle digit\rangle.\langle more\ digits\rangle$ to an integer truncates, and to `1\exp_stop_f:`; otherwise. The $\langle more\ digits\rangle$ part must be a digit, followed by something that does not overflow a `\int_use:N \int_eval:w` construction. The only relevant information about this piece is whether it is zero or not.

```

11198 \cs_new:Npn \_fp_round_s:NNNw #1 #2 #3 #4;
11199 {
11200   \exp_after:wN \_fp_round:NNN
11201   \exp_after:wN #1
11202   \exp_after:wN #2
11203   \int_value:w \int_eval:w
11204   \if_int_odd:w 0 \if_meaning:w 0 #3 1 \fi:
11205     \if_meaning:w 5 #3 1 \fi:
11206     \exp_stop_f:
11207     \if_int_compare:w \int_eval:w #4 > 0 \exp_stop_f:
11208       1 +
11209     \fi:
11210   \fi:
11211   #3
11212 ;
11213 }

```

(End definition for `_fp_round_s:NNNw`.)

`_fp_round_digit:Nw` This function should always be called within an `\int_value:w` or `\int_eval:w` expansion; it may add an extra `\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.

```

11214 \cs_new:Npn \_fp_round_digit:Nw #1 #2;
11215 {
11216   \if_int_odd:w \if_meaning:w 0 #1 1 \else:
11217     \if_meaning:w 5 #1 1 \else:
11218       0 \fi: \fi: \exp_stop_f:
11219     \if_int_compare:w \int_eval:w #2 > 0 \exp_stop_f:
11220       \int_eval:w 1 +
11221     \fi:
11222   \fi:
11223   #1
11224 }

```

(End definition for `_fp_round_digit:Nw`.)

`_fp_round_neg:NNN` 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
`_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`

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.

```

11225 \cs_new_eq:NN \_fp_round_to_ninf_neg:NNN \_fp_round_to_pinf:NNN
11226 \cs_new:Npn \_fp_round_to_zero_neg:NNN #1 #2 #3
11227 {
11228     \if_int_compare:w #3 > 0 \exp_stop_f:
11229         \_fp_round_return_one:
11230     \fi:
11231     0 \exp_stop_f:
11232 }
11233 \cs_new_eq:NN \_fp_round_to_pinf_neg:NNN \_fp_round_to_ninf:NNN
11234 \cs_new_eq:NN \_fp_round_to_nearest_neg:NNN \_fp_round_to_nearest:NNN
11235 \cs_new_eq:NN \_fp_round_to_nearest_ninf_neg:NNN \_fp_round_to_nearest_pinf:NNN
11236 \cs_new:Npn \_fp_round_to_nearest_zero_neg:NNN #1 #2 #3
11237 {
11238     \if_int_compare:w #3 < \c_fp_five_int \else:
11239         \_fp_round_return_one:
11240     \fi:
11241     0 \exp_stop_f:
11242 }
11243 \cs_new_eq:NN \_fp_round_to_nearest_pinf_neg:NNN \_fp_round_to_nearest_ninf:NNN
11244 \cs_new_eq:NN \_fp_round_neg:NNN \_fp_round_to_nearest_neg:NNN

```

(End definition for `_fp_round_neg:NNN` and others.)

23.2 The round function

`_fp_round_o:Nw`

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 will change #1 from `_fp_round_to_nearest:NNN` to one of its analogues.

```

11245 \cs_new:Npn \_fp_round_o:Nw #1#2 @
11246 {
11247     \if_case:w
11248         \int_eval:w \fp_array_count:n {#2} \int_eval_end:
11249             \_fp_round_no_arg_o:Nw #1 \exp:w
11250         \or: \_fp_round:Nwn #1 #2 {0} \exp:w
11251         \or: \_fp_round:Nww #1 #2 \exp:w
11252         \else: \_fp_round:Nwww #1 #2 @ \exp:w
11253     \fi:
11254     \exp_after:wN \exp_end:
11255 }

```

(End definition for `_fp_round_o:Nw`.)

`_fp_round_no_arg_o:Nw`

```

11256 \cs_new:Npn \_fp_round_no_arg_o:Nw #1
11257 {
11258     \cs_if_eq:NNTF #1 \_fp_round_to_nearest:NNN
11259         { \_fp_error:nnnn { fp-num-args } { round () } { 1 } { 3 } }
11260     {

```

```

11261      \__fp_error:nffn { fp-num-args }
11262          { \__fp_round_name_from_cs:N #1 () } { 1 } { 2 }
11263      }
11264      \exp_after:wN \c_nan_fp
11265  }

```

(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.

```

11266  \cs_new:Npn \__fp_round:Nwww #1#2 ; #3 ; \s_#fp \__fp_chk:w #4#5#6 ; #7 @
11267  {
11268      \cs_if_eq:NNTF #1 \__fp_round_to_nearest:NNN
11269      {
11270          \tl_if_empty:nTF {#7}
11271          {
11272              \exp_args:Nc \__fp_round:Nww
11273              {
11274                  \__fp_round_to_nearest
11275                  \if_meaning:w 0 #4 _zero \else:
11276                      \if_case:w #5 \exp_stop_f: _pinf \or: \else: _ninf \fi: \fi:
11277                      :NNN
11278                  }
11279                  #2 ; #3 ;
11280              }
11281              {
11282                  \__fp_error:nnnn { fp-num-args } { round () } { 1 } { 3 }
11283                  \exp_after:wN \c_nan_fp
11284              }
11285          }
11286          {
11287              \__fp_error:nffn { fp-num-args }
11288                  { \__fp_round_name_from_cs:N #1 () } { 1 } { 2 }
11289                  \exp_after:wN \c_nan_fp
11290              }
11291      }

```

(End definition for `__fp_round:Nwww.`)

`__fp_round_name_from_cs:N`

```

11292  \cs_new:Npn \__fp_round_name_from_cs:N #1
11293  {
11294      \cs_if_eq:NNTF #1 \__fp_round_to_zero:NNN { trunc }
11295      {
11296          \cs_if_eq:NNTF #1 \__fp_round_to_ninf:NNN { floor }
11297          {
11298              \cs_if_eq:NNTF #1 \__fp_round_to_pinf:NNN { ceil }
11299                  { round }
11300              }
11301          }
11302      }

```

(End definition for `__fp_round_name_from_cs:N.`)

```

\__fp_round:Nww
\__fp_round:Nwn
\__fp_round_normal:NwNNnw
\__fp_round_normal:NnnwNNnn
\__fp_round_pack:Nw
\__fp_round_normal>NNwNnn
\__fp_round_normal_end:wwNnn
\__fp_round_special:NwwNnn
\__fp_round_special_aux:Nw

11303 \cs_new:Npn \__fp_round:Nww #1#2 ; #3 ;
11304 {
11305   \__fp_small_int:wTF #3; { \__fp_round:Nwn #1#2; }
11306   {
11307     \__fp_invalid_operation_tl_o:ff
11308     { \__fp_round_name_from_cs:N #1 }
11309     { \__fp_array_to_clist:n { #2; #3; } }
11310   }
11311 }
11312 \cs_new:Npn \__fp_round:Nwn #1 \s__fp \__fp_chk:w #2#3#4; #5
11313 {
11314   \if_meaning:w 1 #2
11315     \exp_after:wN \__fp_round_normal:NwNNnw
11316     \exp_after:wN #1
11317     \int_value:w #5
11318   \else:
11319     \exp_after:wN \__fp_exp_after_o:w
11320   \fi:
11321   \s__fp \__fp_chk:w #2#3#4;
11322 }
11323 \cs_new:Npn \__fp_round_normal:NwNNnw #1#2 \s__fp \__fp_chk:w 1#3#4#5;
11324 {
11325   \__fp_decimate:nNnnnn { \c__fp_prec_int - #4 - #2 }
11326   \__fp_round_normal:NnnwNNnn #1 #3 {#4} {#2}
11327 }
11328 \cs_new:Npn \__fp_round_normal:NnnwNNnn #1#2#3#4; #5#6
11329 {
11330   \exp_after:wN \__fp_round_normal>NNwNnn
11331   \int_value:w \__int_eval:w
11332   \if_int_compare:w #2 > 0 \exp_stop_f:
11333     1 \int_value:w #2
11334     \exp_after:wN \__fp_round_pack:Nw
11335     \int_value:w \__int_eval:w 1#3 +
11336   \else:
11337     \if_int_compare:w #3 > 0 \exp_stop_f:
11338       1 \int_value:w #3 +
11339     \fi:
11340   \fi:
11341   \exp_after:wN #5
11342   \exp_after:wN #6
11343   \use_none:nnnnnnnn #3
11344   #1
11345   \__int_eval_end:
11346   0000 0000 0000 0000 ; #6
11347 }
11348 \cs_new:Npn \__fp_round_pack:Nw #1
11349   { \if_meaning:w 2 #1 + 1 \fi: \__int_eval_end: }
11350 \cs_new:Npn \__fp_round_normal>NNwNnn #1 #2
11351 {
11352   \if_meaning:w 0 #2
11353     \exp_after:wN \__fp_round_special:NwwNnn
11354     \exp_after:wN #1
11355   \fi:

```

```

11356     \__fp_pack_twice_four:wNNNNNNNN
11357     \__fp_pack_twice_four:wNNNNNNNN
11358     \__fp_round_normal_end:wwNnn
11359     ; #2
11360   }
11361 \cs_new:Npn \__fp_round_normal_end:wwNnn #1;#2;#3#4#5
11362   {
11363     \exp_after:wN \__fp_exp_after_o:w \exp:w \exp_end_continue_f:w
11364     \__fp_SANITIZE:Nw #3 #4 ; #1 ;
11365   }
11366 \cs_new:Npn \__fp_round_special:NwwNnn #1#2;#3;#4#5#6
11367   {
11368     \if_meaning:w 0 #1
11369       \__fp_case_return:nw
11370         { \exp_after:wN \__fp_zero_fp:N \exp_after:wN #4 }
11371     \else:
11372       \exp_after:wN \__fp_round_special_aux:Nw
11373       \exp_after:wN #4
11374       \__int_value:w \__int_eval:w 1
11375       \if_meaning:w 1 #1 -#6 \else: +#5 \fi:
11376     \fi:
11377   ;
11378 }
11379 \cs_new:Npn \__fp_round_special_aux:Nw #1#2;
11380   {
11381     \exp_after:wN \__fp_exp_after_o:w \exp:w \exp_end_continue_f:w
11382     \__fp_SANITIZE:Nw #1#2; {1000}{0000}{0000}{0000};
11383   }

```

(End definition for `__fp_round:Nww` and others.)

11384

24 l3fp-parse implementation

```

11385 (*initex | package)
11386 (@@=fp)

```

24.1 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.

`__fp_parse:n` Evaluates the *<floating point expression>* and leaves the result in the input stream as an internal floating point number. 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.

TEXhackers note: Registers (integers, toks, etc.) are automatically unpacked, without requiring a function such as `\int_use:N`. Invalid tokens remaining after f-expansion will lead to unrecoverable low-level T_EX errors.

(End definition for `_fp_parse:n`.)

Floating point expressions are composed of numbers, given in various forms, infix operators, such as `+`, `**`, or `,` (which joins two numbers into a list), and prefix operators, such as the unary `-`, 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.

16 Function calls with multiple arguments.

15 Function calls expecting exactly one argument.

13/14 Binary `**` and `^` (right to left).

12 Unary `+`, `-`, `!` (right to left).

10 Binary `*`, `/`, and juxtaposition (implicit `*`).

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, and parentheses accepting commas.

1 Parentheses expecting exactly one argument.

0 Start and end of the expression.

```
\c__fp_prec_funcii_int
\c__fp_prec_func_int
\c__fp_prec_hatii_int
\c__fp_prec_hat_int
\c__fp_prec_not_int
\c__fp_prec_times_int
\c__fp_prec_plus_int
\c__fp_prec_comp_int
\c__fp_prec_and_int
\c__fp_prec_or_int
\c__fp_prec_quest_int
\c__fp_prec_colon_int
\c__fp_prec_comma_int
\c__fp_prec_paren_int
\c__fp_prec_end_int

11387 \int_const:Nn \c__fp_prec_funcii_int { 16 }
11388 \int_const:Nn \c__fp_prec_func_int { 15 }
11389 \int_const:Nn \c__fp_prec_hatii_int { 14 }
11390 \int_const:Nn \c__fp_prec_hat_int { 13 }
11391 \int_const:Nn \c__fp_prec_not_int { 12 }
11392 \int_const:Nn \c__fp_prec_times_int { 10 }
11393 \int_const:Nn \c__fp_prec_plus_int { 9 }
11394 \int_const:Nn \c__fp_prec_comp_int { 7 }
11395 \int_const:Nn \c__fp_prec_and_int { 6 }
11396 \int_const:Nn \c__fp_prec_or_int { 5 }
11397 \int_const:Nn \c__fp_prec_quest_int { 4 }
11398 \int_const:Nn \c__fp_prec_colon_int { 3 }
11399 \int_const:Nn \c__fp_prec_comma_int { 2 }
11400 \int_const:Nn \c__fp_prec_paren_int { 1 }
11401 \int_const:Nn \c__fp_prec_end_int { 0 }
```

(End definition for `\c__fp_prec_funcii_int` and others.)

24.1.1 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 will grow 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 much 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_\dots_o:ww` expand what follows once. This comes at the cost of leaving tokens in the input stack, and we will 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.

24.1.2 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 the calculation $41 - 2^3 * 4 + 5$ will be done. 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 will be 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 $\langle \text{precedence} \rangle$ 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 $\langle \text{number} \rangle$, performing no computation, and finds the following binary $\langle \text{operator} \rangle$. Then it expands to

```
 $\langle \text{number} \rangle$ 
\_fp_parse_infix_\langle \text{operator} \rangle:N \langle \text{precedence} \rangle
```

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_\langle \text{operator} \rangle: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 $\langle \text{precedence} \rangle$ (of the earlier operator) to the `infix` auxiliary for the following $\langle \text{operator} \rangle$, to know whether to perform the computation of the $\langle \text{operator} \rangle$. If it should not be performed, the `infix` auxiliary expands to

```
@ \use_none:n \_fp_parse_infix_\langle \text{operator} \rangle:N
```

and otherwise it calls `_fp_parse_operand:Nw` with the precedence of the $\langle \text{operator} \rangle$ to find its second operand $\langle \text{number}_2 \rangle$ and the next $\langle \text{operator}_2 \rangle$, and expands to

```
@ \_fp_parse_apply_binary:NwNwN
\langle \text{operator} \rangle \langle \text{number}_2 \rangle
@ \_fp_parse_infix_\langle \text{operator}_2 \rangle:N
```

The `infix` function is responsible for comparing precedences, but cannot directly call the computation functions, because the first operand $\langle \text{number} \rangle$ 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 \langle \text{precedence} \rangle` with some of the expansion control removed is

```
\exp_after:wN \_fp_parse_continue:NwN
\exp_after:wN \langle \text{precedence} \rangle
\exp:w \exp_end_continue_f:w
\_fp_parse_one:Nw \langle \text{precedence} \rangle
```

This expands `_fp_parse_one:NwN` 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 will describe various subtleties.

24.1.3 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 comparision 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.

24.1.4 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 `<significand>e<exponent>`, where the `<significand>` is any non-empty string composed of decimal digits and at most one decimal separator (a period), the exponent “`e<exponent>`” is optional and is composed of an exponent mark `e` followed by a possibly empty string of signs `+` or `-` and a non-empty string of decimal digits. The `<significand>` 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 `<exponent>` 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_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 *`<significand>`* 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_mark`, and otherwise it is a case of juxtaposing numbers, such as `2\c_zero`, 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 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 \_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 {1, 2, 4, 10, 13} will not work, 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 will not be expanded if we simply perform f-expansion.

```
\DeclareDocumentCommand {\test} {m} { \fp_eval:n {#1} }
\ExplSyntaxOff
\test { 1 + \X }
```

Of course, spaces will 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 will stop the f-expansion before further macro calls are performed. However, in practice it should be enough: in particular, floating point numbers will correctly be expanded to the underlying `\s_fp ...` structure. The f-expansion is performed by `_fp_parse_expand:w`.

24.2 Main auxiliary functions

`_fp_parse_operand:Nw`

Reads the “...”, performing every computation with a precedence higher than $\langle\text{precedence}\rangle$, then expands to where the $\langle\text{operation}\rangle$ is the first operation with a lower precedence, possibly `end`, and the “...” start just after the $\langle\text{operation}\rangle$.

(End definition for `_fp_parse_operand:Nw.`)

`_fp_parse_infix_+:N`

If `+` has a precedence higher than the $\langle\text{precedence}\rangle$, cleans up a second $\langle\text{operand}\rangle$ and finds the $\langle\text{operation}_2\rangle$ which follows, and expands to Otherwise expands to A similar function exists for each infix operator.

(End definition for `_fp_parse_infix_+:N.`)

`_fp_parse_one:Nw`

Cleans up one or two operands depending on how the precedence of the next operation compares to the $\langle\text{precedence}\rangle$. If the following $\langle\text{operation}\rangle$ has a precedence higher than $\langle\text{precedence}\rangle$, expands to and otherwise expands to

(End definition for `_fp_parse_one:Nw.`)

24.3 Helpers

`_fp_parse_expand:w`

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.

```
11402 \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.

```
11403 \cs_new:Npn \_fp_parse_return_semicolon:w
11404     #1 \fi: \_fp_parse_expand:w { \fi: ; #1 }
```

(End definition for `_fp_parse_return_semicolon:w`.)

`_fp_type_from_scan:N`
`_fp_type_from_scan:w`

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 `_?`.

```
11405 \cs_new:Npx \_fp_type_from_scan:N #1
11406     {
11407         \exp_not:N \exp_after:wN \exp_not:N \_fp_type_from_scan:w
11408         \exp_not:N \token_to_str:N #1 \exp_not:N \q_mark
11409             \tl_to_str:n { s_fp _? } \exp_not:N \q_mark \exp_not:N \q_stop
11410     }
11411 \use:x
11412     {
11413         \cs_new:Npn \exp_not:N \_fp_type_from_scan:w
11414             ##1 \tl_to_str:n { s_fp } ##2 \exp_not:N \q_mark ##3 \exp_not:N \q_stop
11415             {##2}
11416     }
```

(End definition for `_fp_type_from_scan:N` and `_fp_type_from_scan:w`.)

`_fp_parse_digits_vii:N`
`_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`

These functions must be called within an `_int_value:w` or `_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

`<digits> ; <filling 0> ; <length>`

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.

```
11417 \cs_set_protected:Npn \_fp_tmp:w #1 #2 #3
11418     {
11419         \cs_new:cpn { _fp_parse_digits_ #1 :N } ##1
11420             {
11421                 \if_int_compare:w 9 < 1 \token_to_str:N ##1 \exp_stop_f:
11422                     \token_to_str:N ##1 \exp_after:wN #2 \exp:w
11423                 \else:
11424                     \_fp_parse_return_semicolon:w #3 ##1
```

```

11425     \fi:
11426     \_\_fp\_parse\_expand:w
11427   }
11428 }
11429 \_\_fp\_tmp:w {vii} \_\_fp\_parse\_digits\_vi:N { 0000000 ; 7 }
11430 \_\_fp\_tmp:w {vi} \_\_fp\_parse\_digits\_v:N { 000000 ; 6 }
11431 \_\_fp\_tmp:w {v} \_\_fp\_parse\_digits\_iv:N { 00000 ; 5 }
11432 \_\_fp\_tmp:w {iv} \_\_fp\_parse\_digits\_iii:N { 00000 ; 4 }
11433 \_\_fp\_tmp:w {iii} \_\_fp\_parse\_digits\_ii:N { 0000 ; 3 }
11434 \_\_fp\_tmp:w {ii} \_\_fp\_parse\_digits\_i:N { 000 ; 2 }
11435 \_\_fp\_tmp:w {i} \_\_fp\_parse\_digits\_N { 00 ; 1 }
11436 \cs_new:Npn \_\_fp\_parse\_digits\_N { ; ; 0 }

(End definition for \_\_fp\_parse\_digits\_vii:N and others.)

```

24.4 Parsing one number

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 will be split further). Despite the earlier f-expansion, #2 may still be expandable if it was protected by `\exp_not:N`, as may happen with the L^AT_EX 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.

```

11437 \cs_new:Npn \_\_fp\_parse\_one:Nw #1 #2
11438 {
11439   \if_catcode:w \scan_stop: \exp_not:N #2
11440     \exp_after:wN \if_meaning:w \exp_not:N #2 #2 \else:
11441       \exp_after:wN \reverse_if:N
11442     \fi:
11443     \if_meaning:w \scan_stop: #2
11444       \exp_after:wN \exp_after:wN
11445       \exp_after:wN \_\_fp\_parse\_one_fp>NN
11446     \else:
11447       \exp_after:wN \exp_after:wN
11448       \exp_after:wN \_\_fp\_parse\_one_register>NN
11449     \fi:
11450   \else:
11451     \if_int_compare:w 9 < 1 \token_to_str:N #2 \exp_stop_f:
11452       \exp_after:wN \exp_after:wN
11453       \exp_after:wN \_\_fp\_parse\_one_digit>NN
11454     \else:
11455       \exp_after:wN \exp_after:wN
11456       \exp_after:wN \_\_fp\_parse\_one_other>NN
11457     \fi:
11458   \fi:
11459   #1 #2
11460 }

(End definition for \_\_fp\_parse\_one:Nw.)

```

This function receives a *(precedence)* and a control sequence equal to `\scan_stop:` in meaning. There are three cases, dispatched using `__fp_type_from_scan:N`.

- `\s__fp` starts a floating point number, and we call `_fp_exp_after_f:nw`, which f-expands after the floating point.
- `\s__fp_mark` is a premature end, we call `_fp_exp_after_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 L^AT_EX 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 L^AT_EX 2 _{ε} uses `\@unexpandable@protect` rather than `\protect` because `\protect` is often `\scan_stop`: hence “does not exist”.

```

11461 \cs_new:Npn \_fp_parse_one_fp:NN #1#2
11462   {
11463     \cs:w \_fp_exp_after \_fp_type_from_scan:N #2 _f:nw \cs_end:
11464     {
11465       \exp_after:wN \_fp_parse_infix:NN
11466       \exp_after:wN #1 \exp:w \_fp_parse_expand:w
11467     }
11468   #2
11469 }
11470 \cs_new:Npn \_fp_exp_after_mark_f:nw #1
11471   {
11472     \_msg_kernel_expandable_error:nn { kernel } { fp-early-end }
11473     \exp_after:wN \c_nan_fp \exp:w \exp_end_continue_f:w #1
11474   }
11475 \cs_new:cpn { \_fp_exp_after_?_f:nw } #1#2
11476   {
11477     \_msg_kernel_expandable_error:nnn { kernel } { bad-variable } {##2}
11478     \exp_after:wN \c_nan_fp \exp:w \exp_end_continue_f:w #1
11479   }
11480 (*package)
11481 \cs_set_protected:Npn \_fp_tmp:w #1
11482   {
11483     \cs_if_exist:NT #1
11484     {
11485       \cs_gset:cpn { \_fp_exp_after_?_f:nw } ##1##2
11486       {
11487         \exp_after:wn \c_nan_fp \exp:w \exp_end_continue_f:w ##1
11488         \str_if_eq:nnTF {##2} { \protect }
11489         {
11490           \cs_if_eq:NNTF ##2 #1 { \use_i:nn } { \use:n }
11491           { \_msg_kernel_expandable_error:nnn { kernel } { fp-robust-cmd } }
11492         }
11493         { \_msg_kernel_expandable_error:nnn { kernel } { bad-variable } {##2} }
11494       }
11495     }
11496   }
11497 \exp_args:Nc \_fp_tmp:w { @unexpandable@protect }
11498 (/package)

```

(End definition for `_fp_parse_one_fp>NN`, `_fp_exp_after_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  
  \_fp_parse_one_register_wd:w  
  \_fp_parse_one_register_wd:Nw
```

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 `<value>e<exponent>` to a floating point number with `_fp_parse:n` (this is somewhat wasteful). For other registers, the decimal rounding provided by TeX 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_eval:w <decimal value> pt`, and use an auxiliary of `\dim_to_fp:n`, which performs the multiplication by 2^{-16} , correctly rounded.

```
11499 \cs_new:Npn \_fp_parse_one_register:NN #1#2  
11500 {  
11501   \exp_after:wN \_fp_parse_infix_after_operand:NwN  
11502   \exp_after:wN #1  
11503   \exp:w \exp_end_continue_f:w  
11504   \if_meaning:w \box_wd:N #2 \_fp_parse_one_register_wd:w \fi:  
11505   \if_meaning:w \box_ht:N #2 \_fp_parse_one_register_wd:w \fi:  
11506   \if_meaning:w \box_dp:N #2 \_fp_parse_one_register_wd:w \fi:  
11507   \exp_after:wN \_fp_parse_one_register_aux:Nw  
11508   \exp_after:wN #2  
11509   \_int_value:w  
11510   \exp_after:wN \_fp_parse_exponent:N  
11511   \exp:w \_fp_parse_expand:w  
11512 }  
11513 \cs_new:Npx \_fp_parse_one_register_aux:Nw #1  
11514 {  
11515   \exp_not:n  
11516   {  
11517     \exp_after:wN \use:nn  
11518     \exp_after:wN \_fp_parse_one_register_auxii:wwwNw  
11519   }  
11520   \exp_not:N \exp_after:wN { \exp_not:N \tex_the:D #1 }  
11521   ; \exp_not:N \_fp_parse_one_register_dim:ww  
11522   \tl_to_str:n { pt } ; \exp_not:N \_fp_parse_one_register_mu:www  
11523   . \tl_to_str:n { pt } ; \exp_not:N \_fp_parse_one_register_int:www  
11524   \exp_not:N \q_stop  
11525 }  
11526 \use:x  
11527 {  
11528   \cs_new:Npn \exp_not:N \_fp_parse_one_register_auxii:wwwNw  
11529     ##1 . ##2 \tl_to_str:n { pt } ##3 ; ##4##5 \exp_not:N \q_stop  
11530     { ##4 ##1.##2; }  
11531   \cs_new:Npn \exp_not:N \_fp_parse_one_register_mu:www  
11532     ##1 \tl_to_str:n { mu } ; ##2 ;  
11533     { \exp_not:N \_fp_parse_one_register_dim:ww ##1 ; }  
11534 }  
11535 \cs_new:Npn \_fp_parse_one_register_int:www #1; #2.; #3;  
11536   { \_fp_parse:n { #1 e #3 } }  
11537 \cs_new:Npn \_fp_parse_one_register_dim:ww #1; #2;
```

```

11538  {
11539      \exp_after:wN \_fp_from_dim_test:ww
11540      \_int_value:w #2 \exp_after:wN ,
11541      \_int_value:w \_dim_eval:w #1 pt ;
11542  }

```

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.

```

11543 \cs_new:Npn \_fp_parse_one_register_wd:w
11544     #1#2 \exp_after:wN #3#4 \_fp_parse_expand:w
11545  {
11546      #1
11547      \exp_after:wN \_fp_parse_one_register_wd:Nw
11548      #4 \_fp_parse_expand:w e
11549  }
11550 \cs_new:Npn \_fp_parse_one_register_wd:Nw #1#2 ;
11551  {
11552      \exp_after:wN \_fp_from_dim_test:ww
11553      \exp_after:wN 0 \exp_after:wN ,
11554      \_int_value:w \_dim_eval:w
11555      \exp_after:wN \use:n \exp_after:wN { \tex_the:D #1 #2 } ;
11556  }

```

(End definition for `_fp_parse_one_register>NN` and others.)

`_fp_parse_one_digit:NN`

A digit marks the beginning of an explicit floating point number. Once the number is found, we will 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.

```

11557 \cs_new:Npn \_fp_parse_one_digit:NN #
11558  {
11559      \exp_after:wN \_fp_parse_infix_after_operand:NwN
11560      \exp_after:wN #1
11561      \exp:w \exp_end_continue_f:w
11562      \exp_after:wN \_fp_sanitize:wN
11563      \_int_value:w \_int_eval:w 0 \_fp_parse_trim_zeros:N
11564  }

```

(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`.

```

11565 \cs_new:Npn \_fp_parse_one_other:NN #1 #2
11566  {
11567      \if_int_compare:w
11568          \_int_eval:w
11569          ( '#2 \if_int_compare:w '#2 > 'Z - 32 \fi: ) / 26
11570          = 3 \exp_stop_f:
11571      \exp_after:wN \_fp_parse_word:Nw

```

```

11572   \exp_after:wN #1
11573   \exp_after:wN #2
11574   \exp:w \exp_after:wN \__fp_parse_letters:N
11575   \exp:w
11576 \else:
11577   \exp_after:wN \__fp_parse_prefix>NNN
11578   \exp_after:wN #1
11579   \exp_after:wN #2
11580   \cs:w
11581     __fp_parse_prefix_ \token_to_str:N #2 :Nw
11582     \exp_after:wN
11583   \cs_end:
11584   \exp:w
11585 \fi:
11586 \__fp_parse_expand:w
11587 }

```

(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, 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.

```

11588 \cs_new:Npn \__fp_parse_word:Nw #1#2;
11589 {
11590   \cs_if_exist_use:cF { __fp_parse_word:#2:N }
11591   {
11592     \cs_if_exist_use:cF { __fp_parse_caseless_ \str_fold_case:n {#2} :N }
11593     {
11594       \__msg_kernel_expandable_error:nnn
11595         { kernel } { unknown-fp-word } {#2}
11596       \exp_after:wN \c_nan_fp \exp:w \exp_end_continue_f:w
11597       \__fp_parse_infix:NN
11598     }
11599   }
11600   #1
11601 }
11602 \cs_new:Npn \__fp_parse_letters:N #
11603 {
11604   \exp_end_continue_f:w
11605   \if_int_compare:w
11606     \if_catcode:w \scan_stop: \exp_not:N #1
11607     0
11608   \else:
11609     \__int_eval:w
11610       ( '#1 \if_int_compare:w '#1 > 'Z - 32 \fi: ) / 26
11611   \fi:
11612   = 3 \exp_stop_f:
11613   \exp_after:wN #

```

```

11614     \exp:w \exp_after:wN \__fp_parse_letters:N
11615     \exp:w
11616     \else:
11617         \__fp_parse_return_semicolon:w #1
11618     \fi:
11619     \__fp_parse_expand:w
11620 }

```

(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`.

```

11621 \cs_new:Npn \__fp_parse_prefix:NNN #1#2#3
11622 {
11623     \if_meaning:w \scan_stop: #3
11624         \exp_after:wN \__fp_parse_prefix_unknown:NNN
11625         \exp_after:wN #2
11626     \fi:
11627     #3 #1
11628 }
11629 \cs_new:Npn \__fp_parse_prefix_unknown:NNN #1#2#3
11630 {
11631     \cs_if_exist:cTF { __fp_parse_infix_ \token_to_str:N #1 :N }
11632     {
11633         \__msg_kernel_expandable_error:nnn
11634             { kernel } { fp-missing-number } {#1}
11635         \exp_after:wN \c_nan_fp \exp:w \exp_end_continue_f:w
11636         \__fp_parse_infix:NN #3 #1
11637     }
11638     {
11639         \__msg_kernel_expandable_error:nnn
11640             { kernel } { fp-unknown-symbol } {#1}
11641         \__fp_parse_one:Nw #3
11642     }
11643 }
```

(End definition for `__fp_parse_prefix:NNN` and `__fp_parse_prefix_unknown:NNN`.)

24.4.1 Numbers: trimming leading zeros

Numbers will be 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.

```
11644 \cs_new:Npn \_\_fp\_parse\_trim\_zeros:N #1  
11645 {  
11646   \if:w 0 \exp_not:N #1  
11647     \exp_after:wN \_\_fp\_parse\_trim\_zeros:N  
11648     \exp:w  
11649   \else:  
11650     \if:w . \exp_not:N #1  
11651       \exp_after:wN \_\_fp\_parse\_strim\_zeros:N  
11652       \exp:w  
11653     \else:  
11654       \_\_fp\_parse\_trim\_end:w #1  
11655     \fi:  
11656   \fi:  
11657   \_\_fp\_parse\_expand:w  
11658 }  
11659 \cs_new:Npn \_\_fp\_parse\_trim\_end:w #1 \fi: \fi: \_\_fp\_parse\_expand:w  
11660 {  
11661   \fi:  
11662   \fi:  
11663   \if_int_compare:w 9 < 1 \token_to_str:N #1 \exp_stop_f:  
11664     \exp_after:wN \_\_fp\_parse\_large:N  
11665   \else:  
11666     \exp_after:wN \_\_fp\_parse\_zero:  
11667   \fi:  
11668 #1  
11669 }
```

(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”.

```
11670 \cs_new:Npn \_\_fp\_parse\_strim\_zeros:N #1  
11671 {  
11672   \if:w 0 \exp_not:N #1  
11673     - 1  
11674     \exp_after:wN \_\_fp\_parse\_strim\_zeros:N \exp:w  
11675   \else:  
11676     \_\_fp\_parse\_strim\_end:w #1  
11677   \fi:  
11678   \_\_fp\_parse\_expand:w  
11679 }  
11680 \cs_new:Npn \_\_fp\_parse\_strim\_end:w #1 \fi: \_\_fp\_parse\_expand:w  
11681 {  
11682   \fi:  
11683   \if_int_compare:w 9 < 1 \token_to_str:N #1 \exp_stop_f:
```

```

11684     \exp_after:wN \__fp_parse_small:N
11685     \else:
11686     \exp_after:wN \__fp_parse_zero:
11687     \fi:
11688     #1
11689 }

```

(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 will remove everything and leave an exact zero.

```

11690 \cs_new:Npn \__fp_parse_zero:
11691 {
11692     \exp_after:wN ; \exp_after:wN 1
11693     \int_value:w \__fp_parse_exponent:N
11694 }

```

(End definition for `__fp_parse_zero:..`)

24.4.2 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 will leave those digits in the `\int_value:w`, and grab some more, or stop if there are no more digits. Then the `pack_leading` auxiliary puts the various parts in the appropriate order for the processing further up.

```

11695 \cs_new:Npn \__fp_parse_small:N #1
11696 {
11697     \exp_after:wN \__fp_parse_pack_leading:NNNNNww
11698     \int_value:w \int_eval:w 1 \token_to_str:N #1
11699     \exp_after:wN \__fp_parse_small_leading:wwNN
11700     \int_value:w 1
11701     \exp_after:wN \__fp_parse_digits_vii:N
11702     \exp:w \__fp_parse_expand:w
11703 }

```

(End definition for `__fp_parse_small:N`.)

`__fp_parse_small_leading:wwNN` We leave $\langle digits \rangle \langle zeros \rangle$ 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.

```

11704 \cs_new:Npn \__fp_parse_small_leading:wwNN 1 #1 ; #2; #3 #4
11705 {
11706     #1 #2
11707     \exp_after:wN \__fp_parse_pack_trailing:NNNNNNNww
11708     \exp_after:wN 0

```

```

11709     \__int_value:w \__int_eval:w 1
11710     \if_int_compare:w 9 < 1 \token_to_str:N #4 \exp_stop_f:
11711         \token_to_str:N #4
11712         \exp_after:wN \__fp_parse_small_trailing:wwNN
11713         \__int_value:w 1
11714         \exp_after:wN \__fp_parse_digits_vi:N
11715         \exp:w
11716     \else:
11717         0000 0000 \__fp_parse_exponent:Nw #4
11718     \fi:
11719     \__fp_parse_expand:w
11720 }

```

(End definition for `__fp_parse_small_leading:wwNN`.)

`__fp_parse_small_trailing:wwNN` 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.

```

11721 \cs_new:Npn \__fp_parse_small_trailing:wwNN 1 #1 ; #2; #3 #4
11722 {
11723     #1 #2
11724     \if_int_compare:w 9 < 1 \token_to_str:N #4 \exp_stop_f:
11725         \token_to_str:N #4
11726         \exp_after:wN \__fp_parse_small_round:NN
11727         \exp_after:wN #4
11728         \exp:w
11729     \else:
11730         0 \__fp_parse_exponent:Nw #4
11731     \fi:
11732     \__fp_parse_expand:w
11733 }

```

(End definition for `__fp_parse_small_trailing:wwNN`.)

`__fp_parse_pack_trailing:NNNNNNww`
`__fp_parse_pack_leading:NNNNNNww`
`__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.

```

11734 \cs_new:Npn \__fp_parse_pack_trailing:NNNNNNww #1 #2 #3#4#5#6 #7; #8 ;
11735 {
11736     \if_meaning:w 2 #2 + 1 \fi:
11737     ; #8 + #1 ; {#3#4#5#6} {-#7};
11738 }
11739 \cs_new:Npn \__fp_parse_pack_leading:NNNNNNww #1 #2#3#4#5 #6; #7;
11740 {
11741     + #7

```

```

11742      \if_meaning:w 2 #1 \__fp_parse_pack_carry:w \fi:
11743      ; 0 {#2#3#4#5} {#6}
11744  }
11745 \cs_new:Npn \__fp_parse_pack_carry:w \fi: ; 0 #1
11746  { \fi: + 1 ; 0 {1000} }

(End definition for \__fp_parse_pack_trailing:NNNNNNww, \__fp_parse_pack_leading:NNNNNww, and
\__fp_parse_pack_carry:w.)

```

24.4.3 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.

```

11747 \cs_new:Npn \__fp_parse_large:N #1
11748  {
11749      \exp_after:wN \__fp_parse_large_leading:wwNN
11750      \__int_value:w 1 \token_to_str:N #1
11751      \exp_after:wN \__fp_parse_digits_vii:N
11752      \exp:w \__fp_parse_expand:w
11753  }

(End definition for \__fp_parse_large:N.)

```

`__fp_parse_large_leading:wwNN` We shift the exponent by the number of digits in `#1`, namely the target number, 8, minus the $\langle \text{number of zeros} \rangle$ (number of digits missing). Then prepare to pack the 8 first digits. If the $\langle \text{next token} \rangle$ 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 $\langle \text{zeros} \rangle$ to complete the 8 first digits, insert 8 more, and look for an exponent.

```

11754 \cs_new:Npn \__fp_parse_large_leading:wwNN 1 #1 ; #2; #3 #4
11755  {
11756      + \c_fp_half_prec_int - #3
11757      \exp_after:wN \__fp_parse_pack_leading:NNNNNww
11758      \__int_value:w \__int_eval:w 1 #1
11759      \if_int_compare:w 9 < 1 \token_to_str:N #4 \exp_stop_f:
11760          \exp_after:wN \__fp_parse_large_trailing:wwNN
11761          \__int_value:w 1 \token_to_str:N #4
11762          \exp_after:wN \__fp_parse_digits_vi:N
11763          \exp:w
11764      \else:
11765          \if:w . \exp_not:N #4
11766              \exp_after:wN \__fp_parse_small_leading:wwNN
11767              \__int_value:w 1
11768              \cs:w
11769                  __fp_parse_digits_
11770                  \__int_to_roman:w #3
11771                  :N \exp_after:wN
11772                  \cs_end:
11773
11774
11775
11776
11777
11778
11779
11780
11781
11782
11783
11784
11785
11786
11787
11788
11789
11790
11791
11792
11793
11794
11795
11796
11797
11798
11799
11800
11801
11802
11803
11804
11805
11806
11807
11808
11809
11810
11811
11812
11813
11814
11815
11816
11817
11818
11819
11820
11821
11822
11823
11824
11825
11826
11827
11828
11829
11830
11831
11832
11833
11834
11835
11836
11837
11838
11839
11840
11841
11842
11843
11844
11845
11846
11847
11848
11849
11850
11851
11852
11853
11854
11855
11856
11857
11858
11859
11860
11861
11862
11863
11864
11865
11866
11867
11868
11869
11870
11871
11872
11873
11874
11875
11876
11877
11878
11879
11880
11881
11882
11883
11884
11885
11886
11887
11888
11889
11890
11891
11892
11893
11894
11895
11896
11897
11898
11899
11900
11901
11902
11903
11904
11905
11906
11907
11908
11909
11910
11911
11912
11913
11914
11915
11916
11917
11918
11919
11920
11921
11922
11923
11924
11925
11926
11927
11928
11929
11930
11931
11932
11933
11934
11935
11936
11937
11938
11939
11940
11941
11942
11943
11944
11945
11946
11947
11948
11949
11950
11951
11952
11953
11954
11955
11956
11957
11958
11959
11960
11961
11962
11963
11964
11965
11966
11967
11968
11969
11970
11971
11972
11973
11974
11975
11976
11977
11978
11979
11980
11981
11982
11983
11984
11985
11986
11987
11988
11989
11990
11991
11992
11993
11994
11995
11996
11997
11998
11999
119999
120000
120001
120002
120003
120004
120005
120006
120007
120008
120009
120010
120011
120012
120013
120014
120015
120016
120017
120018
120019
120020
120021
120022
120023
120024
120025
120026
120027
120028
120029
120030
120031
120032
120033
120034
120035
120036
120037
120038
120039
120040
120041
120042
120043
120044
120045
120046
120047
120048
120049
120050
120051
120052
120053
120054
120055
120056
120057
120058
120059
120060
120061
120062
120063
120064
120065
120066
120067
120068
120069
120070
120071
120072
120073
120074
120075
120076
120077
120078
120079
120080
120081
120082
120083
120084
120085
120086
120087
120088
120089
120090
120091
120092
120093
120094
120095
120096
120097
120098
120099
1200999
1201000
1201001
1201002
1201003
1201004
1201005
1201006
1201007
1201008
1201009
1201010
1201011
1201012
1201013
1201014
1201015
1201016
1201017
1201018
1201019
1201020
1201021
1201022
1201023
1201024
1201025
1201026
1201027
1201028
1201029
1201030
1201031
1201032
1201033
1201034
1201035
1201036
1201037
1201038
1201039
1201040
1201041
1201042
1201043
1201044
1201045
1201046
1201047
1201048
1201049
1201050
1201051
1201052
1201053
1201054
1201055
1201056
1201057
1201058
1201059
1201060
1201061
1201062
1201063
1201064
1201065
1201066
1201067
1201068
1201069
1201070
1201071
1201072
1201073
1201074
1201075
1201076
1201077
1201078
1201079
1201080
1201081
1201082
1201083
1201084
1201085
1201086
1201087
1201088
1201089
1201090
1201091
1201092
1201093
1201094
1201095
1201096
1201097
1201098
1201099
12010999
12011000
12011001
12011002
12011003
12011004
12011005
12011006
12011007
12011008
12011009
12011010
12011011
12011012
12011013
12011014
12011015
12011016
12011017
12011018
12011019
12011020
12011021
12011022
12011023
12011024
12011025
12011026
12011027
12011028
12011029
12011030
12011031
12011032
12011033
12011034
12011035
12011036
12011037
12011038
12011039
12011040
12011041
12011042
12011043
12011044
12011045
12011046
12011047
12011048
12011049
12011050
12011051
12011052
12011053
12011054
12011055
12011056
12011057
12011058
12011059
12011060
12011061
12011062
12011063
12011064
12011065
12011066
12011067
12011068
12011069
12011070
12011071
12011072
12011073
12011074
12011075
12011076
12011077
12011078
12011079
12011080
12011081
12011082
12011083
12011084
12011085
12011086
12011087
12011088
12011089
12011090
12011091
12011092
12011093
12011094
12011095
12011096
12011097
12011098
12011099
120110999
120111000
120111001
120111002
120111003
120111004
120111005
120111006
120111007
120111008
120111009
120111010
120111011
120111012
120111013
120111014
120111015
120111016
120111017
120111018
120111019
120111020
120111021
120111022
120111023
120111024
120111025
120111026
120111027
120111028
120111029
120111030
120111031
120111032
120111033
120111034
120111035
120111036
120111037
120111038
120111039
120111040
120111041
120111042
120111043
120111044
120111045
120111046
120111047
120111048
120111049
120111050
120111051
120111052
120111053
120111054
120111055
120111056
120111057
120111058
120111059
120111060
120111061
120111062
120111063
120111064
120111065
120111066
120111067
120111068
120111069
120111070
120111071
120111072
120111073
120111074
120111075
120111076
120111077
120111078
120111079
120111080
120111081
120111082
120111083
120111084
120111085
120111086
120111087
120111088
120111089
120111090
120111091
120111092
120111093
120111094
120111095
120111096
120111097
120111098
120111099
1201110999
1201111000
1201111001
1201111002
1201111003
1201111004
1201111005
1201111006
1201111007
1201111008
1201111009
1201111010
1201111011
1201111012
1201111013
1201111014
1201111015
1201111016
1201111017
1201111018
1201111019
1201111020
1201111021
1201111022
1201111023
1201111024
1201111025
1201111026
1201111027
1201111028
1201111029
1201111030
1201111031
1201111032
1201111033
1201111034
1201111035
1201111036
1201111037
1201111038
1201111039
1201111040
1201111041
1201111042
1201111043
1201111044
1201111045
1201111046
1201111047
1201111048
1201111049
1201111050
1201111051
1201111052
1201111053
1201111054
1201111055
1201111056
1201111057
1201111058
1201111059
1201111060
1201111061
1201111062
1201111063
1201111064
1201111065
1201111066
1201111067
1201111068
1201111069
1201111070
1201111071
1201111072
1201111073
1201111074
1201111075
1201111076
1201111077
1201111078
1201111079
1201111080
1201111081
1201111082
1201111083
1201111084
1201111085
1201111086
1201111087
1201111088
1201111089
1201111090
1201111091
1201111092
1201111093
1201111094
1201111095
1201111096
1201111097
1201111098
1201111099
12011110999
12011111000
12011111001
12011111002
12011111003
12011111004
12011111005
12011111006
12011111007
12011111008
12011111009
12011111010
12011111011
12011111012
12011111013
12011111014
12011111015
12011111016
12011111017
12011111018
12011111019
12011111020
12011111021
12011111022
12011111023
12011111024
12011111025
12011111026
12011111027
12011111028
12011111029
12011111030
12011111031
12011111032
12011111033
12011111034
12011111035
12011111036
12011111037
12011111038
12011111039
12011111040
12011111041
12011111042
12011111043
12011111044
12011111045
12011111046
12011111047
12011111048
12011111049
12011111050
12011111051
12011111052
12011111053
12011111054
12011111055
12011111056
12011111057
12011111058
12011111059
12011111060
12011111061
12011111062
12011111063
12011111064
12011111065
12011111066
12011111067
12011111068
12011111069
12011111070
12011111071
12011111072
12011111073
12011111074
12011111075
12011111076
12011111077
12011111078
12011111079
12011111080
12011111081
12011111082
12011111083
12011111084
12011111085
12011111086
12011111087
12011111088
12011111089
12011111090
12011111091
12011111092
12011111093
12011111094
12011111095
12011111096
12011111097
12011111098
12011111099
120111110999
120111111000
120111111001
120111111002
120111111003
120111111004
120111111005
120111111006
120111111007
120111111008
120111111009
120111111010
120111111011
120111111012
120111111013
120111111014
120111111015
120111111016
120111111017
120111111018
120111111019
120111111020
120111111021
120111111022
120111111023
120111111024
120111111025
120111111026
120111111027
120111111028
120111111029
120111111030
120111111031
120111111032
120111111033
120111111034
120111111035
120111111036
120111111037
120111111038
120111111039
120111111040
120111111041
120111111042
120111111043
120111111044
120111111045
120111111046
120111111047
120111111048
120111111049
120111111050
120111111051
120111111052
120111111053
120111111054
120111111055
120111111056
120111111057
120111111058
120111111059
120111111060
120111111061
120111111062
120111111063
120111111064
120111111065
120111111066
120111111067
120111111068
120111111069
120111111070
120111111071
120111111072
120111111073
120111111074
120111111075
120111111076
120111111077
120111111078
120111111079
120111111080
120111111081
120111111082
120111111083
120111111084
120111111085
120111111086
120111111087
120111111088
120111111089
120111111090
120111111091
120111111092
120111111093
120111111094
120111111095
120111111096
120111111097
120111111098
120111111099
1201111110999
1201111111000
1201111111001
1201111111002
1201111111003
1201111111004
1201111111005
1201111111006
1201111111007
1201111111008
1201111111009
1201111111010
1201111111011
1201111111012
1201111111013
1201111111014
1201111111015
1201111111016
1201111111017
1201111111018
1201111111019
1201111111020
1201111111021
1201111111022
1201111111023
1201111111024
1201111111025
1201111111026
1201111111027
1201111111028
1201111111029
1201111111030
1201111111031
1201111111032
1201111111033
1201111111034
1201111111035
1201111111036
1201111111037
1201111111038
1201111111039
1201111111040
1201111111041
1201111111042
1201111111043
1201111111044
1201111111045
1201111111046
1201111111047
1201111111048
1201111111049
1201111111050
1201111111051
1201111111052
1201111111053
1201111111054
1201111111055
1201111111056
1201111111057
1201111111058
1201111111059
1201111111060
1201111111061
1201111111062
1201111111063
1201111111064
1201111111065
1201111111066
1201111111067
1201111111068
1201111111069
1201111111070
1201111111071
1201111111072
1201111111073
1201111111074
1201111111075
1201111111076
1201111111077
1201111111078
1201111111079
1201111111080
1201111111081
1201111111082
1201111111083
1201111111084
1201111111085
1201111111086
1201111111087
1201111111088
1201111111089
1201111111090
1201111111091
1201111111092
1201111111093
1201111111094
1201111111095
1201111111096
1201111111097
1201111111098
1201111111099
12011111110999
12011111111000
12011111111001
12011111111002
12011111111003
12011111111004
12011111111005
12011111111006
12011111111007
12011111111008
120111
```

```

11773           \exp:w
11774     \else:
11775       #2
11776       \exp_after:wN \__fp_parse_pack_trailing:NNNNNNww
11777       \exp_after:wN 0
11778       \__int_value:w 1 0000 0000
11779       \__fp_parse_exponent:Nw #4
11780   \fi:
11781 \fi:
11782 \__fp_parse_expand:w
11783 }

```

(End definition for `__fp_parse_large_leading:wwNN`.)

`__fp_parse_large_trailing:wwNN` 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.

```

11784 \cs_new:Npn \__fp_parse_large_trailing:wwNN 1 #1 ; #2; #3 #4
11785 {
11786   \if_int_compare:w 9 < 1 \token_to_str:N #4 \exp_stop_f:
11787   \exp_after:wN \__fp_parse_pack_trailing:NNNNNNww
11788   \exp_after:wN \c__fp_half_prec_int
11789   \__int_value:w \__int_eval:w 1 #1 \token_to_str:N #4
11790   \exp_after:wN \__fp_parse_large_round:NN
11791   \exp_after:wN #4
11792   \exp:w
11793 \else:
11794   \exp_after:wN \__fp_parse_pack_trailing:NNNNNNww
11795   \__int_value:w \__int_eval:w 7 - #3 \exp_stop_f:
11796   \__int_value:w \__int_eval:w 1 #1
11797   \if:w . \exp_not:N #4
11798   \exp_after:wN \__fp_parse_small_trailing:wwNN
11799   \__int_value:w 1
11800   \cs:w
11801     __fp_parse_digits_
11802     \__int_to_roman:w #3
11803     :N \exp_after:wN
11804     \cs_end:
11805     \exp:w
11806   \else:
11807     #2 0 \__fp_parse_exponent:Nw #4
11808   \fi:
11809 \fi:
11810 \__fp_parse_expand:w
11811 }

```

(End definition for `__fp_parse_large_trailing:wwNN`.)

24.4.4 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). 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.

```

11812 \cs_new:Npn \_\_fp_parse_round_loop:N #1
11813 {
11814     \if_int_compare:w 9 < 1 \token_to_str:N #1 \exp_stop_f:
11815         + 1
11816         \if:w 0 \token_to_str:N #1
11817             \exp_after:wN \_\_fp_parse_round_loop:N
11818             \exp:w
11819         \else:
11820             \exp_after:wN \_\_fp_parse_round_up:N
11821             \exp:w
11822         \fi:
11823     \else:
11824         \_\_fp_parse_return_semicolon:w 0 #1
11825     \fi:
11826     \_\_fp_parse_expand:w
11827 }
11828 \cs_new:Npn \_\_fp_parse_round_up:N #1
11829 {
11830     \if_int_compare:w 9 < 1 \token_to_str:N #1 \exp_stop_f:
11831         + 1
11832         \exp_after:wN \_\_fp_parse_round_up:N
11833         \exp:w
11834     \else:
11835         \_\_fp_parse_return_semicolon:w 1 #1
11836     \fi:
11837     \_\_fp_parse_expand:w
11838 }
```

(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.

```

11839 \cs_new:Npn \_\_fp_parse_round_after:wN #1; #2
11840 {
11841     + #2 \exp_after:wN ;
11842     \_\_int_value:w \_\_int_eval:w #1 + \_\_fp_parse_exponent:N
11843 }
```

(End definition for `__fp_parse_round_after:wN`.)

__fp_parse_small_round:NN

__fp_parse_round_after:wN

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 will 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`.

```

11844 \cs_new:Npn \_fp_parse_small_round:NN #1#2
11845 {
11846     \if_int_compare:w 9 < 1 \token_to_str:N #2 \exp_stop_f:
11847         +
11848         \exp_after:wN \_fp_round_s:NNNw
11849         \exp_after:wN 0
11850         \exp_after:wN #1
11851         \exp_after:wN #2
11852         \_int_value:w \_int_eval:w
11853             \exp_after:wN \_fp_parse_round_after:wN
11854             \_int_value:w \_int_eval:w 0 * \_int_eval:w 0
11855                 \exp_after:wN \_fp_parse_round_loop:N
11856                 \exp:w
11857         \else:
11858             \_fp_parse_exponent:Nw #2
11859         \fi:
11860         \_fp_parse_expand:w
11861     }

```

(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.

```

11862 \cs_new:Npn \_fp_parse_large_round:NN #1#2
11863 {
11864     \if_int_compare:w 9 < 1 \token_to_str:N #2 \exp_stop_f:
11865         +
11866         \exp_after:wN \_fp_round_s:NNNw
11867         \exp_after:wN 0
11868         \exp_after:wN #1
11869         \exp_after:wN #2
11870         \_int_value:w \_int_eval:w
11871             \exp_after:wN \_fp_parse_large_round_aux:wNN
11872             \_int_value:w \_int_eval:w 1
11873                 \exp_after:wN \_fp_parse_round_loop:N
11874             \else: %^^A could be dot, or e, or other
11875                 \exp_after:wN \_fp_parse_large_round_test:NN
11876                 \exp_after:wN #1
11877                 \exp_after:wN #2
11878             \fi:
11879     }

```

```

11880 \cs_new:Npn \__fp_parse_large_round_test:NN #1#2
11881 {
11882     \if:w . \exp_not:N #2
11883         \exp_after:wN \__fp_parse_small_round:NN
11884         \exp_after:wN #1
11885         \exp:w
11886     \else:
11887         \__fp_parse_exponent:Nw #2
11888     \fi:
11889     \__fp_parse_expand:w
11890 }
11891 \cs_new:Npn \__fp_parse_large_round_aux:wNN #1 ; #2 #3
11892 {
11893     + #2
11894     \exp_after:wN \__fp_parse_round_after:wN
11895     \__int_value:w \__int_eval:w #1
11896     \if:w . \exp_not:N #3
11897         + 0 * \__int_eval:w 0
11898         \exp_after:wN \__fp_parse_round_loop:N
11899         \exp:w \exp_after:wN \__fp_parse_expand:w
11900     \else:
11901         \exp_after:wN ;
11902         \exp_after:wN 0
11903         \exp_after:wN #3
11904     \fi:
11905 }

```

(End definition for `__fp_parse_large_round>NN`, `__fp_parse_large_round_test>NN`, and `__fp_parse_large_round_aux>wNN`.)

24.4.5 Number: finding the exponent

Expansion is a little bit tricky here, in part because we accept input where multiplication is implicit.

```

\@@_parse:n { 3.2 erf(0.1) }
\@@_parse:n { 3.2 e\l_my_int }
\@@_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 __int_eval:w ... there if needed.

```
11906 \cs_new:Npn \_\_fp_parse_exponent:Nw #1 #2 \_\_fp_parse_expand:w
11907 {
11908     \exp_after:wN ;
11909     \_\_int_value:w #2 \_\_fp_parse_exponent:N #1
11910 }
```

(End definition for __fp_parse_exponent:Nw.)

__fp_parse_exponent:N
__fp_parse_exponent_aux:N

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, leave an exponent of 0. If there is an 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.

```
11911 \cs_new:Npn \_\_fp_parse_exponent:N #1
11912 {
11913     \if:w e \exp_not:N #1
11914         \exp_after:wN \_\_fp_parse_exponent_aux:N
11915         \exp:w
11916     \else:
11917         0 \_\_fp_parse_return_semicolon:w #1
11918     \fi:
11919         \_\_fp_parse_expand:w
11920 }
11921 \cs_new:Npn \_\_fp_parse_exponent_aux:N #1
11922 {
11923     \if_int_compare:w \if_catcode:w \scan_stop: \exp_not:N #1
11924         0 \else: '#1 \fi: > '9 \exp_stop_f:
11925         0 \exp_after:wN ; \exp_after:wN e
11926     \else:
11927         \exp_after:wN \_\_fp_parse_exponent_sign:N
11928     \fi:
11929     #1
11930 }
```

(End definition for __fp_parse_exponent:N and __fp_parse_exponent_aux:N.)

__fp_parse_exponent_sign:N

Read signs one by one (if there is any).

```
11931 \cs_new:Npn \_\_fp_parse_exponent_sign:N #1
11932 {
11933     \if:w + \if:w - \exp_not:N #1 + \fi: \token_to_str:N #1
11934         \exp_after:wN \_\_fp_parse_exponent_sign:N
11935         \exp:w \exp_after:wN \_\_fp_parse_expand:w
11936     \else:
11937         \exp_after:wN \_\_fp_parse_exponent_body:N
11938         \exp_after:wN #1
11939     \fi:
11940 }
```

(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).

```

11941 \cs_new:Npn \_\_fp_parse_exponent_body:N #1
11942 {
11943     \if_int_compare:w 9 < 1 \token_to_str:N #1 \exp_stop_f:
11944         \token_to_str:N #1
11945         \exp_after:wN \_\_fp_parse_exponent_digits:N
11946         \exp:w
11947     \else:
11948         \_\_fp_parse_exponent_keep:NTF #1
11949             { \_\_fp_parse_return_semicolon:w #1 }
11950             {
11951                 \exp_after:wN ;
11952                 \exp:w
11953             }
11954     \fi:
11955     \_\_fp_parse_expand:w
11956 }
```

(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.

```

11957 \cs_new:Npn \_\_fp_parse_exponent_digits:N #1
11958 {
11959     \if_int_compare:w 9 < 1 \token_to_str:N #1 \exp_stop_f:
11960         \token_to_str:N #1
11961         \exp_after:wN \_\_fp_parse_exponent_digits:N
11962         \exp:w
11963     \else:
11964         \_\_fp_parse_return_semicolon:w #1
11965     \fi:
11966     \_\_fp_parse_expand:w
11967 }
```

(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.

```

11968 \prg_new_conditional:Npnn \_\_fp_parse_exponent_keep:N #1 { TF }
11969 {
11970     \if_catcode:w \scan_stop: \exp_not:N #1
11971         \if_meaning:w \scan_stop: #1
11972             \if_int_compare:w
```

```

11973      \__str_if_eq_x:nn { \s_fp } { \exp_not:N #1 }
11974      = 0 \exp_stop_f:
11975      0
11976      \__msg_kernel_expandable_error:nnn
11977          { kernel } { fp-after-e } { floating-point~ }
11978          \prg_return_true:
11979      \else:
11980          0
11981          \__msg_kernel_expandable_error:nnn
11982              { kernel } { bad-variable } {#1}
11983          \prg_return_false:
11984      \fi:
11985      \else:
11986          \if_int_compare:w
11987              \__str_if_eq_x:nn { \__int_value:w #1 } { \tex_the:D #1 }
11988              = 0 \exp_stop_f:
11989              \__int_value:w #1
11990          \else:
11991              0
11992              \__msg_kernel_expandable_error:nnn
11993                  { kernel } { fp-after-e } { dimension~#1 }
11994          \fi:
11995          \prg_return_false:
11996      \fi:
11997      \else:
11998          0
11999          \__msg_kernel_expandable_error:nnn
12000              { kernel } { fp-missing } { exponent }
12001          \prg_return_true:
12002      \fi:
12003  }

```

(End definition for `__fp_parse_exponent_keep:NTF.`)

24.5 Constants, functions and prefix operators

24.5.1 Prefix operators

`__fp_parse_prefix_+:Nw` A unary + does nothing; we should continue looking for a number.

```

12004 \cs_new_eq:cN { __fp_parse_prefix_+:Nw } \__fp_parse_one:Nw

```

(End definition for `__fp_parse_prefix_+:Nw.`)

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.

This is redefined in `l3fp-extras`.

```

12005 \cs_new:Npn \__fp_parse_apply_unary:NNNwN #1#2#3#4@#5
12006  {
12007      #3 #2 #4 @
12008      \exp:w \exp_end_continue_f:w #5 #1
12009  }

```

(End definition for `__fp_parse_apply_unary:NNNwN.`)

```
\_\_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.

```
12010 \cs_set_protected:Npn \_\_fp\_tmp:w #1#2#3#4  
12011 {  
12012     \cs_new:cpn { \_\_fp\_parse\_prefix\_ #1 :Nw } ##1  
12013     {  
12014         \exp_after:wN \_\_fp\_parse\_apply\_unary:NNNwN  
12015         \exp_after:wN ##1  
12016         \exp_after:wN #4  
12017         \exp_after:wN #3  
12018         \exp:w  
12019         \if_int_compare:w #2 < ##1  
12020             \_\_fp\_parse\_operand:Nw ##1  
12021         \else:  
12022             \_\_fp\_parse\_operand:Nw #2  
12023         \fi:  
12024         \_\_fp\_parse\_expand:w  
12025     }  
12026 }  
12027 \_\_fp\_tmp:w - \c\_\_fp\_prec\_not\_int \_\_fp\_set\_sign\_o:w 2  
12028 \_\_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_strim_zeros:N to trim zeros after the decimal point, rather than the trim_zeros function for zeros before the decimal point.

```
12029 \cs_new:cpn { \_\_fp\_parse\_prefix\_.\!:Nw } #1  
12030 {  
12031     \exp_after:wN \_\_fp\_parse\_infix\_after\_operand:NwN  
12032     \exp_after:wN #1  
12033     \exp:w \exp_end_continue_f:w  
12034     \exp_after:wN \_\_fp\_sanitize:wN  
12035     \_\_int_value:w \_\_int_eval:w 0 \_\_fp\_parse\_strim\_zeros:N  
12036 }
```

(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. Commas will be allowed if the previous precedence is 16 (function with multiple arguments). In this case, find an operand using the precedence 1; otherwise the precedence 0. 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 the array it found as an operand, fetching the following infix operator.

```
12037 \cs_new:cpn { \_\_fp\_parse\_prefix\_(:Nw } #1  
12038 {  
12039     \exp_after:wN \_\_fp\_parse\_lparen\_after:NwN  
12040     \exp_after:wN #1  
12041     \exp:w
```

```

12042     \if_int_compare:w #1 = \c_fp_prec_funcii_int
12043         \_fp_parse_operand:Nw \c_fp_prec_comma_int
12044     \else:
12045         \_fp_parse_operand:Nw \c_fp_prec_paren_int
12046     \fi:
12047     \_fp_parse_expand:w
12048 }
12049 \cs_new:Npx \_fp_parse_lparen_after:NwN #1#2 @ #3
12050 {
12051     \exp_not:N \token_if_eq_meaning:NNTF #3
12052     \exp_not:c { \_fp_parse_infix_ }:N
12053 {
12054     \exp_not:N \_fp_exp_after_array_f:w #2 \s_fp_stop
12055     \exp_not:N \exp_after:wN
12056     \exp_not:N \_fp_parse_infix:NN
12057     \exp_not:N \exp_after:wN #1
12058     \exp_not:N \exp:w
12059     \exp_not:N \_fp_parse_expand:w
12060 }
12061 {
12062     \exp_not:N \_msg_kernel_expandable_error:nnn
12063     { kernel } { fp-missing } { ) }
12064     #2 @
12065     \exp_not:N \use_none:n #3
12066 }
12067 }

```

(End definition for `_fp_parse_prefix_(:Nw` and `_fp_parse_lparen_after:NwN`.)

`_fp_parse_prefix_):Nw`

The right parenthesis can appear as unary prefixes when arguments of a multi-argument function end with a comma, or when there is no argument, as in `max(1,2,)` or in `rand()`. In single-argument functions (precedence 0 rather than 1) forbid this.

```

12068 \cs_new:cpn { \_fp_parse_prefix_ ):Nw } #1
12069 {
12070     \if_int_compare:w #1 = \c_fp_prec_comma_int
12071     \else:
12072         \_msg_kernel_expandable_error:nnn
12073         { kernel } { fp-missing-number } { ) }
12074         \exp_after:wN \c_nan_fp \exp:w \exp_end_continue_f:w
12075     \fi:
12076     \_fp_parse_infix:NN #1 )
12077 }

```

(End definition for `_fp_parse_prefix_):Nw`.)

24.5.2 Constants

`_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`

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`.

```

12078 \cs_set_protected:Npn \_fp_tmp:w #1 #2
12079 {
12080     \cs_new:cpn { \_fp_parse_word_#1:N }
12081     { \exp_after:wN #2 \exp:w \exp_end_continue_f:w \_fp_parse_infix:NN }
12082 }

```

```

12083 \__fp_tmp:w { inf } \c_inf_fp
12084 \__fp_tmp:w { nan } \c_nan_fp
12085 \__fp_tmp:w { pi } \c_pi_fp
12086 \__fp_tmp:w { deg } \c_one_degree_fp
12087 \__fp_tmp:w { true } \c_one_fp
12088 \__fp_tmp:w { false } \c_zero_fp

```

(End definition for `__fp_parse_word_inf:N` and others.)

`__fp_parse_caseless_inf:N` Copies of `__fp_parse_word_...:N` commands, to allow arbitrary case as mandated by the standard.

```

12089 \cs_new_eq:NN \__fp_parse_caseless_inf:N \__fp_parse_word_inf:N
12090 \cs_new_eq:NN \__fp_parse_caseless_infinity:N \__fp_parse_word_inf:N
12091 \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`.)

`__fp_parse_word_pt: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.

```

12092 \cs_set_protected:Npn \__fp_tmp:w #1 #2
12093 {
12094   \cs_new:cpn { __fp_parse_word_#1:N }
12095   {
12096     \__fp_exp_after_f:nw { \__fp_parse_infix:NN }
12097     \s_fp \__fp_chk:w 10 #2 ;
12098   }
12099 }
12100 \__fp_tmp:w {pt} { {1} {1000} {0000} {0000} {0000} }
12101 \__fp_tmp:w {in} { {2} {7227} {0000} {0000} {0000} }
12102 \__fp_tmp:w {pc} { {2} {1200} {0000} {0000} {0000} }
12103 \__fp_tmp:w {cm} { {2} {2845} {2755} {9055} {1181} }
12104 \__fp_tmp:w {mm} { {1} {2845} {2755} {9055} {1181} }
12105 \__fp_tmp:w {dd} { {1} {1070} {0085} {6496} {0630} }
12106 \__fp_tmp:w {cc} { {2} {1284} {0102} {7795} {2756} }
12107 \__fp_tmp:w {nd} { {1} {1066} {9783} {4645} {6693} }
12108 \__fp_tmp:w {nc} { {2} {1280} {3740} {1574} {8031} }
12109 \__fp_tmp:w {bp} { {1} {1003} {7500} {0000} {0000} }
12110 \__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`.

```

12111 \tl_map_inline:nn { {em} {ex} }
12112 {
12113   \cs_new:cpn { __fp_parse_word_#1:N }
12114   {
12115     \exp_after:wN \__fp_from_dim_test:ww
12116     \exp_after:wN 0 \exp_after:wN ,
12117     \int_value:w \__dim_eval:w 1 #1 \exp_after:wN ;
12118     \exp:w \exp_end_continue_f:w \__fp_parse_infix:NN
12119   }
12120 }

```

(End definition for `__fp_parse_word_em:N` and `__fp_parse_word_ex:N`.)

24.5.3 Functions

```

\_\_fp\_parse\_unary\_function:NNN
\_\_fp\_parse\_function:NNN
12121 \cs_new:Npn \_\_fp\_parse\_unary\_function:NNN #1#2#3
12122 {
12123     \exp_after:wN \_\_fp\_parse\_apply\_unary:NNNwN
12124     \exp_after:wN #3
12125     \exp_after:wN #2
12126     \exp_after:wN #1
12127     \exp:w
12128     \_\_fp\_parse\_operand:Nw \c\_fp\_prec\_func\_int \_\_fp\_parse\_expand:w
12129 }
12130 \cs_new:Npn \_\_fp\_parse\_function:NNN #1#2#3
12131 {
12132     \exp_after:wN \_\_fp\_parse\_apply\_unary:NNNwN
12133     \exp_after:wN #3
12134     \exp_after:wN #2
12135     \exp_after:wN #1
12136     \exp:w
12137     \_\_fp\_parse\_operand:Nw \c\_fp\_prec\_funcii\_int \_\_fp\_parse\_expand:w
12138 }
```

(End definition for `__fp_parse_unary_function:NNN` and `__fp_parse_function:NNN`.)

24.6 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 will perform computations until reaching an operation with precedence `\c_fp_prec_end_int` or less, namely, the end of the expression. The marker `\s_fp_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:..`.

```

12139 \cs_new:Npn \_\_fp\_parse:n #1
12140 {
12141     \exp:w
12142     \exp_after:wN \_\_fp\_parse\_after:ww
12143     \exp:w
12144     \_\_fp\_parse\_operand:Nw \c\_fp\_prec\_end\_int
12145     \_\_fp\_parse\_expand:w #1
12146     \s\_fp\_mark \_\_fp\_parse\_infix\_end:N
12147     \s\_fp\_stop
12148 }
12149 \cs_new:Npn \_\_fp\_parse\_after:ww
12150     #1@ \_\_fp\_parse\_infix\_end:N \s\_fp\_stop
12151 { \exp_end: #1 }
```

(End definition for `__fp_parse:n` and `__fp_parse_after:ww`.)

```

\_\_fp\_parse_o:n
12152 \cs_new:Npn \_\_fp\_parse_o:n #1
12153 {
12154     \exp_after:wN \exp_after:wN
12155     \exp_after:wN \_\_fp\_exp\_after_o:w
```

```

12156     \__fp_parse:n {#1}
12157 }

```

(End definition for `__fp_parse_o:n`.)

`__fp_parse_operand:Nw`
`__fp_parse_continue:NwN`

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`.

```

12158 \cs_new:Npn \__fp_parse_operand:Nw #1
12159 {
12160     \exp_end_continue_f:w
12161     \exp_after:wN \__fp_parse_continue:NwN
12162     \exp_after:wN #1
12163     \exp:w \exp_end_continue_f:w
12164     \exp_after:wN \__fp_parse_one:Nw
12165     \exp_after:wN #1
12166     \exp:w
12167 }
12168 \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`

Receives `<precedence> <operand1> @ <operation> <operand2> @ <infix command>`. Builds the appropriate call to the `<operation> #3`.

This is redefined in l3fp-extras.

```

12169 \cs_new:Npn \__fp_parse_apply_binary:NwNwN #1 #2@ #3 #4@ #5
12170 {
12171     \exp_after:wN \__fp_parse_continue:NwN
12172     \exp_after:wN #1
12173     \exp:w \exp_end_continue_f:w \cs:w __fp_#3_o:ww \cs_end: #2 #4
12174     \exp:w \exp_end_continue_f:w #5 #1
12175 }

```

(End definition for `__fp_parse_apply_binary:NwNwN`.)

24.7 Infix operators

`__fp_parse_infix_after_operand:NwN`

```

12176 \cs_new:Npn \__fp_parse_infix_after_operand:NwN #1 #2;
12177 {
12178     \__fp_exp_after_f:nw { \__fp_parse_infix>NN #1 }
12179     #2;
12180 }
12181 \cs_new:Npn \__fp_parse_infix>NN #1 #2
12182 {
12183     \if_catcode:w \scan_stop: \exp_not:N #2
12184     \if_int_compare:w
12185         \__str_if_eq_x:nn { \s_fp_mark } { \exp_not:N #2 }
12186         = 0 \exp_stop_f:
12187         \exp_after:wN \exp_after:wN
12188         \exp_after:wN \__fp_parse_infix_mark:NNN
12189     \else:
12190         \exp_after:wN \exp_after:wN
12191         \exp_after:wN \__fp_parse_infix_juxtapose:N
12192     \fi:

```

```

12193 \else:
12194     \if_int_compare:w
12195         \__int_eval:w
12196             ( '#2 \if_int_compare:w '#2 > 'Z - 32 \fi: ) / 26
12197             = 3 \exp_stop_f:
12198             \exp_after:wN \exp_after:wN
12199             \exp_after:wN \__fp_parse_infix_juxtapose:N
12200 \else:
12201     \exp_after:wN \__fp_parse_infix_check:NNN
12202     \cs:w
12203         __fp_parse_infix_ \token_to_str:N #2 :N
12204         \exp_after:wN \exp_after:wN \exp_after:wN
12205         \cs_end:
12206         \fi:
12207     \fi:
12208     #1
12209     #2
12210 }
12211 \cs_new:Npx \__fp_parse_infix_check:NNN #1#2#3
12212 {
12213     \exp_not:N \if_meaning:w \scan_stop: #1
12214     \exp_not:N \__msg_kernel_expandable_error:nnn
12215         { kernel } { fp-missing } { * }
12216     \exp_not:N \exp_after:wN
12217     \exp_not:c { __fp_parse_infix_*:N }
12218     \exp_not:N \exp_after:wN #2
12219     \exp_not:N \exp_after:wN #3
12220     \exp_not:N \else:
12221         \exp_not:N \exp_after:wN #1
12222         \exp_not:N \exp_after:wN #2
12223         \exp_not:N \exp:w
12224         \exp_not:N \exp_after:wN
12225         \exp_not:N \__fp_parse_expand:w
12226         \exp_not:N \fi:
12227 }

```

(End definition for `__fp_parse_infix_after_operand:NwN.`)

24.7.1 Closing parentheses and commas

As an infix operator, `\s__fp_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.

```
12228 \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.

```
12229 \cs_new:Npn \__fp_parse_infix_end:N #1
12230     { @ \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.

```

12231 \cs_set_protected:Npn \_\_fp\_tmp:w #1
12232 {
12233   \cs_new:Npn #1 ##1
12234   {
12235     \if_int_compare:w ##1 < \c_fp_prec_paren_int
12236       \msg_kernel_expandable_error:nnn { kernel } { fp-extra } { }
12237       \exp_after:wN \_\_fp_parse_infix:NN
12238       \exp_after:wN ##1
12239       \exp:w \exp_after:wN \_\_fp_parse_expand:w
12240     \else:
12241       \exp_after:wN @
12242       \exp_after:wN \use_none:n
12243       \exp_after:wN #1
12244     \fi:
12245   }
12246 }
12247 \exp_args:Nc \_\_fp_tmp:w { \_\_fp_parse_infix_):N }

(End definition for \_\_fp_parse_infix_):N)

```

__fp_parse_infix_,:N __fp_,\o:ww is a complicated way of replacing any number of floating point arguments by nan.

```

12248 \cs_set_protected:Npn \_\_fp_tmp:w #1
12249 {
12250   \cs_new:Npn #1 ##1
12251   {
12252     \if_int_compare:w ##1 > \c_fp_prec_comma_int
12253       \exp_after:wN @
12254       \exp_after:wN \use_none:n
12255       \exp_after:wN #1
12256     \else:
12257       \if_int_compare:w ##1 < \c_fp_prec_comma_int
12258         \_\_fp_parse_infix_comma_error:w
12259       \fi:
12260       \exp_after:wN \_\_fp_parse_infix_comma:w
12261       \exp:w \_\_fp_parse_operand:Nw \c_fp_prec_comma_int
12262       \exp_after:wN \_\_fp_parse_expand:w
12263     \fi:
12264   }
12265 }
12266 \exp_args:Nc \_\_fp_tmp:w { \_\_fp_parse_infix_,:N }
12267 \cs_new:Npn \_\_fp_parse_infix_comma:w #1 @
12268   { #1 @ \use_none:n }
12269 \cs_new:Npn \_\_fp_parse_infix_comma_error:w #1 \exp:w
12270   {
12271     \fi:
12272     \msg_kernel_expandable_error:nn { kernel } { fp-extra-comma }
12273     \exp_after:wN @
12274     \exp_after:wN \_\_fp_parse_apply_binary:NwNwN
12275     \exp_after:wN ,
12276     \exp:w
12277   }

```

```

12278 \cs_set_protected:Npn \__fp_tmp:w #1
12279   {
12280     \cs_new:Npn #1 ##1
12281   {
12282     \if_meaning:w \s_fp ##1
12283       \exp_after:wN \__fp_use_i_until_s:nw
12284       \exp_after:wN #1
12285     \fi:
12286     \exp_after:wN \c_nan_fp
12287     ##1
12288   }
12289 }
12290 \exp_args:Nc \__fp_tmp:w { __fp_,_o:ww }

(End definition for \__fp_parse_infix_,:N and others.)

```

24.7.2 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.

```

12291 \cs_set_protected:Npn \__fp_tmp:w #1#2#3#4
12292   {
12293     \cs_new:Npn #1 ##1
12294   {
12295     \if_int_compare:w ##1 < #3
12296       \exp_after:wN @
12297       \exp_after:wN \__fp_parse_apply_binary:NwNwN
12298       \exp_after:wN #2
12299       \exp:w
12300       \__fp_parse_operand:Nw #4
12301       \exp_after:wN \__fp_parse_expand:w
12302     \else:
12303       \exp_after:wN @
12304       \exp_after:wN \use_none:n
12305       \exp_after:wN #1
12306     \fi:
12307   }
12308 }
12309 \exp_args:Nc \__fp_tmp:w { __fp_parse_infix^:N } ^
12310   \c_fp_prec_hatii_int \c_fp_prec_hat_int
12311 \exp_args:Nc \__fp_tmp:w { __fp_parse_infix/_:N } /
12312   \c_fp_prec_times_int \c_fp_prec_times_int
12313 \exp_args:Nc \__fp_tmp:w { __fp_parse_infix_mul:N } *
12314   \c_fp_prec_times_int \c_fp_prec_times_int
12315 \exp_args:Nc \__fp_tmp:w { __fp_parse_infix_=:N } -
12316   \c_fp_prec_plus_int \c_fp_prec_plus_int
12317 \exp_args:Nc \__fp_tmp:w { __fp_parse_infix_+:N } +
12318   \c_fp_prec_plus_int \c_fp_prec_plus_int
12319 \exp_args:Nc \__fp_tmp:w { __fp_parse_infix_and:N } &
12320   \c_fp_prec_and_int \c_fp_prec_and_int
12321 \exp_args:Nc \__fp_tmp:w { __fp_parse_infix_or:N } |
12322   \c_fp_prec_or_int \c_fp_prec_or_int

```

(End definition for `_fp_parse_infix_+:N` and others.)

24.7.3 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_juxtapose:N`.

```
12323 \cs_new:cpn { \_fp_parse_infix_(:N } #1
12324   { \_fp_parse_infix_juxtapose:N #1 ( }
```

(End definition for `_fp_parse_infix_(:N`.)

`_fp_parse_infix_juxtapose:N`
`_fp_parse_apply_juxtapose:NwwN`

Juxtaposition follows the same scheme as other binary operations, but calls `_fp_parse_apply_juxtapose:NwwN` rather than directly calling `_fp_parse_apply_binary:NwNwN`. This lets us catch errors such as $\dots(1,2,3)\text{pt}$ where one operand of the juxtaposition is not a single number: both #3 and #5 of the apply auxiliary must be empty.

```
12325 \cs_new:Npn \_fp_parse_infix_juxtapose:N #1
12326   {
12327     \if_int_compare:w #1 < \c_fp_prec_times_int
12328       \exp_after:wN @
12329       \exp_after:wN \_fp_parse_apply_juxtapose:NwwN
12330       \exp:w
12331       \_fp_parse_operand:Nw \c_fp_prec_times_int
12332       \exp_after:wN \_fp_parse_expand:w
12333     \else:
12334       \exp_after:wN @
12335       \exp_after:wN \use_none:n
12336       \exp_after:wN \_fp_parse_infix_juxtapose:N
12337     \fi:
12338   }
12339 \cs_new:Npn \_fp_parse_apply_juxtapose:NwwN #1 #2;#30 #4;#50
12340   {
12341     \if_catcode:w ^ \tl_to_str:n { #3 #5 } ^
12342     \else:
12343       \_fp_error:nffn { fp-invalid-ii }
12344       { \_fp_array_to_clist:n { #2; #3 } }
12345       { \_fp_array_to_clist:n { #4; #5 } }
12346       { }
12347     \fi:
12348     \_fp_parse_apply_binary:NwNwN #1 #2;@ * #4;@
12349   }
```

(End definition for `_fp_parse_infix_juxtapose:N` and `_fp_parse_apply_juxtapose:NwwN`.)

24.7.4 Multi-character cases

`_fp_parse_infix_*:N`

```
12350 \cs_set_protected:Npn \_fp_tmp:w #1
12351   {
12352     \cs_new:cpn { \_fp_parse_infix_*:N } ##1##2
12353     {
12354       \if:w * \exp_not:N ##2
```

```

12355     \exp_after:wN #1
12356     \exp_after:wN ##1
12357 \else:
12358     \exp_after:wN \_fp_parse_infix_mul:N
12359     \exp_after:wN ##1
12360     \exp_after:wN ##2
12361 \fi:
12362 }
12363 }
12364 \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
12365 \cs_set_protected:Npn \_fp_tmp:w #1#2#3
12366 {
12367     \cs_new:Npn #1 ##1##2
12368     {
12369         \if:w #2 \exp_not:N ##2
12370         \exp_after:wN #
12371         \exp_after:wN ##1
12372         \exp:w \exp_after:wN \_fp_parse_expand:w
12373     \else:
12374         \exp_after:wN #3
12375         \exp_after:wN ##1
12376         \exp_after:wN ##2
12377     \fi:
12378 }
12379 }
12380 \exp_args:Nc \_fp_tmp:w { _fp_parse_infix_|:N } | \_fp_parse_infix_or:N
12381 \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.)

```

24.7.5 Ternary operator

```

\_fp_parse_infix_?:N
\_fp_parse_infix_::N
12382 \cs_set_protected:Npn \_fp_tmp:w #1#2#3#4
12383 {
12384     \cs_new:Npn #1 ##1
12385     {
12386         \if_int_compare:w ##1 < \c_fp_prec_quest_int
12387             #4
12388             \exp_after:wN @
12389             \exp_after:wN #2
12390             \exp:w
12391             \_fp_parse_operand:Nw #3
12392             \exp_after:wN \_fp_parse_expand:w
12393     \else:
12394         \exp_after:wN @
12395         \exp_after:wN \use_none:n
12396         \exp_after:wN #1
12397     \fi:
12398 }

```

```

12399      }
12400  \exp_args:Nc \_fp_tmp:w { __fp_parse_infix_?:N }
12401    \_fp_ternary:NwwN \c_fp_prec_quest_int { }
12402  \exp_args:Nc \_fp_tmp:w { __fp_parse_infix_::N }
12403    \_fp_ternary_auxii:NwwN \c_fp_prec_colon_int
12404    {
12405      \_msg_kernel_expandable_error:nnnn
12406        { kernel } { fp-missing } { ? } { ~for~?: }
12407    }

```

(End definition for `_fp_parse_infix_?:N` and `_fp_parse_infix_::N`.)

24.7.6 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:NNNNNNN
  \_fp_parse_compare_auxii:NNNNN
  \_fp_parse_compare_end:NNNNw
  \__fp_compare:wNNNNw

\__fp_parse_compare:NNNNNNN
  \cs_new:cpn { __fp_parse_infix_<:N } #1
  { \_fp_parse_compare:NNNNNNN #1 1 0 0 0 0 < }
\__fp_parse_compare:NNNNNNN
  \cs_new:cpn { __fp_parse_infix_=:N } #1
  { \_fp_parse_compare:NNNNNNN #1 1 0 0 0 0 = }
\__fp_parse_compare:NNNNNNN
  \cs_new:cpn { __fp_parse_infix_>:N } #1
  { \_fp_parse_compare:NNNNNNN #1 1 0 0 0 0 > }
\__fp_parse_compare:NNNNNNN
  \cs_new:cpn { __fp_parse_infix_!:N } #1
  {
    \exp_after:wN \_fp_parse_compare:NNNNNNN
    \exp_after:wN #1
    \exp_after:wN 0
    \exp_after:wN 1
    \exp_after:wN 1
    \exp_after:wN 1
    \exp_after:wN 1
  }
\__fp_parse_excl_error:
  {
    \_msg_kernel_expandable_error:nnnn
      { kernel } { fp-missing } { = } { ~after~!. }
  }
\__fp_parse_compare:NNNNNNN #1
  {
    \if_int_compare:w #1 < \c_fp_prec_comp_int
      \exp_after:wN \_fp_parse_compare_auxi:NNNNNNN
      \exp_after:wN \_fp_parse_excl_error:
    \else:
      \exp_after:wN @
      \exp_after:wN \use_none:n
      \exp_after:wN \_fp_parse_compare:NNNNNNN
    \fi:
  }
\__fp_parse_compare_auxi:NNNNNNN #1#2#3#4#5#6#7
  {
    \if_case:w
      \int_eval:w \exp_after:wN ` \token_to_str:N #7 - '< \int_eval_end:
      \_fp_parse_compare_auxii:NNNNN #2#2#4#5#6
    \or: \_fp_parse_compare_auxii:NNNNN #2#3#2#5#6
    \or: \_fp_parse_compare_auxii:NNNNN #2#3#4#2#6
  }

```

```

12447     \or: \_fp_parse_compare_auxii:NNNNN #2#3#4#5#2
12448     \else: #1 \_fp_parse_compare_end:NNNNw #3#4#5#6#7
12449     \fi:
12450   }
12451 \cs_new:Npn \_fp_parse_compare_auxii:NNNNN #1#2#3#4#5
12452   {
12453     \exp_after:wN \_fp_parse_compare_auxi:NNNNNNN
12454     \exp_after:wN \prg_do_nothing:
12455     \exp_after:wN #1
12456     \exp_after:wN #2
12457     \exp_after:wN #3
12458     \exp_after:wN #4
12459     \exp_after:wN #5
12460     \exp:w \exp_after:wN \_fp_parse_expand:w
12461   }
12462 \cs_new:Npn \_fp_parse_compare_end:NNNNw #1#2#3#4#5 \fi:
12463   {
12464     \fi:
12465     \exp_after:wN @
12466     \exp_after:wN \_fp_parse_apply_compare:NwNNNNNwN
12467     \exp_after:wN \c_one_fp
12468     \exp_after:wN #1
12469     \exp_after:wN #2
12470     \exp_after:wN #3
12471     \exp_after:wN #4
12472     \exp:w
12473     \_fp_parse_operand:Nw \c_fp_prec_comp_int \_fp_parse_expand:w #5
12474   }
12475 \cs_new:Npn \_fp_parse_apply_compare:NwNNNNNwN
12476   #1 #2@ #3 #4#5#6#7 #8@ #9
12477   {
12478     \if_int_odd:w
12479       \if_meaning:w \c_zero_fp #3
12480         0
12481       \else:
12482         \if_case:w \_fp_compare_back:ww #8 #2 \exp_stop_f:
12483           #5 \or: #6 \else: #7 \else: #4
12484         \fi:
12485       \fi:
12486       \exp_stop_f:
12487     \exp_after:wN \_fp_parse_apply_compare_aux:NNwN
12488     \exp_after:wN \c_one_fp
12489   \else:
12490     \exp_after:wN \_fp_parse_apply_compare_aux:NNwN
12491     \exp_after:wN \c_zero_fp
12492   \fi:
12493   #1 #8 #9
12494 }
12495 \cs_new:Npn \_fp_parse_apply_compare_aux:NNwN #1 #2 #3; #4
12496   {
12497     \if_meaning:w \_fp_parse_compare>NNNNNNN #4
12498       \exp_after:wN \_fp_parse_continue_compare>NNwNN
12499       \exp_after:wN #1
12500       \exp_after:wN #2

```

```

12501   \exp:w \exp_end_continue_f:w
12502   \_fp_exp_after_o:w #3;
12503   \exp:w \exp_end_continue_f:w
12504 \else:
12505   \exp_after:wN \_fp_parse_continue:NwN
12506   \exp_after:wN #2
12507   \exp:w \exp_end_continue_f:w
12508   \exp_after:wN #1
12509   \exp:w \exp_end_continue_f:w
12510 \fi:
12511 #4 #2
12512 }
12513 \cs_new:Npn \_fp_parse_continue_compare>NNwNN #1#2 #3@ #4#5
12514 { #4 #2 #3@ #1 }

```

(End definition for `_fp_parse_infix_<:N` and others.)

24.8 Candidate: defining new l3fp functions

`\fp_function:Nw` Parse the argument of the function #1 using `_fp_parse_operand:Nw` with a precedence of 16, and pass the function and argument to `_fp_function_apply:nw`.

```

12515 \cs_new:Npn \fp_function:Nw #1
12516 {
12517   \exp_after:wN \_fp_function_apply:nw
12518   \exp_after:wN #1
12519   \exp:w
12520     \_fp_parse_operand:Nw \c_fp_prec_funcii_int \_fp_parse_expand:w
12521 }

```

(End definition for `\fp_function:Nw`.)

`\fp_new_function:Npn`
`_fp_new_function:NNnnn`
`_fp_new_function:Ncfnn`
`_fp_function_args:Nwn`

Save the code provided by the user in the control sequence `_fp_user_#1`. Define #1 to call `_fp_function_apply:nw` after parsing one operand using `_fp_parse_operand:Nw` with precedence 16. The auxiliary `_fp_function_args:Nwn` receives the user function and the number of arguments (half of the number of tokens in the parameter text #2), followed by the operand (as a token list of floating points). It checks the number of arguments, and applies the user function to the arguments (without the outer brace group).

```

12522 \cs_new_protected:Npn \fp_new_function:Npn #1#2#
12523 {
12524   \_fp_new_function:Ncfnn #1
12525   { _fp_user_ \cs_to_str:N #1 }
12526   { \int_eval:n { \tl_count:n {#2} / 2 } }
12527   {#2}
12528 }
12529 \cs_new_protected:Npn \_fp_new_function:NNnnn #1#2#3#4#5
12530 {
12531   \cs_new:Npn #1
12532   {
12533     \exp_after:wN \_fp_function_apply:nw \exp_after:wN
12534     {
12535       \exp_after:wN \_fp_function_args:Nwn
12536       \exp_after:wN #2
12537       \_int_value:w #3 \exp_after:wN ; \exp_after:wN

```

```

12538         }
12539     \exp:w
12540         \__fp_parse_operand:Nw \c_fp_prec_funcii_int \__fp_parse_expand:w
12541     }
12542     \cs_new:Npn #2 #4 {#5}
12543   }
12544 \cs_generate_variant:Nn \__fp_new_function>NNnnn { Ncf }
12545 \cs_new:Npn \__fp_function_args:Nwn #1#2; #3
12546   {
12547     \int_compare:nNnTF { \tl_count:n {#3} } = {#2}
12548     { #1 #3 }
12549   {
12550     \__msg_kernel_expandable_error:nnnn
12551       { kernel } { fp-num-args } { #1() } {#2} {#2}
12552     \c_nan_fp
12553   }
12554 }

```

(End definition for `\fp_new_function:Npn`, `__fp_new_function>NNnnn`, and `__fp_function_args:Nwn`.)

The auxiliary `__fp_function_apply:nw` is called after parsing an operand, so it receives some code #1, then the operand ending with @, then a function such as `__fp_parse_infix_+:N` (but not always of this form, see comparisons for instance). Package the operand (an array) into a token list with floating point items: this is the role of `__fp_function_store:wwNwnn` and `__fp_function_store_end:wnnn`. Then apply `__fp_parse:n` to the code #1 followed by a brace group with this token list. This results in a floating point result, which will correctly be parsed as the next operand of whatever was looking for one. The trailing `\s_fp_mark` is used as a special infix operator to indicate that the next token has already gone through `__fp_parse_infix:NN`.

```

12555 \cs_new:Npn \__fp_function_apply:nw #1#2 @
12556   {
12557     \__fp_parse:n
12558   {
12559     \__fp_function_store:wwNwnn #2
12560       \s_fp_mark \__fp_function_store:wwNwnn ;
12561       \s_fp_mark \__fp_function_store_end:wnnn
12562       \s_fp_stop { } { } {#1}
12563   }
12564   \s_fp_mark
12565 }
12566 \cs_new:Npn \__fp_function_store:wwNwnn
12567   #1; #2 \s_fp_mark #3#4 \s_fp_stop #5#6
12568   { #3 #2 \s_fp_mark #3#4 \s_fp_stop { #5 #6 } { { #1; } } }
12569 \cs_new:Npn \__fp_function_store_end:wnnn
12570   #1 \s_fp_stop #2#3#4
12571   { #4 {#2} }

```

(End definition for `__fp_function_apply:nw`, `__fp_function_store:wwNwnn`, and `__fp_function_store_end:wnnn`.)

24.9 Messages

```

12572 \__msg_kernel_new:nnn { kernel } { fp-deprecated }
12573   { '#1'~deprecated;~use~'#2' }

```

```

12574 \_\_msg_kernel_new:nnn { kernel } { unknown-fp-word }
12575   { Unknown~fp~word~#1. }
12576 \_\_msg_kernel_new:nnn { kernel } { fp-missing }
12577   { Missing~#1~inserted #2. }
12578 \_\_msg_kernel_new:nnn { kernel } { fp-extra }
12579   { Extra~#1~ignored. }
12580 \_\_msg_kernel_new:nnn { kernel } { fp-early-end }
12581   { Premature~end~in~fp~expression. }
12582 \_\_msg_kernel_new:nnn { kernel } { fp-after-e }
12583   { Cannot~use~#1 after~'e'. }
12584 \_\_msg_kernel_new:nnn { kernel } { fp-missing-number }
12585   { Missing~number~before~'#1'. }
12586 \_\_msg_kernel_new:nnn { kernel } { fp-unknown-symbol }
12587   { Unknown~symbol~#1~ignored. }
12588 \_\_msg_kernel_new:nnn { kernel } { fp-extra-comma }
12589   { Unexpected~comma:~extra~arguments~ignored. }
12590 \_\_msg_kernel_new:nnn { kernel } { fp-num-args }
12591   { #1~expects~between~#2~and~#3~arguments. }
12592 (*package)
12593 \cs_if_exist:cT { @unexpandable@protect }
12594   {
12595     \_\_msg_kernel_new:nnn { kernel } { fp-robust-cmd }
12596     { Robust~command~#1 invalid~in~fp~expression! }
12597   }
12598 
```

25 I3fp-logic Implementation

```

12600 (*initex | package)
12601 @C=fp

```

`__fp_parse_word_max:N` Those functions may receive a variable number of arguments.

```

12602 \cs_new:Npn \_\_fp_parse_word_max:N
12603   { \_\_fp_parse_function:NNN \_\_fp_minmax_o:Nw 2 }
12604 \cs_new:Npn \_\_fp_parse_word_min:N
12605   { \_\_fp_parse_function:NNN \_\_fp_minmax_o:Nw 0 }

```

(End definition for `__fp_parse_word_max:N` and `__fp_parse_word_min:N`.)

25.1 Syntax of internal functions

- `__fp_compare_npos:nwnw {<expo1>} <body1> ; {<expo2>} <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.

25.2 Existence test

```
\fp_if_exist_p:N Copies of the cs functions defined in |3basics.
\fp_if_exist_p:c 12606 \prg_new_eq_conditional:NNN \fp_if_exist:N \cs_if_exist:N { TF , T , F , p }
\fp_if_exist:NTF 12607 \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 178.)

25.3 Comparison

`\fp_compare_p:n` Within floating point expressions, comparison operators are treated as operations, so we evaluate #1, then compare with 0.

```
\_fp_compare_return:w 12608 \prg_new_conditional:Npnn \fp_compare:n #1 { p , T , F , TF }
                      {
12609          \exp_after:wN \_fp_compare_return:w
12610          \exp:w \exp_end_continue_f:w \_fp_parse:n {#1}
12611          }
12612          \cs_new:Npn \_fp_compare_return:w \s_fp \_fp_chk:w #1#2;
12613          {
12614          \if_meaning:w 0 #1
12615          \prg_return_false:
12616          \else:
12617          \prg_return_true:
12618          \fi:
12619          }
```

(End definition for `\fp_compare:nTF` and `_fp_compare_return:w`. These functions are documented on page 179.)

`\fp_compare_p:nNn` Evaluate #1 and #3, using an auxiliary to expand both, and feed the two floating point numbers swapped to `_fp_compare_back:ww`, defined below. Compare the result with '#2-'=, which is -1 for <, 0 for =, 1 for > and 2 for ?.

```
\_fp_compare_aux:wn 12621 \prg_new_conditional:Npnn \fp_compare:nNn #1#2#3 { p , T , F , TF }
                      {
12622          \if_int_compare:w
12623          \exp_after:wN \_fp_compare_aux:wn
12624          \exp:w \exp_end_continue_f:w \_fp_parse:n {#1} {#3}
12625          = \_int_eval:w '#2 - '=' \_int_eval_end:
12626          \prg_return_true:
12627          \else:
12628          \prg_return_false:
12629          \fi:
12630          }
12631          \cs_new:Npn \_fp_compare_aux:wn #1; #2
12632          {
12633          \exp_after:wN \_fp_compare_back:ww
12634          \exp:w \exp_end_continue_f:w \_fp_parse:n {#2} #1;
12635          }
```

(End definition for `\fp_compare:nNnTF` and `_fp_compare_aux:wn`. These functions are documented on page 179.)

```
\_\_fp\_compare\_back:ww
\_\_fp\_compare\_nan:w
```

`__fp_compare_back:ww` $\langle y \rangle ; \langle x \rangle ;$
 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.

```
12637 \cs_new:Npn \_\_fp\_compare\_back:ww
12638   \s_fp \_\_fp_chk:w #1 #2 #3;
12639   \s_fp \_\_fp_chk:w #4 #5 #6;
12640   {
12641     \_\_int_value:w
12642       \if_meaning:w 3 #1 \exp_after:wN \_\_fp\_compare\_nan:w \fi:
12643       \if_meaning:w 3 #4 \exp_after:wN \_\_fp\_compare\_nan:w \fi:
12644       \if_meaning:w 2 #5 - \fi:
12645       \if_meaning:w #2 #5
12646         \if_meaning:w #1 #4
12647           \if_meaning:w 1 #
12648             \_\_fp\_compare\_npos:nwnw #6; #3;
12649           \else:
12650             0
12651           \fi:
12652           \else:
12653             \if_int_compare:w #4 < #1 - \fi: 1
12654           \fi:
12655           \else:
12656             \if_int_compare:w #1#4 = 0 \exp_stop_f:
12657               0
12658             \else:
12659               1
12660             \fi:
12661           \fi:
12662           \exp_stop_f:
12663     }
12664 \cs_new:Npn \_\_fp\_compare\_nan:w #1 \fi: \exp_stop_f: { 2 \exp_stop_f: }
```

(End definition for `__fp_compare_back:ww` and `__fp_compare_nan:w`.)

```
\_\_fp\_compare\_npos:nwnw
\_\_fp\_compare\_significand:nnnnnnnn
```

`__fp_compare_npos:nwnw` $\{ \langle \text{expo}_1 \rangle \} \langle \text{body}_1 \rangle ; \{ \langle \text{expo}_2 \rangle \} \langle \text{body}_2 \rangle ;$
 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.

```
12665 \cs_new:Npn \_\_fp\_compare\_npos:nwnw #1#2; #3#4;
12666   {
12667     \if_int_compare:w #1 = #3 \exp_stop_f:
12668       \_\_fp\_compare\_significand:nnnnnnnn #2 #4
12669     \else:
12670       \if_int_compare:w #1 < #3 - \fi: 1
12671     \fi:
```

```

12672     }
12673 \cs_new:Npn \__fp_compare_significand:nnnnnnnn #1#2#3#4#5#6#7#8
12674 {
12675     \if_int_compare:w #1#2 = #5#6 \exp_stop_f:
12676         \if_int_compare:w #3#4 = #7#8 \exp_stop_f:
12677             0
12678         \else:
12679             \if_int_compare:w #3#4 < #7#8 - \fi: 1
12680         \fi:
12681     \else:
12682         \if_int_compare:w #1#2 < #5#6 - \fi: 1
12683     \fi:
12684 }
```

(End definition for `__fp_compare_npos:nwnw` and `__fp_compare_significand:nnnnnnnn`.)

25.4 Floating point expression loops

`\fp_do_until:nn`
`\fp_do_while: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.

```

12685 \cs_new:Npn \fp_do_until:nn #1#2
12686 {
12687     #2
12688     \fp_compare:nF {#1}
12689         { \fp_do_until:nn {#1} {#2} }
12690     }
12691 \cs_new:Npn \fp_do_while:nn #1#2
12692 {
12693     #2
12694     \fp_compare:nT {#1}
12695         { \fp_do_while:nn {#1} {#2} }
12696     }
12697 \cs_new:Npn \fp_until_do:nn #1#2
12698 {
12699     \fp_compare:nF {#1}
12700     {
12701         #2
12702         \fp_until_do:nn {#1} {#2}
12703     }
12704 }
12705 \cs_new:Npn \fp_while_do:nn #1#2
12706 {
12707     \fp_compare:nT {#1}
12708     {
12709         #2
12710         \fp_while_do:nn {#1} {#2}
12711     }
12712 }
```

(End definition for `\fp_do_until:nn` and others. These functions are documented on page 180.)

`\fp_do_until:nNnn`

As above but not using the `nNn` syntax.

`\fp_do_while:nNnn`
`\fp_until_do:nNnn`
`\fp_while_do:nNnn`

```

12713 \cs_new:Npn \fp_do_until:nNnn #1#2#3#4
12714 {
```

```

12715      #4
12716      \fp_compare:nNnF {#1} #2 {#3}
12717          { \fp_do_until:nNnn {#1} #2 {#3} {#4} }
12718      }
12719 \cs_new:Npn \fp_do_while:nNnn #1#2#3#4
12720      {
12721          #4
12722          \fp_compare:nNnT {#1} #2 {#3}
12723              { \fp_do_while:nNnn {#1} #2 {#3} {#4} }
12724      }
12725 \cs_new:Npn \fp_until_do:nNnn #1#2#3#4
12726      {
12727          \fp_compare:nNnF {#1} #2 {#3}
12728              {
12729                  #4
12730                  \fp_until_do:nNnn {#1} #2 {#3} {#4}
12731              }
12732      }
12733 \cs_new:Npn \fp_while_do:nNnn #1#2#3#4
12734      {
12735          \fp_compare:nNnT {#1} #2 {#3}
12736              {
12737                  #4
12738                  \fp_while_do:nNnn {#1} #2 {#3} {#4}
12739              }
12740      }

```

(End definition for `\fp_do_until:nNnn` and others. These functions are documented on page 180.)

\fp_step_function:nnnN
\fp_step_function:nnnc
 $__fp_step:wwwN$
 $__fp_step:NnnnnN$
 $__fp_step:NfnnnnN$

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 will increase.

```

12741 \cs_new:Npn \fp_step_function:nnnN #1#2#3
12742      {
12743          \exp_after:wN \_\_fp_step:wwwN
12744              \exp:w \exp_end_continue_f:w \_\_fp_parse_o:n {#1}
12745              \exp:w \exp_end_continue_f:w \_\_fp_parse_o:n {#2}
12746              \exp:w \exp_end_continue_f:w \_\_fp_parse:n {#3}
12747      }
12748 \cs_generate_variant:Nn \fp_step_function:nnnN { nnnc }
12749 %           \end{macrocode}
12750 % Only \enquote{normal} floating points (not $\pm 0$,
12751 % $\pm\texttt{inf}$, $\texttt{nan}$) can be used as step; if positive,
12752 % call \cs{\_\_fp_step:NnnnnN} with argument |>| otherwise |<|. This
12753 % function has one more argument than its integer counterpart, namely
12754 % the previous value, to catch the case where the loop has made no
12755 % progress. Conversion to decimal is done just before calling the
12756 % user's function.
12757 %           \begin{macrocode}
12758 \cs_new:Npn \_\_fp_step:wwwN #1 ; \s_\_fp \_\_fp_chk:w #2#3#4 ; #5; #6
12759      {
12760          \token_if_eq_meaning:NNTF #2 1
12761              {

```

```

12762     \token_if_eq_meaning:NNTF #3 0
12763         { \__fp_step:NnnmN > }
12764         { \__fp_step:NnnnN < }
12765     }
12766 {
12767     \token_if_eq_meaning:NNTF #2 0
12768         { \__msg_kernel_expandable_error:nnn { kernel } { zero-step } {#6} }
12769         {
12770             \__fp_error:nfnf { fp-bad-step } { }
12771             { \fp_to_t1:n { \s_fp \__fp_chk:w #2#3#4 ; } } {#6}
12772         }
12773         \use_none:nnnn
12774     }
12775     { #1 ; } { \c_nan_fp } { \s_fp \__fp_chk:w #2#3#4 ; } { #5 ; } #6
12776 }
12777 \cs_new:Npn \__fp_step:NnnmN #1#2#3#4#5#6
12778 {
12779     \fp_compare:nNnTF {#2} = {#3}
12780     {
12781         \__fp_error:nffn { fp-tiny-step }
12782         { \fp_to_t1:n {#3} } { \fp_to_t1:n {#4} } {#6}
12783     }
12784     {
12785         \fp_compare:nNnF {#2} #1 {#5}
12786         {
12787             \exp_args:Nf #6 { \fp_to_decimal_dispatch:w #2 }
12788             \__fp_step:NfnnnN
12789             #1 { \__fp_parse:n { #2 + #4 } } {#2} {#4} {#5} #6
12790         }
12791     }
12792 }
12793 \cs_generate_variant:Nn \__fp_step:NnnnN { Nf }

(End definition for \fp_step_function:nnN, \fp_step:wwwN, and \fp_step:NnnnnN. These functions are documented on page 181.)
```

As for \int_step_inline:nnnn, create a global function and apply it, following up with a break point.

```

12794 \cs_new_protected:Npn \fp_step_inline:nnnn
12795 {
12796     \int_gincr:N \g_prg_map_int
12797     \exp_args:NNc \__fp_step>NNnnnn
12798     \cs_gset_protected:Npn
12799     { __prg_map_ \int_use:N \g_prg_map_int :w }
12800 }
12801 \cs_new_protected:Npn \fp_step_variable:nnnN #1#2#3#4#5
12802 {
12803     \int_gincr:N \g_prg_map_int
12804     \exp_args:NNc \__fp_step>NNnnnn
12805     \cs_gset_protected:Npx
12806     { __prg_map_ \int_use:N \g_prg_map_int :w }
12807     {#1} {#2} {#3}
12808     {
12809         \tl_set:Nn \exp_not:N #4 {##1}
```

```

12810           \exp_not:n {#5}
12811       }
12812   }
12813 \cs_new_protected:Npn \__fp_step:NNnnnn #1#2#3#4#5#6
12814   {
12815     #1 #2 ##1 {#6}
12816     \fp_step_function:nnnN {#3} {#4} {#5} #2
12817     \__prg_break_point:Nn \scan_stop: { \int_gdecr:N \g__prg_map_int }
12818   }

```

(End definition for `\fp_step_inline:nnnn`, `\fp_step_variable:nnnNn`, and `__fp_step:NNnnnn`. These functions are documented on page 181.)

```

12819 \__msg_kernel_new:nnn { kernel } { fp-bad-step }
12820   { Invalid~step~size~#2~in~step~function~#3. }
12821 \__msg_kernel_new:nnn { kernel } { fp-tiny-step }
12822   { Tiny~step~size~(#1+#2=#1)~in~step~function~#3. }

```

25.5 Extrema

`__fp_minmax_o:Nw`

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, currently impossible. Since no number is smaller (larger) than that, it will never alter the maximum (minimum). The weird fp-like trailing marker breaks the loop correctly: see the precise definition of `__fp_minmax_loop:Nww`.

```

12823 \cs_new:Npn \__fp_minmax_o:Nw #1#2 @
12824   {
12825     \if_meaning:w 0 #1
12826       \exp_after:wN \__fp_minmax_loop:Nww \exp_after:wN +
12827     \else:
12828       \exp_after:wN \__fp_minmax_loop:Nww \exp_after:wN -
12829     \fi:
12830     #2
12831     \s__fp \__fp_chk:w 2 #1 \s__fp_exact ;
12832     \s__fp \__fp_chk:w { 3 \__fp_minmax_break_o:w } ;
12833   }

```

(End definition for `__fp_minmax_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.

```

12834 \cs_new:Npn \__fp_minmax_loop:Nww
12835   #1 \s__fp \__fp_chk:w #2#3; \s__fp \__fp_chk:w #4#5;
12836   {
12837     \if_meaning:w 3 #4
12838     \if_meaning:w 3 #2
12839       \__fp_minmax_auxi:ww

```

```

12840     \else:
12841         \_\_fp\_minmax\_auxii:ww
12842     \fi:
12843 \else:
12844     \if_int_compare:w
12845         \_\_fp\_compare\_back:ww
12846             \s\_\_fp \_\_fp\_chk:w #4#5;
12847             \s\_\_fp \_\_fp\_chk:w #2#3;
12848             = #1 1 \exp_stop_f:
12849             \_\_fp\_minmax\_auxii:ww
12850     \else:
12851         \_\_fp\_minmax\_auxi:ww
12852     \fi:
12853 \fi:
12854 \_\_fp\_minmax\_loop:Nww #1
12855     \s\_\_fp \_\_fp\_chk:w #2#3;
12856     \s\_\_fp \_\_fp\_chk:w #4#5;
12857 }

```

(End definition for `__fp_minmax_loop:Nww`.)

`__fp_minmax_auxi:ww` Keep the first/second number, and remove the other.

```

12858 \cs_new:Npn \_\_fp\_minmax\_auxi:ww #1 \fi: \fi: #2 \s\_\_fp #3 ; \s\_\_fp #4;
12859     { \fi: \fi: #2 \s\_\_fp #3 ; }
12860 \cs_new:Npn \_\_fp\_minmax\_auxii:ww #1 \fi: \fi: #2 \s\_\_fp #3 ;
12861     { \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.

```

12862 \cs_new:Npn \_\_fp\_minmax\_break_o:w #1 \fi: \fi: #2 \s\_\_fp #3; #4;
12863     { \fi: \_\_fp\_exp\_after_o:w \s\_\_fp #3; }

```

(End definition for `__fp_minmax_break_o:w`.)

25.6 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.

```

12864 \cs_new:cpn { \_\_fp\_not_o:w } #1 \s\_\_fp \_\_fp\_chk:w #2#3; @
12865     {
12866         \if_meaning:w 0 #2
12867             \exp_after:wN \exp_after:wN \exp_after:wN \c\_one_fp
12868         \else:
12869             \exp_after:wN \exp_after:wN \exp_after:wN \c\_zero_fp
12870         \fi:
12871     }

```

(End definition for `__fp_not_o:w`.)

```
\__fp_&_o:ww
\__fp_|_o:ww
\__fp_and_return:wNw
```

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.

```
12872 \group_begin:
12873   \char_set_catcode_letter:N &
12874   \char_set_catcode_letter:N |
12875   \cs_new:Npn \__fp_&_o:ww #1 \s_fp \__fp_chk:w #2#3;
12876   {
12877     \if_meaning:w 0 #2 #1
12878       \__fp_and_return:wNw \s_fp \__fp_chk:w #2#3;
12879     \fi:
12880     \__fp_exp_after_o:w
12881   }
12882   \cs_new:Npn \__fp_|_o:ww { \__fp_&_o:ww \else: }
12883 \group_end:
12884 \cs_new:Npn \__fp_and_return:wNw #1; \fi: #2#3; { \fi: #2 #1; }
```

(End definition for `__fp_&_o:ww`, `__fp_|_o:ww`, and `__fp_and_return:wNw`.)

25.7 Ternary operator

```
\__fp_ternary:NwwN
\__fp_ternary_auxi:NwwN
\__fp_ternary_auxii:NwwN
\__fp_ternary_loop_break:w
  \__fp_ternary_loop:Nw
\__fp_ternary_map_break:
\__fp_ternary_break_point:n
```

The first function receives the test and the true branch of the `?:` ternary operator. It returns the true branch, unless the test branch is zero. In that case, the function returns a very specific `nan`. The second function receives the output of the first function, and the false branch. It returns the previous input, unless that is the special `nan`, in which case we return the false branch.

```
12885 \cs_new:Npn \__fp_ternary:NwwN #1 #2@ #3@ #4
12886   {
12887     \if_meaning:w \__fp_parse_infix_::N #4
12888       \__fp_ternary_loop:Nw
12889         #
12890         \s_fp \__fp_chk:w { \__fp_ternary_loop_break:w } ;
12891       \__fp_ternary_break_point:n { \exp_after:wN \__fp_ternary_auxi:NwwN }
12892       \exp_after:wN #1
12893       \exp:w \exp_end_continue_f:w
12894       \__fp_exp_after_array_f:w #3 \s_fp_stop
12895       \exp_after:wN @
12896       \exp:w
12897         \__fp_parse_operand:Nw \c_fp_prec_colon_int
12898         \__fp_parse_expand:w
12899   \else:
12900     \__msg_kernel_expandable_error:nnnn
12901       { kernel } { fp-missing } { : } { ~for~?~:~ }
12902       \exp_after:wN \__fp_parse_continue:NwN
12903       \exp_after:wN #1
12904       \exp:w \exp_end_continue_f:w
12905       \__fp_exp_after_array_f:w #3 \s_fp_stop
12906       \exp_after:wN #4
12907       \exp_after:wN #1
12908     \fi:
12909   }
```

```

12910 \cs_new:Npn \__fp_ternary_loop_break:w
12911     #1 \fi: #2 \__fp_ternary_break_point:n #3
12912 {
12913     0 = 0 \exp_stop_f: \fi:
12914     \exp_after:wN \__fp_ternary_auxii:NwwN
12915 }
12916 \cs_new:Npn \__fp_ternary_loop:Nw \s__fp \__fp_chk:w #1#2;
12917 {
12918     \if_int_compare:w #1 > 0 \exp_stop_f:
12919         \exp_after:wN \__fp_ternary_map_break:
12920         \fi:
12921         \__fp_ternary_loop:Nw
12922     }
12923 \cs_new:Npn \__fp_ternary_map_break: #1 \__fp_ternary_break_point:n #2 {#2}
12924 \cs_new:Npn \__fp_ternary_auxi:NwwN #1#2@#3@#4
12925 {
12926     \exp_after:wN \__fp_parse_continue:NwN
12927     \exp_after:wN #1
12928     \exp:w \exp_end_continue_f:w
12929     \__fp_exp_after_array_f:w #2 \s__fp_stop
12930     #4 #1
12931 }
12932 \cs_new:Npn \__fp_ternary_auxii:NwwN #1#2@#3@#4
12933 {
12934     \exp_after:wN \__fp_parse_continue:NwN
12935     \exp_after:wN #1
12936     \exp:w \exp_end_continue_f:w
12937     \__fp_exp_after_array_f:w #3 \s__fp_stop
12938     #4 #1
12939 }

(End definition for \__fp_ternary:NwwN and others.)
12940 ⟨/initex | package⟩

```

26 l3fp-basics Implementation

```

12941 ⟨*initex | package⟩
12942 ⟨@@=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.

```

12943 \cs_new:Npn \__fp_parse_word_abs:N
12944     { \__fp_parse_unary_function:NNN \__fp_set_sign_o:w 0 }
12945 \cs_new:Npn \__fp_parse_word_sign:N
12946     { \__fp_parse_unary_function:NNN \__fp_sign_o:w ? }
12947 \cs_new:Npn \__fp_parse_word_sqrt:N
12948     { \__fp_parse_unary_function:NNN \__fp_sqrt_o:w ? }

```

(End definition for `_fp_parse_word_abs:N`, `_fp_parse_word_sign:N`, and `_fp_parse_word_sqrt:N`.)

26.1 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.

26.1.1 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`.

```

12949 \cs_new:cpx { \_fp_-_o:ww } \s_fp
12950   {
12951     \exp_not:c { \_fp+_o:ww }
12952     \exp_not:n { \s_fp \_fp_neg_sign:N }
12953   }

```

(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 `\langle types \rangle` #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 `\langle sign_2 \rangle` (expansion of #1#5) as an argument. If the `\langle types \rangle` are distinct, the result is simply the floating point number with the highest `\langle type \rangle`. Since case 3 (used for two `nan`) also picks the first operand, we can also use it

when $\langle type_1 \rangle$ is greater than $\langle type_2 \rangle$. Also note that we don't need to worry about $\langle sign_2 \rangle$ in that case since the second operand is discarded.

```

12954 \cs_new:cpn { __fp+_o:ww }
12955     \s_fp #1 __fp_chk:w #2 #3 ; \s_fp __fp_chk:w #4 #5
12956 {
12957     \if_case:w
12958         \if_meaning:w #2 #4
12959             #2
12960         \else:
12961             \if_int_compare:w #2 > #4 \exp_stop_f:
12962                 3
12963             \else:
12964                 4
12965             \fi:
12966         \fi:
12967         \exp_stop_f:
12968             \exp_after:wN __fp_add_zeros_o:Nww __int_value:w
12969         \or: \exp_after:wN __fp_add_normal_o:Nww __int_value:w
12970         \or: \exp_after:wN __fp_add_inf_o:Nww __int_value:w
12971         \or: __fp_case_return_i_o:ww
12972         \else: \exp_after:wN __fp_add_return_ii_o:Nww __int_value:w
12973         \fi:
12974         #1 #5
12975         \s_fp __fp_chk:w #2 #3 ;
12976         \s_fp __fp_chk:w #4 #5
12977     }

```

(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.

```

12978 \cs_new:Npn __fp_add_return_ii_o:Nww #1 #2 ; \s_fp __fp_chk:w #3 #4
12979 { __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 .

```

12980 \cs_new:Npn __fp_add_zeros_o:Nww #1 \s_fp __fp_chk:w 0 #2
12981 {
12982     \if_int_compare:w #1 = 20 \exp_stop_f:
12983         \exp_after:wN __fp_add_return_ii_o:Nww
12984     \else:
12985         __fp_case_return_i_o:ww
12986     \fi:
12987     #1
12988     \s_fp __fp_chk:w 0 #2
12989 }

```

(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.

```

12990 \cs_new:Npn __fp_add_inf_o:Nww

```

```

12991      #1 \s_fp \fp_chk:w 2 #2 #3; \s_fp \fp_chk:w 2 #4
12992  {
12993      \if_meaning:w #1 #2
12994          \fp_case_return_i_o:ww
12995      \else:
12996          \fp_case_use:nw
12997          {
12998              \exp_last_unbraced:Nf \fp_invalid_operation_o:Nww
12999                  { \token_if_eq_meaning:NNTF #1 #4 + - }
13000          }
13001      \fi:
13002      \s_fp \fp_chk:w 2 #2 #3;
13003      \s_fp \fp_chk:w 2 #4
13004  }

```

(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.

```

13005 \cs_new:Npn \fp_add_normal_o:Nww #1 \s_fp \fp_chk:w 1 #2
13006  {
13007      \if_meaning:w #1#2
13008          \exp_after:wN \fp_add_npos_o:NnwNnw
13009      \else:
13010          \exp_after:wN \fp_sub_npos_o:NnwNnw
13011      \fi:
13012      #2
13013  }

```

(End definition for $_fp_add_normal_o:Nww$.)

26.1.2 Absolute addition

In this subsection, we perform the addition of two positive normal numbers.

```

\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 $_int_eval:w$, 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.

```

13014 \cs_new:Npn \fp_add_npos_o:NnwNnw #1#2#3 ; \s_fp \fp_chk:w 1 #4 #5
13015  {
13016      \exp_after:wN \fp_sanitize:Nw
13017      \exp_after:wN #1
13018      \int_value:w \int_eval:w
13019      \if_int_compare:w #2 > #5 \exp_stop_f:
13020          #2

```

```

13021      \exp_after:wN \__fp_add_big_i_o:wNww \__int_value:w -
13022      \else:
13023          #5
13024          \exp_after:wN \__fp_add_big_ii_o:wNww \__int_value:w
13025      \fi:
13026      \__int_eval:w #5 - #2 ; #1 #3;
13027  }

```

(End definition for `__fp_add_npos_o:NnwNnw`.)

`__fp_add_big_i_o:wNww` $\langle shift \rangle$; $\langle final\ sign \rangle$ $\langle body_1 \rangle$; $\langle body_2 \rangle$;

Used in `\l3fp-expo`. Shift the significand of the small number, then add with `__fp_add_significand_o:NnnwnnnnnN`.

```

13028 \cs_new:Npn \__fp_add_big_i_o:wNww #1; #2 #3; #4;
13029  {
13030      \__fp_decimate:nNnnnn {#1}
13031      \__fp_add_significand_o:NnnwnnnnnN
13032          #4
13033          #3
13034          #2
13035      }
13036 \cs_new:Npn \__fp_add_big_ii_o:wNww #1; #2 #3; #4;
13037  {
13038      \__fp_decimate:nNnnnn {#1}
13039      \__fp_add_significand_o:NnnwnnnnnN
13040          #3
13041          #4
13042          #2
13043      }

```

(End definition for `__fp_add_big_i_o:wNww` and `__fp_add_big_ii_o:wNww`.)

`__fp_add_significand_o:NnnwnnnnnN` $\langle rounding\ digit \rangle$ $\{ \langle Y_1 \rangle \}$ $\{ \langle Y_2 \rangle \}$
`__fp_add_significand_pack:NNNNNN`

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 $0.99\cdots 95 \rightarrow 1.00\cdots 0$, but this situation always give an exact power of 10, for which it is easy to correct the result at the end.

```

13044 \cs_new:Npn \__fp_add_significand_o:NnnwnnnnnN #1 #2#3 #4; #5#6#7#8
13045  {
13046      \exp_after:wN \__fp_add_significand_test_o:N
13047      \__int_value:w \__int_eval:w 1#5#6 + #2
13048      \exp_after:wN \__fp_add_significand_pack:NNNNNNNN
13049      \__int_value:w \__int_eval:w 1#7#8 + #3 ; #1
13050  }
13051 \cs_new:Npn \__fp_add_significand_pack:NNNNNNNN #1 #2#3#4#5#6#7
13052  {
13053      \if_meaning:w 2 #1
13054          + 1
13055      \fi:
13056      ; #2 #3 #4 #5 #6 #7 ;
13057  }
13058 \cs_new:Npn \__fp_add_significand_test_o:N #1

```

```

13059   {
13060     \if_meaning:w 2 #1
13061       \exp_after:wN \__fp_add_significand_carry_o:wwwNN
13062     \else:
13063       \exp_after:wN \__fp_add_significand_no_carry_o:wwwNN
13064     \fi:
13065   }

```

(End definition for $__fp_add_significand_o:NnnnnnnnN$, $__fp_add_significand_pack:NNNNNNN$, and $__fp_add_significand_test_o:N$.)

$__fp_add_significand_no_carry_o:wwwNN$
 $__fp_add_significand_no_carry_o:wwwNN \langle 8d \rangle ; \langle 6d \rangle ; \langle 2d \rangle ; \langle rounding digit \rangle \langle sign \rangle$

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.

```

13066 \cs_new:Npn \__fp_add_significand_no_carry_o:wwwNN
13067   #1; #2; #3#4 ; #5#6
13068   {
13069     \exp_after:wN \_\_fp_basics_pack_high:NNNNNw
13070     \_int_value:w \_\_int_eval:w 1 #1
13071       \exp_after:wN \_\_fp_basics_pack_low:NNNNNw
13072       \_int_value:w \_\_int_eval:w 1 #2 #3#4
13073         + \_\_fp_round:NNN #6 #4 #5
13074       \exp_after:wN ;
13075   }

```

(End definition for $__fp_add_significand_no_carry_o:wwwNN$.)

$__fp_add_significand_carry_o:wwwNN$
 $__fp_add_significand_carry_o:wwwNN \langle 8d \rangle ; \langle 6d \rangle ; \langle 2d \rangle ; \langle rounding digit \rangle \langle sign \rangle$

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.

```

13076 \cs_new:Npn \__fp_add_significand_carry_o:wwwNN
13077   #1; #2; #3#4; #5#6
13078   {
13079     + 1
13080     \exp_after:wN \_\_fp_basics_pack_weird_high:NNNNNNNNw
13081     \_int_value:w \_\_int_eval:w 1 1 #1
13082       \exp_after:wN \_\_fp_basics_pack_weird_low:NNNNw
13083       \_int_value:w \_\_int_eval:w 1 #2#3 +
13084         \exp_after:wN \_\_fp_round:NNN
13085         \exp_after:wN #6
13086         \exp_after:wN #3
13087         \_int_value:w \_\_fp_round_digit:Nw #4 #5 ;
13088       \exp_after:wN ;
13089   }

```

(End definition for $__fp_add_significand_carry_o:wwwNN$.)

26.1.3 Absolute subtraction

$__fp_sub_npos_o:NnwNnw$
 $__fp_sub_eq_o:NnwNw$
 $__fp_sub_npos_ii_o:NnwNw$
 $__fp_sub_npos_o:NnwNnw \langle sign_1 \rangle \langle exp_1 \rangle \langle body_1 \rangle ; \s_fp __fp_chk:w 1$
 $\langle initial sign_2 \rangle \langle exp_2 \rangle \langle body_2 \rangle ;$

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 $\langle sign_1 \rangle$.

```

13090 \cs_new:Npn \_fp_sub_npos_o:Nnwnw #1#2#3; \s_fp \_fp_chk:w 1 #4#5#6;
13091 {
13092   \if_case:w \_fp_compare_npos:nwnw {#2} #3; {#5} #6; \exp_stop_f:
13093     \exp_after:wN \_fp_sub_eq_o:Nnwnw
13094   \or:
13095     \exp_after:wN \_fp_sub_npos_i_o:Nnwnw
13096   \else:
13097     \exp_after:wN \_fp_sub_npos_ii_o:Nnwnw
13098   \fi:
13099   #1 {#2} #3; {#5} #6;
13100 }
13101 \cs_new:Npn \_fp_sub_eq_o:Nnwnw #1#2; #3; { \exp_after:wN \c_zero_fp }
13102 \cs_new:Npn \_fp_sub_npos_ii_o:Nnwnw #1 #2; #3;
13103 {
13104   \exp_after:wN \_fp_sub_npos_i_o:Nnwnw
13105     \int_value:w \_fp_neg_sign:N #1
13106     #3; #2;
13107 }
```

(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.

```

13108 \cs_new:Npn \_fp_sub_npos_i_o:Nnwnw #1 #2#3; #4#5;
13109 {
13110   \exp_after:wN \_fp_sanitize:Nw
13111   \exp_after:wN #1
13112   \int_value:w \int_eval:w
13113     #2
13114   \if_int_compare:w #2 = #4 \exp_stop_f:
13115     \exp_after:wN \_fp_sub_back_near_o:nnnnnnnnN
13116   \else:
13117     \exp_after:wN \_fp_decimate:nNnnnn \exp_after:wN
13118       { \int_value:w \int_eval:w #2 - #4 - 1 \exp_after:wN }
13119     \exp_after:wN \_fp_sub_back_far_o:NnnnnnnnN
13120   \fi:
13121     #5
13122   #3
13123   #1
13124 }
```

(End definition for `_fp_sub_npos_i_o:Nnwnw`.)

`_fp_sub_back_near_o:nnnnnnnnN`
`_fp_sub_back_near_pack:NNNNNNNw`
`_fp_sub_back_near_after:wNNNNw`

`_fp_sub_back_near_o:nnnnnnnnN {⟨Y1⟩} {⟨Y2⟩} {⟨Y3⟩} {⟨Y4⟩} {⟨X1⟩}`
`{⟨X2⟩} {⟨X3⟩} {⟨X4⟩} ⟨final sign⟩`

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.

```

13125 \cs_new:Npn \__fp_sub_back_near_o:nxxxxxN #1#2#3#4 #5#6#7#8 #9
13126 {
13127   \exp_after:wN \__fp_sub_back_near_after:wNNNNw
13128   \__int_value:w \__int_eval:w 10#5#6 - #1#2 - 11
13129   \exp_after:wN \__fp_sub_back_near_pack:NNNNNNw
13130   \__int_value:w \__int_eval:w 11#7#8 - #3#4 \exp_after:wN ;
13131 }
13132 \cs_new:Npn \__fp_sub_back_near_pack:NNNNNNw #1#2#3#4#5#6#7 ;
13133 { + #1#2 ; {#3#4#5#6} {#7} ; }
13134 \cs_new:Npn \__fp_sub_back_near_after:wNNNNw 10 #1#2#3#4 #5 ;
13135 {
13136   \if_meaning:w 0 #1
13137   \exp_after:wN \__fp_sub_back_shift:wnnnn
13138   \fi:
13139   ; {#1#2#3#4} {#5}
13140 }

```

(End definition for $\langle __fp_sub_back_near_o:nxxxxxN, __fp_sub_back_near_pack:NNNNNNw, __fp_sub_back_near_after:wNNNNw \rangle$.)

```

\__fp_sub_back_shift:wnnnn
\__fp_sub_back_shift_ii:ww
  \__fp_sub_back_shift_iii:NNNNNNNNw
    \__fp_sub_back_shift_iv:nnnnw

```

$\langle __fp_sub_back_shift:wnnnn ; \{Z_1\} \{Z_2\} \{Z_3\} \{Z_4\} \rangle$;

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.

```

13141 \cs_new:Npn \__fp_sub_back_shift:wnnnn ; #1#2
13142 {
13143   \exp_after:wN \__fp_sub_back_shift_ii:ww
13144   \__int_value:w #1 #2 0 ;
13145 }
13146 \cs_new:Npn \__fp_sub_back_shift_ii:ww #1 0 ; #2#3 ;
13147 {
13148   \if_meaning:w 0 #1 0
13149   - 7
13150   - \exp_after:wN \use_i:nnn
13151   \exp_after:wN \__fp_sub_back_shift_iii:NNNNNNNNw
13152   \__int_value:w #2#3 0 ~ 123456789;
13153 \else:
13154   - \__fp_sub_back_shift_iii:NNNNNNNNw #1 123456789;
13155 \fi:
13156 \exp_after:wN \__fp_pack_twice_four:wNNNNNNNN
13157 \exp_after:wN \__fp_pack_twice_four:wNNNNNNNN
13158 \exp_after:wN \__fp_sub_back_shift_iv:nnnnw
13159 \exp_after:wN ;
13160 \__int_value:w

```

```

13161      #1 ~ #2#3 0 ~ 0000 0000 0000 000 ;
13162    }
13163 \cs_new:Npn \__fp_sub_back_shift_iii:NNNNNNNNw #1#2#3#4#5#6#7#8#9; {#8}
13164 \cs_new:Npn \__fp_sub_back_shift_iv:nnnw #1 ; #2 ; { ; #1 ; }

(End definition for \__fp_sub_back_shift:wnnnn and others.)

```

`__fp_sub_back_far_o:NnnwnnnN` $\langle \text{rounding} \rangle \{ \langle Y_1 \rangle \} \{ \langle Y_2 \rangle \}$
 $\langle \text{extra-digits} \rangle ; \{ \langle X_1 \rangle \} \{ \langle X_2 \rangle \} \{ \langle X_3 \rangle \} \{ \langle X_4 \rangle \} \langle \text{final sign} \rangle$

If the difference is greater than $10^{\langle \text{expo}_x \rangle}$, call the `very_far` auxiliary. If the result is less than $10^{\langle \text{expo}_x \rangle}$, 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).

```

13165 \cs_new:Npn \__fp_sub_back_far_o:NnnwnnnN #1 #2#3 #4; #5#6#7#8
13166  {
13167    \if_case:w
13168      \if_int_compare:w 1 #2 = #5#6 \use_i:nnnn #7 \exp_stop_f:
13169        \if_int_compare:w #3 = \use_none:n #7#8 0 \exp_stop_f:
13170          0
13171        \else:
13172          \if_int_compare:w #3 > \use_none:n #7#8 0 - \fi: 1
13173        \fi:
13174      \else:
13175        \if_int_compare:w 1 #2 > #5#6 \use_i:nnnn #7 - \fi: 1
13176        \fi:
13177      \exp_stop_f:
13178        \exp_after:wN \__fp_sub_back Quite_far_o:wwNN
13179      \or: \exp_after:wN \__fp_sub_back very_far_o:wwwNN
13180      \else: \exp_after:wN \__fp_sub_back not_far_o:wwwNN
13181      \fi:
13182      #2 ~ #3 ; #5 #6 ~ #7 #8 ; #1
13183  }

(End definition for \__fp_sub_back_far_o:NnnwnnnN.)
```

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 $\langle \text{rounding} \rangle$ #3 and the $\langle \text{final sign} \rangle$ #4 control whether we get 1 or 0.999999999999999. In the usual round-to-nearest mode, we will get 1 whenever the $\langle \text{rounding} \rangle$ digit is less than or equal to 5 (remember that the $\langle \text{rounding} \rangle$ digit is only equal to 5 if there was no further non-zero digit).

```

13184 \cs_new:Npn \__fp_sub_back_quite_far_o:wwNN #1; #2; #3#4
13185  {
13186    \exp_after:wN \__fp_sub_back_quite_far_i:NN
13187    \exp_after:wN #3
13188    \exp_after:wN #4
13189  }
13190 \cs_new:Npn \__fp_sub_back_quite_far_i:NN #1#2
13191  {
13192    \if_case:w \__fp_round_neg:NNN #2 0 #1
13193      \exp_after:wN \use_i:nn
13194    \else:
13195      \exp_after:wN \use_i:nn
```

```

13196     \fi:
13197     { ; {1000} {0000} {0000} {0000} ; }
13198     { - 1 ; {9999} {9999} {9999} {9999} ; }
13199   }

```

(End definition for `_fp_sub_back Quite_far_o:wwwNN` 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.

```

13200 \cs_new:Npn \_fp_sub_back_not_far_o:wwwNN #1 ~ #2; #3 ~ #4; #5#6
13201   {
13202     - 1
13203     \exp_after:wN \_fp_sub_back_near_after:wNNNNw
13204     \_int_value:w \_int_eval:w 1#30 - #1 - 11
13205     \exp_after:wN \_fp_sub_back_near_pack:NNNNNNw
13206     \_int_value:w \_int_eval:w 11 0000 0000 + #40 - #2
13207     - \exp_after:wN \_fp_round_neg:NNN
13208     \exp_after:wN #6
13209     \use_none:nnnnnn #2 #5
13210     \exp_after:wN ;
13211   }

```

(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.

```

13212 \cs_new:Npn \_fp_sub_back_very_far_o:wwwNN #1#2#3#4#5#6#7
13213   {
13214     \_fp_pack_eight:wNNNNNNNN
13215     \_fp_sub_back_very_far_ii_o:nnNwwNN
13216     { 0 #1#2#3 #4#5#6#7 }
13217     ;
13218   }
13219 \cs_new:Npn \_fp_sub_back_very_far_ii_o:nnNwwNN #1#2 ; #3 ; #4 ~ #5; #6#7
13220   {
13221     \exp_after:wN \_fp_basics_pack_high:NNNNNw
13222     \_int_value:w \_int_eval:w 1#4 - #1 - 1
13223     \exp_after:wN \_fp_basics_pack_low:NNNNNw
13224     \_int_value:w \_int_eval:w 2#5 - #2
13225     - \exp_after:wN \_fp_round_neg:NNN
13226     \exp_after:wN #7

```

```

13227     \_\_int_value:w
13228         \if_int_odd:w \_\_int_eval:w #5 - #2 \_\_int_eval_end:
13229             1 \else: 2 \fi:
13230             \_\_int_value:w \_\_fp_round_digit:Nw #3 #6 ;
13231             \exp_after:wN ;
13232     }

```

(End definition for `__fp_sub_back_very_far_o:wwwwNN` and `__fp_sub_back_very_far_iio:nnNwwNN`.)

26.2 Multiplication

26.2.1 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`.

```

13233 \cs_new:cpxn { \_\_fp_*_o:ww }
13234 {
13235     \_\_fp_mul_cases_o:NnNnww
13236     *
13237     { - 2 + }
13238     \_\_fp_mul_npos_o:Nww
13239     { }
13240 }

```

(End definition for `__fp_*_o:ww`.)

`__fp_mul_cases_o:nNnnww`

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.

```

13241 \cs_new:Npn \_\_fp_mul_cases_o:NnNnww
13242     #1#2#3#4 \s__fp \_\_fp_chk:w #5#6#7; \s__fp \_\_fp_chk:w #8#9
13243 {
13244     \if_case:w \_\_int_eval:w
13245         \if_int_compare:w #5 #8 = 11 ~
13246             1
13247         \else:
13248             \if_meaning:w 3 #8
13249                 3
13250             \else:
13251                 \if_meaning:w 3 #5
13252                     2
13253             \else:

```

```

13254           \if_int_compare:w #5 #8 = 10 ~
13255               9 #2 - 2
13256       \else:
13257           (#5 #2 #8) / 2 * 2 + 7
13258       \fi:
13259   \fi:
13260 \fi:
13261 \fi:
13262 \if_meaning:w #6 #9 - 1 \fi:
13263     \_int_eval_end:
13264     \_fp_case_use:nw { #3 0 }
13265     \or: \_fp_case_use:nw { #3 2 }
13266     \or: \_fp_case_return_i_o:ww
13267     \or: \_fp_case_return_ii_o:ww
13268     \or: \_fp_case_return_o:Nww \c_zero_fp
13269     \or: \_fp_case_return_o:Nww \c_minus_zero_fp
13270     \or: \_fp_case_use:nw { \_fp_invalid_operation_o:Nww #1 }
13271     \or: \_fp_case_use:nw { \_fp_invalid_operation_o:Nww #1 }
13272     \or: \_fp_case_return_o:Nww \c_inf_fp
13273     \or: \_fp_case_return_o:Nww \c_minus_inf_fp
13274     #4
13275   \fi:
13276   \s_fp \_fp_chk:w #5 #6 #7;
13277   \s_fp \_fp_chk:w #8 #9
13278 }

```

(End definition for `_fp_mul_cases_o:nNnnww`.)

26.2.2 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, `_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.

```

13279 \cs_new:Npn \_fp_mul_npos_o:Nww
13280     #1 \s_fp \_fp_chk:w #2 #3 #4 #5 ; \s_fp \_fp_chk:w #6 #7 #8 #9 ;
13281 {
13282     \exp_after:wN \_fp_sanitize:Nw
13283     \exp_after:wN #1
13284     \_int_value:w \_int_eval:w
13285     #4 + #8
13286     \_fp_mul_significand_o:nnnnNnnnn #5 #1 #9
13287 }

```

(End definition for `_fp_mul_npos_o:Nww`.)

```
\_fp_mul_significand_o:nnnnNnnnn {<X1>} {<X2>} {<X3>} {<X4>} <sign>
{<Y1>} {<Y2>} {<Y3>} {<Y4>}
```

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 `_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 `_int_eval:w`.

```

13288 \cs_new:Npn \_fp_mul_significand_o:nnnnNnnnn #1#2#3#4 #5 #6#7#8#9
13289 {
13290   \exp_after:wN \_fp_mul_significand_test_f:NNN
13291   \exp_after:wN #5
13292   \_int_value:w \_int_eval:w 99990000 + #1##6 +
13293   \exp_after:wN \_fp_mul_significand_keep:NNNNNw
13294   \_int_value:w \_int_eval:w 99990000 + #1##7 + #2##6 +
13295   \exp_after:wN \_fp_mul_significand_keep:NNNNNw
13296   \_int_value:w \_int_eval:w 99990000 + #1##8 + #2##7 + #3##6 +
13297   \exp_after:wN \_fp_mul_significand_drop:NNNNNw
13298   \_int_value:w \_int_eval:w 99990000 + #1##9 + #2##8 + #3##7 + #4##6 +
13299   \exp_after:wN \_fp_mul_significand_drop:NNNNNw
13300   \_int_value:w \_int_eval:w 99990000 + #2##9 + #3##8 + #4##7 +
13301   \exp_after:wN \_fp_mul_significand_drop:NNNNNw
13302   \_int_value:w \_int_eval:w 99990000 + #3##9 + #4##8 +
13303   \exp_after:wN \_fp_mul_significand_drop:NNNNNw
13304   \_int_value:w \_int_eval:w 100000000 + #4##9 ;
13305   ; \exp_after:wN ;
13306 }
13307 \cs_new:Npn \_fp_mul_significand_drop:NNNNNw #1#2#3#4#5 #6;
13308 { #1#2#3#4#5 ; + #6 }
13309 \cs_new:Npn \_fp_mul_significand_keep:NNNNNw #1#2#3#4#5 #6;
13310 { #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`

$$_fp_mul_significand_test_f:NNN \langle sign \rangle 1 \langle digits 1-8 \rangle ; \langle digits 9-12 \rangle ; \\ \langle digits 13-16 \rangle ; + \langle digits 17-20 \rangle + \langle digits 21-24 \rangle + \langle digits 25-28 \rangle + \langle digits 29-32 \rangle ; \exp_{after}:wN ;$$

If the $\langle digit 1 \rangle$ 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 $\langle digit 1 \rangle$ is zero, we care about digits 17 and 18, and whether further digits are zero.

```

13311 \cs_new:Npn \_fp_mul_significand_test_f:NNN #1 #2 #3
13312 {
13313   \if_meaning:w 0 #3
13314     \exp_after:wN \_fp_mul_significand_small_f:NNwwwN
13315   \else:
13316     \exp_after:wN \_fp_mul_significand_large_f:NwwNNNN
13317   \fi:
13318   #1 #3
13319 }
```

(End definition for `_fp_mul_significand_test_f:NNN`.)

`_fp_mul_significand_large_f:NwwNNNN` In this branch, $\langle digit\ 1 \rangle$ is non-zero. The result is thus $\langle digits\ 1-16 \rangle$, 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 $\langle rounding\ digit \rangle$, suitable for `_fp_round:NNN`.

```

13320 \cs_new:Npn \_fp_mul_significand_large_f:NwwNNNN #1 #2; #3; #4#5#6#7; +
13321 {
13322     \exp_after:wN \_fp_basics_pack_high:NNNNNw
13323     \_int_value:w \_int_eval:w 1#2
13324     \exp_after:wN \_fp_basics_pack_low:NNNNNw
13325     \_int_value:w \_int_eval:w 1#3#4#5#6#7
13326     + \exp_after:wN \_fp_round:NNN
13327     \exp_after:wN #1
13328     \exp_after:wN #7
13329     \_int_value:w \_fp_round_digit:Nw
13330 }
```

(End definition for `_fp_mul_significand_large_f:NwwNNNN`.)

`_fp_mul_significand_small_f:NNwwwN` In this branch, $\langle digit\ 1 \rangle$ is zero. Our result will thus be $\langle digits\ 2-17 \rangle$, 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.

```

13331 \cs_new:Npn \_fp_mul_significand_small_f:NNwwwN #1 #2#3; #4#5; #6; + #7
13332 {
13333     - 1
13334     \exp_after:wN \_fp_basics_pack_high:NNNNNw
13335     \_int_value:w \_int_eval:w 1#3#4
13336     \exp_after:wN \_fp_basics_pack_low:NNNNNw
13337     \_int_value:w \_int_eval:w 1#5#6#7
13338     + \exp_after:wN \_fp_round:NNN
13339     \exp_after:wN #1
13340     \exp_after:wN #7
13341     \_int_value:w \_fp_round_digit:Nw
13342 }
```

(End definition for `_fp_mul_significand_small_f:NNwwwN`.)

26.3 Division

26.3.1 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:NnNnww` are provided as the fourth argument here.

```
13343 \cs_new:cpx { _fp_/_o:ww }
```

```

13344  {
13345    \__fp_mul_cases_o:NnNnww
13346    /
13347    { - }
13348    \__fp_div_npos_o:Nww
13349    {
13350      \or:
13351        \__fp_case_use:nw
13352          { \__fp_division_by_zero_o>NNww \c_inf_fp / }
13353        \or:
13354          \__fp_case_use:nw
13355            { \__fp_division_by_zero_o>NNww \c_minus_inf_fp / }
13356        }
13357    }

```

(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}
{\langle A_1 \rangle} {\langle A_2 \rangle} {\langle A_3 \rangle} {\langle A_4 \rangle} ; \s__fp \__fp_chk:w 1 <sign_Z> {\exp Z}
{\langle Z_1 \rangle} {\langle Z_2 \rangle} {\langle Z_3 \rangle} {\langle Z_4 \rangle} ;

```

We want to compute A/Z . As for multiplication, $__fp_sanitize:Nw$ checks for overflow or underflow; we provide it with the $\langle final\ sign\rangle$, 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 $\{\langle A_i \rangle\}$, then the four $\{\langle Z_i \rangle\}$, a semi-colon, and the $\langle final\ sign\rangle$, used for rounding at the end.

```

13358 \cs_new:Npn \__fp_div_npos_o:Nww
13359   #1 \s__fp \__fp_chk:w 1 #2 #3 #4 ; \s__fp \__fp_chk:w 1 #5 #6 #7#8#9;
13360   {
13361     \exp_after:wN \__fp_SANITIZE:Nw
13362     \exp_after:wN #1
13363     \__int_value:w \__int_eval:w
13364       #3 - #6
13365     \exp_after:wN \__fp_div_significand_i_o:wnnw
13366       \__int_value:w \__int_eval:w #7 \use_i:nnnn #8 + 1 ;
13367       #4
13368       {[#7]}{[#8]}#9 ;
13369       #1
13370   }

```

(End definition for $__fp_div_npos_o:Nww$.)

26.3.2 Work plan

In this subsection, we explain how to avoid overflowing T_EX'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 will be 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

$$\text{\textbackslash int_eval:n}\left\{\frac{A_1 A_2}{Z_1 + 1} - 1\right\} \leq 10^4 \frac{A}{Z}$$

Subtracting 1 at the end takes care of the fact that ε - \TeX 's `_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

$$Q_A = \text{\textbackslash int_eval:n}\left\{\frac{10 \cdot A_1 A_2}{\lfloor 10^{-3} \cdot Z_1 Z_2 \rfloor + 1} - 1\right\}.$$

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 ε - \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 :

$$\begin{aligned}
10^5 B &= A_1 A_2 0 + 10 \cdot 0 \cdot A_3 A_4 - 10 \cdot Z_1 \cdot Z_2 Z_3 Z_4 \cdot Q_A \\
&< A_1 A_2 0 \cdot \left(1 - 10 \cdot \frac{Z_1 \cdot Z_2 Z_3 Z_4}{y}\right) + \frac{3}{2} \cdot 10 \cdot Z_1 \cdot Z_2 Z_3 Z_4 + 10 \\
&\leq \frac{A_1 A_2 0 \cdot (y - 10 \cdot Z_1 \cdot Z_2 Z_3 Z_4)}{y} + \frac{3}{2} y + 10 \\
&\leq \frac{A_1 A_2 0 \cdot 1}{y} + \frac{3}{2} y + 10 \leq \frac{10^9 A}{y} + 1.6 \cdot y.
\end{aligned}$$

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 T_EX's bounds in all cases!

We will 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 will be 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

$$P = \text{\textbackslash int_eval:n} \left\{ \frac{2 \cdot E_1 E_2}{Z_1 Z_2} \right\},$$

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.

26.3.3 Implementing the significand division

```
\_\_fp\_div\_significand\_i\_o:wnnw
  \_\_fp\_div\_significand\_i\_o:wnnw <y> ; {<A1>} {<A2>} {<A3>} {<A4>}
    {<Z1>} {<Z2>} {<Z3>} {<Z4>} ; <sign>
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  $\langle \text{continuation} \rangle$  arguments for 4 consecutive calls to  $\_\_fp\_div\_significand\_calc:wnnnnnnnn$ . Each of these calls will need  $\langle y \rangle$  (#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.
```

```
13371 \cs_new:Npn \_\_fp\_div\_significand\_i\_o:wnnw #1 ; #2#3 #4 ;
13372   {
13373     \exp_after:wN \_\_fp\_div\_significand\_test_o:w
13374     \_\_int_value:w \_\_int_eval:w
13375       \exp_after:wN \_\_fp\_div\_significand\_calc:wnnnnnnnn
13376       \_\_int_value:w \_\_int_eval:w 999999 + #2 #3 0 / #1 ;
13377       #2 #3 ;
13378       #4
13379       { \exp_after:wN \_\_fp\_div\_significand\_ii:wwn \_\_int_value:w #1 }
13380       { \exp_after:wN \_\_fp\_div\_significand\_ii:wwn \_\_int_value:w #1 }
13381       { \exp_after:wN \_\_fp\_div\_significand\_ii:wwn \_\_int_value:w #1 }
13382       { \exp_after:wN \_\_fp\_div\_significand\_iii:wwnnnnn \_\_int_value:w #1 }
13383   }
```

(End definition for $__fp_div_significand_i_o:wnnw$.)

```
\_\_fp\_div\_significand\_calc:wnnnnnnnn <math>10^6 + Q_A</math> ; <math>\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 \text{continuation} \rangle \}
```

expands to

$$\langle 10^6 + Q_A \rangle \langle \text{continuation} \rangle ; \langle B_1 \rangle \langle B_2 \rangle ; \{\langle B_3 \rangle\} \{\langle B_4 \rangle\} \{\langle Z_1 \rangle\} \{\langle Z_2 \rangle\} \{\langle Z_3 \rangle\} \\ \{\langle Z_4 \rangle\}$$

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 `\langle 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 $\#1, \#2, \dots$ 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.

```

13384 \cs_new:Npn \__fp_div_significand_calc:wwnnnnnnn 1#1
13385   {
13386     \if_meaning:w 1 #1
13387       \exp_after:wN \__fp_div_significand_calc_i:wwnnnnnnn
13388     \else:
13389       \exp_after:wN \__fp_div_significand_calc_ii:wwnnnnnnn
13390     \fi:
13391   }
13392 \cs_new:Npn \__fp_div_significand_calc_i:wwnnnnnnn #1; #2;#3#4 #5#6#7#8 #9
13393   {
13394     1 1 #1
13395     #9 \exp_after:wN ;
13396     \__int_value:w \__int_eval:w \c_fp_Bigg_leading_shift_int
13397       + #2 - #1 * #5 - #5#60
13398     \exp_after:wN \__fp_pack_Bigg:NNNNNNw
13399     \__int_value:w \__int_eval:w \c_fp_Bigg_middle_shift_int
13400       + #3 - #1 * #6 - #70
13401     \exp_after:wN \__fp_pack_Bigg:NNNNNNw
13402     \__int_value:w \__int_eval:w \c_fp_Bigg_middle_shift_int
13403       + #4 - #1 * #7 - #80
13404     \exp_after:wN \__fp_pack_Bigg:NNNNNNw
13405     \__int_value:w \__int_eval:w \c_fp_Bigg_trailing_shift_int
13406       - #1 * #8 ;

```

```

13407      {#5}{#6}{#7}{#8}
13408    }
13409 \cs_new:Npn \__fp_div_significand_calc_ii:wwnnnnnnn #1; #2;#3#4 #5#6#7#8 #9
13410  {
13411    1 0 #1
13412    #9 \exp_after:wN ;
13413    \__int_value:w \__int_eval:w \c_fp_Bigg_leading_shift_int
13414      + #2 - #1 * #5
13415    \exp_after:wN \__fp_pack_Bigg:NNNNNNw
13416    \__int_value:w \__int_eval:w \c_fp_Bigg_middle_shift_int
13417      + #3 - #1 * #6
13418    \exp_after:wN \__fp_pack_Bigg:NNNNNNw
13419    \__int_value:w \__int_eval:w \c_fp_Bigg_middle_shift_int
13420      + #4 - #1 * #7
13421    \exp_after:wN \__fp_pack_Bigg:NNNNNNw
13422    \__int_value:w \__int_eval:w \c_fp_Bigg_trailing_shift_int
13423      - #1 * #8 ;
13424  {#5}{#6}{#7}{#8}
13425 }

```

(End definition for `__fp_div_significand_calc:wwnnnnnnn`, `__fp_div_significand_calc_i:wwnnnnnnn`, and `__fp_div_significand_calc_ii:wwnnnnnnn`.)

`__fp_div_significand_ii:wwn` $\langle y \rangle ; \langle B_1 \rangle ; \{ \langle B_2 \rangle \} \{ \langle B_3 \rangle \} \{ \langle B_4 \rangle \} \{ \langle Z_1 \rangle \}$
 $\{ \langle Z_2 \rangle \} \{ \langle Z_3 \rangle \} \{ \langle Z_4 \rangle \} \langle \text{continuations} \rangle \langle \text{sign} \rangle$

Compute Q_B by evaluating $\langle B_1 \rangle \langle B_2 \rangle /y - 1$. The result will be output to the left, in an `__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 .

```

13426 \cs_new:Npn \__fp_div_significand_ii:wwn #1; #2;#3
13427  {
13428    \exp_after:wN \__fp_div_significand_pack:NNN
13429    \__int_value:w \__int_eval:w
13430      \exp_after:wN \__fp_div_significand_calc:wwnnnnnnn
13431      \__int_value:w \__int_eval:w 999999 + #2 #3 0 / #1 ; #2 #3 ;
13432 }

```

(End definition for `__fp_div_significand_ii:wwn`.)

`__fp_div_significand_iii:wwnnnnn` $\langle y \rangle ; \langle E_1 \rangle ; \{ \langle E_2 \rangle \} \{ \langle E_3 \rangle \} \{ \langle E_4 \rangle \}$
 $\{ \langle Z_1 \rangle \} \{ \langle Z_2 \rangle \} \{ \langle Z_3 \rangle \} \{ \langle Z_4 \rangle \} \langle \text{sign} \rangle$

We compute $P \simeq 2E/Z$ by rounding $2E_1 E_2 / Z_1 Z_2$. Note the first 0, which multiplies Q_D by 10: we will 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 .

```

13433 \cs_new:Npn \__fp_div_significand_iii:wwnnnnn #1; #2;#3#4#5 #6#7
13434  {
13435    0
13436    \exp_after:wN \__fp_div_significand_iv:wwnnnnnnn
13437    \__int_value:w \__int_eval:w ( 2 * #2 #3) / #6 #7 ; % <- P
13438      #2 ; {#3} {#4} {#5}
13439      {#6} {#7}
13440 }

```

(End definition for `_fp_div_significand_iv:wwnnnnnn`.)

```
\_fp_div_significand_iv:wwnnnnnn
\_\_fp\_div\_significand\_v:Nw
\_\_fp\_div\_significand\_vi:Nw
```

`__fp_div_significand_iv:wwnnnnnn` $\langle P \rangle ; \langle E_1 \rangle ; \{\langle E_2 \rangle\} \{\langle E_3 \rangle\} \{\langle E_4 \rangle\} \{\langle Z_1 \rangle\} \{\langle Z_2 \rangle\} \{\langle Z_3 \rangle\} \{\langle Z_4 \rangle\} \langle sign \rangle$

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 $\langle rounding \rangle$ digit. This $\langle rounding \rangle$ 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: $T = 0$.

```
13441 \cs_new:Npn \_fp_div_significand_iv:wwnnnnnnn #1; #2;#3#4#5 #6#7#8#9
13442 {
13443     + 5 * #1
13444     \exp_after:wN \_fp_div_significand_vi:Nw
13445     \_int_value:w \_int_eval:w -20 + 2*#2#3 - #1*#6#7 +
13446         \exp_after:wN \_fp_div_significand_v:NN
13447         \_int_value:w \_int_eval:w 199980 + 2*#4 - #1*#8 +
13448             \exp_after:wN \_fp_div_significand_v:NN
13449             \_int_value:w \_int_eval:w 200000 + 2*#5 - #1*#9 ;
13450 }
13451 \cs_new:Npn \_fp_div_significand_v:NN #1#2 { #1#2 \_int_eval_end: + }
13452 \cs_new:Npn \_fp_div_significand_vi:Nw #1#2;
13453 {
13454     \if_meaning:w 0 #1
13455         \if_int_compare:w \_int_eval:w #2 > 0 + 1 \fi:
13456     \else:
13457         \if_meaning:w - #1 - \else: + \fi: 1
13458     \fi:
13459 ;
13460 }
```

(End definition for `_fp_div_significand_iv:wwnnnnnn`, `_fp_div_significand_v:Nw`, and `_fp_div_significand_vi:Nw`.)

`_fp_div_significand_pack:NNN` At this stage, we are in the following situation: T_EX 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, ε = 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).

```
13461 \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`

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 \tilde{Q}_A (the tilde denoting the contribution from the other Q_i) is at most 99999, and $10^6 + \tilde{Q}_A = 10 \dots$.

It is now time to round. This depends on how many digits the final result will have.

```
13462 \cs_new:Npn \_fp_div_significand_test_o:w 10 #1
13463 {
13464     \if_meaning:w 0 #1
13465         \exp_after:wN \_fp_div_significand_small_o:wwwNNNNwN
13466     \else:
13467         \exp_after:wN \_fp_div_significand_large_o:wwwNNNNwN
13468     \fi:
13469     #1
13470 }
```

(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:NNNNNw` and `_fp_basics_pack_high:NNNNNw`. We finally get to use the ⟨final sign⟩ which has been sitting there for a while.

```
13471 \cs_new:Npn \_fp_div_significand_small_o:wwwNNNNwN
13472     0 #1; #2; #3; #4#5#6#7#8; #9
13473 {
13474     \exp_after:wN \_fp_basics_pack_high:NNNNNw
13475     \_int_value:w \_int_eval:w 1 #1#
13476         \exp_after:wN \_fp_basics_pack_low:NNNNNw
13477         \_int_value:w \_int_eval:w 1 #3#4#5#6#7
13478         + \_fp_round:NNN #9 #7 #8
13479         \exp_after:wN ;
13480 }
```

(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.

```
13481 \cs_new:Npn \_fp_div_significand_large_o:wwwNNNNwN
13482     #1; #2; #3; #4#5#6#7#8; #9
13483 {
13484     +
13485     \exp_after:wN \_fp_basics_pack_weird_high:NNNNNNNNw
13486     \_int_value:w \_int_eval:w 1 #1 #2
13487     \exp_after:wN \_fp_basics_pack_weird_low:NNNNw
13488     \_int_value:w \_int_eval:w 1 #3 #4 #5 #6 +
13489     \exp_after:wN \_fp_round:NNN
13490     \exp_after:wN #9
13491     \exp_after:wN #6
13492     \_int_value:w \_fp_round_digit:Nw #7 #8 ;
13493     \exp_after:wN ;
13494 }
```

(End definition for `_fp_div_significand_large_o:wwwNNNNwN`.)

26.4 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.

```
13495 \cs_new:Npn \_fp_sqrt_o:w #1 \s_fp \_fp_chk:w #2#3#4; @
13496 {
13497     \if_meaning:w 0 #2 \_fp_case_return_same_o:w \fi:
13498     \if_meaning:w 2 #3
13499         \_fp_case_use:nw { \_fp_invalid_operation_o:nw { sqrt } }
13500     \fi:
13501     \if_meaning:w 1 #2 \else: \_fp_case_return_same_o:w \fi:
13502     \_fp_sqrt_npos_o:w
13503     \s_fp \_fp_chk:w #2 #3 #4;
13504 }
```

(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 $17782794 \simeq 10^{7.25}$.

```
13505 \cs_new:Npn \_fp_sqrt_npos_o:w \s_fp \_fp_chk:w 1 0 #1#2#3#4#5;
13506 {
13507     \exp_after:wN \_fp_sanitize:Nw
13508     \exp_after:wN 0
13509     \_int_value:w \_int_eval:w
13510     \if_int_odd:w #1 \exp_stop_f:
13511         \exp_after:wN \_fp_sqrt_npos_auxi_o:wwnnN
```

```

13512      \fi:
13513      #1 / 2
13514      \_\_fp_sqrt_Newton_o:wwn 56234133; 0; {\#2\#3} {\#4\#5} 0
13515    }
13516 \cs_new:Npn \_\_fp_sqrt_npos_auxi_o:wwnnN #1 / 2 #2; 0; #3#4#5
13517  {
13518    ( #1 + 1 ) / 2
13519    \_\_fp_pack_eight:wNNNNNNNNN
13520    \_\_fp_sqrt_npos_auxii_o:wNNNNNNNN
13521    ;
13522    0 #3 #4
13523  }
13524 \cs_new:Npn \_\_fp_sqrt_npos_auxii_o:wNNNNNNNNN #1; #2#3#4#5#6#7#8#9
13525  { \_\_fp_sqrt_Newton_o:wwn 17782794; 0; {#1} {\#2\#3\#4\#5\#6\#7\#8\#9} }

```

(End definition for $__fp_sqrt_npos_o:w$, $__fp_sqrt_npos_auxi_o:wwnnN$, and $__fp_sqrt_npos_auxii_o:wNNNNNNNNN$.)

$__fp_sqrt_Newton_o:wwn$ 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

$$x' = \left[\frac{x + [10^8 a_1/x]}{2} \right] \geq \frac{x + 10^8 a_1/x - 1/2 + 1/(2x)}{2} \geq \sqrt{10^8 a_1} - \frac{1}{4} + \frac{1}{4x}.$$

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:wwn` 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 $[10^7, 10^8]$.

```

13526 \cs_new:Npn \_fp_sqrt_Newton_o:wwn #1; #2; #3
13527 {
13528     \if_int_compare:w #1 = #2 \exp_stop_f:
13529         \exp_after:wN \_fp_sqrt_auxi_o:NNNNwnnN
13530         \int_value:w \int_eval:w 9999 9999 +
13531             \exp_after:wN \_fp_use_none_until_s:w
13532     \fi:
13533     \exp_after:wN \_fp_sqrt_Newton_o:wwn
13534     \int_value:w \int_eval:w (#1 + #3 * 1 0000 0000 / #1) / 2 ;
13535     #1; {#3}
13536 }
```

(End definition for `_fp_sqrt_Newton_o:wwn`.)

`_fp_sqrt_auxi_o:NNNNwnnN`

This function is followed by $10^8 + x - 1$, which has 9 digits starting with 1, then ; $\{a_1\} \{a_2\} \langle a' \rangle$. 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` will be 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, $y < 1$.

```

13537 \cs_new:Npn \_fp_sqrt_auxi_o:NNNNwnnN 1 #1#2#3#4#5;
13538 {
13539     \_fp_sqrt_auxii_o:NnnnnnnnnN
13540     \_fp_sqrt_auxiii_o:wnnnnnnnn
13541     {#1#2#3#4} {#5} {2499} {9988} {7500}
13542 }
```

(End definition for `_fp_sqrt_auxi_o:NNNNwnnN`.)

`_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)

$$10^{4j}z = \left[\left(\lfloor 10^{4j}(a - y^2) \rfloor - 257 \right) \cdot (0.5 \cdot 10^8) \middle/ \lfloor 10^8 y + 1 \rfloor \right].$$

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 T_EX'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 $[b/c] \leq b/c + 1/2$, 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 $-#4*#4 - 2*#3*#5 - 2*#2*#6$ which may be as low as $-5 \cdot 10^8$, we need to use the `pack_big` functions, and the `big` shifts.

```

13543 \cs_new:Npn \__fp_sqrt_auxii_o:NnnnnnnN #1 #2#3#4#5#6 #7#8#9
13544 {
13545   \exp_after:wN #1
13546   \__int_value:w \__int_eval:w \c_fp_big_leading_shift_int
13547     + #7 - #2 * #2
13548   \exp_after:wN \__fp_pack_big:NNNNNNw
13549   \__int_value:w \__int_eval:w \c_fp_big_middle_shift_int
13550     - 2 * #2 * #3
13551   \exp_after:wN \__fp_pack_big:NNNNNNw
13552   \__int_value:w \__int_eval:w \c_fp_big_middle_shift_int
13553     + #8 - #3 * #3 - 2 * #2 * #4
13554   \exp_after:wN \__fp_pack_big:NNNNNNw
13555   \__int_value:w \__int_eval:w \c_fp_big_middle_shift_int
13556     - 2 * #3 * #4 - 2 * #2 * #5
13557   \exp_after:wN \__fp_pack_big:NNNNNNw
13558   \__int_value:w \__int_eval:w \c_fp_big_middle_shift_int
13559     + #9 000 0000 - #4 * #4 - 2 * #3 * #5 - 2 * #2 * #6
13560   \exp_after:wN \__fp_pack_big:NNNNNNw
13561   \__int_value:w \__int_eval:w \c_fp_big_middle_shift_int
13562     - 2 * #4 * #5 - 2 * #3 * #6
13563   \exp_after:wN \__fp_pack_big:NNNNNNw
13564   \__int_value:w \__int_eval:w \c_fp_big_middle_shift_int
13565     - #5 * #5 - 2 * #4 * #6
13566   \exp_after:wN \__fp_pack_big:NNNNNNw
13567   \__int_value:w \__int_eval:w
13568     \c_fp_big_middle_shift_int
13569     - 2 * #5 * #6
13570   \exp_after:wN \__fp_pack_big:NNNNNNw
13571   \__int_value:w \__int_eval:w
13572     \c_fp_big_trailing_shift_int
13573     - #6 * #6 ;
13574 %
13575   - 257 ) * 5000 0000 / (#2#3 + 1) + 10 0000 0000 ;
13576   {#2}{#3}{#4}{#5}{#6} {#7}{#8}#9
13577 }
```

(End definition for `__fp_sqrt_auxii_o:NnnnnnnN`.)

```

\fp_sqrt_auxiii_o:wmnnnnnn
\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_auxiii_o:NnnnnnnnN`, which completes the expression

$$10^{4j}z = \left[(\lfloor 10^{4j}(a - y^2) \rfloor - 257) \cdot (0.5 \cdot 10^8) \Big/ \lfloor 10^8y + 1 \rfloor \right]$$

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

$$10^{4j}z \geq \frac{(20000 - 257)(0.5 \cdot 10^8)}{\lfloor 10^8y + 1 \rfloor} \geq (20000 - 257) \cdot 0.5 > 0.$$

```

13578 \cs_new:Npn \fp_sqrt_auxiii_o:wnnnnnnnn
13579   #1; #2#3#4#5#6#7#8#9
13580   {
13581     \if_int_compare:w #1 > 1 \exp_stop_f:
13582       \exp_after:wN \fp_sqrt_auxiv_o:NNNNNw
13583       \int_value:w \int_eval:w (#1#2 %)
13584     \else:
13585       \if_int_compare:w #1#2 > 1 \exp_stop_f:
13586         \exp_after:wN \fp_sqrt_auxv_o:NNNNNw
13587         \int_value:w \int_eval:w (#1#2#3 %)
13588       \else:
13589         \if_int_compare:w #1#2#3 > 1 \exp_stop_f:
13590           \exp_after:wN \fp_sqrt_auxvi_o:NNNNNw
13591           \int_value:w \int_eval:w (#1#2#3#4 %)
13592         \else:
13593           \exp_after:wN \fp_sqrt_auxvii_o:NNNNNw
13594           \int_value:w \int_eval:w (#1#2#3#4#5 %)
13595         \fi:
13596       \fi:
13597     \fi:
13598   }
13599 \cs_new:Npn \fp_sqrt_auxiv_o:NNNNNw 1#1#2#3#4#5#6;
13600   { \fp_sqrt_auxviii_o:nnnnnnnn {#1#2#3#4#5#6} {00000000} }
13601 \cs_new:Npn \fp_sqrt_auxv_o:NNNNNw 1#1#2#3#4#5#6;
13602   { \fp_sqrt_auxviii_o:nnnnnnnn {000#1#2#3#4#5} {#60000} }
13603 \cs_new:Npn \fp_sqrt_auxvi_o:NNNNNw 1#1#2#3#4#5#6;
13604   { \fp_sqrt_auxviii_o:nnnnnnnn {0000000#1} {#2#3#4#5#6} }
13605 \cs_new:Npn \fp_sqrt_auxvii_o:NNNNNw 1#1#2#3#4#5#6;
13606   {
13607     \if_int_compare:w #1#2 = 0 \exp_stop_f:
13608       \exp_after:wN \fp_sqrt_auxxx_o:Nnnnnnnn
13609     \fi:

```

```

13610      \__fp_sqrt_auxviii_o:nnnnnnnn {00000000} {000#1#2#3#4#5}
13611  }

```

(End definition for $__fp_sqrt_auxviii_o:wnnnnnnn$ and others.)

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 $y'^2 = (y + z)^2$.

```

13612 \cs_new:Npn \__fp_sqrt_auxviii_o:nnnnnnnn #1#2 #3#4#5#6#7
13613  {
13614      \exp_after:wN \__fp_sqrt_auxix_o:wnwnw
13615      \int_value:w \int_eval:w #3
13616      \exp_after:wN \__fp_basics_pack_low>NNNNW
13617      \int_value:w \int_eval:w #1 + 1#4#5
13618      \exp_after:wN \__fp_basics_pack_low>NNNNW
13619      \int_value:w \int_eval:w #2 + 1#6#7 ;
13620  }
13621 \cs_new:Npn \__fp_sqrt_auxix_o:wnwnw #1; #2#3; #4#5;
13622  {
13623      \__fp_sqrt_auxii_o:NnnnnnnnN
13624      \__fp_sqrt_auxiii_o:wnnnnnnnn {#1}{#2}{#3}{#4}{#5}
13625  }

```

(End definition for $__fp_sqrt_auxviii_o:nnnnnnn$ and $__fp_sqrt_auxix_o:wnwnw$.)

At this stage, $j = 6$ and $10^{24}z < 10^7$, hence

$$10^7 + 1/2 > 10^{24}z + 1/2 \geq (10^{24}(a - y^2) - 258) \cdot (0.5 \cdot 10^8) / (10^8y + 1),$$

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 $a \leq 1 - 10^{-16}$.

```

13626 \cs_new:Npn \__fp_sqrt_auxx_o:Nnnnnnnn #1#2#3 #4#5#6#7#8
13627  {
13628      \exp_after:wN \__fp_sqrt_auxxi_o:wnnnN
13629      \int_value:w \int_eval:w
13630      (#8 + 2499) / 5000 * 5000 ;
13631      {#4} {#5} {#6} {#7} ;
13632  }
13633 \cs_new:Npn \__fp_sqrt_auxxi_o:wnnnN #1; #2; #3#4#5
13634  {
13635      \__fp_sqrt_auxii_o:NnnnnnnnN

```

```

13636     \_fp_sqrt_auxxii_o:nnnnnnnnw
13637     #2 {#1}
13638     {#3} { #4 + 1 } #5
13639   }

```

(End definition for `_fp_sqrt_auxx_o:Nnnnnnnn` and `_fp_sqrt_auxxi_o:wwnnN`.)

`_fp_sqrt_auxxii_o:nnnnnnnnw`
`_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.

```

13640 \cs_new:Npn \_fp_sqrt_auxxii_o:nnnnnnnnw 0; #1#2#3#4#5#6#7#8 #9;
13641   {
13642     \if_int_compare:w #1#2 > 0 \exp_stop_f:
13643     \if_int_compare:w #1#2 = 1 \exp_stop_f:
13644       \if_int_compare:w #3#4 = 0 \exp_stop_f:
13645         \if_int_compare:w #5#6 = 0 \exp_stop_f:
13646           \if_int_compare:w #7#8 = 0 \exp_stop_f:
13647             \_fp_sqrt_auxxiii_o:w
13648           \fi:
13649         \fi:
13650       \fi:
13651     \fi:
13652     \exp_after:wN \_fp_sqrt_auxxiv_o:wnnnnnnnN
13653     \int_value:w 9998
13654   \else:
13655     \exp_after:wN \_fp_sqrt_auxxiv_o:wnnnnnnnN
13656     \int_value:w 10000
13657   \fi:
13658   ;
13659 }
13660 \cs_new:Npn \_fp_sqrt_auxxiii_o:w \fi: \fi: \fi: \fi: \fi: #1 \fi: ;
13661   {
13662     \fi: \fi: \fi: \fi: \fi:
13663     \_fp_sqrt_auxxiv_o:wnnnnnnnN 9999 ;
13664   }

```

(End definition for `_fp_sqrt_auxxii_o:nnnnnnnnw` 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.

```

13665 \cs_new:Npn \__fp_sqrt_auxxiv_o:wnnnnnnnnN #1; #2#3#4#5#6 #7#8#9
13666 {
13667     \exp_after:wN \__fp_basics_pack_high:NNNNNw
13668     \__int_value:w \__int_eval:w 1 0000 0000 + #2#3
13669     \exp_after:wN \__fp_basics_pack_low:NNNNNw
13670     \__int_value:w \__int_eval:w 1 0000 0000
13671     + #4#5
13672     \if_int_compare:w #6 > #1 \exp_stop_f: + 1 \fi:
13673     + \exp_after:wN \__fp_round:NNN
13674         \exp_after:wN 0
13675         \exp_after:wN 0
13676         \__int_value:w
13677             \exp_after:wN \use_i:nn
13678             \exp_after:wN \__fp_round_digit:Nw
13679             \__int_value:w \__int_eval:w #6 + 19999 - #1 ;
13680     \exp_after:wN ;
13681 }

```

(End definition for `__fp_sqrt_auxxiv_o:wnnnnnnnnN`.)

26.5 About the sign

`__fp_sign_o:w` Find the sign of the floating point: `nan`, `+0`, `-0`, `+1` or `-1`.

```

\__fp_sign_aux_o:w \cs_new:Npn \__fp_sign_o:w ? \s__fp \__fp_chk:w #1#2; @
13682 {
13683     \if_case:w #1 \exp_stop_f:
13684         \__fp_case_return_same_o:w
13685     \or: \exp_after:wN \__fp_sign_aux_o:w
13686     \or: \exp_after:wN \__fp_sign_aux_o:w
13687     \else: \__fp_case_return_same_o:w
13688     \fi:
13689     \s__fp \__fp_chk:w #1 #2;
13690 }
13691 \cs_new:Npn \__fp_sign_aux_o:w \s__fp \__fp_chk:w #1 #2 #3 ;
13692 { \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`.

```

13693 \cs_new:Npn \__fp_set_sign_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
13694 {
13695     \exp_after:wN \__fp_exp_after_o:w
13696     \exp_after:wN \s__fp
13697     \exp_after:wN \__fp_chk:w
13698     \exp_after:wN #2
13699     \__int_value:w
13700         \if_case:w #3 \exp_stop_f: #1 \or: 1 \or: 0 \fi: \exp_stop_f:
13701         #4;
13702 }

```

(End definition for `__fp_set_sign_o:w`.)

13704 ⟨/initex | package⟩

27 l3fp-extended implementation

```
13705  {*initex | package}
13706  <@@=fp>
```

27.1 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

```
{⟨a1⟩} {⟨a2⟩} {⟨a3⟩} {⟨a4⟩} {⟨a5⟩} {⟨a6⟩} ;
```

where each ⟨a_i⟩ is exactly 4 digits (ranging from 0000 to 9999), except ⟨a₁⟩, 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 $a = \sum_{i=1}^6 a_i \cdot 10^{-4i}$.

Most functions we define here have the form They perform the ⟨calculation⟩ on the two ⟨operands⟩, then feed the result (6 brace groups followed by a semicolon) to the ⟨continuation⟩, responsible for the next step of the calculation. Some functions only accept an N-type ⟨continuation⟩. This allows constructions such as

```
\_fp_fixed_add:wn ⟨X1⟩ ; ⟨X2⟩ ;
\_fp_fixed_mul:wn ⟨X3⟩ ;
\_fp_fixed_add:wn ⟨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.

27.2 Helpers for numbers with extended precision

\c_fp_one_fixed_t1 The fixed-point number 1, used in l3fp-expo.

```
13707  \tl_const:Nn \c\_fp\_one\_fixed_t1
13708  { {10000} {0000} {0000} {0000} {0000} {0000} ; }
```

(End definition for \c_fp_one_fixed_t1.)

_fp_fixed_continue:wn This function simply calls the next function.

```
13709  \cs_new:Npn \_fp_fixed_continue:wn #1; #2 { #2 #1; }
```

(End definition for _fp_fixed_continue:wn.)

_fp_fixed_add_one:wN This function adds 1 to the fixed point $\langle a \rangle$, by changing a_1 to $10000 + a_1$, then calls the $\langle \text{continuation} \rangle$. This requires $a_1 + 10000 < 2^{31}$.

```
13710 \cs_new:Npn \_fp_fixed_add_one:wN #1#2; #3
13711 {
13712     \exp_after:wN #3 \exp_after:wN
13713         { \_int_value:w \_int_eval:w \c_fp_myriad_int + #1 } #2 ;
13714 }
```

(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 $[0, 5 \cdot 10^8 - 1]$.

```
13715 \cs_new:Npn \_fp_fixed_div_myriad:wn #1#2#3#4#5#6;
13716 {
13717     \exp_after:wN \_fp_fixed_mul_after:wwn
13718     \_int_value:w \_int_eval:w \c_fp_leading_shift_int
13719         \exp_after:wN \_fp_pack:NNNNw
13720         \_int_value:w \_int_eval:w \c_fp_trailing_shift_int
13721             + #1 ; {#2}{#3}{#4}{#5};
13722 }
```

(End definition for _fp_fixed_div_myriad:wn.)

_fp_fixed_mul_after:wwn The fixed point operations which involve multiplication end by calling this auxiliary. It braces the last block of digits, and places the $\langle \text{continuation} \rangle$ #3 in front.

```
13723 \cs_new:Npn \_fp_fixed_mul_after:wwn #1; #2; #3 { #3 {#1} #2; }
```

(End definition for _fp_fixed_mul_after:wwn.)

27.3 Multiplying a fixed point number by a short one

_fp_fixed_mul_short:wwn 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 $\langle \text{continuation} \rangle \{ \langle c_1 \rangle \} \dots \{ \langle c_6 \rangle \}$; 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: a second operand of {0001}-{0000}-{0000} will leave the first operand unchanged (rather than dividing it by 10^4 , as _fp_fixed_mul:wwn would).

```
13724 \cs_new:Npn \_fp_fixed_mul_short:wwn #1#2#3#4#5#6; #7#8#9;
13725 {
13726     \exp_after:wN \_fp_fixed_mul_after:wwn
13727     \_int_value:w \_int_eval:w \c_fp_leading_shift_int
13728         + #1*#7
13729         \exp_after:wN \_fp_pack:NNNNw
13730         \_int_value:w \_int_eval:w \c_fp_middle_shift_int
13731             + #1*#8 + #2*#7
13732             \exp_after:wN \_fp_pack:NNNNw
13733             \_int_value:w \_int_eval:w \c_fp_middle_shift_int
13734                 + #1*#9 + #2*#8 + #3*#7
13735                 \exp_after:wN \_fp_pack:NNNNw
13736                 \_int_value:w \_int_eval:w \c_fp_middle_shift_int
13737                     + #2*#9 + #3*#8 + #4*#7
```

```

13738     \exp_after:wN \_fp_pack>NNNNw
13739     \_int_value:w \_int_eval:w \c_fp_middle_shift_int
13740         + #3*#9 + #4*#8 + #5*#7
13741     \exp_after:wN \_fp_pack>NNNNw
13742     \_int_value:w \_int_eval:w \c_fp_trailing_shift_int
13743         + #4*#9 + #5*#8 + #6*#7
13744         + ( #5*#9 + #6*#8 + #6*#9 / \c_fp_myriad_int )
13745     / \c_fp_myriad_int ; ;
13746 }

```

(End definition for `_fp_fixed_mul_short:wnn`.)

27.4 Dividing a fixed point number by a small integer

Divides the fixed point number $\langle a \rangle$ by the (small) integer $0 < \langle n \rangle < 10^4$ and feeds the result to the *(continuation)*. There is no bound on a_i .

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 will have 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.

```

13747 \cs_new:Npn \fp_fixed_div_int:wwN #1#2#3#4#5#6 ; #7 ; #8
13748 {
13749     \exp_after:wN \fp_fixed_div_int_after:Nw
13750     \exp_after:wN #8
13751     \_int_value:w \_int_eval:w - 1
13752         \fp_fixed_div_int:wnN
13753         #1; {#7} \fp_fixed_div_int_auxi:wnn
13754         #2; {#7} \fp_fixed_div_int_auxi:wnn
13755         #3; {#7} \fp_fixed_div_int_auxi:wnn
13756         #4; {#7} \fp_fixed_div_int_auxi:wnn
13757         #5; {#7} \fp_fixed_div_int_auxi:wnn
13758         #6; {#7} \fp_fixed_div_int_auxii:wnn ;
13759 }

```

```

13760 \cs_new:Npn \__fp_fixed_div_int:wnN #1; #2 #3
13761 {
13762     \exp_after:wN #3
13763     \_int_value:w \_int_eval:w #1 / #2 - 1 ;
13764     {#2}
13765     {#1}
13766 }
13767 \cs_new:Npn \__fp_fixed_div_int_auxi:wnn #1; #2 #3
13768 {
13769     + #1
13770     \exp_after:wN \__fp_fixed_div_int_pack:Nw
13771     \_int_value:w \_int_eval:w 9999
13772     \exp_after:wN \__fp_fixed_div_int:wnN
13773     \_int_value:w \_int_eval:w #3 - #1*#2 \_int_eval_end:
13774 }
13775 \cs_new:Npn \__fp_fixed_div_int_auxii:wnn #1; #2 #3 { + #1 + 2 ; }
13776 \cs_new:Npn \__fp_fixed_div_int_pack:Nw #1 #2; { + #1; {#2} }
13777 \cs_new:Npn \__fp_fixed_div_int_after:Nw #1 #2; { #1 {#2} }

(End definition for \__fp_fixed_div_int:wwN and others.)

```

27.5 Adding and subtracting fixed points

`__fp_fixed_add:wwn`
`__fp_fixed_sub:wwn`
`__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 *(continuation)*. 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 *(continuation)* as arguments. After going down through the various level, we go back up, packing digits and bringing the *(continuation)* (#8, then #7) from the end of the argument list to its start.

```

13778 \cs_new:Npn \__fp_fixed_add:wwn { \__fp_fixed_add:Nnnnnwnn + }
13779 \cs_new:Npn \__fp_fixed_sub:wwn { \__fp_fixed_sub:Nnnnnwnn - }
13780 \cs_new:Npn \__fp_fixed_add:Nnnnnwnn #1 #2#3#4#5 #6; #7#8
13781 {
13782     \exp_after:wN \__fp_fixed_add_after:NNNNNwn
13783     \_int_value:w \_int_eval:w 9 9999 9998 + #2#3 #1 #7#8
13784     \exp_after:wN \__fp_fixed_add_pack:NNNNNwn
13785     \_int_value:w \_int_eval:w 1 9999 9998 + #4#5
13786     \__fp_fixed_add:nnNnnnwn #6 #1
13787 }
13788 \cs_new:Npn \__fp_fixed_add:nnNnnnwn #1#2 #3 #4#5 #6#7 ; #8
13789 {
13790     #3 #4#5
13791     \exp_after:wN \__fp_fixed_add_pack:NNNNNwn
13792     \_int_value:w \_int_eval:w 2 0000 0000 #3 #6#7 + #1#2 ; {#8} ;
13793 }
13794 \cs_new:Npn \__fp_fixed_add_pack:NNNNNwn #1 #2#3#4#5 #6; #7
13795     { + #1 ; {#7} {#2#3#4#5} {#6} }
13796 \cs_new:Npn \__fp_fixed_add_after:NNNNNwn 1 #1 #2#3#4#5 #6; #7
13797     { #7 {#1#2#3#4#5} {#6} }

(End definition for \__fp_fixed_add:wwn and others.)

```

27.6 Multiplying fixed points

```
\_\_fp\_fixed\_mul:wwn
\_\_fp\_fixed\_mul:nnnnnnnw
```

Computes $a \times b$ and feeds the result to $\langle\text{continuation}\rangle$. 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 $\langle\text{continuation}\rangle$ as arguments. It writes the end of the expression, including the right parenthesis and the denominator of the fraction. The $\langle\text{continuation}\rangle$ is finally placed in front of the 6 brace groups by `__fp_fixed_mul_after:wwn`.

```
13798 \cs_new:Npn \_\_fp\_fixed\_mul:wwn #1#2#3#4 #5; #6#7#8#9
13799 {
13800   \exp_after:wN \_\_fp\_fixed\_mul\_after:wwn
13801   \_\_int\_value:w \_\_int\_eval:w \c__fp\_leading\_shift\_int
13802     \exp_after:wN \_\_fp\_pack:NNNNNw
13803     \_\_int\_value:w \_\_int\_eval:w \c__fp\_middle\_shift\_int
13804       + #1*#6
13805     \exp_after:wN \_\_fp\_pack:NNNNNw
13806     \_\_int\_value:w \_\_int\_eval:w \c__fp\_middle\_shift\_int
13807       + #1*#7 + #2*#6
13808     \exp_after:wN \_\_fp\_pack:NNNNNw
13809     \_\_int\_value:w \_\_int\_eval:w \c__fp\_middle\_shift\_int
13810       + #1*#8 + #2*#7 + #3*#6
13811     \exp_after:wN \_\_fp\_pack:NNNNNw
13812     \_\_int\_value:w \_\_int\_eval:w \c__fp\_middle\_shift\_int
13813       + #1*#9 + #2*#8 + #3*#7 + #4*#6
13814     \exp_after:wN \_\_fp\_pack:NNNNNw
13815     \_\_int\_value:w \_\_int\_eval:w \c__fp\_trailing\_shift\_int
13816       + #2*#9 + #3*#8 + #4*#7
13817       + ( #3*#9 + #4*#8
13818         + \_\_fp\_fixed\_mul:nnnnnnnw #5 {#6}{#7} {#1}{#2}
13819       )
13820 \cs_new:Npn \_\_fp\_fixed\_mul:nnnnnnnw #1#2 #3#4 #5#6 #7#8 ;
13821 {
13822   #1*#4 + #2*#3 + #5*#8 + #6*#7 ) / \c__fp_myriad_int
```

```

13823     + #1*#3 + #5*#7 ; ;
13824 }
```

(End definition for `_fp_fixed_mul:wwn` and `_fp_fixed_mul:nnnnnnnw`.)

27.7 Combining product and sum of fixed points

`_fp_fixed_mul_add:wwn` Compute $a \times b + c$, $c - a \times b$, and $1 - a \times b$ and feed the result to the *{continuation}*. 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 will 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 `+ c5 c6 ; {continuation}`. The `+ c5 c6` piece, which is omitted for `_fp_fixed_one_minus_mul:wwn`, will be taken in the integer expression for the 10^{-24} level.

```

13825 \cs_new:Npn \_fp_fixed_mul_add:wwn #1; #2; #3#4#5#6#7#8;
13826 {
13827   \exp_after:wN \_fp_fixed_mul_after:wwn
13828   \_int_value:w \_int_eval:w \c_fp_big_leading_shift_int
13829   \exp_after:wN \_fp_pack_big:NNNNNNw
13830   \_int_value:w \_int_eval:w \c_fp_big_middle_shift_int + #3 #4
13831   \_fp_fixed_mul_add:Nwnnnnnn +
13832   + #5 #6 ; #2 ; #1 ; #2 ; +
13833   + #7 #8 ; ;
13834 }
13835 \cs_new:Npn \_fp_fixed_mul_sub_back:wwn #1; #2; #3#4#5#6#7#8;
13836 {
13837   \exp_after:wN \_fp_fixed_mul_after:wwn
13838   \_int_value:w \_int_eval:w \c_fp_big_leading_shift_int
13839   \exp_after:wN \_fp_pack_big:NNNNNNw
13840   \_int_value:w \_int_eval:w \c_fp_big_middle_shift_int + #3 #4
13841   \_fp_fixed_mul_add:Nwnnnnnn -
13842   + #5 #6 ; #2 ; #1 ; #2 ; -
13843   + #7 #8 ; ;
```

```

13844      }
13845  \cs_new:Npn \__fp_fixed_one_minus_mul:wwn #1; #2;
13846  {
13847      \exp_after:wN \__fp_fixed_mul_after:wwn
13848      \__int_value:w \__int_eval:w \c_fp_big_leading_shift_int
13849          \exp_after:wN \__fp_pack_big:NNNNNNw
13850          \__int_value:w \__int_eval:w \c_fp_big_middle_shift_int + 1 0000 0000
13851          \__fp_fixed_mul_add:Nwnnwnnn -
13852          ; #2 ; #1 ; #2 ; -
13853          ;
13854      }

```

(End definition for `__fp_fixed_mul_add:wwn`, `__fp_fixed_mul_sub_back:wwn`, and `__fp_fixed_minus_mul:wwn`.)

`\fp fixed mul add:Nwnnwnnn`

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 $a \cdot b$ products use the sign #1. Note that #2 is empty for `__fp_fixed_one_minus_mul:wwn`. 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.

```

13855  \cs_new:Npn \__fp_fixed_mul_add:Nwnnwnnn #1 #2; #3#4#5#6; #7#8#9
13856  {
13857      #1 #7*#3
13858      \exp_after:wN \__fp_pack_big:NNNNNNw
13859      \__int_value:w \__int_eval:w \c_fp_big_middle_shift_int
13860          #1 #7*#4 #1 #8*#3
13861          \exp_after:wN \__fp_pack_big:NNNNNNw
13862          \__int_value:w \__int_eval:w \c_fp_big_middle_shift_int
13863              #1 #7*#5 #1 #8*#4 #1 #9*#3 #2
13864              \exp_after:wN \__fp_pack_big:NNNNNNw
13865              \__int_value:w \__int_eval:w \c_fp_big_middle_shift_int
13866                  #1 \__fp_fixed_mul_add:nnnnwnnnn {#7}{#8}{#9}
13867  }

```

(End definition for `__fp_fixed_mul_add:Nwnnwnnn`.)

`\fp fixed mul add:nnnnwnnn`

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

$$\begin{aligned} & b_1 + a_4 \cdot b_2 + a_3 \cdot b_3 + a_2 \cdot b_4 + a_1 \\ & b_2 + a_4 \cdot b_3 + a_3 \cdot b_4 + a_2. \end{aligned}$$

Obviously, those expressions make no mathematical sense: we will 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 $\langle b \rangle$.

```

13868  \cs_new:Npn \__fp_fixed_mul_add:nnnnwnnn #1#2#3#4#5; #6#7#8#9
13869  {
13870      ( #1*#9 + #2*#8 + #3*#7 + #4*#6 )
13871      \exp_after:wN \__fp_pack_big:NNNNNNw
13872      \__int_value:w \__int_eval:w \c_fp_big_trailing_shift_int
13873          \__fp_fixed_mul_add:nnnnwnnn
13874              { #6 + #4*#7 + #3*#8 + #2*#9 + #1 }

```

```

13875      { #7 + #4*#8 + #3*#9 + #2 }
13876      {#1} #5;
13877      {#6}
13878  }

```

(End definition for `_fp_fixed_mul_add:nnnnwnnwN`.)

`_fp_fixed_mul_add:nnnnwnnwN` 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.

```

13879  \cs_new:Npn \_fp_fixed_mul_add:nnnnwnnwN #1#2 #3#4#5; #6#7#8; #9
13880  {
13881      #9 (#4* #1 **#7)
13882      #9 (#5*#6+#4* #2 **#7+#3*#8) / \c_fp_myriad_int
13883  }

```

(End definition for `_fp_fixed_mul_add:nnnnwnnwN`.)

27.8 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` 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 will have 12 digits, hopefully starting with many zeros.

```

13884  \cs_new:Npn \_fp_ep_to_fixed:wwn #1,#2
13885  {
13886      \exp_after:wN \_fp_ep_to_fixed_auxi:www
13887      \_int_value:w \_int_eval:w 1 0000 0000 + #2 \exp_after:wN ;
13888      \exp:w \exp_end_continue_f:w
13889      \prg_replicate:nn { 4 - \int_max:nn {#1} { -32 } } { 0 } ;
13890  }
13891  \cs_new:Npn \_fp_ep_to_fixed_auxi:www 1#1; #2; #3#4#5#6#7;
13892  {
13893      \_fp_pack_eight:wNNNNNNNN
13894      \_fp_pack_twice_four:wNNNNNNNN
13895      \_fp_pack_twice_four:wNNNNNNNN
13896      \_fp_pack_twice_four:wNNNNNNNN
13897      \_fp_ep_to_fixed_auxii:nnnnnnwn ;
13898      #2 #1#3#4#5#6#7 0000 !
13899  }
13900 \cs_new:Npn \_fp_ep_to_fixed_auxii:nnnnnnwn #1#2#3#4#5#6#7; #8! #9
13901  { #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_auxi:nnnnnnnnwn`.)

`_fp_ep_to_ep:wwN` 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).

```

13902 \cs_new:Npn \_fp_ep_to_ep:wwN #1,#2#3#4#5#6#7; #8
13903 {
13904     \exp_after:wN #8
13905     \int_value:w \int_eval:w #1 + 4
13906     \exp_after:wN \use_i:nn
13907     \exp_after:wN \_fp_ep_to_ep_loop:N
13908     \int_value:w \int_eval:w 1 0000 0000 + #2 \int_eval_end:
13909     #3#4#5#6#7 ; ;
13910 }
13911 \cs_new:Npn \_fp_ep_to_ep_loop:N #1
13912 {
13913     \if_meaning:w 0 #1
13914     - 1
13915     \else:
13916         \_fp_ep_to_ep_end:www #1
13917     \fi:
13918     \_fp_ep_to_ep_loop:N
13919 }
13920 \cs_new:Npn \_fp_ep_to_ep_end:www
13921     #1 \fi: \_fp_ep_to_ep_loop:N #2; #3!
13922 {
13923     \fi:
13924     \if_meaning:w ; #1
13925     - 2 * \c_fp_max_exponent_int
13926     \_fp_ep_to_ep_zero:ww
13927     \fi:
13928     \_fp_pack_twice_four:wNNNNNNNN
13929     \_fp_pack_twice_four:wNNNNNNNN
13930     \_fp_pack_twice_four:wNNNNNNNN
13931     \_fp_use_i:ww , ;
13932     #1 #2 0000 0000 0000 0000 0000 0000 ;
13933 }
13934 \cs_new:Npn \_fp_ep_to_ep_zero:ww \fi: #1; #2; #3;
13935     { \fi: , {1000}{0000}{0000}{0000}{0000}{0000} ; }
```

(End definition for `_fp_ep_to_ep:wwN` and others.)

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].

```

13936 \cs_new:Npn \__fp_ep_compare:www #1,#2#3#4#5#6#7;
13937   { \__fp_ep_compare_aux:www {#1}{#2}{#3}{#4}{#5}; #6#7; }
13938 \cs_new:Npn \__fp_ep_compare_aux:www #1:#2:#3,#4#5#6#7#8#9;
13939   {
13940     \if_case:w
13941       \__fp_compare_npos:nwnw #1; {#3}{#4}{#5}{#6}{#7}; \exp_stop_f:
13942         \if_int_compare:w #2 = #8#9 \exp_stop_f:
13943           0
13944         \else:
13945           \if_int_compare:w #2 < #8#9 - \fi: 1
13946         \fi:
13947       \or: 1
13948     \else: -1
13949     \fi:
13950   }

```

(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].

```

13951 \cs_new:Npn \__fp_ep_mul:wwwn #1,#2; #3,#4;
13952   {
13953     \__fp_ep_to_ep:wnN #3,#4;
13954     \__fp_fixed_continue:wn
13955     {
13956       \__fp_ep_to_ep:wnN #1,#2;
13957       \__fp_ep_mul_raw:wwwn
13958     }
13959     \__fp_fixed_continue:wn
13960   }
13961 \cs_new:Npn \__fp_ep_mul_raw:wwwn #1,#2; #3,#4; #5
13962   {
13963     \__fp_fixed_mul:wnN #2; #4;
13964     { \exp_after:wnN #5 \__int_value:w \__int_eval:w #1 + #3 , }
13965   }

```

(End definition for `__fp_ep_mul:wwwn` and `__fp_ep_mul_raw:wwwn`.)

27.9 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 $\langle n_1 \rangle \dots \langle n_6 \rangle$ and the 6 blocks $\langle d_1 \rangle \dots \langle d_6 \rangle$, the condition translates to $\langle n_1 \rangle, \langle d_1 \rangle \in [1000, 9999]$.

We will first find an integer estimate $a \simeq 10^8/\langle d \rangle$ by computing

$$\begin{aligned}\alpha &= \left[\frac{10^9}{\langle d_1 \rangle + 1} \right] \\ \beta &= \left[\frac{10^9}{\langle d_1 \rangle} \right] \\ a &= 10^3\alpha + (\beta - \alpha) \cdot \left(10^3 - \left[\frac{\langle d_2 \rangle}{10} \right] \right) - 1250,\end{aligned}$$

where $\left[\frac{\bullet}{\bullet} \right]$ 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 will 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 $\left[\frac{\langle d_2 \rangle}{10} \right]$ will at most underestimate $10^{-1}(\langle d_2 \rangle + 1)$ by 0.5, as can be checked for each possible last digit of $\langle d_2 \rangle$. Then,

$$10^7\langle d \rangle a < \left(10^3\langle d_1 \rangle + \left[\frac{\langle d_2 \rangle}{10} \right] + \frac{1}{2} \right) \left(\left(10^3 - \left[\frac{\langle d_2 \rangle}{10} \right] \right) \beta + \left[\frac{\langle d_2 \rangle}{10} \right] \alpha - 1250 \right) \quad (1)$$

$$< \left(10^3\langle d_1 \rangle + \left[\frac{\langle d_2 \rangle}{10} \right] + \frac{1}{2} \right) \quad (2)$$

$$\left(\left(10^3 - \left[\frac{\langle d_2 \rangle}{10} \right] \right) \left(\frac{10^9}{\langle d_1 \rangle} + \frac{1}{2} \right) + \left[\frac{\langle d_2 \rangle}{10} \right] \left(\frac{10^9}{\langle d_1 \rangle + 1} + \frac{1}{2} \right) - 1250 \right) \quad (3)$$

$$< \left(10^3\langle d_1 \rangle + \left[\frac{\langle d_2 \rangle}{10} \right] + \frac{1}{2} \right) \left(\frac{10^{12}}{\langle d_1 \rangle} - \left[\frac{\langle d_2 \rangle}{10} \right] \frac{10^9}{\langle d_1 \rangle(\langle d_1 \rangle + 1)} - 750 \right) \quad (4)$$

We recognize a quadratic polynomial in $\left[\langle d_2 \rangle / 10 \right]$ with a negative leading coefficient: this polynomial is bounded above, according to $(\left[\langle d_2 \rangle / 10 \right] + a)(b - c[\langle d_2 \rangle / 10]) \leq (b + ca)^2 / (4c)$. 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..., 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

$$10^7 \langle d \rangle a > \left(10^3 \langle d_1 \rangle + \left[\frac{\langle d_2 \rangle}{10} \right] - \frac{1}{2} \right) \left(\frac{10^{12}}{\langle d_1 \rangle} - \left[\frac{\langle d_2 \rangle}{10} \right] \frac{10^9}{\langle d_1 \rangle (\langle d_1 \rangle + 1)} - 1750 \right)$$

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 will give 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 TeX 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 *continuation* once we are done. First normalize the inputs so that both first block lie in [1000, 9999], then call `__fp_ep_div_esti:wwwn` *(denominator)* *(numerator)*, responsible for estimating the inverse of the denominator.

```

13966 \cs_new:Npn \__fp_ep_div:wwwn #1,#2; #3,#4;
13967   {
13968     \__fp_ep_to_ep:wwN #1,#2;
13969     \__fp_fixed_continue:wn
13970     {
13971       \__fp_ep_to_ep:wwN #3,#4;
13972       \__fp_ep_div_esti:wwwn
13973     }
13974 }
```

(End definition for `__fp_ep_div:wwwn`.)

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 will 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_epsi:vn>NNNNn`, which computes $10^{-9}a / (1 - \epsilon)$, that is, $1 / (10 \langle d \rangle)$ and we finally multiply this by the numerator #8.

```

13975 \cs_new:Npn \__fp_ep_div_esti:wwwn #1,#2#3; #4,
13976   {
13977     \exp_after:wN \__fp_ep_div_estii:wwnnwwn
13978     \int_value:w \int_eval:w 10 0000 0000 / ( #2 + 1 )
13979     \exp_after:wN ;
13980     \int_value:w \int_eval:w #4 - #1 + 1 ,
13981     {#2} #3;
13982   }
13983 \cs_new:Npn \__fp_ep_div_estii:wwnnwwn #1; #2,#3#4#5; #6; #7
13984   {
```

```

13985     \exp_after:wN \_fp_ep_div_estiii:NNNNNwwn
13986     \_int_value:w \_int_eval:w 10 0000 0000 - 1750
13987     + #1 000 + (10 0000 0000 / #3 - #1) * (1000 - #4 / 10) ;
13988     {#3}{#4}#5; #6; { #7 #2, }
13989   }
13990 \cs_new:Npn \_fp_ep_div_estiii:NNNNNwwn 1#1#2#3#4#5#6; #7;
13991   {
13992     \_fp_fixed_mul_short:wwn #7; {#1}{#2#3#4#5}{#6};
13993     \_fp_ep_div_epsi:wnNNNNn {#1#2#3#4}#5#6
13994     \_fp_fixed_mul:wwn
13995   }

```

(End definition for `_fp_ep_div_esti:wwwn`, `_fp_ep_div_estii:wwnnwwn`, and `_fp_ep_div-estiii:NNNNNwwn`.)

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 `epsi` 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`.

```

13996 \cs_new:Npn \_fp_ep_div_epsi:wnNNNNn #1#2#3#4#5#6;
13997   {
13998     \exp_after:wN \_fp_ep_div_epsii:wwnNNNNn
13999     \_int_value:w \_int_eval:w 1 9998 - #2
14000     \exp_after:wN \_fp_ep_div_eps_pack:NNNNNw
14001     \_int_value:w \_int_eval:w 1 9999 9998 - #3#4
14002     \exp_after:wN \_fp_ep_div_eps_pack:NNNNNw
14003     \_int_value:w \_int_eval:w 2 0000 0000 - #5#6 ; ;
14004   }
14005 \cs_new:Npn \_fp_ep_div_eps_pack:NNNNNw #1#2#3#4#5#6;
14006   { + #1 ; {#2#3#4#5} {#6} }
14007 \cs_new:Npn \_fp_ep_div_epsi:wnNNNNn 1#1; #2; #3#4#5#6#7#8
14008   {
14009     \_fp_fixed_mul:wwn {0000}{#1}#2; {0000}{#1}#2;
14010     \_fp_fixed_add_one:wn
14011     \_fp_fixed_mul:wwn {10000} {#1} #2 ;
14012     {
14013       \_fp_fixed_mul_short:wwn {0000}{#1}#2; {#3}{#4#5#6#7}{#8000};
14014       \_fp_fixed_div_myriad:wn
14015       \_fp_fixed_mul:wwn
14016     }
14017     \_fp_fixed_add:wwn {#3}{#4#5#6#7}{#8000}{0000}{0000}{0000};
14018   }

```

(End definition for `_fp_ep_div_epsi:wnNNNNn`, `_fp_ep_div_eps_pack:NNNNNw`, and `_fp_ep-div_epsi:wwnNNNNn`.)

27.10 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 will replace 10^8 by a slightly larger number which will ensure that $r^2x \geq 10^4$. 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 $y = 10^{-4}r^2x$ (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 $x^{-1/2} = 10^{-2}ry^{-1/2}$.

```
\_\_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 10^4x (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.

```
14019 \cs_new:Npn \_\_fp\_ep\_isqrt:wwn #1,#2;
14020 {
14021     \_\_fp\_ep\_to\_ep:wwN #1,#2;
14022     \_\_fp\_ep\_isqrt\_auxi:wwn
14023 }
14024 \cs_new:Npn \_\_fp\_ep\_isqrt\_auxi:wwn #1,
14025 {
14026     \exp_after:wN \_\_fp\_ep\_isqrt\_auxii:wwnnwn
14027     \_\_int\_value:w \_\_int\_eval:w
14028     \int_if_odd:nTF {#1}
14029     { (1 - #1) / 2 , 535 , { 0 } { } }
14030     { 1 - #1 / 2 , 168 , { } { 0 } }
14031 }
14032 \cs_new:Npn \_\_fp\_ep\_isqrt\_auxii:wwnnwn #1, #2, #3#4 #5#6; #7
14033 {
14034     \_\_fp\_ep\_isqrt\_esti:wwnnwn #2, 0, #5, {#3} {#4}
14035     {#5} #6 ; { #7 #1 , }
14036 }
```

(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 $((\langle prev \rangle + 1.005 \cdot 10^8 \text{ or } 10^9) / (\langle prev \rangle \cdot x)) / 2$, 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 estiii. 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_epsii:wwN`, which computes $1/\sqrt{y}$, and we finally multiply the result by r .

```

14037 \cs_new:Npn \_fp_ep_isqrt_esti:wwwnnwn #1, #2, #3, #4
14038 {
14039     \if_int_compare:w #1 = #2 \exp_stop_f:
14040         \exp_after:wN \_fp_ep_isqrt_estii:wwwnnwn
14041     \fi:
14042     \exp_after:wN \_fp_ep_isqrt_esti:wwwnnwn
14043     \int_value:w \int_eval:w
14044         (#1 + 1 0050 0000 #4 / (#1 * #3)) / 2 ,
14045     #1, #3, {#4}
14046 }
14047 \cs_new:Npn \_fp_ep_isqrt_estii:wwwnnwn #1, #2, #3, #4#5
14048 {
14049     \exp_after:wN \_fp_ep_isqrt_estiii:NNNNNwwwn
14050     \int_value:w \int_eval:w 1000 0000 + #2 * #2 #5 * 5
14051     \exp_after:wN , \int_value:w \int_eval:w 10000 + #2 ;
14052 }
14053 \cs_new:Npn \_fp_ep_isqrt_estiii:NNNNNwwwn #1#2#3#4#5#6, #7#8; #9;
14054 {
14055     \fp_fixed_mul_short:wwn #9; {#1} {#2#3#4#5} {#600} ;
14056     \fp_ep_isqrt_epsii:wwN
14057     \fp_fixed_mul_short:wwn {#7} {#80} {0000} ;
14058 }
```

(End definition for `_fp_ep_isqrt_esti:wwwnnwn`, `_fp_ep_isqrt_estii:wwwnnwn`, and `_fp_ep_isqrt_estiii:NNNNNwwwn`.)

`_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`.

```

14059 \cs_new:Npn \_fp_ep_isqrt_epsii:wwN #1;
14060 {
14061     \fp_fixed_sub:wwn {15000}{0000}{0000}{0000}{0000}; #1;
14062     \fp_ep_isqrt_epsii:wwN #1;
14063     \fp_ep_isqrt_epsii:wwN #1;
14064     \fp_ep_isqrt_epsii:wwN #1;
14065 }
14066 \cs_new:Npn \_fp_ep_isqrt_epsii:wwN #1; #2;
14067 {
14068     \fp_fixed_mul:wwn #1; #1;
14069     \fp_fixed_mul_sub_back:wwnn #2;
14070     {15000}{0000}{0000}{0000}{0000};
14071     \fp_fixed_mul:wwn #1;
14072 }
```

(End definition for `_fp_ep_isqrt_epsii:wwN` and `_fp_ep_isqrt_epsii:wwN`.)

27.11 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` 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.

```

14073 \cs_new:Npn \_fp_ep_to_float_o:wwN #1,
14074   { + \int_eval:w #1 \_fp_fixed_to_float_o:wwN }
14075 \cs_new:Npn \_fp_ep_inv_to_float_o:wwN #1,#2;
14076   {
14077     \_fp_ep_div:wwwn 1,{1000}{0000}{0000}{0000}{0000}; #1,#2;
14078     \_fp_ep_to_float_o:wwN
14079   }

```

(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.

```

14080 \cs_new:Npn \_fp_fixed_inv_to_float_o:wN
14081   { \_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 `|3fp-trig`.

```

14082 \cs_new:Npn \_fp_fixed_to_float_rad_o:wN #1;
14083   {
14084     \_fp_fixed_mul:wwn #1; {5729}{5779}{5130}{8232}{0876}{7981};
14085     { \_fp_ep_to_float_o:wwN 2, }
14086   }

```

(End definition for `_fp_fixed_to_float_rad_o:wN`.)

`_fp_fixed_to_float_o:wN` 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

```

14087 \cs_new:Npn \_fp_fixed_to_float_o:Nw #1#2; { \_fp_fixed_to_float_o:wN #2; #1 }
14088 \cs_new:Npn \_fp_fixed_to_float_o:wN #1#2#3#4#5#6; #7
14089   {
14090     + \int_eval:w \c_fp_block_int % for the 8-digit-at-the-start thing.
14091     \exp_after:wN \exp_after:wN
14092     \exp_after:wN \_fp_fixed_to_loop:N
14093     \exp_after:wN \use_none:n
14094     \int_value:w \int_eval:w
14095       1 0000 0000 + #1 \exp_after:wN \_fp_use_none_stop_f:n
14096       \int_value:w 1#2 \exp_after:wN \_fp_use_none_stop_f:n
14097       \int_value:w 1#3#4 \exp_after:wN \_fp_use_none_stop_f:n
14098       \int_value:w 1#5#6
14099     \exp_after:wN ;
14100     \exp_after:wN ;
14101   }
14102 \cs_new:Npn \_fp_fixed_to_loop:N #1
14103   {

```

¹¹Bruno: I must double check this assumption.

```

14104     \if_meaning:w 0 #1
14105         - 1
14106         \exp_after:wN \__fp_fixed_to_loop:N
14107     \else:
14108         \exp_after:wN \__fp_fixed_to_loop_end:w
14109         \exp_after:wN #1
14110     \fi:
14111 }
14112 \cs_new:Npn \__fp_fixed_to_loop_end:w #1 #2 ;
14113 {
14114     \if_meaning:w ; #1
14115         \exp_after:wN \__fp_fixed_to_float_zero:w
14116     \else:
14117         \exp_after:wN \__fp_pack_twice_four:wNNNNNNNN
14118         \exp_after:wN \__fp_pack_twice_four:wNNNNNNNN
14119         \exp_after:wN \__fp_fixed_to_float_pack:ww
14120         \exp_after:wN ;
14121     \fi:
14122     #1 #2 0000 0000 0000 0000 ;
14123 }
14124 \cs_new:Npn \__fp_fixed_to_float_zero:w ; 0000 0000 0000 0000 ;
14125 {
14126     - 2 * \c_fp_max_exponent_int ;
14127     {0000} {0000} {0000} {0000} ;
14128 }
14129 \cs_new:Npn \__fp_fixed_to_float_pack:ww #1 ; #2#3 ;
14130 {
14131     \if_int_compare:w #2 > 4 \exp_stop_f:
14132         \exp_after:wN \__fp_fixed_to_float_round_up:wnnnnw
14133     \fi:
14134     ; #1 ;
14135 }
14136 \cs_new:Npn \__fp_fixed_to_float_round_up:wnnnnw ; #1#2#3#4 ;
14137 {
14138     \exp_after:wN \__fp_basics_pack_high>NNNNw
14139     \__int_value:w \__int_eval:w 1 #1#2
14140     \exp_after:wN \__fp_basics_pack_low>NNNNw
14141     \__int_value:w \__int_eval:w 1 #3#4 + 1 ;
14142 }

```

(End definition for `__fp_fixed_to_float_o:wN` and `__fp_fixed_to_float_o:Nw`.)

14143 ⟨/initex | package⟩

28 I3fp-expo implementation

```

14144 {*initex | package}
14145 <@=fp>
\__fp_parse_word_exp:N
\__fp_parse_word_ln:N
14146 \cs_new:Npn \__fp_parse_word_exp:N
14147     { \__fp_parse_unary_function:NNN \__fp_exp_o:w ? }
14148 \cs_new:Npn \__fp_parse_word_ln:N
14149     { \__fp_parse_unary_function:NNN \__fp_ln_o:w ? }

```

(End definition for `_fp_parse_word_exp:N` and `_fp_parse_word_ln:N`.)

28.1 Logarithm

28.1.1 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, $c \in [1, 10]$ will be 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.

28.1.2 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_vi_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
14150 \tl_const:Nn \c_fp_ln_i_fixed_t1 { {0000}{0000}{0000}{0000}{0000}{0000};}
14151 \tl_const:Nn \c_fp_ln_ii_fixed_t1 { {6931}{4718}{0559}{9453}{0941}{7232};}
14152 \tl_const:Nn \c_fp_ln_iii_fixed_t1 { {10986}{1228}{8668}{1096}{9139}{5245};}
14153 \tl_const:Nn \c_fp_ln_iv_fixed_t1 { {13862}{9436}{1119}{8906}{1883}{4464};}
14154 \tl_const:Nn \c_fp_ln_vi_fixed_t1 { {17917}{5946}{9228}{0550}{0081}{2477};}
14155 \tl_const:Nn \c_fp_ln_vii_fixed_t1 { {19459}{1014}{9055}{3133}{0510}{5353};}
14156 \tl_const:Nn \c_fp_ln_viii_fixed_t1 { {20794}{4154}{1679}{8359}{2825}{1696};}
14157 \tl_const:Nn \c_fp_ln_ix_fixed_t1 { {21972}{2457}{7336}{2193}{8279}{0490};}
14158 \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.)

28.1.3 Sign, exponent, and special numbers

`_fp_ln_o:w` 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`.

```

14159 \cs_new:Npn \_fp_ln_o:w #1 \s_fp \_fp_chk:w #2#3#4; @
14160   {

```

```

14161   \if_meaning:w 2 #3
14162     \_fp_case_use:nw { \_fp_invalid_operation_o:nw { ln } }
14163   \fi:
14164   \if_case:w #2 \exp_stop_f:
14165     \_fp_case_use:nw
14166       { \_fp_division_by_zero_o:Nnw \c_minus_inf_fp { ln } }
14167   \or:
14168   \else:
14169     \_fp_case_return_same_o:w
14170   \fi:
14171   \_fp_ln_npos_o:w \s_fp \_fp_chk:w #2#3#4;
14172 }

```

(End definition for $_fp_ln_o:w$.)

28.1.4 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.

```

14173 \cs_new:Npn \_fp_ln_npos_o:w \s_fp \_fp_chk:w 10#1#2#3;
14174   { %^A todo: ln(1) should be "exact zero", not "underflow"
14175     \exp_after:wN \_fp_sanitize:Nw
14176     \_int_value:w % for the overall sign
14177     \if_int_compare:w #1 < 1 \exp_stop_f:
14178       2
14179     \else:
14180       0
14181   \fi:
14182   \exp_after:wN \exp_stop_f:
14183   \_int_value:w \_int_eval:w % for the exponent
14184     \_fp_ln_significand:NNNNnnnN #2#3
14185     \_fp_ln_exponent:wn {#1}
14186 }

```

(End definition for $_fp_ln_npos_o:w$.)

$_fp_ln_significand:NNNNnnnN$

$_fp_ln_significand:NNNNnnnN \langle X_1 \rangle \{ \langle X_2 \rangle \} \{ \langle X_3 \rangle \} \{ \langle X_4 \rangle \} \langle \text{continuation} \rangle$
This function expands to

$\langle \text{continuation} \rangle \{ \langle Y_1 \rangle \} \{ \langle Y_2 \rangle \} \{ \langle Y_3 \rangle \} \{ \langle Y_4 \rangle \} \{ \langle Y_5 \rangle \} \{ \langle Y_6 \rangle \} ;$

where $Y = -\ln(X)$ as an extended fixed point.

```

14187 \cs_new:Npn \_fp_ln_significand:NNNNnnnN #1#2#3#4
14188   {
14189     \exp_after:wN \_fp_ln_x_ii:wnnnn
14190     \_int_value:w
14191     \if_case:w #1 \exp_stop_f:
14192     \or:
14193       \if_int_compare:w #2 < 4 \exp_stop_f:
14194         \_int_eval:w 10 - #2
14195       \else:
14196         6
14197       \fi:
14198     \or: 4
14199     \or: 3

```

```

14200      \or: 2
14201      \or: 2
14202      \or: 2
14203      \else: 1
14204      \fi:
14205      ; { #1 #2 #3 #4 }
14206  }

```

(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 $1 + x = 1 + ac \in [1.7, 2.4]$.

```

14207 \cs_new:Npn \_fp_ln_x_ii:wnnnn #1; #2#3#4#5
14208  {
14209      \exp_after:wN \_fp_ln_div_after:Nw
14210      \cs:w c_fp_ln_\_int_to_roman:w #1 _fixed_tl \exp_after:wN \cs_end:
14211      \_int_value:w
14212          \exp_after:wN \_fp_ln_x_iv:wnnnnnnnn
14213          \_int_value:w \_int_eval:w
14214          \exp_after:wN \_fp_ln_x_iii_var:NNNNNw
14215          \_int_value:w \_int_eval:w 9999 9990 + #1*#2#3 +
14216          \exp_after:wN \_fp_ln_x_iii:NNNNNw
14217          \_int_value:w \_int_eval:w 10 0000 0000 + #1*#4#5 ;
14218          {20000} {0000} {0000}
14219      } %^^A todo: reoptimize (a generalization attempt failed).
14220 \cs_new:Npn \_fp_ln_x_iii:NNNNNw #1#2 #3#4#5#6 #7;
14221  { #1#2; {#3#4#5#6} {#7} }
14222 \cs_new:Npn \_fp_ln_x_iii_var:NNNNNw #1 #2#3#4#5 #6;
14223  {
14224      #1#2#3#4#5 + 1 ;
14225      {#1#2#3#4#5} {#6}
14226  }

```

The Taylor series will be 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 eTeX 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 $B = 10^4A - Q_1Z$. Then

$$\begin{aligned} 10^4B &\leq A_1A_2.A_3A_4 - \left(\frac{A_1A_2}{y} - \frac{3}{2}\right)10^4Z \\ &\leq A_1A_2\left(1 - \frac{10^4Z}{y}\right) + 1 + \frac{3}{2}y \\ &\leq 10^8\frac{A}{y} + 1 + \frac{3}{2}y \end{aligned}$$

In the same way, and using $1.7 \cdot 10^4 \leq y \leq 2.4 \cdot 10^4$, and convexity, we get

$$\begin{aligned} 10^4A &= 2 \cdot 10^4 \\ 10^4B &\leq 10^8\frac{A}{y} + 1.6y \leq 4.7 \cdot 10^4 \\ 10^4C &\leq 10^8\frac{B}{y} + 1.6y \leq 5.8 \cdot 10^4 \\ 10^4D &\leq 10^8\frac{C}{y} + 1.6y \leq 6.3 \cdot 10^4 \\ 10^4E &\leq 10^8\frac{D}{y} + 1.6y \leq 6.5 \cdot 10^4 \\ 10^4F &\leq 10^8\frac{E}{y} + 1.6y \leq 6.6 \cdot 10^4 \end{aligned}$$

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 \langle 1 or 2 \rangle \langle 8d \rangle ; {\langle 4d \rangle} {\langle 4d \rangle} \langle fixed-tl \rangle
```

The number is x . Compute y by adding 1 to the five first digits.

```
14227 \cs_new:Npn \_\_fp\_ln\_x\_iv:wnnnnnnnn #1; #2#3#4#5 #6#7#8#9
14228 {
14229   \exp_after:wN \_\_fp\_div\_significand\_pack:NNN
14230   \_int\_value:w \_int\_eval:w
14231   \_\_fp\_ln\_div\_i:w #1 ;
14232   #6 #7 ; {#8} {#9}
14233   {#2} {#3} {#4} {#5}
14234   { \exp_after:wN \_\_fp\_ln\_div\_ii:wwn \_int\_value:w #1 }
14235   { \exp_after:wN \_\_fp\_ln\_div\_ii:wwn \_int\_value:w #1 }
14236   { \exp_after:wN \_\_fp\_ln\_div\_ii:wwn \_int\_value:w #1 }
14237   { \exp_after:wN \_\_fp\_ln\_div\_ii:wwn \_int\_value:w #1 }
14238   { \exp_after:wN \_\_fp\_ln\_div\_vi:wwn \_int\_value:w #1 }
14239 }
14240 \cs_new:Npn \_\_fp\_ln\_div\_i:w #1;
14241 {
14242   \exp_after:wN \_\_fp\_div\_significand\_calc:wnnnnnnnn
14243   \_int\_value:w \_int\_eval:w 999999 + 2 0000 0000 / #1 ; % Q1
14244 }
14245 \cs_new:Npn \_\_fp\_ln\_div\_ii:wwn #1; #2;#3 % y; B1;B2 <- for k=1
14246 {
```

```

14247     \exp_after:wN \_fp_div_significand_pack:NNN
14248     \_int_value:w \_int_eval:w
14249     \exp_after:wN \_fp_div_significand_calc:wwnnnnnnn
14250     \_int_value:w \_int_eval:w 999999 + #2 #3 / #1 ; % Q2
14251     #2 #3 ;
14252 }
14253 \cs_new:Npn \_fp_ln_div_vi:wwn #1; #2;#3#4#5 #6#7#8#9 %y;F1;F2F3F4x1x2x3x4
14254 {
14255     \exp_after:wN \_fp_div_significand_pack:NNN
14256     \_int_value:w \_int_eval:w 1000000 + #2 #3 / #1 ; % Q6
14257 }

```

We now have essentially

$$_fp_ln_div_after:Nw \langle fixed tl \rangle _fp_div_significand_pack:NNN 10^6 + Q_1 _fp_div_significand_pack:NNN 10^6 + Q_2 _fp_div_significand_pack:NNN 10^6 + Q_3 _fp_div_significand_pack:NNN 10^6 + Q_4 _fp_div_significand_pack:NNN 10^6 + Q_5 _fp_div_significand_pack:NNN 10^6 + Q_6 ; \langle exponent \rangle ; \langle continuation \rangle$$

where $\langle fixed tl \rangle$ holds the logarithm of a number in $[1, 10]$, and $\langle exponent \rangle$ 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

$$_fp_ln_div_after:Nw \langle fixed-tl \rangle \langle 1d \rangle ; \langle 4d \rangle ; \langle exponent \rangle ;$$

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.

```

14258 \cs_new:Npn \_fp_ln_div_after:Nw #1#2;
14259 {
14260     \if_meaning:w 0 #2
14261         \exp_after:wN \_fp_ln_t_small:Nw
14262     \else:
14263         \exp_after:wN \_fp_ln_t_large>NNw
14264         \exp_after:wN -
14265         \fi:
14266         #1
14267     }
14268 \cs_new:Npn \_fp_ln_t_small:Nw #1 #2; #3; #4; #5; #6; #7;
14269 {
14270     \exp_after:wN \_fp_ln_t_large>NNw
14271     \exp_after:wN + % <sign>
14272     \exp_after:wN #1
14273     \_int_value:w \_int_eval:w 9999 - #2 \exp_after:wN ;
14274     \_int_value:w \_int_eval:w 9999 - #3 \exp_after:wN ;
14275     \_int_value:w \_int_eval:w 9999 - #4 \exp_after:wN ;
14276     \_int_value:w \_int_eval:w 9999 - #5 \exp_after:wN ;
14277     \_int_value:w \_int_eval:w 9999 - #6 \exp_after:wN ;
14278     \_int_value:w \_int_eval:w 1 0000 - #7 ;
14279 }

```

```
\_\_fp\_ln\_t\_large:NNw <sign><fixed tl> <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.

```

14280 \cs_new:Npn \_\_fp\_ln\_t\_large:NNw #1 #2 #3; #4; #5; #6; #7; #8;
14281 {
14282   \exp_after:wN \_\_fp\_ln\_square\_t\_after:w
14283   \int_value:w \_\_int\_eval:w 9999 0000 + #3*#3
14284   \exp_after:wN \_\_fp\_ln\_square\_t\_pack:NNNNNw
14285   \int_value:w \_\_int\_eval:w 9999 0000 + 2*#3*#4
14286   \exp_after:wN \_\_fp\_ln\_square\_t\_pack:NNNNNw
14287   \int_value:w \_\_int\_eval:w 9999 0000 + 2*#3*#5 + #4*#4
14288   \exp_after:wN \_\_fp\_ln\_square\_t\_pack:NNNNNw
14289   \int_value:w \_\_int\_eval:w 9999 0000 + 2*#3*#6 + 2*#4*#5
14290   \exp_after:wN \_\_fp\_ln\_square\_t\_pack:NNNNNw
14291   \int_value:w \_\_int\_eval:w 1 0000 0000 + 2*#3*#7 + 2*#4*#6 + #5*#5
14292   + (2*#3*#8 + 2*#4*#7 + 2*#5*#6) / 1 0000
14293   % ; ;
14294   \exp_after:wN \_\_fp\_ln\_twice\_t\_after:w
14295   \int_value:w \_\_int\_eval:w -1 + 2*#3
14296   \exp_after:wN \_\_fp\_ln\_twice\_t\_pack:Nw
14297   \int_value:w \_\_int\_eval:w 9999 + 2*#4
14298   \exp_after:wN \_\_fp\_ln\_twice\_t\_pack:Nw
14299   \int_value:w \_\_int\_eval:w 9999 + 2*#5
14300   \exp_after:wN \_\_fp\_ln\_twice\_t\_pack:Nw
14301   \int_value:w \_\_int\_eval:w 9999 + 2*#6
14302   \exp_after:wN \_\_fp\_ln\_twice\_t\_pack:Nw
14303   \int_value:w \_\_int\_eval:w 9999 + 2*#7
14304   \exp_after:wN \_\_fp\_ln\_twice\_t\_pack:Nw
14305   \int_value:w \_\_int\_eval:w 10000 + 2*#8 ; ;
14306   { \_\_fp\_ln\_c:NwNw #1 }
14307   #2
14308 }
14309 \cs_new:Npn \_\_fp\_ln\_twice\_t\_pack:Nw #1 #2; { + #1 ; {#2} }
14310 \cs_new:Npn \_\_fp\_ln\_twice\_t\_after:w #1; { ; ; {#1} }
14311 \cs_new:Npn \_\_fp\_ln\_square\_t\_pack:NNNNNw #1 #2#3#4#5 #6;
14312 { + #1#2#3#4#5 ; {#6} }
14313 \cs_new:Npn \_\_fp\_ln\_square\_t\_after:w 1 0 #1#2#3 #4;
14314 { \_\_fp\_ln\_Taylor:wwNw {#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 {⟨T1⟩} {⟨T2⟩} {⟨T3⟩} {⟨T4⟩} {⟨T5⟩} {⟨T6⟩} ; ;
{⟨(2t)1⟩} {⟨(2t)2⟩} {⟨(2t)3⟩} {⟨(2t)4⟩} {⟨(2t)5⟩} {⟨(2t)6⟩} ; { \_\_fp\_ln\_c:NwNw <sign>} <fixed tl> <exponent>; <continuation>
```

And we want to compute

$$\ln\left(\frac{1+t}{1-t}\right) = 2t \left(1 + T \left(\frac{1}{3} + T \left(\frac{1}{5} + T \left(\frac{1}{7} + T \left(\frac{1}{9} + \dots\right)\right)\right)\right)\right)$$

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 ($< 10^4$).

```

14315 \cs_new:Npn \_fp_ln_Taylor:wwNw
14316   { \_fp_ln_Taylor_loop:www 21 ; {0000}{0000}{0000}{0000}{0000} ; }
14317 \cs_new:Npn \_fp_ln_Taylor_loop:www #1; #2; #3;
14318   {
14319     \if_int_compare:w #1 = 1 \exp_stop_f:
14320       \_fp_ln_Taylor_break:w
14321     \fi:
14322     \exp_after:wN \_fp_fixed_div_int:wwN \c_fp_one_fixed_tl #1;
14323     \_fp_fixed_add:wwn #2;
14324     \_fp_fixed_mul:wwn #3;
14325   {
14326     \exp_after:wN \_fp_ln_Taylor_loop:www
14327       \_int_value:w \_int_eval:w #1 - 2 ;
14328   }
14329   #3;
14330 }
14331 \cs_new:Npn \_fp_ln_Taylor_break:w \fi: #1 \_fp_fixed_add:wwn #2#3; #4 ;
14332   {
14333     \fi:
14334     \exp_after:wN \_fp_fixed_mul:wwn
14335     \exp_after:wN { \_int_value:w \_int_eval:w 10000 + #2 } #3;
14336   }

```

(End definition for $_fp_ln_Taylor:wwNw$.)

```

\_fp_ln_c:NwNw      \_fp_ln_c:NwNw <sign> {<r1>} {<r2>} {<r3>} {<r4>} {<r5>} {<r6>} ; <fixed tl>
<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 will be at least $\ln(10/7) \simeq 0.35$.

```

14337 \cs_new:Npn \_fp_ln_c:NwNw #1 #2; #3
14338   {
14339     \if_meaning:w + #1
14340       \exp_after:wN \exp_after:wN \exp_after:wN \_fp_fixed_sub:wwn
14341     \else:
14342       \exp_after:wN \exp_after:wN \exp_after:wN \_fp_fixed_add:wwn
14343     \fi:
14344     #3 #2 ;
14345   }

```

(End definition for $_fp_ln_c:NwNw$.)

```

\_fp_ln_exponent:wn  \_fp_ln_exponent:wn {<s1>} {<s2>} {<s3>} {<s4>} {<s5>} {<s6>} ;
{<exponent>}

```

Compute $\langle exponent \rangle$ times $\ln(10)$. Apart from the cases where $\langle exponent \rangle$ is 0 or 1, the result will necessarily be 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.

```

14346 \cs_new:Npn \__fp_ln_exponent:wn #1; #2
14347 {
14348     \if_case:w #2 \exp_stop_f:
14349         0 \__fp_case_return:nw { \__fp_fixed_to_float_o:Nw 2 }
14350     \or:
14351         \exp_after:wN \__fp_ln_exponent_one:ww \__int_value:w
14352     \else:
14353         \if_int_compare:w #2 > 0 \exp_stop_f:
14354             \exp_after:wN \__fp_ln_exponent_small:NNww
14355             \exp_after:wN 0
14356             \exp_after:wN \__fp_fixed_sub:wwn \__int_value:w
14357         \else:
14358             \exp_after:wN \__fp_ln_exponent_small:NNww
14359             \exp_after:wN 2
14360             \exp_after:wN \__fp_fixed_add:wwn \__int_value:w -
14361         \fi:
14362     \fi:
14363     #2; #1;
14364 }
```

Now we painfully write all the cases.¹² No overflow nor underflow can happen, except when computing $\ln(1)$.

```

14365 \cs_new:Npn \__fp_ln_exponent_one:ww 1; #1;
14366 {
14367     0
14368     \exp_after:wN \__fp_fixed_sub:wwn \c__fp_ln_x_fixed_tl #1;
14369     \__fp_fixed_to_float_o:wN 0
14370 }
```

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.

```

14371 \cs_new:Npn \__fp_ln_exponent_small:NNww #1#2#3; #4#5#6#7#8#9;
14372 {
14373     4
14374     \exp_after:wN \__fp_fixed_mul:wwn
14375         \c__fp_ln_x_fixed_tl
14376         {#3}{0000}{0000}{0000}{0000} ;
14377     #2
14378         {0000}{#4}{#5}{#6}{#7}{#8};
14379     \__fp_fixed_to_float_o:wN #1
14380 }
```

(End definition for $__fp_ln_exponent:wn$.)

¹²Bruno: do rounding.

28.2 Exponential

28.2.1 Sign, exponent, and special numbers

```
\_fp_exp_o:w
14381 \cs_new:Npn \_fp_exp_o:w #1 \s_fp \_fp_chk:w #2#3#4; @
14382 {
14383     \if_case:w #2 \exp_stop_f:
14384         \_fp_case_return_o:Nw \c_one_fp
14385     \or:
14386         \exp_after:wN \_fp_exp_normal_o:w
14387     \or:
14388         \if_meaning:w 0 #3
14389             \exp_after:wN \_fp_case_return_o:Nw
14390             \exp_after:wN \c_inf_fp
14391         \else:
14392             \exp_after:wN \_fp_case_return_o:Nw
14393             \exp_after:wN \c_zero_fp
14394         \fi:
14395     \or:
14396         \_fp_case_return_same_o:w
14397     \fi:
14398     \s_fp \_fp_chk:w #2#3#4;
14399 }
```

(End definition for `_fp_exp_o:w`.)

```
\_fp_exp_normal_o:w
\_fp_exp_pos_o:Nnwnw
\_fp_exp_overflow>NN
14400 \cs_new:Npn \_fp_exp_normal_o:w \s_fp \_fp_chk:w 1#1
14401 {
14402     \if_meaning:w 0 #1
14403         \_fp_exp_pos_o>NNnwnw + \_fp_fixed_to_float_o:wN
14404     \else:
14405         \_fp_exp_pos_o>NNnwnw - \_fp_fixed_inv_to_float_o:wN
14406     \fi:
14407 }
14408 \cs_new:Npn \_fp_exp_pos_o>NNnwnw #1#2#3 \fi: #4#5;
14409 {
14410     \fi:
14411     \if_int_compare:w #4 > \c_fp_max_exp_exponent_int
14412         \token_if_eq_charcode:NNTF + #1
14413             { \_fp_exp_overflow>NN \_fp_overflow:w \c_inf_fp }
14414             { \_fp_exp_overflow>NN \_fp_underflow:w \c_zero_fp }
14415         \exp:w
14416     \else:
14417         \exp_after:wN \_fp_sanitize:Nw
14418         \exp_after:wN 0
14419         \_int_value:w #1 \_int_eval:w
14420         \if_int_compare:w #4 < 0 \exp_stop_f:
14421             \exp_after:wN \use_i:nn
14422         \else:
14423             \exp_after:wN \use_ii:nn
14424         \fi:
14425     {
14426         0
```

```

14427      \__fp_decimate:nNnnnn { - #4 }
14428          \__fp_exp_Taylor:Nnnwn
14429      }
14430      {
14431          \__fp_decimate:nNnnnn { \c__fp_prec_int - #4 }
14432              \__fp_exp_pos_large:NnnNwn
14433      }
14434      #5
14435      {#4}
14436      #1 #2 0
14437      \exp:w
14438      \fi:
14439      \exp_after:wN \exp_end:
14440  }
14441 \cs_new:Npn \__fp_exp_overflow>NN #1#2
14442  {
14443      \exp_after:wN \exp_after:wN
14444      \exp_after:wN #1
14445      \exp_after:wN #2
14446  }

```

(End definition for `__fp_exp_normal_o:w`, `__fp_exp_pos_o:Nnwnw`, 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.

```

14447 \cs_new:Npn \__fp_exp_Taylor:Nnnwn #1#2#3 #4; #5 #6
14448  {
14449      #6
14450      \__fp_pack_twice_four:wNNNNNNNN
14451      \__fp_pack_twice_four:wNNNNNNNN
14452      \__fp_pack_twice_four:wNNNNNNNN
14453      \__fp_exp_Taylor_ii:ww
14454      ; #2#3#4 0000 0000 ;
14455  }
14456 \cs_new:Npn \__fp_exp_Taylor_ii:ww #1; #2;
14457  { \__fp_exp_Taylor_loop:www 10 ; #1 ; #1 ; \s_stop }
14458 \cs_new:Npn \__fp_exp_Taylor_loop:www #1; #2; #3;
14459  {
14460      \if_int_compare:w #1 = 1 \exp_stop_f:
14461          \exp_after:wN \__fp_exp_Taylor_break:Nww
14462      \fi:
14463      \__fp_fixed_div_int:wwN #3 ; #1 ;
14464      \__fp_fixed_add_one:wN
14465      \__fp_fixed_mul:wwN #2 ;
14466  {
14467      \exp_after:wN \__fp_exp_Taylor_loop:www
14468      \__int_value:w \__int_eval:w #1 - 1 ;
14469      #2 ;
14470  }
14471  }
14472 \cs_new:Npn \__fp_exp_Taylor_break:Nww #1 #2; #3 \s_stop
14473  { \__fp_fixed_add_one:wN #2 ; }

```

(End definition for `_fp_exp_Taylor:Nnnwn`, `_fp_exp_Taylor_loop:www`, and `_fp_exp_Taylor_break:Nww`.)

```
\_fp_exp_pos_large:NnnNwn
\fp_exp_large_after:www
    \fp_exp_large:w
        \fp_exp_large_v:wN
        \fp_exp_large_iv:wN
\fp_exp_large_iii:wN
\fp_exp_large_ii:wN
    \fp_exp_large_i:wN
\fp_exp_large_:wN
```

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 having the auxiliary for one exponent call the auxiliary for the next exponent. The current total is expressed by leaving the exponent behind in the input stream (we are currently within an `\int_eval:w`), and keeping track of a fixed point number, #1 for the numbered auxiliaries. Our usage of `\if_case:w` is somewhat dirty for optimization: TeX jumps to the appropriate case, but we then close the `\if_case:w` "by hand", using `\or:` and `\fi:` as delimiters.

```
14474 \cs_new:Npn \fp_exp_pos_large:NnnNwn #1#2#3 #4#5; #6
14475 {
14476     \exp_after:wN \exp_after:wN
14477     \cs:w \fp_exp_large_ \int_to_roman:w #6 :wN \exp_after:wN \cs_end:
14478     \exp_after:wN \c_fp_one_fixed_tl
14479     \int_value:w #3 #4 \exp_stop_f:
14480     #5 00000 ;
14481 }
14482 \cs_new:Npn \fp_exp_large:w #1 \or: #2 \fi:
14483 { \fi: \fp_fixed_mul:www #1; }
14484 \cs_new:Npn \fp_exp_large_v:wN #1; #2
14485 {
14486     \if_case:w #2 ~ \exp_after:wN \fp_fixed_continue:wn \or:
14487     + 4343 \fp_exp_large:w {8806}{8182}{2566}{2921}{5872}{6150} \or:
14488     + 8686 \fp_exp_large:w {7756}{0047}{2598}{6861}{0458}{3204} \or:
14489     + 13029 \fp_exp_large:w {6830}{5723}{7791}{4884}{1932}{7351} \or:
14490     + 17372 \fp_exp_large:w {6015}{5609}{3095}{3052}{3494}{7574} \or:
14491     + 21715 \fp_exp_large:w {5297}{7951}{6443}{0315}{3251}{3576} \or:
14492     + 26058 \fp_exp_large:w {4665}{6719}{0099}{3379}{5527}{2929} \or:
14493     + 30401 \fp_exp_large:w {4108}{9724}{3326}{3186}{5271}{5665} \or:
14494     + 34744 \fp_exp_large:w {3618}{6973}{3140}{0875}{3856}{4102} \or:
14495     + 39087 \fp_exp_large:w {3186}{9209}{6113}{3900}{6705}{9685} \or:
14496 \fi:
14497 #1;
14498 \fp_exp_large_iv:wN
14499 }
14500 \cs_new:Npn \fp_exp_large_iv:wN #1; #2
14501 {
14502     \if_case:w #2 ~ \exp_after:wN \fp_fixed_continue:wn \or:
14503     + 435 \fp_exp_large:w {1970}{0711}{1401}{7046}{9938}{8888} \or:
14504     + 869 \fp_exp_large:w {3881}{1801}{9428}{4368}{5764}{8232} \or:
14505     + 1303 \fp_exp_large:w {7646}{2009}{8905}{4704}{8893}{1073} \or:
14506     + 1738 \fp_exp_large:w {1506}{3559}{7005}{0524}{9009}{7592} \or:
14507     + 2172 \fp_exp_large:w {2967}{6283}{8402}{3667}{0689}{6630} \or:
14508     + 2606 \fp_exp_large:w {5846}{4389}{5650}{2114}{7278}{5046} \or:
14509     + 3041 \fp_exp_large:w {1151}{7900}{5080}{6878}{2914}{4154} \or:
14510     + 3475 \fp_exp_large:w {2269}{1083}{0850}{6857}{8724}{4002} \or:
14511     + 3909 \fp_exp_large:w {4470}{3047}{3316}{5442}{6408}{6591} \or:
14512 \fi:
```

```

14513      #1;
14514      \_\_fp\_exp\_large\_iii:wN
14515  }
14516 \cs_new:Npn \_\_fp\_exp\_large\_iii:wN #1; #2
14517  {
14518      \if_case:w #2 ~ \exp_after:wN \_\_fp_fixed_continue:wn \or:
14519      + 44 \_\_fp\_exp\_large:w {2688}{1171}{4181}{6135}{4484}{1263} \or:
14520      + 87 \_\_fp\_exp\_large:w {7225}{9737}{6812}{5749}{2581}{7748} \or:
14521      + 131 \_\_fp\_exp\_large:w {1942}{4263}{9524}{1255}{9365}{8421} \or:
14522      + 174 \_\_fp\_exp\_large:w {5221}{4696}{8976}{4143}{9505}{8876} \or:
14523      + 218 \_\_fp\_exp\_large:w {1403}{5922}{1785}{2837}{4107}{3977} \or:
14524      + 261 \_\_fp\_exp\_large:w {3773}{0203}{0092}{9939}{8234}{0143} \or:
14525      + 305 \_\_fp\_exp\_large:w {1014}{2320}{5473}{5004}{5094}{5533} \or:
14526      + 348 \_\_fp\_exp\_large:w {2726}{3745}{7211}{2566}{5673}{6478} \or:
14527      + 391 \_\_fp\_exp\_large:w {7328}{8142}{2230}{7421}{7051}{8866} \or:
14528  \fi:
14529  #1;
14530  \_\_fp\_exp\_large\_ii:wN
14531 }
14532 \cs_new:Npn \_\_fp\_exp\_large\_ii:wN #1; #2
14533 {
14534      \if_case:w #2 ~ \exp_after:wN \_\_fp_fixed_continue:wn \or:
14535      + 5 \_\_fp\_exp\_large:w {2202}{6465}{7948}{0671}{6516}{9579} \or:
14536      + 9 \_\_fp\_exp\_large:w {4851}{6519}{5409}{7902}{7796}{9107} \or:
14537      + 14 \_\_fp\_exp\_large:w {1068}{6474}{5815}{2446}{2146}{9905} \or:
14538      + 18 \_\_fp\_exp\_large:w {2353}{8526}{6837}{0199}{8540}{7900} \or:
14539      + 22 \_\_fp\_exp\_large:w {5184}{7055}{2858}{7072}{4640}{8745} \or:
14540      + 27 \_\_fp\_exp\_large:w {1142}{0073}{8981}{5684}{2836}{6296} \or:
14541      + 31 \_\_fp\_exp\_large:w {2515}{4386}{7091}{9167}{0062}{6578} \or:
14542      + 35 \_\_fp\_exp\_large:w {5540}{6223}{8439}{3510}{0525}{7117} \or:
14543      + 40 \_\_fp\_exp\_large:w {1220}{4032}{9431}{7840}{8020}{0271} \or:
14544  \fi:
14545  #1;
14546  \_\_fp\_exp\_large\_i:wN
14547 }
14548 \cs_new:Npn \_\_fp\_exp\_large\_i:wN #1; #2
14549 {
14550      \if_case:w #2 ~ \exp_after:wN \_\_fp_fixed_continue:wn \or:
14551      + 1 \_\_fp\_exp\_large:w {2718}{2818}{2845}{9045}{2353}{6029} \or:
14552      + 1 \_\_fp\_exp\_large:w {7389}{0560}{9893}{0650}{2272}{3043} \or:
14553      + 2 \_\_fp\_exp\_large:w {2008}{5536}{9231}{8766}{7740}{9285} \or:
14554      + 2 \_\_fp\_exp\_large:w {5459}{8150}{0331}{4423}{9078}{1103} \or:
14555      + 3 \_\_fp\_exp\_large:w {1484}{1315}{9102}{5766}{0342}{1116} \or:
14556      + 3 \_\_fp\_exp\_large:w {4034}{2879}{3492}{7351}{2260}{8387} \or:
14557      + 4 \_\_fp\_exp\_large:w {1096}{6331}{5842}{8458}{5992}{6372} \or:
14558      + 4 \_\_fp\_exp\_large:w {2980}{9579}{8704}{1728}{2747}{4359} \or:
14559      + 4 \_\_fp\_exp\_large:w {8103}{0839}{2757}{5384}{0077}{1000} \or:
14560  \fi:
14561  #1;
14562  \_\_fp\_exp\_large\_wN
14563 }
14564 \cs_new:Npn \_\_fp\_exp\_large\_wN #1; #2
14565 {
14566      \if_case:w #2 ~ \exp_after:wN \_\_fp_fixed_continue:wn \or:

```

```

14567    + 1 \__fp_exp_large:w {1105}{1709}{1807}{5647}{6248}{1171} \or:
14568    + 1 \__fp_exp_large:w {1221}{4027}{5816}{0169}{8339}{2107} \or:
14569    + 1 \__fp_exp_large:w {1349}{8588}{0757}{6003}{1039}{8374} \or:
14570    + 1 \__fp_exp_large:w {1491}{8246}{9764}{1270}{3178}{2485} \or:
14571    + 1 \__fp_exp_large:w {1648}{7212}{7070}{0128}{1468}{4865} \or:
14572    + 1 \__fp_exp_large:w {1822}{1188}{0039}{0508}{9748}{7537} \or:
14573    + 1 \__fp_exp_large:w {2013}{7527}{0747}{0476}{5216}{2455} \or:
14574    + 1 \__fp_exp_large:w {2225}{5409}{2849}{2467}{6045}{7954} \or:
14575    + 1 \__fp_exp_large:w {2459}{6031}{1115}{6949}{6638}{0013} \or:
14576    \fi:
14577    #1;
14578    \__fp_exp_large_after:wwn
14579  }
14580 \cs_new:Npn \__fp_exp_large_after:wwn #1; #2; #3
14581  {
14582    \__fp_exp_Taylor:Nnnwn ? { } { } 0 #2; {} #3
14583    \__fp_fixed_mul:wwn #1;
14584  }

```

(End definition for `__fp_exp_pos_large:NnnNwn` and others.)

28.3 Power

Raising a number a to a power b leads to many distinct situations.

a^b	$-\infty$	$(-\infty, -0)$	$-p/5^k$	± 0	$+p/5^k$	$(0, \infty)$	$+\infty$	<code>NaN</code>
$+\infty$	$+0$		$+0$	$+1$		$+\infty$	$+\infty$	<code>NaN</code>
$(1, \infty)$	$+0$		$+ a ^b$	$+1$		$+ a ^b$	$+\infty$	<code>NaN</code>
$+1$	$+1$		$+1$	$+1$		$+1$	$+1$	$+1$
$(0, 1)$	$+\infty$		$+ a ^b$	$+1$		$+ a ^b$	$+0$	<code>NaN</code>
$+0$	$+\infty$		$+\infty$	$+1$		$+0$	$+0$	<code>NaN</code>
-0	$+\infty$	<code>NaN</code>	$(-1)^p \infty$	$+1$	$(-1)^p 0$	$+0$	$+0$	<code>NaN</code>
$(-1, 0)$	$+\infty$	<code>NaN</code>	$(-1)^p a ^b$	$+1$	$(-1)^p a ^b$	<code>NaN</code>	$+0$	<code>NaN</code>
-1	$+1$	<code>NaN</code>	$(-1)^p$	$+1$	$(-1)^p$	<code>NaN</code>	$+1$	<code>NaN</code>
$(-\infty, -1)$	$+0$	<code>NaN</code>	$(-1)^p a ^b$	$+1$	$(-1)^p a ^b$	<code>NaN</code>	$+\infty$	<code>NaN</code>
$-\infty$	$+0$	$+0$	$(-1)^p 0$	$+1$	$(-1)^p \infty$	<code>NaN</code>	$+\infty$	<code>NaN</code>
<code>NaN</code>	<code>NaN</code>	<code>NaN</code>	<code>NaN</code>	$+1$	<code>NaN</code>	<code>NaN</code>	<code>NaN</code>	<code>NaN</code>

We distinguished in this table the cases of finite (positive or negative) exponents of the form $b = p/q$ with q odd (hence necessarily a power of 5), as $(-1)^{p/q} = (-1)^p$ 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 $+\infty$, 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 $\{ b a \}$, is inserted between a and b). Then do some tests to find the final sign of the result if it exists.

```

14585 \cs_new:cpn { __fp_ \iow_char:N \^ _o:ww }
14586     \s_fp \__fp_chk:w #1#2#3; \s_fp \__fp_chk:w #4#5#6;
14587 {
14588     \if_meaning:w 0 #4
14589         \__fp_case_return_o:Nw \c_one_fp
14590     \fi:
14591     \if_case:w #2 \exp_stop_f:
14592         \exp_after:wN \use_i:nn
14593     \or:
14594         \__fp_case_return_o:Nw \c_nan_fp
14595     \else:
14596         \exp_after:wN \__fp_pow_neg:www
14597         \exp:w \exp_end_continue_f:w \exp_after:wN \use:nn
14598     \fi:
14599 {
14600     \if_meaning:w 1 #1
14601         \exp_after:wN \__fp_pow_normal_o:ww
14602     \else:
14603         \exp_after:wN \__fp_pow_zero_or_inf:ww
14604     \fi:
14605     \s_fp \__fp_chk:w #1#2#3;
14606 }
14607 { \s_fp \__fp_chk:w #4#5#6; \s_fp \__fp_chk:w #1#2#3; }
14608 \s_fp \__fp_chk:w #4#5#6;
14609 }
```

(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 $b = -\infty$.

```

14610 \cs_new:Npn \__fp_pow_zero_or_inf:ww
14611     \s_fp \__fp_chk:w #1#2; \s_fp \__fp_chk:w #3#4
14612 {
14613     \if_meaning:w 1 #4
14614         \__fp_case_return_same_o:w
14615     \fi:
14616     \if_meaning:w #1 #4
14617         \__fp_case_return_o:Nw \c_zero_fp
14618     \fi:
14619     \if_meaning:w 2 #1
14620         \__fp_case_return_o:Nw \c_inf_fp
14621     \fi:
14622     \if_meaning:w 2 #3
14623         \__fp_case_return_o:Nw \c_inf_fp
14624     \else:
14625         \__fp_case_use:nw
```

```

14626   {
14627     \__fp_division_by_zero_o:NNww \c_inf_fp ^
14628       \s_fp \__fp_chk:w #1 #2 ;
14629   }
14630   \fi:
14631   \s_fp \__fp_chk:w #3#4
14632 }
```

(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 .

```

14633 \cs_new:Npn \__fp_pow_normal_o:ww
14634   \s_fp \__fp_chk:w 1 #1#2#3; \s_fp \__fp_chk:w #4#5
14635   {
14636     \if_int_compare:w \__str_if_eq_x:nn { #2 #3 }
14637       { 1 {1000} {0000} {0000} {0000} } = 0 \exp_stop_f:
14638       \if_int_compare:w #4 #1 = 32 \exp_stop_f:
14639         \exp_after:wN \__fp_case_return_ii_o:ww
14640       \fi:
14641       \__fp_case_return_o:Nww \c_one_fp
14642     \fi:
14643     \if_case:w #4 \exp_stop_f:
14644       \or:
14645         \exp_after:wN \__fp_pow_npos_o:Nww
14646         \exp_after:wN #5
14647       \or:
14648         \if_meaning:w 2 #5 \exp_after:wN \reverse_if:N \fi:
14649         \if_int_compare:w #2 > 0 \exp_stop_f:
14650           \exp_after:wN \__fp_case_return_o:Nww
14651           \exp_after:wN \c_inf_fp
14652         \else:
14653           \exp_after:wN \__fp_case_return_o:Nww
14654           \exp_after:wN \c_zero_fp
14655         \fi:
14656       \or:
14657         \__fp_case_return_ii_o:ww
14658       \fi:
14659       \s_fp \__fp_chk:w 1 #1 {#2} #3 ;
14660       \s_fp \__fp_chk:w #4 #5
14661 }
```

(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.

```
14662 \cs_new:Npn \_\_fp\_pow\_npos\_o:Nww #1 \s\_fp \_\_fp\_chk:w 1#2#3
14663 {
14664     \exp_after:wN \_\_fp\_sanitize:Nw
14665     \exp_after:wN
14666     \_\_int\_value:w
14667     \if:w #1 \if_int_compare:w #3 > 0 \exp_stop_f: 0 \else: 2 \fi:
14668         \exp_after:wN \_\_fp\_pow\_npos\_aux:NNnww
14669         \exp_after:wN +
14670         \exp_after:wN \_\_fp\_fixed\_to\_float_o:wN
14671     \else:
14672         \exp_after:wN \_\_fp\_pow\_npos\_aux:NNnww
14673         \exp_after:wN -
14674         \exp_after:wN \_\_fp\_fixed\_inv\_to\_float_o:wN
14675     \fi:
14676     {#3}
14677 }
```

(End definition for $__fp_pow_npos_o:Nww$.)

```
\_\_fp\_pow\_npos\_aux:NNnww
```

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 $-\ln(x)$.

```
14678 \cs_new:Npn \_\_fp\_pow\_npos\_aux:NNnww #1#2#3#4#5; \s\_fp \_\_fp\_chk:w 1#6#7#8;
14679 {
14680     #1
14681     \_\_int\_eval:w
14682         \_\_fp\_ln\_significand:NNNNnnnnN #4#5
14683         \_\_fp\_pow\_exponent:wnN {#3}
14684         \_\_fp\_fixed\_mul:wwn #8 {0000}{0000} ;
14685         \_\_fp\_pow_B:wwN #7;
14686         #1 #2 0 % fixed_to_float_o:wN
14687     }
14688 \cs_new:Npn \_\_fp\_pow\_exponent:wnN #1; #2
14689 {
14690     \if_int_compare:w #2 > 0 \exp_stop_f:
14691         \exp_after:wN \_\_fp\_pow\_exponent:Nwnnnnnw % n\ln(10) - (-\ln(x))
14692         \exp_after:wN +
14693     \else:
14694         \exp_after:wN \_\_fp\_pow\_exponent:Nwnnnnnw % -(|n|\ln(10) + (-\ln(x)))
14695         \exp_after:wN -
14696     \fi:
14697     #2; #1;
14698 }
14699 \cs_new:Npn \_\_fp\_pow\_exponent:Nwnnnnnw #1#2; #3#4#5#6#7#8;
14700 { %^^A todo: use that in ln.
14701     \exp_after:wN \_\_fp\_fixed\_mul\_after:wwn
14702     \_\_int\_value:w \_\_int\_eval:w \c\_fp\_leading\_shift\_int
```

```

14703 \exp_after:wN \_fp_pack>NNNNw
14704 \_int_value:w \_int_eval:w \c_fp_middle_shift_int
14705 #1#2*23025 - #1 #3
14706 \exp_after:wN \_fp_pack>NNNNw
14707 \_int_value:w \_int_eval:w \c_fp_middle_shift_int
14708 #1 #2*8509 - #1 #4
14709 \exp_after:wN \_fp_pack>NNNNw
14710 \_int_value:w \_int_eval:w \c_fp_middle_shift_int
14711 #1 #2*2994 - #1 #5
14712 \exp_after:wN \_fp_pack>NNNNw
14713 \_int_value:w \_int_eval:w \c_fp_middle_shift_int
14714 #1 #2*0456 - #1 #6
14715 \exp_after:wN \_fp_pack>NNNNw
14716 \_int_value:w \_int_eval:w \c_fp_trailing_shift_int
14717 #1 #2*8401 - #1 #7
14718 #1 ( #2*7991 - #8 ) / 1 0000 ; ;
14719 }
14720 \cs_new:Npn \_fp_pow_B:wwN #1#2#3#4#5#6; #7;
14721 {
14722 \if_int_compare:w #7 < 0 \exp_stop_f:
14723 \exp_after:wN \_fp_pow_C_neg:w \_int_value:w -
14724 \else:
14725 \if_int_compare:w #7 < 22 \exp_stop_f:
14726 \exp_after:wN \_fp_pow_C_pos:w \_int_value:w
14727 \else:
14728 \exp_after:wN \_fp_pow_C_overflow:w \_int_value:w
14729 \fi:
14730 \fi:
14731 #7 \exp_after:wN ;
14732 \_int_value:w \_int_eval:w 10 0000 + #1 \_int_eval_end:
14733 #2#3#4#5#6 0000 0000 0000 0000 0000 ; %^^A todo: how many 0?
14734 }
14735 \cs_new:Npn \_fp_pow_C_overflow:w #1; #2; #3
14736 {
14737 + 2 * \c_fp_max_exponent_int
14738 \exp_after:wN \_fp_fixed_continue:wn \c_fp_one_fixed_tl
14739 }
14740 \cs_new:Npn \_fp_pow_C_neg:w #1 ; 1
14741 {
14742 \exp_after:wN \exp_after:wN \exp_after:wN \_fp_pow_C_pack:w
14743 \prg_replicate:nn {#1} {0}
14744 }
14745 \cs_new:Npn \_fp_pow_C_pos:w #1; 1
14746 { \_fp_pow_C_pos_loop:wn #1; }
14747 \cs_new:Npn \_fp_pow_C_pos_loop:wn #1; #2
14748 {
14749 \if_meaning:w 0 #1
14750 \exp_after:wN \_fp_pow_C_pack:w
14751 \exp_after:wN #2
14752 \else:
14753 \if_meaning:w 0 #2
14754 \exp_after:wN \_fp_pow_C_pos_loop:wn \_int_value:w
14755 \else:
14756 \exp_after:wN \_fp_pow_C_overflow:w \_int_value:w

```

```

14757     \fi:
14758     \_int_eval:w #1 - 1 \exp_after:wN ;
14759   \fi:
14760 }
14761 \cs_new:Npn \_fp_pow_C_pack:w
14762   { \exp_after:wN \_fp_exp_large_v:wN \c_fp_one_fixed_tl }

(End definition for \_fp_pow_npos_aux:NNnww.)

```

_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, $(-0.1)^{**}(12345.67)$ will give +0 rather than complaining that the sign is not defined.

```

14763 \cs_new:Npn \_fp_pow_neg:www \s_fp \_fp_chk:w #1#2; #3; #4;
14764   {
14765     \if_case:w \_fp_pow_neg_case:w #4 ;
14766       \exp_after:wN \_fp_pow_neg_aux:wNN
14767     \or:
14768       \if_int_compare:w \_int_eval:w #1 / 2 = 1 \exp_stop_f:
14769         \_fp_invalid_operation_o:Nww ^ #3; #4;
14770         \exp:w \exp_end_continue_f:w
14771         \exp_after:wN \exp_after:wN
14772         \exp_after:wN \_fp_use_none_until_s:w
14773       \fi:
14774     \fi:
14775     \_fp_exp_after_o:w
14776     \s_fp \_fp_chk:w #1#2;
14777   }
14778 \cs_new:Npn \_fp_pow_neg_aux:wNN #1 \s_fp \_fp_chk:w #2#3
14779   {
14780     \exp_after:wN \_fp_exp_after_o:w
14781     \exp_after:wN \s_fp
14782     \exp_after:wN \_fp_chk:w
14783     \exp_after:wN #2
14784     \_int_value:w \_int_eval:w 2 - #3 \_int_eval_end:
14785   }

```

(End definition for _fp_pow_neg:www and _fp_pow_neg_aux:wNN.)

_fp_pow_neg_case:w
_fp_pow_neg_case_aux:nnnn
_fp_pow_neg_case_aux:w

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 divided by some power of 5, 0 if the number is an odd integer divided by some power of 5, 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. The idea is to repeatedly multiply the number by 5 (by halving the mantissa and shifting the exponent) until the mantissa is odd (this can only happen at most 53 times since $2^{54} > 10^{16}$): if the resulting exponent is larger than 16 the parity is even, if it is exactly 16 the parity is odd, and otherwise we should return 1. Of course there is a shortcut: we stop as soon as the exponent exceeds 16.

```

14786 \cs_new:Npn \_fp_pow_neg_case:w \s_fp \_fp_chk:w #1#2#3;
14787   {

```

```

14788     \if_case:w #1 \exp_stop_f:
14789         -1
14790     \or:  \fp_pow_neg_case_aux:nnnnn #3
14791     \or:  -1
14792     \else: 1
14793     \fi:
14794     \exp_stop_f:
14795 }
14796 \cs_new:Npn \fp_pow_neg_case_aux:nnnnn #1#2#3#4#5
14797 { \fp_pow_neg_case_aux:w #1 ; #2 #3 ; #4 #5 ; }
14798 \cs_new:Npn \fp_pow_neg_case_aux:w #1 ; #2 ; #3 ;
14799 {
14800     \if_int_compare:w #1 > \c_fp_prec_int
14801         -1
14802     \else:
14803         \if_int_odd:w #3 \exp_stop_f:
14804             \if_int_compare:w #1 = \c_fp_prec_int
14805                 0
14806             \else:
14807                 1
14808             \fi:
14809         \else:
14810             \exp_after:wN \fp_pow_neg_case_aux:w
14811             \int_value:w \int_eval:w #1 + 1 \exp_after:wN ;
14812             \int_value:w \int_eval:w (#2 + 1) / 2 - 1 \exp_after:wN ;
14813             \int_value:w \int_eval:w
14814             \if_int_odd:w #2 \exp_stop_f: 5000 0000 + \fi: #3 / 2 ;
14815         \fi:
14816     \fi:
14817 }

```

(End definition for `\fp_pow_neg_case:w`, `\fp_pow_neg_case_aux:nnnnn`, and `\fp_pow_neg_case_aux:w`.)

```

14818 </initex | package>

```

29 I3fp-trig Implementation

```

14819 <*initex | package>
14820 <@=fp>

\fp_parse_word_acos:N Unary functions.
\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

```

```

14833     \cs_new:cpx { __fp_parse_word_#1d:N }
14834     {
14835         \exp_not:N \__fp_parse_unary_function>NNN
14836         \exp_not:c { __fp_#1_o:w }
14837         \exp_not:N \use_i:nn
14838     }
14839 }
```

(End definition for `__fp_parse_word_acos:N` and others.)

`__fp_parse_word_acot:N`

```

14840 \cs_new:Npn \__fp_parse_word_acot:N
14841   { \__fp_parse_function>NNNN \__fp_acot_o:Nw \use_i:nn }
14842 \cs_new:Npn \__fp_parse_word_acotd:N
14843   { \__fp_parse_function>NNNN \__fp_acot_o:Nw \use_i:nn }
14844 \cs_new:Npn \__fp_parse_word_atan:N
14845   { \__fp_parse_function>NNNN \__fp_atan_o:Nw \use_i:nn }
14846 \cs_new:Npn \__fp_parse_word_atand:N
14847   { \__fp_parse_function>NNNN \__fp_atan_o:Nw \use_i:nn }
```

(End definition for `__fp_parse_word_acot:N` and others.)

29.1 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.

29.1.1 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_i: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` will be 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:ww` which converts the result of the series to a floating point directly rather than taking its inverse, since $\sin(x) = \#3 \sin|x|$.

```

14848 \cs_new:Npn \__fp_sin_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
14849   {
14850     \if_case:w #2 \exp_stop_f:
14851       \__fp_case_return_same_o:w
14852     \or: \__fp_case_use:nw
14853       {
14854         \__fp_trig>NNNNNwn #1 \__fp_sin_series_o:NNwwww
14855         \__fp_ep_to_float_o:wwN #3 0
14856       }
14857     \or: \__fp_case_use:nw
14858       { \__fp_invalid_operation_o:fw { #1 { sin } { sind } } }
14859     \else: \__fp_case_return_same_o:w
14860     \fi:
14861     \s__fp \__fp_chk:w #2 #3 #4;
14862   }

```

(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 will 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 $\cos(x) = +\sin(\pi/2 + |x|)$.

```

14863 \cs_new:Npn \__fp_cos_o:w #1 \s__fp \__fp_chk:w #2#3; @
14864   {
14865     \if_case:w #2 \exp_stop_f:
14866       \__fp_case_return_o:Nw \c_one_fp
14867     \or: \__fp_case_use:nw
14868       {
14869         \__fp_trig>NNNNNwn #1 \__fp_sin_series_o:NNwwww
14870         \__fp_ep_to_float_o:wwN 0 2
14871       }
14872     \or: \__fp_case_use:nw
14873       { \__fp_invalid_operation_o:fw { #1 { cos } { cosd } } }
14874     \else: \__fp_case_return_same_o:w
14875     \fi:
14876     \s__fp \__fp_chk:w #2 #3;
14877   }

```

(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 $\csc(x) = \#3(\sin|x|)^{-1}$.

```

14878 \cs_new:Npn \__fp_csc_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
14879   {
14880     \if_case:w #2 \exp_stop_f:
14881       \__fp_cot_zero_o:Nfw #3 { #1 { csc } { cscd } }
14882     \or: \__fp_case_use:nw
14883       {

```

```

14884           \_fp_trig:NNNNNwn #1 \_fp_sin_series_o:NNwwww
14885           \_fp_ep_inv_to_float_o:wwN #3 0
14886       }
14887   \or:   \_fp_case_use:nw
14888       { \_fp_invalid_operation_o:fw { #1 { csc } { csed } } }
14889   \else: \_fp_case_return_same_o:w
14890   \fi:
14891   \s_fp \_fp_chk:w #2 #3 #4;
14892 }

```

(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 $\sec(x) = +1/\sin(\pi/2 + |x|)$.

```

14893 \cs_new:Npn \_fp_sec_o:w #1 \s_fp \_fp_chk:w #2#3; @
14894 {
14895     \if_case:w #2 \exp_stop_f:
14896         \_fp_case_return_o:Nw \c_one_fp
14897     \or:   \_fp_case_use:nw
14898         {
14899             \_fp_trig:NNNNNwn #1 \_fp_sin_series_o:NNwwww
14900             \_fp_ep_inv_to_float_o:wwN 0 2
14901         }
14902     \or:   \_fp_case_use:nw
14903         { \_fp_invalid_operation_o:fw { #1 { sec } { secd } } }
14904     \else: \_fp_case_return_same_o:w
14905     \fi:
14906     \s_fp \_fp_chk:w #2 #3;
14907 }

```

(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 $_fp_tan_series_o:NNwwww$, 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.

```

14908 \cs_new:Npn \_fp_tan_o:w #1 \s_fp \_fp_chk:w #2#3#4; @
14909 {
14910     \if_case:w #2 \exp_stop_f:
14911         \_fp_case_return_same_o:w
14912     \or:   \_fp_case_use:nw
14913         {
14914             \_fp_trig:NNNNNwn #1
14915             \_fp_tan_series_o:NNwwww 0 #3 1
14916         }
14917     \or:   \_fp_case_use:nw
14918         { \_fp_invalid_operation_o:fw { #1 { tan } { tand } } }
14919     \else: \_fp_case_return_same_o:w
14920     \fi:
14921     \s_fp \_fp_chk:w #2 #3 #4;
14922 }

```

(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.

```

14923 \cs_new:Npn \_fp_cot_o:w #1 \s_fp \_fp_chk:w #2#3#4; @
14924   {
14925     \if_case:w #2 \exp_stop_f:
14926       \_fp_cot_zero_o:Nfw #3 { #1 { cot } { cotd } }
14927     \or: \_fp_case_use:nw
14928       {
14929         \_fp_trig:NNNNNwn #1
14930         \_fp_tan_series_o:NNwww 2 #3 3
14931       }
14932     \or: \_fp_case_use:nw
14933       { \_fp_invalid_operation_o:fw { #1 { cot } { cotd } } }
14934   \else: \_fp_case_return_same_o:w
14935   \fi:
14936   \s_fp \_fp_chk:w #2 #3 #4;
14937 }
14938 \cs_new:Npn \_fp_cot_zero_o:Nfw #1#2#3 \fi:
14939   {
14940     \fi:
14941     \token_if_eq_meaning:NNTF 0 #1
14942       { \exp_args:NNf \_fp_division_by_zero_o:Nnw \c_inf_fp }
14943       { \exp_args:NNf \_fp_division_by_zero_o:Nnw \c_minus_inf_fp }
14944     {#2}
14945   }

```

(End definition for `_fp_cot_o:w` and `_fp_cot_zero_o:Nfw`.)

29.1.2 Distinguishing small and large arguments

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.

```

14946 \cs_new:Npn \__fp_trig:NNNNNwn #1#2#3#4#5 \s__fp \__fp_chk:w 1#6#7#8;
14947 {
14948     \exp_after:wN #2
14949     \exp_after:wN #3
14950     \exp_after:wN #4
14951     \__int_value:w \__int_eval:w #5
14952     \exp_after:wN \exp_after:wN \exp_after:wN \exp_after:wN
14953     \if_int_compare:w #7 > #1 0 1 \exp_stop_f:
14954         #1 \__fp_trig_large:ww \__fp_trigd_large:ww
14955     \else:
14956         #1 \__fp_trig_small:ww \__fp_trigd_small:ww
14957     \fi:
14958     #7 ,#8{0000}{0000};
14959 }
```

(End definition for `__fp_trig:NNNNNwn`.)

29.1.3 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 will be 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.

```

14960 \cs_new:Npn \__fp_trig_small:ww #1,#2;
14961 { \__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.

```

14962 \cs_new:Npn \__fp_trigd_small:ww #1,#2;
14963 {
14964     \__fp_ep_mul_raw:wwwwN
14965     -1,{1745}{3292}{5199}{4329}{5769}{2369}; #1,#2;
14966     \__fp_trig_small:ww
14967 }
```

(End definition for `__fp_trigd_small:ww`.)

29.1.4 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 $\text{__fp_trigd_small:ww}$. 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).

```

14968 \cs_new:Npn \_\_fp_trigd_large:ww #1, #2#3#4#5#6#7;
14969 {
14970     \exp_after:wN \_\_fp_pack_eight:wNNNNNNNN
14971     \exp_after:wN \_\_fp_pack_eight:wNNNNNNNN
14972     \exp_after:wN \_\_fp_pack_twice_four:wNNNNNNNN
14973     \exp_after:wN \_\_fp_pack_twice_four:wNNNNNNNN
14974     \exp_after:wN \_\_fp_trigd_large_auxi:nnnnwNNNN
14975     \exp_after:wN ;
14976     \exp:w \exp_end_continue_f:w
14977     \prg_replicate:nn { \int_max:nn { 22 - #1 } { 0 } } { 0 }
14978     #2#3#4#5#6#7 0000 0000 0000 !
14979 }
14980 \cs_new:Npn \_\_fp_trigd_large_auxi:nnnnwNNNN #1#2#3#4#5; #6#7#8#9
14981 {
14982     \exp_after:wN \_\_fp_trigd_large_auxii:wNw
14983     \__int_value:w \__int_eval:w #1 + #2
14984     - (#1 + #2 - 4) / 9 * 9 \__int_eval_end:
14985     #3;
14986     #4; #5{#6#7#8#9};
14987 }
14988 \cs_new:Npn \_\_fp_trigd_large_auxii:wNw #1; #2#3;
14989 {
14990     + (#1#2 - 4) / 9 * 2
14991     \exp_after:wN \_\_fp_trigd_large_auxiii:www
14992     \__int_value:w \__int_eval:w #1#2
14993     - (#1#2 - 4) / 9 * 9 \__int_eval_end: #3 ;
14994 }
14995 \cs_new:Npn \_\_fp_trigd_large_auxiii:www #1; #2; #3!
14996 {
14997     \if_int_compare:w #1 < 4500 \exp_stop_f:
14998         \exp_after:wN \_\_fp_use_i_until_s:nw
14999         \exp_after:wN \_\_fp_fixed_continue:wn
15000     \else:
15001         + 1
15002     \fi:
15003     \_\_fp_fixed_sub:wwn {9000}{0000}{0000}{0000}{0000}{0000};
15004     {#1}#2{0000}{0000};
15005     { \_\_fp_trigd_small:ww 2, }
15006 }

```

(End definition for $\text{__fp_trigd_large:ww}$ and others.)

29.1.5 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.

`_fp_trig_inverse_two_pi:` This macro expands to `, , !` or `, !` followed by 10112 decimals of $10^{-16}/(2\pi)$. The number of decimals we really need is the maximum exponent plus the number of digits we will need later, 52, plus 12 (4 – 1 groups of 4 digits). We store the decimals as a control sequence name, and convert it to a token list when required: strings take up less memory than their token list representation.

```

15007 \cs_new:Npx \_fp_trig_inverse_two_pi:
15008 {
15009   \exp_not:n { \exp_after:wN \use_none:n \token_to_str:N }
15010   \cs:w , , !
15011   0000000000000000159154943091895335768883763372514362034459645740 ~
15012   4564487476673440588967976342265350901138027662530859560728427267 ~
15013   5795803689291184611457865287796741073169983922923996693740907757 ~
15014   3077746396925307688717392896217397661693362390241723629011832380 ~
15015   1142226997557159404618900869026739561204894109369378440855287230 ~
15016   9994644340024867234773945961089832309678307490616698646280469944 ~
15017   8652187881574786566964241038995874139348609983868099199962442875 ~
15018   585171178858431117518761605465475369880097394603647593337680593 ~
15019   0249449663530532715677550322032477781639716602294674811959816584 ~
15020   0606016803035998133911987498832786654435279755070016240677564388 ~
15021   8495713108801221993761476813777647378906330680464579784817613124 ~
15022   2731406996077502450029775985708905690279678513152521001631774602 ~
15023   0924811606240561456203146484089248459191435211575407556200871526 ~
15024   6068022171591407574745827225977462853998751553293908139817724093 ~
15025   5825479707332871904069997590765770784934703935898280871734256403 ~
15026   6689511662545705943327631268650026122717971153211259950438667945 ~
15027   0376255608363171169525975812822494162333431451061235368785631136 ~
15028   3669216714206974696012925057833605311960859450983955671870995474 ~
15029   6510431623815517580839442979970999505254387566129445883306846050 ~
15030   7852915151410404892988506388160776196993073410389995786918905980 ~
15031   9373777206187543222718930136625526123878038753888110681406765434 ~
15032   0828278526933426799556070790386060352738996245125995749276297023 ~

```

15033	5940955843011648296411855777124057544494570217897697924094903272	~
15034	9477021664960356531815354400384068987471769158876319096650696440	~
15035	4776970687683656778104779795450353395758301881838687937766124814	~
15036	9530599655802190835987510351271290432315804987196868777594656634	~
15037	6221034204440855497850379273869429353661937782928735937843470323	~
15038	0237145837923557118636341929460183182291964165008783079331353497	~
15039	7909974586492902674506098936890945883050337030538054731232158094	~
15040	3197676032283131418980974982243833517435698984750103950068388003	~
15041	9786723599608024002739010874954854787923568261139948903268997427	~
15042	0834961149208289037767847430355045684560836714793084567233270354	~
15043	8539255620208683932409956221175331839402097079357077496549880868	~
15044	6066360968661967037474542102831219251846224834991161149566556037	~
15045	9696761399312829960776082779901007830360023382729879085402387615	~
15046	5744543092601191005433799838904654921248295160707285300522721023	~
15047	6017523313173179759311050328155109373913639645305792607180083617	~
15048	9548767246459804739772924481092009371257869183328958862839904358	~
15049	6866663975673445140950363732719174311388066383072592302759734506	~
15050	0548212778037065337783032170987734966568490800326988506741791464	~
15051	6835082816168533143361607309951498531198197337584442098416559541	~
15052	5225064339431286444038388356150879771645017064706751877456059160	~
15053	8716857857939226234756331711132998655941596890719850688744230057	~
15054	5191977056900382183925622033874235362568083541565172971088117217	~
15055	9593683256488518749974870855311659830610139214454460161488452770	~
15056	2511411070248521739745103866736403872860099674893173561812071174	~
15057	0478899368886556923078485023057057144063638632023685201074100574	~
15058	8592281115721968003978247595300166958522123034641877365043546764	~
15059	6456565971901123084767099309708591283646669191776938791433315566	~
15060	5066981321641521008957117286238426070678451760111345080069947684	~
15061	2235698962488051577598095339708085475059753626564903439445420581	~
15062	7886435683042000315095594743439252544850674914290864751442303321	~
15063	3324569511634945677539394240360905438335528292434220349484366151	~
15064	4663228602477666660495314065734357553014090827988091478669343492	~
15065	2737602634997829957018161964321233140475762897484082891174097478	~
15066	2637899181699939487497715198981872666294601830539583275209236350	~
15067	685389228468247259972528300766856937583659722919824429747406163	~
15068	8183113958306744348516928597383237392662402434501997809940402189	~
15069	6134834273613676449913827154166063424829363741850612261086132119	~
15070	9863346284709941839942742955915628333990480382117501161211667205	~
15071	1912579303552929241134403116134112495318385926958490443846807849	~
15072	0973982808855297045153053991400988698840883654836652224668624087	~
15073	2540140400911787421220452307533473972538149403884190586842311594	~
15074	6322744339066125162393106283195323883392131534556381511752035108	~
15075	7459558201123754359768155340187407394340363397803881721004531691	~
15076	8295194879591767395417787924352761740724605939160273228287946819	~
15077	3649128949714953432552723591659298072479985806126900733218844526	~
15078	7943350455801952492566306204876616134365339902087545208555344144	~
15079	0990512982727454659118132223284051166615650709837557433729548631	~
15080	2041121716380915606161165732000083306114606181280326258695951602	~
15081	4632166138576614804719932707771316441201594960110632830520759583	~
15082	4850305079095584982982186740289838551383239570208076397550429225	~
15083	9847647071016426974384504309165864528360324933604354657237557916	~
15084	1366324120457809969715663402215880545794313282780055246132088901	~
15085	8742121092448910410052154968097113720754005710963406643135745439	~
15086	915976943578892079342561778302237011486424925239248728713132021	~

15087	7667360756645598272609574156602343787436291321097485897150713073	~
15088	9104072643541417970572226547980381512759579124002534468048220261	~
15089	7342299001020483062463033796474678190501811830375153802879523433	~
15090	4195502135689770912905614317878792086205744999257897569018492103	~
15091	2420647138519113881475640209760554895793785141404145305151583964	~
15092	2823265406020603311891586570272086250269916393751527887360608114	~
15093	55694842103224077727241651364234366992716340309405307480652685	~
15094	0930165892136921414312937134106157153714062039784761842650297807	~
15095	8606266969960809184223476335047746719017450451446166382846208240	~
15096	8673595102371302904443779408535034454426334130626307459513830310	~
15097	2293146934466832851766328241515210179422644395718121717021756492	~
15098	196444939653222187658488244511909401340504432139858628621083179	~
15099	3939608443898019147873897723310286310131486955212620518278063494	~
15100	5711866277825659883100535155231665984394090221806314454521212978	~
15101	9734471488741258268223860236027109981191520568823472398358013366	~
15102	0683786328867928619732367253606685216856320119489780733958419190	~
15103	6659583867852941241871821727987506103946064819585745620060892122	~
15104	8416394373846549589932028481236433466119707324309545859073361878	~
15105	6290631850165106267576851216357588696307451999220010776676830946	~
15106	9814975622682434793671310841210219520899481912444048751171059184	~
15107	4139907889455775184621619041530934543802808938628073237578615267	~
15108	7971143323241969857805637630180884386640607175368321362629671224	~
15109	2609428540110963218262765120117022552929289655594608204938409069	~
15110	0760692003954646191640021567336017909631872891998634341086903200	~
15111	5796637103128612356988817640364252540837098108148351903121318624	~
15112	7228181050845123690190646632235938872454630737272808789830041018	~
15113	9485913673742589418124056729191238003306344998219631580386381054	~
15114	2457893450084553280313511884341007373060595654437362488771292628	~
15115	9807423539074061786905784443105274262641767830058221486462289361	~
15116	9296692992033046693328438158053564864073184440599549689353773183	~
15117	6726613130108623588021288043289344562140479789454233736058506327	~
15118	0439981932635916687341943656783901281912202816229500333012236091	~
15119	8587559201959081224153679499095448881099758919890811581163538891	~
15120	6339402923722049848375224236209100834097566791710084167957022331	~
15121	7897107102928884897013099533995424415335060625843921452433864640	~
15122	343244065731747755340540448100617761256908474646143297654390008	~
15123	382652114521016236643119798731902751191441213616962045693602633	~
15124	6102355962140467029012156796418735746835873172331004745963339773	~
15125	2477044918885134415363760091537564267438450166221393719306748706	~
15126	2881595464819775192207710236743289062690709117919412776212245117	~
15127	2354677115640433357720616661564674474627305622913332030953340551	~
15128	3841718194605321501426328000879551813296754972846701883657425342	~
15129	5016994231069156343106626043412205213831587971115075454063290657	~
15130	0248488648697402872037259869281149360627403842332874942332178578	~
15131	7750735571857043787379693402336902911446961448649769719434527467	~
15132	4429603089437192540526658890710662062575509930379976658367936112	~
15133	8137451104971506153783743579555867972129358764463093757203221320	~
15134	2460565661129971310275869112846043251843432691552928458573495971	~
15135	5042565399302112184947232132380516549802909919676815118022483192	~
15136	5127372199792134331067642187484426215985121676396779352982985195	~
15137	8545392106957880586853123277545433229161989053189053725391582222	~
15138	9232597278133427818256064882333760719681014481453198336237910767	~
15139	1255017528826351836492103572587410356573894694875444694018175923	~
15140	060937082814650185742532496921276462427832210765473750568198834	~

```

15141      5641035458027261252285503154325039591848918982630498759115406321 ~
15142      0354263890012837426155187877318375862355175378506956599570028011 ~
15143      5841258870150030170259167463020842412449128392380525772514737141 ~
15144      2310230172563968305553583262840383638157686828464330456805994018 ~
15145      7001071952092970177990583216417579868116586547147748964716547948 ~
15146      8312140431836079844314055731179349677763739898930227765607058530 ~
15147      4083747752640947435070395214524701683884070908706147194437225650 ~
15148      2823145872995869738316897126851939042297110721350756978037262545 ~
15149      8141095038270388987364516284820180468288205829135339013835649144 ~
15150      3004015706509887926715417450706686888783438055583501196745862340 ~
15151      8059532724727843829259395771584036885940989939255241688378793572 ~
15152      7967951654076673927031256418760962190243046993485989199060012977 ~
15153      7469214532970421677817261517850653008552559997940209969455431545 ~
15154      2745856704403686680428648404512881182309793496962721836492935516 ~
15155      2029872469583299481932978335803459023227052612542114437084359584 ~
15156      944338363838831775184116088171125127923374577219339820819005406 ~
15157      3292937775306906607415304997682647124407768817248673421685881509 ~
15158      9133422075930947173855159340808957124410634720893194912880783576 ~
15159      3115829400549708918023366596077070927599010527028150868897828549 ~
15160      4340372642729262103487013992868853550062061514343078665396085995 ~
15161      00587149391416520653020700852656240747036605333805263766757 ~
15162      2018839497277047222153633851135483463624619855425993871933367482 ~
15163      0422097449956672702505446423243957506869591330193746919142980999 ~
15164      3424230550172665212092414559625960554427590951996824313084279693 ~
15165      7113207021049823238195747175985519501864630940297594363194450091 ~
15166      9150616049228764323192129703446093584259267276386814363309856853 ~
15167      2786024332141052330760658841495858718197071242995959226781172796 ~
15168      4438853796763139274314227953114500064922126500133268623021550837
15169      \cs_end:
15170      }

```

(End definition for `_fp_trig_inverse_two_pi`.)

The exponent #1 is between 1 and 10000. We discard the integer part of $10^{\#1-16}/(2\pi)$, that is, the first #1 digits of $10^{-16}/(2\pi)$, because it yields an integer contribution to $x/(2\pi)$. The `auxii` auxiliary discards 64 digits at a time thanks to spaces inserted in the result of `_fp_trig_inverse_two_pi`, while `auxiii` discards 8 digits at a time, and `auxiv` discards digits one at a time. Then 64 digits are packed into groups of 4 and the `auxv` auxiliary is called.

```

15171  \cs_new:Npn \_fp_trig_large:ww #1, #2#3#4#5#6;
15172  {
15173    \exp_after:wN \_fp_trig_large_auxi:wwwww
15174    \int_value:w \int_eval:w (#1 - 32) / 64 \exp_after:wN ,
15175    \int_value:w \int_eval:w (#1 - 4) / 8 \exp_after:wN ,
15176    \int_value:w #1 \_fp_trig_inverse_two_pi: ;
15177    {#2}{#3}{#4}{#5} ;
15178  }
15179  \cs_new:Npn \_fp_trig_large_auxi:wwwww #1, #2, #3, #4!
15180  {
15181    \prg_replicate:nn {#1} { \_fp_trig_large_auxii:ww }
15182    \prg_replicate:nn {#2 - #1 * 8}
15183      { \_fp_trig_large_auxiii:wwNNNNNNNN }
15184    \prg_replicate:nn {#3 - #2 * 8}
15185      { \_fp_trig_large_auxiv:wN }

```

```

15186     \prg_replicate:nn { 8 } { \__fp_pack_twice_four:wNNNNNNNNN }
15187     \__fp_trig_large_auxv:www
15188     ;
15189 }
15190 \cs_new:Npn \__fp_trig_large_auxii:ww #1; #2 ~ { #1; }
15191 \cs_new:Npn \__fp_trig_large_auxiii:wNNNNNNNN
15192   #1; #2#3#4#5#6#7#8#9 { #1; }
15193 \cs_new:Npn \__fp_trig_large_auxiv:wN #1; #2 { #1; }

```

(End definition for `__fp_trig_large:ww` and others.)

```

\__fp_trig_large_auxv:www
  \__fp_trig_large_auxvi:wNNNNNNNN
\__fp_trig_large_pack:NNNNNw

```

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 some 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:wwn`), 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.

```

15194 \cs_new:Npn \__fp_trig_large_auxv:www #1; #2; #3;
15195   {
15196     \exp_after:wN \__fp_use_i_until_s:nw
15197     \exp_after:wN \__fp_trig_large_auxvii:w
15198     \__int_value:w \__int_eval:w \c_fp_leading_shift_int
15199     \prg_replicate:nn { 13 }
15200       { \__fp_trig_large_auxvi:wNNNNNNNN }
15201       + \c_fp_trailing_shift_int - \c_fp_middle_shift_int
15202     \__fp_use_i_until_s:nw
15203     ; #3 #1 ; ;
15204   }
15205 \cs_new:Npn \__fp_trig_large_auxvi:wNNNNNNNN #1; #2#3#4#5#6#7#8#9
15206   {
15207     \exp_after:wN \__fp_trig_large_pack:NNNNNw
15208     \__int_value:w \__int_eval:w \c_fp_middle_shift_int
15209     + #2*#9 + #3*#8 + #4*#7 + #5*#6
15210     #1; {#2}{#3}{#4}{#5} {#7}{#8}{#9}
15211   }
15212 \cs_new:Npn \__fp_trig_large_pack:NNNNNw #1#2#3#4#5#6;
15213   { + #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
\__fp_trig_large_auxviii:w
\__fp_trig_large_auxix:Nw
\__fp_trig_large_auxx:wNNNNNN
\__fp_trig_large_auxxi: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.

```

15214 \cs_new:Npn \_fp_trig_large_auxvii:w #1#2#3
15215 {
15216   \exp_after:wN \_fp_trig_large_auxviii:ww
15217   \int_value:w \int_eval:w (#1#2#3 - 62) / 125 ;
15218   #1#2#3
15219 }
15220 \cs_new:Npn \_fp_trig_large_auxviii:ww #1;
15221 {
15222   + #1
15223   \if_int_odd:w #1 \exp_stop_f:
15224     \exp_after:wN \_fp_trig_large_auxix:Nw
15225     \exp_after:wN -
15226   \else:
15227     \exp_after:wN \_fp_trig_large_auxix:Nw
15228     \exp_after:wN +
15229   \fi:
15230 }
15231 \cs_new:Npn \_fp_trig_large_auxix:Nw
15232 {
15233   \exp_after:wN \_fp_use_i_until_s:nw
15234   \exp_after:wN \_fp_trig_large_auxxi:w
15235   \int_value:w \int_eval:w \c_fp_leading_shift_int
15236   \prg_replicate:nn { 13 }
15237   { \_fp_trig_large_auxx:wNNNNN }
15238   + \c_fp_trailing_shift_int - \c_fp_middle_shift_int
15239 ;
15240 }
15241 \cs_new:Npn \_fp_trig_large_auxx:wNNNNN #1; #2 #3#4#5#6
15242 {
15243   \exp_after:wN \_fp_trig_large_pack:NNNNNw
15244   \int_value:w \int_eval:w \c_fp_middle_shift_int
15245   #2 8 * #3#4#5#6
15246   #1; #2
15247 }
15248 \cs_new:Npn \_fp_trig_large_auxxi:w #1;
15249 {
15250   \exp_after:wN \_fp_ep_mul_raw:wwwwN
15251   \int_value:w \int_eval:w 0 \_fp_ep_to_ep_loop:N #1 ; ; !
15252   0,{7853}{9816}{3397}{4483}{0961}{5661};
15253   \_fp_trig_small:ww
15254 }
```

(End definition for `_fp_trig_large_auxvii:w` and others.)

29.1.6 Computing the power series

Here we receive a conversion function `_fp_ep_to_float_o:wwN` or `_fp_ep_inv_to_float_o:wwN`, a $\langle sign \rangle$ (0 or 2), a (non-negative) $\langle octant \rangle$ delimited by a dot, a $\langle fixed point \rangle$ 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 will control 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_sanitze:Nw` checks for overflow and underflow.

```

15255 \cs_new:Npn \__fp_sin_series_o:NNwww #1#2#3. #4;
15256   {
15257     \__fp_fixed_mul:wwn #4; #4;
15258     {
15259       \exp_after:wN \__fp_sin_series_aux_o:NNwww
15260       \exp_after:wN #1
15261       \int_value:w
15262         \if_int_odd:w \int_eval:w (#3 + 2) / 4 \int_eval_end:
15263           #2
15264         \else:
15265           \if_meaning:w #2 0 2 \else: 0 \fi:
15266         \fi:
15267         {#3}
15268     }
15269   }
15270 \cs_new:Npn \__fp_sin_series_aux_o:NNwww #1#2#3 #4; #5,#6;
15271   {
15272     \if_int_odd:w \int_eval:w #3 / 2 \int_eval_end:
15273       \exp_after:wN \use_i:nn
15274     \else:
15275       \exp_after:wN \use_ii:nn
15276     \fi:
15277     { % 1/18!
15278       \__fp_fixed_mul_sub_back:wwnn {0000}{0000}{0000}{0001}{5619}{2070};
15279                     #4;{0000}{0000}{0000}{0477}{9477}{3324};
15280       \__fp_fixed_mul_sub_back:wwnn #4;{0000}{0000}{0011}{4707}{4559}{7730};
15281       \__fp_fixed_mul_sub_back:wwnn #4;{0000}{0000}{2087}{6756}{9878}{6810};
15282       \__fp_fixed_mul_sub_back:wwnn #4;{0000}{0027}{5573}{1922}{3985}{8907};
15283       \__fp_fixed_mul_sub_back:wwnn #4;{0000}{2480}{1587}{3015}{8730}{1587};
15284       \__fp_fixed_mul_sub_back:wwnn #4;{0013}{8888}{8888}{8888}{8889};
15285       \__fp_fixed_mul_sub_back:wwnn #4;{0416}{6666}{6666}{6666}{6667};
15286       \__fp_fixed_mul_sub_back:wwnn #4;{5000}{0000}{0000}{0000}{0000};

```

```

15287     \_fp_fixed_mul_sub_back:wwwn#4;{10000}{0000}{0000}{0000}{0000}{0000};
15288     { \_fp_fixed_continue:wn 0, }
15289 }
15290 { % 1/17!
15291     \_fp_fixed_mul_sub_back:wwwn {0000}{0000}{0000}{0028}{1145}{7254};
15292             #4;{0000}{0000}{0000}{7647}{1637}{3182};
15293     \_fp_fixed_mul_sub_back:wwwn #4;{0000}{0000}{0160}{5904}{3836}{8216};
15294     \_fp_fixed_mul_sub_back:wwwn #4;{0000}{0002}{5052}{1083}{8544}{1719};
15295     \_fp_fixed_mul_sub_back:wwwn #4;{0000}{0275}{5731}{9223}{9858}{9065};
15296     \_fp_fixed_mul_sub_back:wwwn #4;{0001}{9841}{2698}{4126}{9841}{2698};
15297     \_fp_fixed_mul_sub_back:wwwn #4;{0083}{3333}{3333}{3333}{3333}{3333};
15298     \_fp_fixed_mul_sub_back:wwwn #4;{1666}{6666}{6666}{6666}{6666}{6667};
15299     \_fp_fixed_mul_sub_back:wwwn#4;{10000}{0000}{0000}{0000}{0000}{0000};
15300     { \_fp_ep_mul:wwwn 0, } #5,#6;
15301 }
15302 {
15303     \exp_after:wN \_fp_sanitize:Nw
15304     \exp_after:wN #2
15305     \_int_value:w \_int_eval:w #1
15306 }
15307 #2
15308 }

```

(End definition for `_fp_sin_series_o:NNwww` and `_fp_sin_series_aux_o:NNwww`.)

`_fp_tan_series_o:NNwww`
`_fp_tan_series_aux_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

$$\tan(x) \simeq \frac{x(1 - x^2(a_1 - x^2(a_2 - x^2(a_3 - x^2(a_4 - x^2a_5)))))}{1 - x^2(b_1 - x^2(b_2 - x^2(b_3 - x^2(b_4 - x^2b_5))))}.$$

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.

```

15309 \cs_new:Npn \_fp_tan_series_o:NNwww #1#2#3. #4;
15310 {
15311     \_fp_fixed_mul:wwn #4; #4;
15312 {
15313     \exp_after:wN \_fp_tan_series_aux_o:Nnwww
15314     \_int_value:w
15315     \if_int_odd:w \_int_eval:w #3 / 2 \_int_eval_end:
15316         \exp_after:wN \reverse_if:N
15317     \fi:

```

```

15318      \if_meaning:w #1#2 2 \else: 0 \fi:
15319      {#3}
15320    }
15321  }
15322 \cs_new:Npn \__fp_tan_series_aux_o:Nnwww #1 #2 #3; #4,#5;
15323  {
15324    \__fp_fixed_mul_sub_back:wwnn {0000}{0000}{1527}{3493}{0856}{7059};
15325          #3; {0000}{0159}{6080}{0274}{5257}{6472};
15326    \__fp_fixed_mul_sub_back:wwnn #3; {0002}{4571}{2320}{0157}{2558}{8481};
15327    \__fp_fixed_mul_sub_back:wwnn #3; {0115}{5830}{7533}{5397}{3168}{2147};
15328    \__fp_fixed_mul_sub_back:wwnn #3; {1929}{8245}{6140}{3508}{7719}{2982};
15329    \__fp_fixed_mul_sub_back:wwnn #3; {10000}{0000}{0000}{0000}{0000}{0000};
15330    { \__fp_ep_mul:wwwn 0, } #4,#5;
15331  {
15332    \__fp_fixed_mul_sub_back:wwnn {0000}{0007}{0258}{0681}{9408}{4706};
15333          #3; {0000}{1243}{7175}{1399}{6151}{7670};
15334    \__fp_fixed_mul_sub_back:wwnn #3; {0019}{2638}{4588}{9232}{8861}{3691};
15335    \__fp_fixed_mul_sub_back:wwnn #3; {0536}{6357}{0691}{4344}{6852}{4252};
15336    \__fp_fixed_mul_sub_back:wwnn #3; {5263}{1578}{9473}{6842}{1052}{6315};
15337    \__fp_fixed_mul_sub_back:wwnn#3; {10000}{0000}{0000}{0000}{0000}{0000};
15338  {
15339    \reverse_if:N \if_int_odd:w
15340        \__int_eval:w (#2 - 1) / 2 \__int_eval_end:
15341        \exp_after:wN \__fp_reverse_args:Nww
15342    \fi:
15343    \__fp_ep_div:wwwn 0,
15344  }
15345  {
15346    \exp_after:wN \__fp_sanitize:Nw
15347    \exp_after:wN #1
15348    \__int_value:w \__int_eval:w \__fp_ep_to_float_o:wwN
15349  }
15350  }
15351  #
15352 }

```

(End definition for `__fp_tan_series_o:NNwww` and `__fp_tan_series_aux_o:Nnwww`.)

29.2 Inverse trigonometric functions

All inverse trigonometric functions (arcsine, arccosine, arctangent, arccotangent, arcosecant, and arcsecant) are based on a function often denoted `atan2`. This function 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 will give 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 will 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.

29.2.1 Arctangent and arccotangent

`__fp_atan_o:Nw`
`__fp_acot_o:Nw`
`__fp_atan_dispatch_o>NNnNw`

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. Here, we dispatch according to the number of arguments. The one-argument versions of arctangent and arccotangent are special cases of the two-argument ones: $\text{atan}(y) = \text{atan}(y, 1) = \text{acot}(1, y)$ and $\text{acot}(x) = \text{atan}(1, x) = \text{acot}(x, 1)$.

15353 `\cs_new:Npn __fp_atan_o:Nw`

```

15354  {
15355    \_fp_atan_dispatch_o:NNnNw
15356      \_fp_acotii_o:Nww \_fp_atanii_o:Nww { atan }
15357  }
15358 \cs_new:Npn \_fp_acot_o:Nw
15359  {
15360    \_fp_atan_dispatch_o:NNnNw
15361      \_fp_atanii_o:Nww \_fp_acotii_o:Nww { acot }
15362  }
15363 \cs_new:Npn \_fp_atan_dispatch_o:NNnNw #1#2#3#4#5@{
15364  {
15365    \if_case:w
15366      \_int_eval:w \_fp_array_count:n {#5} - 1 \_int_eval_end:
15367        \exp_after:wN #1 \exp_after:wN #4 \c_one_fp #5
15368        \exp:w
15369    \or: #2 #4 #5 \exp:w
15370    \else:
15371      \_msg_kernel_expandable_error:nnnn
15372        { kernel } { fp-num-args } { #3() } { 1 } { 2 }
15373        \exp_after:wN \c_nan_fp \exp:w
15374    \fi:
15375    \exp_after:wN \exp_end:
15376  }

```

(End definition for $_fp_atan_o:Nw$, $_fp_acot_o:Nw$, and $_fp_atan_dispatch_o:NNnNw$.)

$_fp_atanii_o:Nww$ If either operand is `nan`, we return it. If both are normal, we call $_fp_atan_normal_o:NNnNw$. If both are zero or both infinity, we call $_fp_atan_inf_o:NNNw$ with argument 2, leading to a result among $\{\pm\pi/4, \pm 3\pi/4\}$ (in degrees, $\{\pm 45, \pm 135\}$). 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 $\{\pm 0, \pm\pi\}$ (in degrees, $\{\pm 0, \pm 180\}$). Since $\text{acot}(x, y) = \text{atan}(y, x)$, $_fp_acotii_o:ww$ simply reverses its two arguments.

```

15377 \cs_new:Npn \_fp_atanii_o:Nww
15378   #1 \s_fp \_fp_chk:w #2#3#4; \s_fp \_fp_chk:w #5
15379  {
15380    \if_meaning:w 3 #2 \_fp_case_return_i_o:ww \fi:
15381    \if_meaning:w 3 #5 \_fp_case_return_ii_o:ww \fi:
15382    \if_case:w
15383      \if_meaning:w #2 #5
15384        \if_meaning:w 1 #2 10 \else: 0 \fi:
15385    \else:
15386      \if_int_compare:w #2 > #5 \exp_stop_f: 1 \else: 2 \fi:
15387    \fi:
15388    \exp_stop_f:
15389      \_fp_case_return:nw { \_fp_atan_inf_o:NNNw #1 #3 2 }
15390      \or: \_fp_case_return:nw { \_fp_atan_inf_o:NNNw #1 #3 4 }
15391      \or: \_fp_case_return:nw { \_fp_atan_inf_o:NNNw #1 #3 0 }
15392    \fi:
15393    \_fp_atan_normal_o:NNnNw #1
15394    \s_fp \_fp_chk:w #2#3#4;
15395    \s_fp \_fp_chk:w #5
15396  }
15397 \cs_new:Npn \_fp_acotii_o:Nww #1#2; #3;

```

```

15398 { \__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; 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, atan z will be computed to be 0, and the result will be $[\#3/2] \cdot \pi/4$ if the sign #5 of x is positive, and $[(7 - \#3)/2] \cdot \pi/4$ for negative x , where the divisions are rounded up.

```

15399 \cs_new:Npn \__fp_atan_inf_o:NNNw #1#2#3 \s__fp \__fp_chk:w #4#5#6;
15400 {
15401   \exp_after:wN \__fp_atan_combine_o:NwwwwN
15402   \exp_after:wN #2
15403   \__int_value:w \__int_eval:w
15404   \if_meaning:w 2 #5 7 - \fi: #3 \exp_after:wN ;
15405   \c__fp_one_fixed_tl
15406   {0000}{0000}{0000}{0000}{0000}{0000};
15407   0,{0000}{0000}{0000}{0000}{0000}; #1
15408 }
```

(End definition for __fp_atan_inf_o:NNNw.)

__fp_atan_normal_o:NNnwNnw

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.

```

15409 \cs_new_protected:Npn \__fp_atan_normal_o:NNnwNnw
15410   #1 \s__fp \__fp_chk:w 1#2#3#4; \s__fp \__fp_chk:w 1#5#6#7;
15411 {
15412   \__fp_atan_test_o:NwwNwwN
15413   #2 #3, #4{0000}{0000};
15414   #5 #6, #7{0000}{0000}; #1
15415 }
```

(End definition for __fp_atan_normal_o:NNnwNnw.)

__fp_atan_test_o:NwwNwwN

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 what z will be, 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:wnwwnw after the operands have been ordered.

```

15416 \cs_new:Npn \__fp_atan_test_o:NwwNwwN #1#2,#3; #4#5,#6;
15417 {
15418   \exp_after:wN \__fp_atan_combine_o:NwwwwN
15419   \exp_after:wN #1
```

```

15420     \_\_int_value:w \_\_int_eval:w
15421         \if_meaning:w 2 #4
15422             7 - \_\_int_eval:w
15423         \fi:
15424         \if_int_compare:w
15425             \_\_fp_ep_compare:www #2,#3; #5,#6; > 0 \exp_stop_f:
15426             3 -
15427             \exp_after:wN \_\_fp_reverse_args:Nww
15428         \fi:
15429         \_\_fp_atan_div:wnwwnw #2,#3; #5,#6;
15430     }

```

(End definition for $__fp_atan_test_o:NwwNwwN$.)

```

\_\_fp_atan_div:wnwwnw
\_\_fp_atan_near:wwwn
\_\_fp_atan_near_aux:wwn

```

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 $\text{atan} \frac{b-a}{a+b}$. Otherwise, the octant is 0 (again, combined with earlier terms) and we wish to compute $\text{atan} \frac{a}{b}$. In any case, call $__fp_atan_auxi:ww$ followed by z , as a comma-delimited exponent and a fixed point number.

```

15431 \cs_new:Npn \_\_fp_atan_div:wnwwnw #1,#2#3; #4,#5#6;
15432     {
15433         \if_int_compare:w
15434             \_\_int_eval:w 41421 * #5 < #2 000
15435                 \if_case:w \_\_int_eval:w #4 - #1 \_\_int_eval_end: 00 \or: 0 \fi:
15436                 \exp_stop_f:
15437                 \exp_after:wN \_\_fp_atan_near:wwwn
15438             \fi:
15439             0
15440             \_\_fp_ep_div:wwwn #1,{#2}#3; #4,{#5}#6;
15441             \_\_fp_atan_auxi:ww
15442         }
15443 \cs_new:Npn \_\_fp_atan_near:wwwn
15444     0 \_\_fp_ep_div:wwwn #1,#2; #3,
15445     {
15446         1
15447         \_\_fp_ep_to_fixed:wwn #1 - #3, #2;
15448         \_\_fp_atan_near_aux:wwn
15449     }
15450 \cs_new:Npn \_\_fp_atan_near_aux:wwn #1; #2;
15451     {
15452         \_\_fp_fixed_add:wwn #1; #2;
15453         { \_\_fp_fixed_sub:wwn #2; #1; { \_\_fp_ep_div:wwwn 0, } 0, }
15454     }

```

(End definition for $__fp_atan_div:wnwwnw$, $__fp_atan_near:wwwn$, and $__fp_atan_near_aux:wwn$.)

```

\_\_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.

```

15455 \cs_new:Npn \_\_fp_atan_auxi:ww #1,#2;
15456     { \_\_fp_ep_to_fixed:wwn #1,#2; \_\_fp_atan_auxii:w #1,#2; }

```

```

15457 \cs_new:Npn \fp_atan_auxi:w #1;
15458 {
15459     \fp_fixed_mul:wwn #1; #1;
15460 {
15461     \fp_atan_Taylor_loop:www 39 ;
15462     {0000}{0000}{0000}{0000}{0000} ;
15463 }
15464 ! #1;
15465 }

```

(End definition for `\fp_atan_auxi:ww` and `\fp_atan_auxi:w`)

`\fp_atan_Taylor_loop:www`
`\fp_atan_Taylor_break:w`

We compute the series of $(\tan z)/z$. A typical intermediate stage has $\#1 = 2k - 1$, $\#2 = \frac{1}{2k+1} - z^2(\frac{1}{2k+3} - z^2(\cdots - z^2 \frac{1}{39}))$, 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 $(\tan z)/z$, 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.

```

15466 \cs_new:Npn \fp_atan_Taylor_loop:www #1; #2; #3;
15467 {
15468     \if_int_compare:w #1 = -1 \exp_stop_f:
15469         \fp_atan_Taylor_break:w
15470     \fi:
15471     \exp_after:wN \fp_fixed_div_int:wwN \c_fp_one_fixed_tl #1;
15472     \fp_rrot:www \fp_fixed_mul_sub_back:wwn #2; #3;
15473 {
15474     \exp_after:wN \fp_atan_Taylor_loop:www
15475     \int_value:w \int_eval:w #1 - 2 ;
15476 }
15477 #3;
15478 }
15479 \cs_new:Npn \fp_atan_Taylor_break:w
15480     \fi: #1 \fp_fixed_mul_sub_back:wwn #2; #3 !
15481 { \fi: ; #2 ; }

```

(End definition for `\fp_atan_Taylor_loop:www` and `\fp_atan_Taylor_break:w`)

`\fp_atan_combine_o:NwwwwN`
`\fp_atan_combine_aux:ww`

This receives a $\langle sign \rangle$, an $\langle octant \rangle$, a fixed point value of $(\tan z)/z$, a fixed point number 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

$$\langle sign \rangle \left(\left\lceil \frac{\langle octant \rangle}{2} \right\rceil \frac{\pi}{4} + (-1)^{\langle octant \rangle} \frac{\tan z}{z} \cdot z \right), \quad (11)$$

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 $\#3 = \frac{\tan z}{z}$ with $\#6$, the adjusted z . Otherwise, multiply $\#3 = \frac{\tan z}{z}$ 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`.

```

15482 \cs_new:Npn \fp_atan_combine_o:NwwwwN #1 #2; #3; #4; #5; #6; #7
15483 {

```

```

15484     \exp_after:wN \_fp_sanitize:Nw
15485     \exp_after:wN #1
15486     \_int_value:w \_int_eval:w
15487     \if_meaning:w 0 #2
15488     \exp_after:wN \use_i:nn
15489     \else:
15490     \exp_after:wN \use_ii:nn
15491     \fi:
15492     { #5 \_fp_fixed_mul:wwn #3; #6; }
15493     {
15494     \_fp_fixed_mul:wwn #3; #4;
15495     {
15496     \exp_after:wN \_fp_atan_combine_aux:ww
15497     \_int_value:w \_int_eval:w #2 / 2 ; #2;
15498     }
15499     }
15500     { #7 \_fp_fixed_to_float_o:wn \_fp_fixed_to_float_rad_o:wn }
15501     #1
15502   }
15503 \cs_new:Npn \_fp_atan_combine_aux:ww #1; #2;
15504   {
15505   \_fp_fixed_mul_short:wwn
15506   {7853}{9816}{3397}{4483}{0961}{5661};
15507   {#1}{0000}{0000};
15508   {
15509   \if_int_odd:w #2 \exp_stop_f:
15510   \exp_after:wN \_fp_fixed_sub:wwn
15511   \else:
15512   \exp_after:wN \_fp_fixed_add:wwn
15513   \fi:
15514   }
15515 }

```

(End definition for `_fp_atan_combine_o:NwwwwN` and `_fp_atan_combine_aux:ww`.)

29.2.2 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).

```

15516 \cs_new:Npn \_fp_asin_o:w #1 \s_fp \_fp_chk:w #2#3; @
15517   {
15518   \if_case:w #2 \exp_stop_f:
15519   \_fp_case_return_same_o:w
15520   \or:
15521   \_fp_case_use:nw
15522   { \_fp_asin_normal_o:NfwNnnnnw #1 { #1 { asin } { asind } } }
15523   \or:
15524   \_fp_case_use:nw
15525   { \_fp_invalid_operation_o:fw { #1 { asin } { asind } } }
15526   \else:
15527   \_fp_case_return_same_o:w

```

```

15528     \fi:
15529     \s_fp \fp_chk:w #2 #3;
15530   }

```

(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.

```

15531 \cs_new:Npn \fp_acos_o:w #1 \s_fp \fp_chk:w #2#3; @
15532   {
15533     \if_case:w #2 \exp_stop_f:
15534       \fp_case_use:nw { \fp_atan_inf_o>NNNw #1 0 4 }
15535     \or:
15536       \fp_case_use:nw
15537         {
15538           \fp_asin_normal_o:NfwNnnnnw #1 { #1 { acos } { acosd } }
15539             \fp_reverse_args:Nww
15540         }
15541     \or:
15542       \fp_case_use:nw
15543         { \fp_invalid_operation_o:fw { #1 { acos } { acosd } } }
15544     \else:
15545       \fp_case_return_same_o:w
15546     \fi:
15547     \s_fp \fp_chk:w #2 #3;
15548   }

```

(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.

```

15549 \cs_new:Npn \fp_asin_normal_o:NfwNnnnnw
15550   #1#2#3 \s_fp \fp_chk:w 1#4#5#6#7#8#9;
15551   {
15552     \if_int_compare:w #5 < 1 \exp_stop_f:
15553       \exp_after:wN \fp_use_none_until_s:w
15554     \fi:
15555     \if_int_compare:w \int_eval:w #5 + #6#7 + #8#9 = 1000 0001 ~
15556       \exp_after:wN \fp_use_none_until_s:w
15557     \fi:
15558     \fp_use_i:ww
15559     \fp_invalid_operation_o:fw {#2}
15560       \s_fp \fp_chk:w 1#4{#5}{#6}{#7}{#8}{#9};
15561     \fp_asin_auxi_o:NnNww
15562       #1 {#3} #4 #5,{#6}{#7}{#8}{#9}{0000}{0000};
15563   }

```

(End definition for `\fp_asin_normal_o:NfwNnnnnw`.)

```
\__fp_asin_auxi_o:NnNww
\__fp_asin_isqrt:wn
```

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`.

```
15564 \cs_new:Npn \__fp_asin_auxi_o:NnNww #1#2#3#4,#5;
15565 {
15566     \__fp_ep_to_fixed:wwn #4,#5;
15567     \__fp_asin_isqrt:wn
15568     \__fp_ep_mul:wwwn #4,#5;
15569     \__fp_ep_to_ep:wwN
15570     \__fp_fixed_continue:wn
15571     { #2 \__fp_atan_test_o:NwwNwwN #3 }
15572     0 1,{1000}{0000}{0000}{0000}{0000}; #1
15573 }
15574 \cs_new:Npn \__fp_asin_isqrt:wn #1;
15575 {
15576     \exp_after:wN \__fp_fixed_sub:wn \c__fp_one_fixed_tl #1;
15577     {
15578         \__fp_fixed_add_one:wN #1;
15579         \__fp_fixed_continue:wn { \__fp_ep_mul:wwwn 0, } 0,
15580     }
15581     \__fp_ep_isqrt:wn
15582 }
```

(End definition for `__fp_asin_auxi_o:NnNww` and `__fp_asin_isqrt:wn`.)

29.2.3 Arcosecant 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 arcosecant of ± 0 raises an invalid operation exception. The arcosecant 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 `acsco`) as an argument for invalid operation exceptions.

```
15583 \cs_new:Npn \__fp_acsc_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
15584 {
15585     \if_case:w \if_meaning:w 2 #2 #3 \fi: #2 \exp_stop_f:
15586         \__fp_case_use:nw
15587             { \__fp_invalid_operation_o:fw { #1 { acsc } { acsco } } }
15588     \or: \__fp_case_use:nw
15589         { \__fp_acsc_normal_o:NfwNnw #1 { #1 { acsc } { acsco } } }
15590     \or: \__fp_case_return_o:Nw \c_zero_fp
15591     \or: \__fp_case_return_same_o:w
15592     \else: \__fp_case_return_o:Nw \c_minus_zero_fp
15593     \fi:
15594     \s__fp \__fp_chk:w #2 #3 #4;
```

15595 }

(End definition for $\text{_fp_acsc_o:w}.$)

_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 $\text{_fp_reverse_args:Nww}$ following precisely that appearing in $\text{_fp_acos_o:w}.$

```
15596 \cs_new:Npn \_fp_asec_o:w #1 \s_fp \_fp_chk:w #2#3; @
15597 {
15598     \if_case:w #2 \exp_stop_f:
15599         \_fp_case_use:nw
15600             { \_fp_invalid_operation_o:fw { #1 { asec } { asecd } } }
15601     \or:
15602         \_fp_case_use:nw
15603             {
15604                 \_fp_acsc_normal_o:NfwNnw #1 { #1 { asec } { asecd } }
15605                     \_fp_reverse_args:Nww
15606             }
15607     \or: \_fp_case_use:nw { \_fp_atan_inf_o>NNNw #1 0 4 }
15608     \else: \_fp_case_return_same_o:w
15609     \fi:
15610     \s_fp \_fp_chk:w #2 #3;
15611 }
```

(End definition for $\text{_fp_asec_o:w}.$)

$\text{_fp_acsc_normal_o:NfwNnw}$ 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 $\text{_fp_asin_auxi_o:NnNww}$ (with all the appropriate arguments). This computes what we want thanks to $\text{acsc}(x) = \text{asin}(1/x)$ and $\text{asec}(x) = \text{acos}(1/x)$.

```
15612 \cs_new:Npn \_fp_acsc_normal_o:NfwNnw #1#2#3 \s_fp \_fp_chk:w 1#4#5#6;
15613 {
15614     \int_compare:nNnTF {#5} < 1
15615         {
15616             \_fp_invalid_operation_o:fw {#2}
15617                 \s_fp \_fp_chk:w 1#4{#5}#6;
15618         }
15619         {
15620             \_fp_ep_div:wwwn
15621                 1,{1000}{0000}{0000}{0000}{0000};
15622                 #5,#6{0000}{0000};
15623                 { \_fp_asin_auxi_o:NnNww #1 {#3} #4 }
15624         }
15625 }
```

(End definition for $\text{_fp_acsc_normal_o:NfwNnw}.$)

15626 ⟨/initex | package⟩

30 13fp-convert implementation

```
15627  {*initex | package}
15628  <@@=fp>
```

30.1 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 will be the `dot` auxiliary, which removes a trailing dot if any. We then clean-up with the `end` auxiliary, keeping only the number.

```
15629  \cs_new:Npn \_fp_trim_zeros:w #1 ;
15630  {
15631      \_fp_trim_zeros_loop:w #1
15632          ; \_fp_trim_zeros_loop:w 0; \_fp_trim_zeros_dot:w .; \s_stop
15633      }
15634  \cs_new:Npn \_fp_trim_zeros_loop:w #1 0; #2 { #2 #1 ; #2 }
15635  \cs_new:Npn \_fp_trim_zeros_dot:w #1 .; { \_fp_trim_zeros_end:w #1 ; }
15636  \cs_new:Npn \_fp_trim_zeros_end:w #1 ; #2 \s_stop { #1 }
```

(End definition for `_fp_trim_zeros:w` and others.)

30.2 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:n`

```
15637  \cs_new:Npn \fp_to_scientific:N #1
15638  { \exp_after:wN \_fp_to_scientific_dispatch:w #1 }
15639  \cs_generate_variant:Nn \fp_to_scientific:N { c }
15640  \cs_new:Npn \fp_to_scientific:n
15641  {
15642      \exp_after:wN \_fp_to_scientific_dispatch:w
15643      \exp:w \exp_end_continue_f:w \_fp_parse:n
15644  }
```

(End definition for `\fp_to_scientific:N` and `\fp_to_scientific:n`. These functions are documented on page 178.)

`_fp_to_scientific_dispatch:w`
`_fp_to_scientific_normal:wnnnnn`
`_fp_to_scientific_normal:wnw`

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.

```
15645  \cs_new:Npn \_fp_to_scientific_dispatch:w \s_fp \_fp_chk:w #1#2
15646  {
15647      \if_meaning:w 2 #2 \exp_after:wN - \exp:w \exp_end_continue_f:w \fi:
15648      \if_case:w #1 \exp_stop_f:
15649          \_fp_case_return:nw { 0.00000000000000e0 }
15650      \or: \exp_after:wN \_fp_to_scientific_normal:wnnnnn
15651      \or:
```

```

15652   \__fp_case_use:nw
15653   {
15654     \__fp_invalid_operation:nnw
15655     { \fp_to_scientific:N \c__fp_overflowing_fp }
15656     { fp_to_scientific }
15657   }
15658 \or:
15659   \__fp_case_use:nw
15660   {
15661     \__fp_invalid_operation:nnw
15662     { \fp_to_scientific:N \c_zero_fp }
15663     { fp_to_scientific }
15664   }
15665 \fi:
15666   \s__fp \__fp_chk:w #1 #2
15667 }
15668 \cs_new:Npn \__fp_to_scientific_normal:wnnnnn
15669   \s__fp \__fp_chk:w 1 #1 #2 #3#4#5#6 ;
15670   {
15671     \exp_after:wN \__fp_to_scientific_normal:wNw
15672     \exp_after:wN e
15673     \__int_value:w \__int_eval:w #2 - 1
15674     ; #3 #4 #5 #6 ;
15675   }
15676 \cs_new:Npn \__fp_to_scientific_normal:wNw #1 ; #2#3;
15677   { #2.#3 #1 }

(End definition for \__fp_to_scientific_dispatch:w, \__fp_to_scientific_normal:wnnnnn, and \__fp_to_scientific_normal:wNw.)

```

30.3 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

```

15678 \cs_new:Npn \fp_to_decimal:N #1
15679   { \exp_after:wN \__fp_to_decimal_dispatch:w #1 }
15680 \cs_generate_variant:Nn \fp_to_decimal:N { c }
15681 \cs_new:Npn \fp_to_decimal:n
15682   {
15683     \exp_after:wN \__fp_to_decimal_dispatch:w
15684     \exp:w \exp_end_continue_f:w \__fp_parse:n
15685   }

```

(End definition for \fp_to_decimal:N and \fp_to_decimal:n. These functions are documented on page 177.)

__fp_to_decimal_dispatch:w The structure is similar to __fp_to_scientific_dispatch: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.

```

15686 \cs_new:Npn \__fp_to_decimal_dispatch:w \s__fp \__fp_chk:w #1#2
15687 {
15688     \if_meaning:w 2 #2 \exp_after:wN - \exp:w \exp_end_continue_f:w \fi:
15689     \if_case:w #1 \exp_stop_f:
15690         \__fp_case_return:nw { 0 }
15691     \or: \exp_after:wN \__fp_to_decimal_normal:wnnnnn
15692     \or:
15693         \__fp_case_use:nw
15694         {
15695             \__fp_invalid_operation:nnw
15696                 { \fp_to_decimal:N \c__fp_overflowing_fp }
15697                 { fp_to_decimal }
15698         }
15699     \or:
15700         \__fp_case_use:nw
15701         {
15702             \__fp_invalid_operation:nnw
15703                 { 0 }
15704                 { fp_to_decimal }
15705         }
15706     \fi:
15707     \s__fp \__fp_chk:w #1 #2
15708 }

15709 \cs_new:Npn \__fp_to_decimal_normal:wnnnnn
15710     \s__fp \__fp_chk:w 1 #1 #2 #3#4#5#6 ;
15711 {
15712     \int_compare:nNnTF {#2} > 0
15713     {
15714         \int_compare:nNnTF {#2} < \c__fp_prec_int
15715         {
15716             \__fp_decimate:nNnnnn { \c__fp_prec_int - #2 }
15717                 \__fp_to_decimal_large:Nnnw
15718         }
15719         {
15720             \exp_after:wN \exp_after:wN
15721                 \exp_after:wN \__fp_to_decimal_huge:wnnnn
15722                 \prg_replicate:nn { #2 - \c__fp_prec_int } { 0 } ;
15723         }
15724     {#3} {#4} {#5} {#6}
15725 }
15726 {
15727     \exp_after:wN \__fp_trim_zeros:w
15728     \exp_after:wN 0
15729     \exp_after:wN .
15730     \exp:w \exp_end_continue_f:w \prg_replicate:nn { - #2 } { 0 }
15731     #3#4#5#6 ;
15732 }
15733 }

15734 \cs_new:Npn \__fp_to_decimal_large:Nnnw #1#2#3#4;
15735 {
15736     \exp_after:wN \__fp_trim_zeros:w \__int_value:w
15737         \if_int_compare:w #2 > 0 \exp_stop_f:
15738             #2
15739     \fi:

```

```

15740      \exp_stop_f:
15741      #3.#4 ;
15742  }
15743 \cs_new:Npn \__fp_to_decimal_huge:wnnnn #1; #2#3#4#5 { #2#3#4#5 #1 }

(End definition for \__fp_to_decimal_dispatch:w and others.)

```

30.4 Token list representation

\fp_to_tl:N
\fp_to_tl:c
\fp_to_tl:n

These three public functions evaluate their argument, then pass it to __fp_to_tl_dispatch:w.

```

15744 \cs_new:Npn \fp_to_tl:N #1 { \exp_after:wN \__fp_to_tl_dispatch:w #1 }
15745 \cs_generate_variant:Nn \fp_to_tl:N { c }
15746 \cs_new:Npn \fp_to_tl:n
15747  {
15748      \exp_after:wN \__fp_to_tl_dispatch:w
15749      \exp:w \exp_end_continue_f:w \__fp_parse:n
15750  }

```

(End definition for \fp_to_tl:N and \fp_to_tl:n. These functions are documented on page 178.)

__fp_to_tl_dispatch:w
__fp_to_tl_normal:nnnn
 __fp_to_tl_scientific:wnnnn
__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.

```

15751 \cs_new:Npn \__fp_to_tl_dispatch:w \s__fp \__fp_chk:w #1#2
15752  {
15753      \if_meaning:w 2 #2 \exp_after:wN - \exp:w \exp_end_continue_f:w \fi:
15754      \if_case:w #1 \exp_stop_f:
15755          \__fp_case_return:nw { 0 }
15756      \or: \exp_after:wN \__fp_to_tl_normal:nnnnn
15757      \or: \__fp_case_return:nw { inf }
15758      \else: \__fp_case_return:nw { nan }
15759      \fi:
15760  }
15761 \cs_new:Npn \__fp_to_tl_normal:nnnnn #1
15762  {
15763      \int_compare:nTF
15764          { -2 <= #1 <= \c_fp_prec_int }
15765          { \__fp_to_decimal_normal:wnnnnn }
15766          { \__fp_to_tl_scientific:wnnnnn }
15767          \s__fp \__fp_chk:w 1 0 {#1}
15768      }
15769 \cs_new:Npn \__fp_to_tl_scientific:wnnnnn
15770     \s__fp \__fp_chk:w 1 #1 #2 #3#4#5#6 ;
15771  {
15772      \exp_after:wN \__fp_to_tl_scientific:wNw
15773      \exp_after:wN e
15774      \__int_value:w \__int_eval:w #2 - 1
15775      ; #3 #4 #5 #6 ;
15776  }
15777 \cs_new:Npn \__fp_to_tl_scientific:wNw #1 ; #2#3;
15778  { \__fp_trim_zeros:w #2.#3 ; #1 }

(End definition for \__fp_to_tl_dispatch:w and others.)

```

30.5 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.

30.6 Convert to dimension or integer

`\fp_to_dim:N` These three public functions rely on `\fp_to_decimal:n` internally.

```
15779 \cs_new:Npn \fp_to_dim:N #1
1580   { \fp_to_decimal:N #1 pt }
1581   \cs_generate_variant:Nn \fp_to_dim:N { c }
1582   \cs_new:Npn \fp_to_dim:n #1
1583     { \fp_to_decimal:n {#1} pt }
```

(End definition for `\fp_to_dim:N` and `\fp_to_dim:n`. These functions are documented on page 177.)

`\fp_to_int:N` These three public functions evaluate their argument, then pass it to `\fp_to_int_- dispatch:w`.

```
15784 \cs_new:Npn \fp_to_int:N #1 { \exp_after:wN \__fp_to_int_dispatch:w #1 }
15785 \cs_generate_variant:Nn \fp_to_int:N { c }
15786 \cs_new:Npn \fp_to_int:n
15787   {
15788     \exp_after:wN \__fp_to_int_dispatch:w
15789     \exp:w \exp_end_continue_f:w \__fp_parse:n
15790   }
```

(End definition for `\fp_to_int:N` and `\fp_to_int:n`. These functions are documented on page 178.)

`__fp_to_int_dispatch:w` To convert to an integer, first round to 0 places (to the nearest integer), then express the result as a decimal number: the definition of `__fp_to_decimal_dispatch:w` is such that there will be no trailing dot nor zero.

```
15791 \cs_new:Npn \__fp_to_int_dispatch:w #1;
15792   {
15793     \exp_after:wN \__fp_to_decimal_dispatch:w \exp:w \exp_end_continue_f:w
15794     \__fp_round:Nwn \__fp_round_to_nearest>NNN #1; { 0 }
15795 }
```

(End definition for `__fp_to_int_dispatch:w`.)

30.7 Convert from a dimension

`\dim_to_fp:n`

```
\__fp_from_dim_test:ww
\__fp_from_dim:wNw
\__fp_from_dim:wNNnnnnnn
\__fp_from_dim:wnnnnnnnw
```

The dimension expression (which can in fact be a glue expression) is evaluated, converted to a number (*i.e.*, expressed in scaled points), then multiplied by $2^{-16} = 0.0000152587890625$ to give a value expressed in points. The auxiliary `__fp_mul_npos_o:Nww` expects the desired *final sign* and two floating point operands (of the form `\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.

```
15796 \cs_new:Npn \dim_to_fp:n #1
15797   {
```

```

15798     \exp_after:wN \__fp_from_dim_test:ww
15799     \exp_after:wN 0
15800     \exp_after:wN ,
15801     \__int_value:w \etex_glueexpr:D #1 ;
15802   }
15803 \cs_new:Npn \__fp_from_dim_test:ww #1, #2
15804   {
15805     \if_meaning:w 0 #2
15806       \__fp_case_return:nw { \exp_after:wN \c_zero_fp }
15807     \else:
15808       \exp_after:wN \__fp_from_dim:wNw
15809       \__int_value:w \__int_eval:w #1 - 4
15810       \if_meaning:w - #2
15811         \exp_after:wN , \exp_after:wN 2 \__int_value:w
15812       \else:
15813         \exp_after:wN , \exp_after:wN 0 \__int_value:w #2
15814       \fi:
15815     \fi:
15816   }
15817 \cs_new:Npn \__fp_from_dim:wNw #1,#2#3;
15818   {
15819     \__fp_pack_twice_four:wNNNNNNNN \__fp_from_dim:wNNnnnnnn ;
15820     #3 000 0000 00 {10}987654321; #2 {#1}
15821   }
15822 \cs_new:Npn \__fp_from_dim:wNNnnnnnn #1; #2#3#4#5#6#7#8#9
15823   { \__fp_from_dim:wnnnnwNn #1 {#2#300} {0000} ; }
15824 \cs_new:Npn \__fp_from_dim:wnnnnwNn #1; #2#3#4#5#6; #7#8
15825   {
15826     \__fp_mul_npos_o:Nww #7
15827     \s_fp \__fp_chk:w 1 #7 {#5} #1 ;
15828     \s_fp \__fp_chk:w 1 0 {#8} {1525} {8789} {0625} {0000} ;
15829     \prg_do_nothing:
15830   }

```

(End definition for `\dim_to_fp:n` and others. These functions are documented on page 154.)

30.8 Use and eval

\fp_use:N Those public functions are simple copies of the decimal conversions.

\fp_use:c 15831 `\cs_new_eq:NN \fp_use:N \fp_to_decimal:N`

\fp_eval:n 15832 `\cs_generate_variant:Nn \fp_use:N { c }`

15833 `\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 178.)

\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.

15834 `\cs_new:Npn \fp_abs:n #1`

15835 `{ \fp_to_decimal:n { abs __fp_parse:n {#1} } }`

(End definition for `\fp_abs:n`. This function is documented on page 191.)

\fp_max:nn Similar to `\fp_abs:n`, for consistency with `\int_max:nn`, etc.

\fp_min:nn 15836 `\cs_new:Npn \fp_max:nn #1#2`

```

15837   { \fp_to_decimal:n { max ( \fp_parse:n {#1} , \fp_parse:n {#2} ) } }
15838   \cs_new:Npn \fp_min:nn #1#2
15839     { \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 191.)

30.9 Convert an array of floating points to a comma list

`_fp_array_to_clist:n`
`_fp_array_to_clist_loop:Nw`

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.

```

15840 \cs_new:Npn \_fp_array_to_clist:n #1
15841   {
15842     \tl_if_empty:nF {#1}
15843       {
15844         \_fp_expand:n
15845           {
15846             { \use_ii:nn }
15847             \_fp_array_to_clist_loop:Nw #1 { ? \prg_break: } ;
15848             \prg_break_point:
15849           }
15850       }
15851     }
15852 \cs_new:Npx \_fp_array_to_clist_loop:Nw #1#2;
15853   {
15854     \exp_not:N \use_none:n #1
15855     \exp_not:N \exp_after:wN
15856       {
15857         \exp_not:N \exp_after:wN ,
15858         \exp_not:N \exp_after:wN \c_space_tl
15859         \exp_not:N \exp:w
15860         \exp_not:N \exp_end_continue_f:w
15861         \exp_not:N \fp_to_tl_dispatch:w #1 #2 ;
15862       }
15863     \exp_not:N \_fp_array_to_clist_loop:Nw
15864   }

```

(End definition for `_fp_array_to_clist:n` and `_fp_array_to_clist_loop:Nw`.)

15865 `</initex | package>`

31 I3fp-random Implementation

15866 `<*initex | package>`

```
15867 ⟨@C=fp⟩
```

`__fp_parse_word_rand:N` Those functions may receive a variable number of arguments. We won't use the argument `?N`.

```
15868 \cs_new:Npn \__fp_parse_word_rand:N
15869   { \__fp_parse_function:NNN \__fp_rand_o:Nw ? }
15870 \cs_new:Npn \__fp_parse_word_randint:N
15871   { \__fp_parse_function:NNN \__fp_randint_o:Nw ? }
```

(End definition for `__fp_parse_word_rand:N` and `__fp_parse_word_randint:N`.)

31.1 Engine support

At present, X_ET_EX, pT_EX and upT_EX do not provide random numbers, while Lu_AT_EX and pdfT_EX provide the primitive `\pdftex_uniformdeviate:D` (`\pdfuniformdeviate` in pdfT_EX and `\uniformdeviate` in Lu_AT_EX). We write the test twice simply in order to write the `false` branch first.

```
15872 \cs_if_exist:NF \pdftex_uniformdeviate:D
15873   {
15874     \__msg_kernel_new:nnn { kernel } { fp-no-random }
15875       { Random-numbers-unavailable }
15876     \cs_new:Npn \__fp_rand_o:Nw ? #1 @
15877       {
15878         \__msg_kernel_expandable_error:nn { kernel } { fp-no-random }
15879           \exp_after:wN \c_nan_fp
15880         }
15881       \cs_new_eq:NN \__fp_randint_o:Nw \__fp_rand_o:Nw
15882     }
15883 \cs_if_exist:NT \pdftex_uniformdeviate:D
15884   {
```

`__fp_rand_uniform:`
`\c__fp_rand_size_int`
`\c__fp_rand_four_int`
`\c__fp_rand_eight_int`

The `\pdftex_uniformdeviate:D` primitive gives a pseudo-random integer in a range $[0, n - 1]$ of the user's choice. This number is meant to be uniformly distributed, but is produced by rescaling a uniform pseudo-random integer in $[0, 2^{28} - 1]$. For instance, setting n to (any multiple of) 2^{29} gives only even values. Thus it is only safe to call `\pdftex_uniformdeviate:D` with argument 2^{28} . This integer is also used in the implementation of `\int_rand:nn`. We will also use variants of this number rounded down to multiples of 10^4 and 10^8 .

```
15885 \cs_new:Npn \__fp_rand_uniform:
15886   { \pdftex_uniformdeviate:D \c__fp_rand_size_int }
15887 \int_const:Nn \c__fp_rand_size_int { 268 435 456 }
15888 \int_const:Nn \c__fp_rand_four_int { 268 430 000 }
15889 \int_const:Nn \c__fp_rand_eight_int { 200 000 000 }
```

(End definition for `__fp_rand_uniform:` and others.)

`__fp_rand_myriads:n`
`__fp_rand_myriads_loop:nn`
`__fp_rand_myriads_get:w`
`__fp_rand_myriads_last:`
`__fp_rand_myriads_last:w`

Used as `__fp_rand_myriads:n {XXX}` with one input character per block of four digit we want. Given a pseudo-random integer from the primitive, we extract 2 blocks of digits if possible, namely if the integer is less than 2×10^8 . If that's not possible, we try to extract 1 block, which succeeds in the range $[2 \times 10^8, 26843 \times 10^4]$. For the 5456 remaining possible values we just throw away the random integer and get a new one. Depending on whether we got 2, 1, or 0 blocks, remove the same number of characters from the input stream with `\use_i:nnn`, `\use_i:nn` or nothing.

```

15890 \cs_new:Npn \__fp_rand_myriads:n #1
15891 {
15892     \__fp_rand_myriads_loop:nn #1
15893     { ? \use_i_delimit_by_q_stop:nw \__fp_rand_myriads_last: }
15894     { ? \use_none_delimit_by_q_stop:w } \q_stop
15895 }
15896 \cs_new:Npn \__fp_rand_myriads_loop:nn #1#2
15897 {
15898     \use_none:n #2
15899     \exp_after:wN \__fp_rand_myriads_get:w
15900     \__int_value:w \__fp_rand_uniform: ; {#1}{#2}
15901 }
15902 \cs_new:Npn \__fp_rand_myriads_get:w #1 ;
15903 {
15904     \if_int_compare:w #1 < \c__fp_rand_eight_int
15905         \exp_after:wN \use_none:n
15906         \__int_value:w \__int_eval:w
15907         \c__fp_rand_eight_int + #1 \__int_eval_end:
15908         \exp_after:wN \use_i:nnn
15909     \else:
15910         \if_int_compare:w #1 < \c__fp_rand_four_int
15911             \exp_after:wN \use_none:nnnnn
15912             \__int_value:w \__int_eval:w
15913             \c__fp_rand_four_int + #1 \__int_eval_end:
15914             \exp_after:wN \exp_after:wN \exp_after:wN \use_i:nn
15915         \fi:
15916     \fi:
15917     \__fp_rand_myriads_loop:nn
15918 }
15919 \cs_new:Npn \__fp_rand_myriads_last:
15920 {
15921     \exp_after:wN \__fp_rand_myriads_last:w
15922     \__int_value:w \__fp_rand_uniform: ;
15923 }
15924 \cs_new:Npn \__fp_rand_myriads_last:w #1 ;
15925 {
15926     \if_int_compare:w #1 < \c__fp_rand_four_int
15927         \exp_after:wN \use_none:nnnnn
15928         \__int_value:w \__int_eval:w
15929         \c__fp_rand_four_int + #1 \__int_eval_end:
15930     \else:
15931         \exp_after:wN \__fp_rand_myriads_last:
15932     \fi:
15933 }

```

(End definition for `__fp_rand_myriads:n` and others.)

31.2 Random floating point

`__fp_rand_o:Nw` First we check that `random` was called without argument. Then get four blocks of four digits.

`__fp_rand_o:W`

```

15934 \cs_new:Npn \__fp_rand_o:Nw ? #1 @
15935 {
15936     \tl_if_empty:nTF {#1}

```

```

15937 { \__fp_rand_o: }
15938 {
15939     \__msg_kernel_expandable_error:nnnn
15940     { kernel } { fp-num-args } { rand() } { 0 } { 0 }
15941     \exp_after:wN \c_nan_fp
15942 }
15943 }
15944 \cs_new:Npn \__fp_rand_o:
15945     { \__fp_parse_o:n { . \__fp_rand_myriads:n { xxxx } } }

(End definition for \__fp_rand_o:Nw, \__fp_rand_o:, and \__fp_rand_o:w.)

```

31.3 Random integer

__fp_randint_o:Nw
__fp_randint_badarg:w
__fp_randint_e:w
__fp_randint_e:wnn
__fp_randint_e:wwNnn
__fp_randint_e:wwwNnn
__fp_randint_narrow_e:nnnn
__fp_randint_wide_e:nnnn
__fp_randint_wide_e:wnnn

Enforce that there is one argument (then add first argument 1) or two arguments. Enforce that they are integers in $(-10^{16}, 10^{16})$ and ordered. We distinguish narrow ranges (less than 2^{28}) from wider ones.

For narrow ranges, compute the number n of possible outputs as an integer using \fp_to_int:n, and reduce a pseudo-random 28-bit integer r modulo n . On its own, this is not uniform when $[0, 2^{28} - 1]$ does not divide evenly into intervals of size n . The auxiliary __fp_randint_e:wwwNnn discards the pseudo-random integer if it lies in an incomplete interval, and repeats.

For wide ranges we use the same code except for the last eight digits which use __fp_rand_myriads:n. It is not safe to combine the first digits with the last eight as a single string of digits, as this may exceed 16 digits and be rounded. Instead, we first add the first few digits (times 10^8) to the lower bound. The result is compared to the upper bound and the process repeats if needed.

```

15946 \cs_new:Npn \__fp_randint_o:Nw ? #1 @
15947     {
15948         \if_case:w
15949             \__int_eval:w \__fp_array_count:n {#1} - 1 \__int_eval_end:
15950             \exp_after:wN \__fp_randint_e:w \c_one_fp #1
15951         \or: \__fp_randint_e:w #1
15952         \else:
15953             \__msg_kernel_expandable_error:nnnn
15954             { kernel } { fp-num-args } { randint() } { 1 } { 2 }
15955             \exp_after:wN \c_nan_fp \exp:w
15956         \fi:
15957         \exp_after:wN \exp_end:
15958     }
15959 \cs_new:Npn \__fp_randint_badarg:w \s_fp \__fp_chk:w #1#2#3;
15960     {
15961         \__fp_int:wTF \s_fp \__fp_chk:w #1#2#3;
15962         {
15963             \if_meaning:w 1 #1
15964             \if_int_compare:w
15965                 \use_i_delimit_by_q_stop:nw #3 \q_stop > \c_fp_prec_int
15966                 1 \exp_stop_f:
15967             \fi:
15968         \fi:
15969     }
15970     { 1 \exp_stop_f: }
15971 }

```

```

15972 \cs_new:Npn \__fp_randint_e:w #1; #2;
15973 {
15974     \if_case:w
15975         \__fp_randint_badarg:w #1;
15976         \__fp_randint_badarg:w #2;
15977         \fp_compare:nNnTF { #1; } > { #2; } { 1 } { 0 } \exp_stop_f:
15978         \exp_after:wN \exp_after:wN \exp_after:wN \__fp_randint_e:wnn
15979         \__fp_parse:n { #2; - #1; } { #1; } { #2; }
15980     \or:
15981         \__fp_invalid_operation_tl_o:ff
15982             { randint } { \__fp_array_to_clist:n { #1; #2; } }
15983             \exp:w
15984         \fi:
15985     }
15986 \cs_new:Npn \__fp_randint_e:wnn #1;
15987 {
15988     \exp_after:wN \__fp_randint_e:wwNnn
15989     \__int_value:w \__fp_rand_uniform: \exp_after:wN ;
15990     \exp:w \exp_end_continue_f:w
15991     \fp_compare:nNnTF { #1 ; } < \c_fp_rand_size_int
15992         { \fp_to_int:n { #1 ; + 1 } ; \__fp_randint_narrow_e:nnnn }
15993         { \fp_to_int:n { floor(#1 ; * 1e-8 + 1) } ; \__fp_randint_wide_e:nnnn }
15994 }
15995 \cs_new:Npn \__fp_randint_e:wwNnn #1 ; #2 ;
15996 {
15997     \exp_after:wN \__fp_randint_e:wwwNnn
15998     \__int_value:w \int_mod:nn {#1} {#2} ; #1 ; #2 ;
15999 }
16000 \cs_new:Npn \__fp_randint_e:wwwNnn #1 ; #2 ; #3 ; #4
16001 {
16002     \int_compare:nNnTF { #2 - #1 + #3 } > \c_fp_rand_size_int
16003     {
16004         \exp_after:wN \__fp_randint_e:wwNnn
16005             \__int_value:w \__fp_rand_uniform: ; #3 ; #4
16006     }
16007     { #4 {#1} {#3} }
16008 }
16009 \cs_new:Npn \__fp_randint_narrow_e:nnnn #1#2#3#4
16010     { \__fp_parse_o:n { #3 + #1 } \exp:w }
16011 \cs_new:Npn \__fp_randint_wide_e:nnnn #1#2#3#4
16012 {
16013     \exp_after:wN \exp_after:wN
16014     \exp_after:wN \__fp_randint_wide_e:wnnn
16015     \__fp_parse:n { #3 + #1e8 + \__fp_rand_myriads:n { xx } }
16016     {#2} {#3} {#4}
16017 }
16018 \cs_new:Npn \__fp_randint_wide_e:wnnn #1 ; #2#3#4
16019 {
16020     \fp_compare:nNnTF { #1 ; } > {#4}
16021     {
16022         \exp_after:wN \__fp_randint_e:wwNnn
16023             \__int_value:w \__fp_rand_uniform: ; #2 ;
16024             \__fp_randint_wide_e:nnnn {#3} {#4}
16025     }

```

```
16026     { \__fp_exp_after_o:w #1 ; \exp:w }
16027 }
```

(End definition for `__fp_randint_o:Nw` and others.)

End the initial conditional that ensures these commands are only defined in pdfTeX and LuaTeX.

```
16028 }
16029 ⟨/initex | package⟩
```

32 I3fp-assign implementation

```
16030 ⟨*initex | package⟩
16031 ⟨@C=fp⟩
```

32.1 Assigning values

\fp_new:N Floating point variables are initialized to be $+0$.

```
16032 \cs_new_protected:Npn \fp_new:N #1
16033   { \cs_new_eq:NN #1 \c_zero_fp }
16034 \cs_generate_variant:Nn \fp_new:N {c}
```

(End definition for `\fp_new:N`. This function is documented on page 176.)

\fp_set:Nn Simply use `__fp_parse:n` within various f-expanding assignments.

```
\fp_set:cn 16035 \cs_new_protected:Npn \fp_set:Nn #1#2
\fp_gset:Nn 16036   { \tl_set:Nx #1 { \exp_not:f { \__fp_parse:n {#2} } } }
\fp_gset:cn 16037 \cs_new_protected:Npn \fp_gset:Nn #1#2
\fp_const:Nn 16038   { \tl_gset:Nx #1 { \exp_not:f { \__fp_parse:n {#2} } } }
\fp_const:cn 16039 \cs_new_protected:Npn \fp_const:Nn #1#2
16040   { \tl_const:Nx #1 { \exp_not:f { \__fp_parse:n {#2} } } }
16041 \cs_generate_variant:Nn \fp_set:Nn {c}
16042 \cs_generate_variant:Nn \fp_gset:Nn {c}
16043 \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 176.)

\fp_set_eq:NN Copying a floating point is the same as copying the underlying token list.

```
\fp_set_eq:cN 16044 \cs_new_eq:NN \fp_set_eq:NN \tl_set_eq:NN
\fp_set_eq:Nc 16045 \cs_new_eq:NN \fp_gset_eq:NN \tl_gset_eq:NN
\fp_set_eq:cc 16046 \cs_generate_variant:Nn \fp_set_eq:NN { c , Nc , cc }
\fp_gset_eq:NN 16047 \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 177.)

\fp_gset_eq:cN Setting a floating point to zero: copy `\c_zero_fp`.

```
\fp_zero:c 16048 \cs_new_protected:Npn \fp_zero:N #1 { \fp_set_eq:NN #1 \c_zero_fp }
\fp_gzero:N 16049 \cs_new_protected:Npn \fp_gzero:N #1 { \fp_gset_eq:NN #1 \c_zero_fp }
\fp_gzero:c 16050 \cs_generate_variant:Nn \fp_zero:N { c }
16051 \cs_generate_variant:Nn \fp_gzero:N { c }
```

(End definition for `\fp_zero:N` and `\fp_gzero:N`. These functions are documented on page 176.)

```

\fp_zero_new:N Set the floating point to zero, or define it if needed.
\fp_zero_new:c 16052 \cs_new_protected:Npn \fp_zero_new:N #1
\fp_gzero_new:N 16053 { \fp_if_exist:NTF #1 { \fp_zero:N #1 } { \fp_new:N #1 } }
\fp_gzero_new:c 16054 \cs_new_protected:Npn \fp_gzero_new:N #1
16055 { \fp_if_exist:NTF #1 { \fp_gzero:N #1 } { \fp_new:N #1 } }
16056 \cs_generate_variant:Nn \fp_zero_new:N { c }
16057 \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 176.)

```

32.2 Updating values

These match the equivalent functions in `\3int` and `\3skip`.

```

\fp_add:Nn For the sake of error recovery we should not simply set #1 to  $\#1 \pm (\#2)$ : 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 16058 \cs_new_protected:Npn \fp_add:Nn { \_fp_add:NNNn \fp_set:Nn + }
\fp_gsub:Nn 16059 \cs_new_protected:Npn \fp_gadd:Nn { \_fp_add:NNNn \fp_gset:Nn + }
\fp_gsub:cn 16060 \cs_new_protected:Npn \fp_sub:Nn { \_fp_add:NNNn \fp_set:Nn - }
16061 \cs_new_protected:Npn \fp_gsub:Nn { \_fp_add:NNNn \fp_gset:Nn - }
\_\_fp_add:NNNn 16062 \cs_new_protected:Npn \_\_fp_add:NNNn #1#2#3#4
16063 { #1 #3 { #3 #2 \_fp_parse:n {#4} } }
16064 \cs_generate_variant:Nn \fp_add:Nn { c }
16065 \cs_generate_variant:Nn \fp_gadd:Nn { c }
16066 \cs_generate_variant:Nn \fp_sub:Nn { c }
16067 \cs_generate_variant:Nn \fp_gsub:Nn { c }

(End definition for \fp_add:Nn and others. These functions are documented on page 177.)

```

32.3 Showing values

`\fp_show:N` This shows the result of computing its argument. The input of `_msg_show_-variable:NNNnn` must start with `>~` (or be empty).

```

\fp_show:c 16068 \cs_new_protected:Npn \fp_show:N #1
16069 {
16070 \_msg_show_variable:NNNnn #1 \fp_if_exist:NTF ? { }
16071 { > ~ \token_to_str:N #1 = \fp_to_tl:N #1 }
16072 }
\fp_show:n 16073 \cs_new_protected:Npn \fp_show:n
16074 { \_msg_show_wrap:Nn \fp_to_tl:n }
16075 \cs_generate_variant:Nn \fp_show:N { c }

(End definition for \fp_show:N and \fp_show:n. These functions are documented on page 183.)

```

`\fp_log:N` Redirect output of `\fp_show:N` and `\fp_show:n` to the log.

```

\fp_log:c 16076 \cs_new_protected:Npn \fp_log:N
16077 { \_msg_log_next: \fp_show:N }
\fp_log:n 16078 \cs_new_protected:Npn \fp_log:n
16079 { \_msg_log_next: \fp_show:n }
16080 \cs_generate_variant:Nn \fp_log:N { c }

(End definition for \fp_log:N and \fp_log:n. These functions are documented on page 183.)

```

32.4 Some useful constants and scratch variables

`\c_one_fp` Some constants.

```
16081 \fp_const:Nn \c_e_fp { 2.718 2818 2845 9045 }
16082 \fp_const:Nn \c_one_fp { 1 }
```

(End definition for `\c_one_fp` and `\c_e_fp`. These variables are documented on page 182.)

`\c_pi_fp` We simply round π to and $\pi/180$ to 16 significant digits.

```
16083 \fp_const:Nn \c_pi_fp { 3.141 5926 5358 9793 }
16084 \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 182.)

`\l_tmpa_fp` Scratch variables are simply initialized there.

```
16085 \fp_new:N \l_tmpa_fp
16086 \fp_new:N \l_tmpb_fp
16087 \fp_new:N \g_tmpa_fp
16088 \fp_new:N \g_tmpb_fp
```

(End definition for `\l_tmpa_fp` and others. These variables are documented on page 182.)

```
16089 </initex | package>
```

33 I3sort implementation

```
16090 <*initex | package>
```

```
16091 <@=sort>
```

33.1 Variables

`\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 will only use `\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.

```
16092 \int_new:N \l__sort_length_int
16093 \int_new:N \l__sort_min_int
16094 \int_new:N \l__sort_top_int
16095 \int_new:N \l__sort_max_int
16096 \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.

```
16097 \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.

```
16098 \int_new:N \l__sort_begin_int
16099 \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.

```
16100 \int_new:N \l__sort_A_int
16101 \int_new:N \l__sort_B_int
16102 \int_new:N \l__sort_C_int
```

(End definition for \l__sort_A_int, \l__sort_B_int, and \l__sort_C_int.)

33.2 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.

```
16103 \cs_new_protected:Npn \_\_sort_shrink_range:
16104 {
16105     \int_set:Nn \l__sort_A_int
16106     { \l__sort_true_max_int - \l__sort_min_int + 1 }
16107     \int_set:Nn \l__sort_block_int { \c_max_register_int / 2 }
16108     \_\_sort_shrink_range_loop:
16109     \int_set:Nn \l__sort_max_int
16110     {
16111         \int_compare:nNnTF
16112         { \l__sort_block_int * 3 / 2 } > \l__sort_A_int
16113         {
16114             \l__sort_min_int
16115             + ( \l__sort_A_int - 1 ) / 2
16116             + \l__sort_block_int / 4
16117             - 1
16118         }
16119         { \l__sort_true_max_int - \l__sort_block_int / 2 }
16120     }
16121 }
16122 \cs_new_protected:Npn \_\_sort_shrink_range_loop:
16123 {
16124     \if_int_compare:w \l__sort_A_int < \l__sort_block_int
16125         \tex_divide:D \l__sort_block_int 2 \exp_stop_f:
16126         \exp_after:wN \_\_sort_shrink_range_loop:
16127     \fi:
```

16128 }

(End definition for `_sort_shrink_range:` and `_sort_shrink_range_loop::`)

`_sort_compute_range:`
`_sort_redefine_compute_range:`
`\c_sort_max_length_int`

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 ConTeXt 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::`. The L^AT_EX3 format mode is easiest: no `\toks` are ever allocated so available `\toks` range from 0 to `\c_max_register_int` and we precompute the result of `_sort_shrink_range::`.

```
16129  {*package}
16130  \cs_new_protected:Npn \_sort_compute_range:
16131  {
16132      \int_set:Nn \l__sort_min_int { \tex_count:D 15 + 1 }
16133      \int_set:Nn \l__sort_true_max_int { \c_max_register_int + 1 }
16134      \_sort_shrink_range:
16135      \if_meaning:w \loctoks \tex undefined:D \else:
16136          \if_meaning:w \loctoks \scan_stop: \else:
16137              \_sort_redefine_compute_range:
16138              \_sort_compute_range:
16139          \fi:
16140      \fi:
16141  }
16142  \cs_new_protected:Npn \_sort_redefine_compute_range:
16143  {
16144      \cs_if_exist:cTF { ver@elocalloc.sty }
16145      {
16146          \cs_gset_protected:Npn \_sort_compute_range:
16147          {
16148              \int_set:Nn \l__sort_min_int { \tex_count:D 265 }
16149              \int_set_eq:NN \l__sort_true_max_int \e@alloc@top
16150              \_sort_shrink_range:
16151          }
16152      }
16153  {
16154      \cs_gset_protected:Npn \_sort_compute_range:
16155      {
16156          \int_set:Nn \l__sort_min_int { \tex_count:D 265 }
16157          \int_set:Nn \l__sort_true_max_int { \tex_count:D 275 }
16158          \_sort_shrink_range:
16159      }
16160  }
16161  }
16162  \cs_if_exist:NT \loctoks { \_sort_redefine_compute_range: }
16163  \tl_map_inline:nn { \lastallocatedtoks \c_syst_last_allocated_toks }
16164  {
```

```

16165   \cs_if_exist:NT #1
16166   {
16167     \cs_gset_protected:Npn \__sort_compute_range:
16168     {
16169       \int_set:Nn \l__sort_min_int { #1 + 1 }
16170       \int_set:Nn \l__sort_true_max_int { \c_max_register_int + 1 }
16171       \__sort_shrink_range:
16172     }
16173   }
16174 }
16175 </package>
16176 <initex>
16177 \int_const:Nn \c__sort_max_length_int
16178   { ( \c_max_register_int + 1 ) * 3 / 4 }
16179 \cs_new_protected:Npn \__sort_compute_range:
16180   {
16181     \int_set:Nn \l__sort_min_int { 0 }
16182     \int_set:Nn \l__sort_true_max_int { \c_max_register_int + 1 }
16183     \int_set:Nn \l__sort_max_int { \c__sort_max_length_int }
16184   }
16185 </initex>

(End definition for \__sort_compute_range:, \__sort_redefine_compute_range:, and \c__sort_max_length_int.)

```

33.3 Protected user commands

`__sort_main:NNNnNn` 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, all further items are entirely ignored after raising an error. Secondly, sort the array of `\toks` registers, using the user-defined sorting function, #6. Finally, unpack the `\toks` registers (now sorted) into a variable of the right type, by `x`-expanding the code in #4, specific to each type of list.

```

16186 \cs_new_protected:Npn \__sort_main:NNNnNn #1#2#3#4#5#6
16187   {
16188     \group_begin:
16189     <package> \__sort_disable_toksdef:
16190     \__sort_compute_range:
16191     \int_set_eq:NN \l__sort_top_int \l__sort_min_int
16192     #2 #5
16193     {
16194       \if_int_compare:w \l__sort_top_int = \l__sort_max_int
16195         \__sort_too_long_error:NNw #3 #5
16196       \fi:
16197       \tex_toks:D \l__sort_top_int {##1}
16198       \int_incr:N \l__sort_top_int
16199     }
16200     \int_set:Nn \l__sort_length_int
16201     { \l__sort_top_int - \l__sort_min_int }
16202     \cs_set:Npn \__sort_compare:nn ##1 ##2 { #6 }
16203     \int_set:Nn \l__sort_block_int { 1 }
16204     \__sort_level:
16205     \use:x
16206   {

```

```

16207         \group_end:
16208             #1 \exp_not:N #5 {#4}
16209     }
16210 }

```

(End definition for `_sort_main:NNNnNn`.)

- \seq_sort:Nn** The first argument to `_sort_main:NNNnNn` is the final assignment function used, either `\tl_set:Nn` or `\tl_gset:Nn` to control local versus global results. The second argument is what mapping function is used when storing items to `\toks` registers, and the third breaks away from the loop. The fourth is used to build back the correct kind of list from the contents of the `\toks` registers, including the leading `\s_seq`. Fifth and sixth arguments are the variable to sort, and the sorting method as inline code.

```

16211 \cs_new_protected:Npn \seq_sort:Nn
16212 {
16213     \_sort_main:NNNnNn \tl_set:Nn
16214         \seq_map_inline:Nn \seq_map_break:n
16215         { \s_seq \_sort_toks:NN \exp_not:N \_seq_item:n }
16216     }
16217 \cs_generate_variant:Nn \seq_sort:Nn { c }
16218 \cs_new_protected:Npn \seq_gsort:Nn
16219 {
16220     \_sort_main:NNNnNn \tl_gset:Nn
16221         \seq_map_inline:Nn \seq_map_break:n
16222         { \s_seq \_sort_toks:NN \exp_not:N \_seq_item:n }
16223     }
16224 \cs_generate_variant:Nn \seq_gsort:Nn { c }

```

(End definition for `\seq_sort:Nn` and `\seq_gsort:Nn`. These functions are documented on page 61.)

- \tl_sort:Nn** Again, use `\tl_set:Nn` or `\tl_gset:Nn` to control the scope of the assignment. Mapping through the token list is done with `\tl_map_inline:Nn`, and producing the token list is very similar to sequences, removing `_seq_item:n`.

```

16225 \cs_new_protected:Npn \tl_sort:Nn
16226 {
16227     \_sort_main:NNNnNn \tl_set:Nn
16228         \tl_map_inline:Nn \tl_map_break:n
16229         { \_sort_toks:NN \prg_do_nothing: \prg_do_nothing: }
16230     }
16231 \cs_generate_variant:Nn \tl_sort:Nn { c }
16232 \cs_new_protected:Npn \tl_gsort:Nn
16233 {
16234     \_sort_main:NNNnNn \tl_gset:Nn
16235         \tl_map_inline:Nn \tl_map_break:n
16236         { \_sort_toks:NN \prg_do_nothing: \prg_do_nothing: }
16237     }
16238 \cs_generate_variant:Nn \tl_gsort:Nn { c }

```

(End definition for `\tl_sort:Nn` and `\tl_gsort:Nn`. These functions are documented on page 43.)

- \clist_sort:Nn** The case of empty comma-lists is a little bit special as usual, and filtered out: there is nothing to sort in that case. Otherwise, the input is done with `\clist_map_inline:Nn`, and the output requires some more elaborate processing than for sequences and token lists. The first comma must be removed. An item must be wrapped in an extra set `_sort_clist:NNn`

of braces if it contains either the space or the comma characters. This is taken care of by `\clist_wrap_item:n`, but `__sort_toks:NN` would simply feed `\tex_the:D \tex_toks:D <number>` as an argument to that function; hence we need to expand this argument once to unpack the register.

```

16239 \cs_new_protected:Npn \clist_sort:Nn
16240   { \__sort_clist>NNn \tl_set:Nn }
16241 \cs_new_protected:Npn \clist_gsort:Nn
16242   { \__sort_clist>NNn \tl_gset:Nn }
16243 \cs_generate_variant:Nn \clist_sort:Nn { c }
16244 \cs_generate_variant:Nn \clist_gsort:Nn { c }
16245 \cs_new_protected:Npn \__sort_clist>NNn #1#2#3
16246   {
16247     \clist_if_empty:NF #2
16248   {
16249     \__sort_main>NNNnNn #1
16250       \clist_map_inline:Nn \clist_map_break:n
16251     {
16252       \exp_last_unbraced:Nf \use_none:n
16253         { \__sort_toks:NN \exp_args:No \__clist_wrap_item:n }
16254     }
16255   #2 {#3}
16256 }
16257 }
```

(End definition for `\clist_sort:Nn`, `\clist_gsort:Nn`, and `__sort_clist>NNn`. These functions are documented on page 101.)

`__sort_toks:NN` Unpack the various `\toks` registers, from `\l__sort_min_int` to `\l__sort_top_int - 1`.
`__sort_toks>NNw` The functions #1 and #2 allow us to treat the three data structures in a unified way:

- for sequences, they are `\exp_not:N __seq_item:n`, expanding to the `__seq_item:n` separator, as expected;
- for token lists, they expand to nothing;
- for comma lists, they expand to `\exp_args:No \clist_wrap_item:n`, taking care of unpacking the register before letting the undocumented internal `clist` function `\clist_wrap_item:n` do the work of putting a comma and possibly braces.

```

16258 \cs_new:Npn \__sort_toks:NN #1#2
16259   { \__sort_toks>NNw #1 #2 \l__sort_min_int ; }
16260 \cs_new:Npn \__sort_toks>NNw #1#2#3 ;
16261   {
16262     \if_int_compare:w #3 < \l__sort_top_int
16263       #1 #2 { \tex_the:D \tex_toks:D #3 }
16264       \exp_after:wN \__sort_toks>NNw \exp_after:wN #1 \exp_after:wN #2
16265         \__int_value:w \__int_eval:w #3 + 1 \exp_after:wN ;
16266     \fi:
16267 }
```

(End definition for `__sort_toks:NN` and `__sort_toks>NNw`.)

33.4 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.

```

16268 \cs_new_protected:Npn \__sort_level:
16269  {
16270   \if_int_compare:w \l__sort_block_int < \l__sort_length_int
16271     \l__sort_end_int \l__sort_min_int
16272     \__sort_blocks:
16273     \tex_advance:D \l__sort_block_int \l__sort_block_int
16274     \exp_after:wN \__sort_level:
16275   \fi:
16276 }

```

(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 will index 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 $\leq \text{top}$. 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.

```

16277 \cs_new_protected:Npn \__sort_merge_blocks:
16278  {
16279   \l__sort_begin_int \l__sort_end_int
16280   \tex_advance:D \l__sort_end_int \l__sort_block_int
16281   \if_int_compare:w \l__sort_end_int < \l__sort_top_int
16282     \l__sort_A_int \l__sort_end_int
16283     \tex_advance:D \l__sort_end_int \l__sort_block_int
16284     \if_int_compare:w \l__sort_end_int > \l__sort_top_int
16285       \l__sort_end_int \l__sort_top_int
16286     \fi:
16287     \l__sort_B_int \l__sort_A_int
16288     \l__sort_C_int \l__sort_top_int
16289     \__sort_copy_block:
16290     \int_decr:N \l__sort_A_int
16291     \int_decr:N \l__sort_B_int
16292     \int_decr:N \l__sort_C_int
16293     \exp_after:wN \__sort_merge_blocks_aux:
16294     \exp_after:wN \__sort_merge_blocks:
16295   \fi:
16296 }

```

(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`.

```

16297 \cs_new_protected:Npn \__sort_copy_block:
16298 {
16299     \tex_toks:D \l__sort_C_int \tex_toks:D \l__sort_B_int
16300     \int_incr:N \l__sort_C_int
16301     \int_incr:N \l__sort_B_int
16302     \if_int_compare:w \l__sort_B_int = \l__sort_end_int
16303         \use_i:nn
16304     \fi:
16305     \__sort_copy_block:
16306 }
```

(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.

```

16307 \cs_new_protected:Npn \__sort_merge_blocks_aux:
16308 {
16309     \exp_after:wN \__sort_compare:nn \exp_after:wN
16310     { \tex_the:D \tex_toks:D \exp_after:wN \l__sort_A_int \exp_after:wN }
16311     \exp_after:wN { \tex_the:D \tex_toks:D \l__sort_C_int }
16312     \prg_do_nothing:
16313     \__sort_return_mark:N
16314     \__sort_return_mark:N
16315     \__sort_return_none_error:
16316 }
```

(End definition for `__sort_merge_blocks_aux:..`)

`\sort_return_same:` The marker removes one token. Each comparison should call `\sort_return_same:` or `\sort_return_swapped:` exactly once. If neither is called, `__sort_return_none_error:` is called.

```

16317 \cs_new_protected:Npn \sort_return_same: #1 \__sort_return_mark:N
16318 { #1 \__sort_return_mark:N \__sort_return_two_error:w \__sort_return_same: }
16319 \cs_new_protected:Npn \sort_return_swapped: #1 \__sort_return_mark:N
16320 { #1 \__sort_return_mark:N \__sort_return_two_error:w \__sort_return_swapped: }
16321 \cs_new_protected:Npn \__sort_return_mark:N #1 { }
16322 \cs_new_protected:Npn \__sort_return_none_error:
16323 {
16324     \__msg_kernel_error:nxxx { sort } { return-none }
16325     { \tex_the:D \tex_toks:D \l__sort_A_int }
16326     { \tex_the:D \tex_toks:D \l__sort_C_int }
16327     \__sort_return_same:
16328 }
16329 \cs_new_protected:Npn \__sort_return_two_error:w
```

```

16330      #1 \__sort_return_none_error:
16331      { \__msg_kernel_error:nn { sort } { return-two } }

```

(End definition for `__sort_return_same:` and others. These functions are documented on page ??.)

`__sort_return_same:` 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.

```

16332  \cs_new_protected:Npn \__sort_return_same:
16333  {
16334      \tex_toks:D \l__sort_B_int \tex_toks:D \l__sort_C_int
16335      \int_decr:N \l__sort_B_int
16336      \int_decr:N \l__sort_C_int
16337      \if_int_compare:w \l__sort_C_int < \l__sort_top_int
16338          \use_i:nn
16339      \fi:
16340      \__sort_merge_blocks_aux:
16341  }

```

(End definition for `__sort_return_same:..`)

`__sort_return_swapped:` 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:..`

```

16342  \cs_new_protected:Npn \__sort_return_swapped:
16343  {
16344      \tex_toks:D \l__sort_B_int \tex_toks:D \l__sort_A_int
16345      \int_decr:N \l__sort_B_int
16346      \int_decr:N \l__sort_A_int
16347      \if_int_compare:w \l__sort_A_int < \l__sort_begin_int
16348          \__sort_merge_blocks_end: \use_i:nn
16349      \fi:
16350      \__sort_merge_blocks_aux:
16351  }

```

(End definition for `__sort_return_swapped:..`)

`__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`.

```

16352  \cs_new_protected:Npn \__sort_merge_blocks_end:
16353  {
16354      \tex_toks:D \l__sort_B_int \tex_toks:D \l__sort_C_int
16355      \int_decr:N \l__sort_B_int
16356      \int_decr:N \l__sort_C_int
16357      \if_int_compare:w \l__sort_B_int < \l__sort_begin_int
16358          \use_i:nn
16359      \fi:
16360      \__sort_merge_blocks_end:
16361  }

```

(End definition for `_sort_merge_blocks_end`.)

33.5 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_recursion_tail \q_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 will be 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> {} \q_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 `<conditional> {<item>}`, 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 `\q_mark {<code>}`, and expands to `<code> <sorted list>`. Sorting the two parts of the list around the pivot is done with

```
\_\_sort_quick_split:NnNn #2 ... \q_mark
{
    \_\_sort_quick_split:NnNn #1 ... \q_mark {<code>}
    {<pivot>}
}
```

Items which are larger than the `<pivot>` are sorted, then placed after code that sorts the smaller items, and after the (braced) `<pivot>`.

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 `<conditional> {<item>}` 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 `<end-loop>` 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 `<end-loop>` 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 \q_stop once in a while, otherwise sorting lists of more than a few thousand items would exhaust a typical TeX'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 will sort the prepared list and perform the post action placed after \q_mark, namely removing the trailing \s_stop and \q_stop and leaving \exp_stop_f: to stop f-expansion.

```
16362 \cs_new:Npn \tl_sort:nN #1#2
16363   {
16364     \exp_not:f
16365     {
16366       \tl_if_blank:nF {#1}
16367       {
16368         \__sort_quick_prepare:Nnnn #2 { } { }
16369         #1
16370         { \__prg_break_point: \__sort_quick_prepare_end:NNNnw }
16371         \q_stop
16372       }
16373     }
16374   }
16375 \cs_new:Npn \__sort_quick_prepare:Nnnn #1#2#3#4
16376   {
16377     \__prg_break: #4 \__prg_break_point:
16378     \__sort_quick_prepare:Nnnn #1 { #2 #3 } { #1 {#4} }
16379   }
16380 \cs_new:Npn \__sort_quick_prepare_end:NNNnw #1#2#3#4#5 \q_stop
16381   {
16382     \__sort_quick_split:NnNn #4 \__sort_quick_end:nnTFNn { }
16383     \q_mark { \__sort_quick_cleanup:w \exp_stop_f: }
16384     \s_stop \q_stop
16385   }
16386 \cs_new:Npn \__sort_quick_cleanup:w #1 \s_stop \q_stop {#1}
```

(End definition for \tl_sort:nN and others. These functions are documented on page 43.)

```

\_\_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.

```

16387 \cs_new:Npn \_\_sort_quick_split:NnNn #1#2#3#4
16388  {
16389    #3 {#2} {#4} \_\_sort_quick_only_ii:NnnnnNn \_\_sort_quick_only_i:NnnnnNn
16390      \_\_sort_quick_single_end:nnnwnw
16391      { #3 {#4} } { } { } {#2}
16392    }
16393 \cs_new:Npn \_\_sort_quick_only_i:NnnnnNn #1#2#3#4#5#6#7
16394  {
16395    #6 {#5} {#7} \_\_sort_quick_split_ii:NnnnnNn \_\_sort_quick_only_i:NnnnnNn
16396      \_\_sort_quick_only_i_end:nnnwnw
16397      { #6 {#7} } { #3 #2 } { } {#5}
16398    }
16399 \cs_new:Npn \_\_sort_quick_only_ii:NnnnnNn #1#2#3#4#5#6#7
16400  {
16401    #6 {#5} {#7} \_\_sort_quick_only_ii:NnnnnNn \_\_sort_quick_split_i:NnnnnNn
16402      \_\_sort_quick_only_ii_end:nnnwnw
16403      { #6 {#7} } { } { #4 #2 } {#5}
16404    }
16405 \cs_new:Npn \_\_sort_quick_split_i:NnnnnNn #1#2#3#4#5#6#7
16406  {
16407    #6 {#5} {#7} \_\_sort_quick_split_ii:NnnnnNn \_\_sort_quick_split_i:NnnnnNn
16408      \_\_sort_quick_split_end:nnnwnw
16409      { #6 {#7} } { #3 #2 } {#4} {#5}
16410    }
16411 \cs_new:Npn \_\_sort_quick_split_ii:NnnnnNn #1#2#3#4#5#6#7
16412  {
16413    #6 {#5} {#7} \_\_sort_quick_split_ii:NnnnnNn \_\_sort_quick_split_i:NnnnnNn
16414      \_\_sort_quick_split_end:nnnwnw
16415      { #6 {#7} } {#3} { #4 #2 } {#5}
16416    }

```

(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 `\q_mark`. To avoid a memory problem described earlier, all of the ending functions read #6 until `\q_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.

```

16417 \cs_new:Npn \__sort_quick_end:nnTFNn #1#2#3#4#5#6 {#5}
16418 \cs_new:Npn \__sort_quick_single_end:nnnw #1#2#3#4 \q_mark #5#6 \q_stop
16419 { #5 {#3} #6 \q_stop }
16420 \cs_new:Npn \__sort_quick_only_i_end:nnnw #1#2#3#4 \q_mark #5#6 \q_stop
16421 {
16422     \__sort_quick_split:NnNn #1
16423         \__sort_quick_end:nnTFNn { } \q_mark {#5}
16424 {#3}
16425 #6 \q_stop
16426 }
16427 \cs_new:Npn \__sort_quick_only_ii_end:nnnw #1#2#3#4 \q_mark #5#6 \q_stop
16428 {
16429     \__sort_quick_split:NnNn #2
16430         \__sort_quick_end:nnTFNn { } \q_mark { #5 {#3} }
16431 #6 \q_stop
16432 }
16433 \cs_new:Npn \__sort_quick_split_end:nnnw #1#2#3#4 \q_mark #5#6 \q_stop
16434 {
16435     \__sort_quick_split:NnNn #2 \__sort_quick_end:nnTFNn { } \q_mark
16436 {
16437     \__sort_quick_split:NnNn #1
16438         \__sort_quick_end:nnTFNn { } \q_mark {#5}
16439 {#3}
16440 }
16441 #6 \q_stop
16442 }
```

(End definition for `__sort_quick_end:nnTFNn` and others.)

33.6 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`: getting rid of the final assignment. This error recovery won't work in a group.

```

16443 \cs_new_protected:Npn \__sort_error:
16444 {
16445     \cs_set_eq:NN \__sort_merge_blocks_aux: \prg_do_nothing:
16446     \cs_set_eq:NN \__sort_merge_blocks: \prg_do_nothing:
16447     \cs_set_protected:Npn \__sort_level: \use:x ##1 { \group_end: }
16448 }
```

(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.

```

16449  {*package}
16450  \cs_new_protected:Npn \__sort_disable_toksdef:
16451    { \cs_set_eq:NN \toksdef \__sort_disabled_toksdef:n }
16452  \cs_new_protected:Npn \__sort_disabled_toksdef:n #1
16453    {
16454      \__msg_kernel_error:nnn { sort } { toksdef }
16455      { \token_to_str:N #1 }
16456      \__sort_error:
16457      \tex_toksdef:D #1
16458    }
16459  \__msg_kernel_new:nnnn { sort } { toksdef }
16460  { Allocation-of-\iow_char:N\toks-registers-impossible-while-sorting. }
16461  {
16462    The-comparison-code-used-for-sorting-a-list-has-attempted-to-
16463    define#1-as-a-new-\iow_char:N\toks-register-using-\iow_char:N\newtoks-
16464    or-a-similar-command.-The-list-will-not-be-sorted.
16465  }
16466  </package>

```

(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.

```

16467  \cs_new_protected:Npn \__sort_too_long_error:NNw #1#2 \fi:
16468  {
16469    \fi:
16470    \__msg_kernel_error:nnxxx { sort } { too-large }
16471    { \token_to_str:N #2 }
16472    { \int_eval:n { \l__sort_true_max_int - \l__sort_min_int } }
16473    { \int_eval:n { \l__sort_top_int - \l__sort_min_int } }
16474    #1 \__sort_error:
16475  }
16476  \__msg_kernel_new:nnnn { sort } { too-large }
16477  { The-list#1-is-too-long-to-be-sorted-by-TeX. }
16478  {
16479    TeX-has#2-toks-registers-still-available:-
16480    this-only-allows-to-sort-with-up-to#3-
16481    items.-All-extra-items-will-be-deleted.
16482  }

```

(End definition for `__sort_too_long_error:NNw`.)

```

16483  \__msg_kernel_new:nnnn { sort } { return-none }
16484  { The-comparison-code-did-not-return. }
16485  {
16486    When-sorting-a-list,-the-code-to-compare-items#1-and#2-
16487    did-not-call-
16488    \iow_char:N\sort_return_same: ~nor~
16489    \iow_char:N\sort_return_swapped: .-
16490    Exactly-one-of-these-should-be-called.
16491  }
16492  \__msg_kernel_new:nnnn { sort } { return-two }
16493  { The-comparison-code-returned-multiple-times. }
16494  {
16495    When-sorting-a-list,-the-code-to-compare-items-called-

```

```

16496      \iow_char:N\sort_return_same: ~or~
16497      \iow_char:N\sort_return_swapped: ~multiple~times.~
16498      Exactly~one~of~these~should~be~called.
16499  }

```

33.7 Deprecated functions

\sort_ordered: These functions were renamed for consistency.

\sort_reversed:

```

16500  \cs_new_protected:Npn \sort_ordered:
16501  {
16502      \__msg_kernel_warning:nxxxx { kernel } { deprecated-command }
16503      { 2018-12-31 }
16504      { \token_to_str:N \sort_ordered: }
16505      { \token_to_str:N \sort_return_same: }
16506      \cs_gset_eq:NN \sort_ordered: \sort_return_same:
16507      \sort_return_same:
16508  }
16509  \cs_new_protected:Npn \sort_reversed:
16510  {
16511      \__msg_kernel_warning:nxxxx { kernel } { deprecated-command }
16512      { 2018-12-31 }
16513      { \token_to_str:N \sort_reversed: }
16514      { \token_to_str:N \sort_return_swapped: }
16515      \cs_gset_eq:NN \sort_reversed: \sort_return_swapped:
16516      \sort_return_swapped:
16517  }

```

(End definition for \sort_ordered: and \sort_reversed:.)

16518 </initex | package>

34 I3tl-analysis implementation

34.1 Internal functions

\s__tl: The format used to store token lists internally uses the scan mark \s__tl as a delimiter.

(End definition for \s__tl.)

__tl_analysis_map_inline:nn __tl_analysis_map_inline:nn {\langle token list\rangle} {\langle inline function\rangle}

Applies the ⟨inline function⟩ to each individual ⟨token⟩ in the ⟨token list⟩. The ⟨inline function⟩ receives three arguments:

- ⟨tokens⟩, which both o-expand and x-expand to the ⟨token⟩. The detailed form of ⟨token⟩ may change in later releases.
- ⟨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).
- ⟨char code⟩, a decimal representation of the character code of the token, -1 if it is a control sequence (with ⟨catcode⟩ 0).

For optimizations in `l3regex` (when matching control sequences), it may be useful to provide a `__tl_analysis_from_str_map_inline:nn` function, perhaps named `__str_analysis_map_inline:nn`.

34.2 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 `token` (even begin-group and end-group character tokens) in a way amenable to manipulating tokens individually. The best we can do is to find `tokens` which both o-expand and x-expand to the given `token`. 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

`<tokens> \s__t1 <catcode> <char code> \s__t1`

The `tokens` 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 `<catcode>` is given as a single hexadecimal digit, 0 for control sequences. The `<char code>` is given as a decimal number, -1 for control sequences.

Using delimited arguments lets us build the `tokens` progressively when doing an encoding conversion in `l3str-convert`. On the other hand, the delimiter `\s__t1` may not appear unbraced in `tokens`. This is not a problem because we are careful to wrap control sequences in braces (as an argument to `\exp_not:n`) when converting from a general token list to the internal format.

The current rule for converting a `token` to a balanced set of `tokens` which both o-expands and x-expands to it is the following.

- A control sequence `\cs` becomes `\exp_not:n { \cs } \s__t1 0 -1 \s__t1`.
- A begin-group character `{` becomes `\exp_after:wN { \if_false: } \fi: \s__t1 1 <char code> \s__t1`.
- An end-group character `}` becomes `\if_false: { \fi: } \s__t1 2 <char code> \s__t1`.
- A character with any other category code becomes `\exp_not:n {<character>} \s__t1 <hex catcode> <char code> \s__t1`.

¹⁶⁵¹⁹ `(*initex | package)`

¹⁶⁵²⁰ `@@=tl_analysis`

34.3 Variables and helper functions

`\s__t1` The scan mark `\s__t1` 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.

16521 `__scan_new:N \s__tl`

(End definition for `\s__tl`.)

`\l__tl_analysis_internal_t1` This token list variable is used to hand the argument of `\tl_show_analysis:n` to `\tl-show_analysis:N`.

16522 `\tl_new:N \l__tl_analysis_internal_t1`

(End definition for `\l__tl_analysis_internal_t1`.)

`\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`.

16523 `\cs_new_eq:NN \l__tl_analysis_token ?`

16524 `\cs_new_eq:NN \l__tl_analysis_char_token ?`

(End definition for `\l__tl_analysis_token` and `\l__tl_analysis_char_token`.)

`\l__tl_analysis_normal_int` The number of normal (N-type argument) tokens since the last special token.

16525 `\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.

16526 `\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.

16527 `\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.

16528 `\int_new:N \l__tl_analysis_type_int`

(End definition for `\l__tl_analysis_type_int`.)

`\g__tl_analysis_result_t1` The result of the conversion is stored in this token list, with a succession of items of the form

`<tokens> \s__tl <catcode> <char code> \s__tl`

16529 `\tl_new:N \g__tl_analysis_result_t1`

(End definition for `\g__tl_analysis_result_t1`.)

`_tl_analysis_extract_charcode:` 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 ‘`(char)`’.

```

16530 \cs_new:Npn \_tl_analysis_extract_charcode:
16531 {
16532     \exp_after:wN \_tl_analysis_extract_charcode_aux:w
16533         \token_to_meaning:N \l__tl_analysis_token
16534 }
16535 \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` 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.

```

16536 \cs_new:Npn \_tl_analysis_cs_space_count:NN #1 #2
16537 {
16538     \exp_after:wN #1
16539     \int_value:w \int_eval:w 0
16540     \exp_after:wN \_tl_analysis_cs_space_count:w
16541         \token_to_str:N #2
16542         \fi: \_tl_analysis_cs_space_count_end:w ; ~ !
16543 }
16544 \cs_new:Npn \_tl_analysis_cs_space_count:w #1 ~
16545 {
16546     \if_false: #1 #1 \fi:
16547     + 1
16548     \_tl_analysis_cs_space_count:w
16549 }
16550 \cs_new:Npn \_tl_analysis_cs_space_count_end:w ; #1 \fi: #2 !
16551 { \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`.)

34.4 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 TeX. 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.

To ease the difficult first pass, we first do some setup with `_tl_analysis_setup:n`. Active characters set equal to non-active characters cause trouble, so we disable all active

characters by setting them equal to `undefined` locally. We also set there the escape character to be printable (backslash, but this later oscillates between slash and backslash): this makes it possible 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 (we eliminate those in the setup step);
- 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 will 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 will be 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.

```

16552 \cs_new_protected:Npn \_tl_analysis:n #1
16553 {
16554     \group_begin:
16555         \group_align_safe_begin:
16556             \_tl_analysis_setup:n {#1}
16557             \_tl_analysis_a:n {#1}
16558             \_tl_analysis_b:n {#1}
16559         \group_align_safe_end:
16560     \group_end:
16561 }
```

(End definition for `_tl_analysis:n`.)

34.5 Setup

`_tl_analysis_setup:n` Active characters can cause problems later on in the processing, so the first step is to disable them, by setting them to `undefined`. Since Unicode contains too many characters to loop over all of them, we instead loop over the input token list as a string: any active character in the token list must appear in its string representation. The string is shortened a little by making the escape character unprintable. The active space must be disabled

separately (the loop skips over it otherwise), and we end the loop by feeding an odd non-N-type argument to the looping macro. For pTeX and upTeX we skip characters beyond [0, 255] because \\lccode only allows those values.

```

16562 \cs_new_protected:Npn \__tl_analysis_setup:n #1
16563 {
16564     \int_set:Nn \tex_escapechar:D { -1 }
16565     \exp_after:wN \__tl_analysis_disable_loop:N
16566         \tl_to_str:n {#1} { ~ } { ? \__prg_break: }
16567     \__prg_break_point:
16568     \scan_stop:
16569 }
16570 \group_begin:
16571     \char_set_catcode_active:N \^^@
16572     \cs_new_protected:Npn \__tl_analysis_disable_loop:N #1
16573 {
16574     \tex_lccode:D 0 = '#1 ~
16575     \tex_lowercase:D { \tex_let:D ^^@ } \tex_undefined:D
16576     \__tl_analysis_disable_loop:N
16577 }
16578 \cs_if_exist:NT \ptex_kanjiskip:D
16579 {
16580     \cs_gset_protected:Npn \__tl_analysis_disable_loop:N #1
16581     {
16582         \use_none:n #1 \scan_stop:
16583         \if_int_compare:w 256 > '#1 \exp_stop_f:
16584             \tex_lccode:D 0 = '#1 ~
16585             \tex_lowercase:D { \tex_let:D ^^@ } \tex_undefined:D
16586         \fi:
16587         \__tl_analysis_disable_loop:N
16588     }
16589 }
16590 \group_end:

```

(End definition for __tl_analysis_setup:n and __tl_analysis_disable_loop:N.)

34.6 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.

After the setup step, 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 undefined 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 will be 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 `__tl_analysis_nesting_int`. This reaches -1 when we read the closing brace.

```

16591 \cs_new_protected:Npn \__tl_analysis_a:n #1
16592   {
16593     \int_set:Nn \tex_escapechar:D { 92 }
16594     \int_zero:N \l__tl_analysis_normal_int
16595     \int_zero:N \l__tl_analysis_index_int
16596     \int_zero:N \l__tl_analysis_nesting_int
16597     \if_false: { \fi: \__tl_analysis_a_loop:w #1 }
16598     \int_decr:N \l__tl_analysis_index_int
16599   }

```

(End definition for `__tl_analysis_a:n`.)

`__tl_analysis_a_loop:w` Read one character and check its type.

```

16600 \cs_new_protected:Npn \__tl_analysis_a_loop:w
16601   { \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` 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).

```
16602 \cs_new_protected:Npn \__tl_analysis_a_type:w
```

```

16603  {
16604    \l__tl_analysis_type_int =
16605      \if_meaning:w \l__tl_analysis_token \c_space_token
16606        0
16607      \else:
16608        \if_catcode:w \exp_not:N \l__tl_analysis_token \c_group_begin_token
16609          1
16610        \else:
16611          \if_catcode:w \exp_not:N \l__tl_analysis_token \c_group_end_token
16612            - 1
16613        \else:
16614          2
16615        \fi:
16616      \fi:
16617    \fi:
16618    \exp_stop_f:
16619    \if_case:w \l__tl_analysis_type_int
16620      \exp_after:wN \__tl_analysis_a_space:w
16621      \or: \exp_after:wN \__tl_analysis_a_bgroup:w
16622      \or: \exp_after:wN \__tl_analysis_a_safe:N
16623      \else: \exp_after:wN \__tl_analysis_a_egroup:w
16624      \fi:
16625  }

```

(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: if it is a control sequence the result starts with the escape character; otherwise it is a true blank space, whose string representation is also a blank space. We test for that in `__tl_analysis_a_space_test:w`, after grabbing as `\l__tl_analysis_char_token` the first character of the string representation. 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 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.

```

16626  \cs_new_protected:Npn \__tl_analysis_a_space:w
16627  {
16628    \tex_afterassignment:D \__tl_analysis_a_space_test:w
16629    \exp_after:wN \cs_set_eq:NN
16630    \exp_after:wN \l__tl_analysis_char_token
16631    \token_to_str:N
16632  }
16633  \cs_new_protected:Npn \__tl_analysis_a_space_test:w
16634  {
16635    \if_meaning:w \l__tl_analysis_char_token \c_space_token
16636      \tex_toks:D \l__tl_analysis_index_int { \exp_not:n { ~ } }
16637      \__tl_analysis_a_store:
16638    \else:
16639      \int_incr:N \l__tl_analysis_normal_int

```

```
16640     \fi:  
16641         \_\_tl\_analysis_a\_loop:w  
16642     }  
  
(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_test:w
```

The token might be either a true character token with catcode 1 or 2, or it could be a control sequence. The only tricky case is if the character code happens to be equal to the escape character: then we change the escape character from backslash to solidus or back, so that the string representation of the true character and of a control sequence set equal to it start differently. Then probe what the first character of that string representation is: this is the place where we need `\l_t1_analysis_char_token` to be a separate control sequence from `\l_t1_analysis_token`, to compare them.

`_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 will 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_t1_analysis_type_int`. The number of normal tokens, 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).

```

16674 \cs_new_protected:Npn \__tl_analysis_a_store:
16675   {
16676     \tex_advance:D \l_t1_analysis_nesting_int \l_t1_analysis_type_int
16677     \if_int_compare:w \tex_lccode:D 0 = '\ \exp_stop_f:
16678       \tex_multiply:D \l_t1_analysis_type_int 2 \exp_stop_f:
16679     \fi:
16680     \tex_skip:D \l_t1_analysis_index_int
16681       = \l_t1_analysis_normal_int sp plus \l_t1_analysis_type_int sp \scan_stop:
16682     \int_incr:N \l_t1_analysis_index_int
16683     \int_zero:N \l_t1_analysis_normal_int
16684     \if_int_compare:w \l_t1_analysis_nesting_int = -1 \exp_stop_f:
16685       \cs_set_eq:NN \__tl_analysis_a_loop:w \scan_stop:
16686     \fi:
16687   }

```

(End definition for `__tl_analysis_a_store`.)

`__tl_analysis_a_safe:N` This should be the simplest case: since the upcoming token is safe, we can simply grab it in a second pass. However, other branches of the code must pass their tokens through `\string`, hence we do it here as well, with some optimizations. If the token is a single character (including space), the `\if_charcode:w` test yields true, and we simply 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_t1_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.

```

16688 \cs_new_protected:Npn \__tl_analysis_a_safe:N #
16689   {
16690     \if_charcode:w
16691       \scan_stop:

```

```

16692      \exp_after:wN \use_none:n \token_to_str:N #1 \prg_do_nothing:
16693      \scan_stop:
16694      \int_incr:N \l__tl_analysis_normal_int
16695  \else:
16696      \__tl_analysis_cs_space_count:NN \__tl_analysis_a_cs:ww #1
16697  \fi:
16698  \__tl_analysis_a_loop:w
16699 }
16700 \cs_new_protected:Npn \__tl_analysis_a_cs:ww #1; #2;
16701 {
16702     \if_int_compare:w #1 > 0 \exp_stop_f:
16703         \tex_skip:D \l__tl_analysis_index_int
16704             = \__int_eval:w \l__tl_analysis_normal_int + 1 sp \scan_stop:
16705         \tex_advance:D \l__tl_analysis_index_int #1 \exp_stop_f:
16706             \l__tl_analysis_normal_int #2 \exp_stop_f:
16707     \else:
16708         \tex_advance:D \l__tl_analysis_normal_int #2 \exp_stop_f:
16709     \fi:
16710 }

```

(End definition for `__tl_analysis_a_safe:N` and `__tl_analysis_a_cs:ww`.)

34.7 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`
`__tl_analysis_b_loop:w`

Start the loop with the index 0. No need for an end-marker: the loop will stop by itself when the last index is read. We will repeatedly oscillate between reading long stretches of normal tokens, and reading special tokens.

```

16711 \cs_new_protected:Npn \__tl_analysis_b:n #1
16712 {
16713     \tl_gset:Nx \g__tl_analysis_result_tl
16714     {
16715         \__tl_analysis_b_loop:w 0; #1
16716         \__prg_break_point:
16717     }
16718 }
16719 \cs_new:Npn \__tl_analysis_b_loop:w #1;
16720 {
16721     \exp_after:wN \__tl_analysis_b_normals:ww
16722     \__int_value:w \tex_skip:D #1 ; #1 ;
16723 }

```

(End definition for `__tl_analysis_b:n` and `__tl_analysis_b_loop:w`.)

`__tl_analysis_b_normals:ww`
`__tl_analysis_b_normal:wwN`

The first argument is the number of normal tokens which remain to be read, and the 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 `\s__tl`, hence must be hidden behind braces in the result.

```

16724 \cs_new:Npn \__tl_analysis_b_normals:ww #1;
16725 {
16726     \if_int_compare:w #1 = 0 \exp_stop_f:
16727         \__tl_analysis_b_special:w
16728     \fi:
16729     \__tl_analysis_b_normal:wwN #1;
16730 }
16731 \cs_new:Npn \__tl_analysis_b_normal:wwN #1; #2; #3
16732 {
16733     \exp_not:n { \exp_not:n { #3 } } \s__tl
16734     \if_charcode:w
16735         \scan_stop:
16736         \exp_after:wN \use_none:n \token_to_str:N #3 \prg_do_nothing:
16737         \scan_stop:
16738         \exp_after:wN \__tl_analysis_b_char:Nww
16739     \else:
16740         \exp_after:wN \__tl_analysis_b_cs:Nww
16741     \fi:
16742     #3 #1; #2;
16743 }

```

(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.

```

16744 \cs_new:Npx \__tl_analysis_b_char:Nww #1
16745 {
16746     \exp_not:N \if_meaning:w #1 \exp_not:N \tex_undefined:D
16747         \token_to_str:N D \exp_not:N \else:
16748         \exp_not:N \if_catcode:w #1 \c_catcode_other_token
16749             \token_to_str:N C \exp_not:N \else:
16750             \exp_not:N \if_catcode:w #1 \c_catcode_letter_token
16751                 \token_to_str:N B \exp_not:N \else:
16752                 \exp_not:N \if_catcode:w #1 \c_math_toggle_token      3 \exp_not:N \else:
16753                 \exp_not:N \if_catcode:w #1 \c_alignment_token      4 \exp_not:N \else:
16754                 \exp_not:N \if_catcode:w #1 \c_math_superscript_token 7 \exp_not:N \else:
16755                 \exp_not:N \if_catcode:w #1 \c_math_subscript_token 8 \exp_not:N \else:
16756                 \exp_not:N \if_catcode:w #1 \c_space_token
16757                     \token_to_str:N A \exp_not:N \else:
16758                         6
16759                     \exp_not:n { \fi: \fi: \fi: \fi: \fi: \fi: }
16760                     \exp_not:N \__int_value:w '#1 \s__tl
16761                     \exp_not:N \exp_after:wN \exp_not:N \__tl_analysis_b_normals:ww
16762                         \exp_not:N \__int_value:w \exp_not:N \__int_eval:w - 1 +
16763 }

```

(End definition for `__tl_analysis_b_char:Nww`.)

`__tl_analysis_b_cs:Nww` 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.

```

16764 \cs_new:Npn \__tl_analysis_b_cs:Nww #1
16765 {

```

```

16766     0 -1 \s__tl
16767     \_tl_analysis_cs_space_count:NN \_tl_analysis_b_cs_test:ww #1
16768   }
16769 \cs_new:Npn \_tl_analysis_b_cs_test:ww #1 ; #2 ; #3 ; #4 ;
16770   {
16771     \exp_after:wN \_tl_analysis_b_normals:ww
16772     \_int_value:w \_int_eval:w
16773     \if_int_compare:w #1 = 0 \exp_stop_f:
16774       #3
16775     \else:
16776       \tex_skip:D \_int_eval:w #4 + #1 \_int_eval_end:
16777     \fi:
16778     - #2
16779     \exp_after:wN ;
16780     \_int_value:w \_int_eval:w #4 + #1 ;
16781   }

```

(End definition for `_tl_analysis_b_cs:Nww` and `_tl_analysis_b_cs_test:ww`.)

`_tl_analysis_b_special: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.

```

16782 \group_begin:
16783   \char_set_catcode_other:N A
16784   \cs_new:Npn \_tl_analysis_b_special:w
16785     \fi: \_tl_analysis_b_normal:wwN 0 ; #1 ;
16786   {
16787     \fi:
16788     \if_int_compare:w #1 = \l__tl_analysis_index_int
16789       \exp_after:wN \_prg_break:
16790     \fi:
16791     \tex_the:D \tex_toks:D #1 \s__tl
16792     \if_case:w \etex_gluestretch:D \tex_skip:D #1 \exp_stop_f:
16793       A
16794       \or: 1
16795       \or: 1
16796       \else: 2
16797     \fi:
16798     \if_int_odd:w \etex_gluestretch:D \tex_skip:D #1 \exp_stop_f:
16799       \exp_after:wN \_tl_analysis_b_special_char:wn \_int_value:w
16800     \else:
16801       \exp_after:wN \_tl_analysis_b_special_space:w \_int_value:w
16802     \fi:
16803     \_int_eval:w 1 + #1 \exp_after:wN ;
16804     \token_to_str:N
16805   }
16806 \group_end:
16807 \cs_new:Npn \_tl_analysis_b_special_char:wn #1 ; #2
16808   {
16809     \_int_value:w '#2 \s__tl
16810     \_tl_analysis_b_loop:w #1 ;
16811   }

```

```

16812 \cs_new:Npn \__tl_analysis_b_special_space:w #1 ; ~
16813   {
16814     32 \s__tl
16815     \__tl_analysis_b_loop:w #1 ;
16816   }

(End definition for \__tl_analysis_b_special:w, \__tl_analysis_b_special_char:wN, and \__tl-
analysis_b_special_space:w.)
```

34.8 Mapping through the analysis

`__tl_analysis_map_inline:nn`

First obtain the analysis of the token list into `\g__tl_analysis_result_tl`. To allow nested mappings, increase the nesting depth `\g__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.

```

16817 \cs_new_protected:Npn \__tl_analysis_map_inline:nn #1
16818   {
16819     \__tl_analysis:n {#1}
16820     \int_gincr:N \g__prg_map_int
16821     \exp_args:Nc \__tl_analysis_map_inline_aux:Nn
16822     { _\__tl_analysis_map_inline_ \int_use:N \g__prg_map_int :wNw }
16823   }
16824 \cs_new_protected:Npn \__tl_analysis_map_inline_aux:Nn #1#2
16825   {
16826     \cs_gset_protected:Npn #1 ##1 \s__tl ##2 ##3 \s__tl
16827     {
16828       \use_none:n ##2
16829       #2
16830       #1
16831     }
16832     \exp_after:wN #1
16833     \g__tl_analysis_result_tl
16834     \s__tl { ? \tl_map_break: } \s__tl
16835     \__prg_break_point:Nn \tl_map_break: { \int_gdecr:N \g__prg_map_int }
16836   }
```

(End definition for `__tl_analysis_map_inline:nn` and `__tl_analysis_map_inline_aux:Nn`.)

34.9 Showing the results

`\tl_show_analysis:N`

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.

```

16837 \cs_new_protected:Npn \tl_show_analysis:N #1
16838   {
16839     \tl_if_exist:NTF #1
16840     {
16841       \exp_args:No \__tl_analysis:n {#1}
16842       \__msg_show_pre:nxxxxx { LaTeX / kernel } { show-tl-analysis }
16843       { \token_to_str:N #1 } { \tl_if_empty:NTF #1 { } { ? } } { } { }
```

```

16844      \__tl_analysis_show:
16845    }
16846    { \tl_show:N #1 }
16847  }
16848 \cs_new_protected:Npn \tl_show_analysis:n #1
16849  {
16850    \__tl_analysis:n {#1}
16851    \__msg_show_pre:nxxxxx { LaTeX / kernel } { show-tl-analysis }
16852    { } { \tl_if_empty:nTF {#1} { } { ? } } { } { }
16853    \__tl_analysis_show:
16854  }
16855 \cs_new_protected:Npn \__tl_analysis_show:
16856  {
16857    \group_begin:
16858    \exp_args:NNx
16859    \group_end:
16860    \__msg_show_wrap:n
16861    {
16862      \exp_after:wN \__tl_analysis_show_loop:wNw \g__tl_analysis_result_tl
16863      \s__tl { ? \__prg_break: } \s__tl
16864      \__prg_break_point:
16865    }
16866  }

```

(End definition for `\tl_show_analysis:N`, `\tl_show_analysis:n`, and `__tl_analysis_show:`. These functions are documented on page 195.)

`__tl_analysis_show_loop:wNw` 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.

```

16867 \cs_new:Npn \__tl_analysis_show_loop:wNw #1 \s__tl #2 #3 \s__tl
16868  {
16869    \use_none:n #2
16870    \exp_not:n { \\ > \\ }
16871    \if_int_compare:w "#2 = 0 \exp_stop_f:
16872      \exp_after:wN \__tl_analysis_show_cs:n
16873    \else:
16874      \if_int_compare:w "#2 = 13 \exp_stop_f:
16875        \exp_after:wN \exp_after:wN
16876        \exp_after:wN \__tl_analysis_show_active:n
16877      \else:
16878        \exp_after:wN \exp_after:wN
16879        \exp_after:wN \__tl_analysis_show_normal:n
16880      \fi:
16881    \fi:
16882    {#1}
16883    \__tl_analysis_show_loop:wNw
16884  }

```

(End definition for `__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 TeX's alignment status.

```

16885 \cs_new:Npn \__tl_analysis_show_normal:n #1
16886 {
16887     \exp_after:wN \token_to_str:N #1 ~
16888     ( \exp_after:wN \token_to_meaning:N #1 )
16889 }

```

(End definition for `__tl_analysis_show_normal:n`.)

`__tl_analysis_show_value:N` This expands to the value of #1 if it has any.

```

16890 \cs_new:Npn \__tl_analysis_show_value:N #1
16891 {
16892     \token_if_expandable:NF #1
16893     {
16894         \token_if_chardef:NTF      #1 \__prg_break: { }
16895         \token_if_mathchardef:NTF #1 \__prg_break: { }
16896         \token_if_dim_register:NTF #1 \__prg_break: { }
16897         \token_if_int_register:NTF #1 \__prg_break: { }
16898         \token_if_skip_register:NTF #1 \__prg_break: { }
16899         \token_if_toks_register:NTF #1 \__prg_break: { }
16900         \use_none:nnn
16901         \__prg_break_point:
16902         \use:n { \exp_after:wN = \tex_the:D #1 }
16903     }
16904 }

```

(End definition for `__tl_analysis_show_value:N`.)

`__tl_analysis_show_cs:n`
`__tl_analysis_show_active:n`
`__tl_analysis_show_long:nn`
`__tl_analysis_show_long_aux:nnnn`

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`.

```

16905 \cs_new:Npn \__tl_analysis_show_cs:n #1
16906     { \exp_args:No \__tl_analysis_show_long:nn {#1} { control-sequence= } }
16907 \cs_new:Npn \__tl_analysis_show_active:n #1
16908     { \exp_args:No \__tl_analysis_show_long:nn {#1} { active-character= } }
16909 \cs_new:Npn \__tl_analysis_show_long:nn #1
16910     {
16911         \__tl_analysis_show_long_aux:oofn
16912         { \token_to_str:N #1 }
16913         { \token_to_meaning:N #1 }
16914         { \__tl_analysis_show_value:N #1 }
16915     }
16916 \cs_new:Npn \__tl_analysis_show_long_aux:nnnn #1#2#3#4
16917     {
16918         \int_compare:nNnTF
16919         { \str_count:n { #1 ~ ( #4 #2 #3 ) } }
16920         > { \l_iow_line_count_int - 3 }
16921         {
16922             \str_range:nnn { #1 ~ ( #4 #2 #3 ) } { 1 }
16923             {
16924                 \l_iow_line_count_int - 3
16925                 - \str_count:N \c__tl_analysis_show_etc_str
16926             }
16927             \c__tl_analysis_show_etc_str
16928     }

```

```

16929      { #1 ~ ( #4 #2 #3 ) }
16930    }
16931 \cs_generate_variant:Nn \__tl_analysis_show_long_aux:nnnn { oof }

(End definition for \__tl_analysis_show_cs:n and others.)

```

34.10 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.

```

16932 \tl_const:Nx \c__tl_analysis_show_etc_str % (
16933   { \token_to_str:N \ETC. }

(End definition for \c__tl_analysis_show_etc_str.)

16934 \__msg_kernel_new:nnn { kernel } { show-tl-analysis }
16935   {
16936     \The~token~list~\tl_if_empty:nF {#1} { #1 ~ }
16937     \tl_if_empty:nTF {#2}
16938     { \is~empty }
16939     { \contains~the~tokens: }
16940   }
16941 </initex | package>

```

35 I3tl-build implementation

```

16942 <*initex | package>
16943 <@=tlbuild>

```

35.1 Variables and helper functions

\l__tl_build_start_index_int Integers pointing to the starting index (currently always starts at zero), and the current index. The corresponding \toks are accessed directly by number.

```

16944 \int_new:N \l__tl_build_start_index_int
16945 \int_new:N \l__tl_build_index_int

```

(End definition for \l__tl_build_start_index_int and \l__tl_build_index_int.)

\l__tl_build_result_t1 The resulting token list is normally built in one go by unpacking all \toks in some range. In the rare cases where there are too many __tl_build_one:n commands, leading to the depletion of registers, the contents of the current set of \toks is unpacked into \l__tl_build_result_t1. This prevents overflow from affecting the end-user (beyond an obvious performance hit).

```
16946 \tl_new:N \l__tl_build_result_t1
```

(End definition for \l__tl_build_result_t1.)

__tl_build_unpack: __tl_build_unpack_loop:w The various pieces of the token list are built in \toks from the start_index (inclusive) to the (current) index (excluded). Those \toks are unpacked and stored in order in the result token list. Optimizations would be possible here, for instance, unpacking 10 \toks at a time with a macro expanding to \the\toks#10... \the\toks#19, but this should be kept for much later.

```
16947 \cs_new_protected:Npn \__tl_build_unpack:
```

```
16948 {
16949     \tl_put_right:Nx \l__tl_build_result_tl
16950     {
16951         \exp_after:wN \__tl_build_unpack_loop:w
16952         \int_use:N \l__tl_build_start_index_int
16953         \__prg_break_point:
16954     }
16955 }
16956 \cs_new:Npn \__tl_build_unpack_loop:w #1 ;
16957 {
16958     \if_int_compare:w #1 = \l__tl_build_index_int
16959     \exp_after:wN \__prg_break:
16960     \fi:
16961     \tex_the:D \tex_toks:D #1 \exp_stop_f:
16962     \exp_after:wN \__tl_build_unpack_loop:w
16963     \int_use:N \__int_eval:w #1 + 1 ;
16964 }
```

(End definition for `_tl_build_unpack`: and `_tl_build_unpack_loop:w`.)

35.2 Building the token list

`_tl_build:Nw` Similar to what is done for coffins: redefine some command, here `_tl_build_end_-aux:n` to hold the relevant assignment (see `_tl_build_end:` for details). Then initialize the start index and the current index at zero, and empty the `result` token list.

```
\_tl_gbuild_x:Nw
\_\_tl_build_aux:NNw 16965 \cs_new_protected:Npn \_\_tl_build:Nw
16966  { \_\_tl_build_aux:NNw \tl_set:Nn }
16967 \cs_new_protected:Npn \_\_tl_build_x:Nw
16968  { \_\_tl_build_aux:NNw \tl_set:Nx }
16969 \cs_new_protected:Npn \_\_tl_gbuild:Nw
16970  { \_\_tl_build_aux:NNw \tl_gset:Nn }
16971 \cs_new_protected:Npn \_\_tl_gbuild_x:Nw
16972  { \_\_tl_build_aux:NNw \tl_gset:Nx }
16973 \cs_new_protected:Npn \_\_tl_build_aux:NNw #1#2
16974  {
16975      \group_begin:
16976          \cs_set:Npn \_\_tl_build_end_assignment:n
16977              { \group_end: #1 #2 }
16978          \int_zero:N \l_\_tl_build_start_index_int
16979          \int_zero:N \l_\_tl_build_index_int
16980          \tl_clear:N \l_\_tl_build_result_tl
16981      }
```

(End definition for _t1 build:Nw and others.)

`_tl_build_end:` When we are done building a token list, unpack all `\toks` into the `result` token list, and expand this list before closing the group. The `_tl_build_end_assignment:n` function is defined by `_tl_build_aux:NW` to end the group and hold the relevant assignment. Its value outside is irrelevant, but just in case, we set it to a function which would clean up the contents of `\l_tl_build_result_tl`.

```
16982 \cs_new_protected:Npn \__tl_build_end:
16983 {
16984     \__tl_build_unpack:
16985     \exp_args:No
```

```

16986     \__tl_build_end_assignment:n \l__tl_build_result_tl
16987   }
16988 \cs_new_eq:NN \__tl_build_end_assignment:n \use_none:n
(End definition for \__tl_build_end: and \__tl_build_end_assignment:n.)

```

__tl_build_one:n Store the tokens in a free `\toks`, then move the pointer to the next one. If we overflow, **__tl_build_one:o** unpack the current `\toks`, and reset the current index, preparing to fill more `\toks`. This **__tl_build_one:x** could be optimized by avoiding to read #1, using `\afterassignment`.

```

16989 \cs_new_protected:Npn \__tl_build_one:n #1
16990   {
16991     \tex_toks:D \l__tl_build_index_int {#1}
16992     \int_incr:N \l__tl_build_index_int
16993     \if_int_compare:w \l__tl_build_index_int > \c_max_register_int
16994       \__tl_build_unpack:
16995       \l__tl_build_index_int \l__tl_build_start_index_int
16996     \fi:
16997   }
16998 \cs_new_protected:Npn \__tl_build_one:o #1
16999   {
17000     \tex_toks:D \l__tl_build_index_int \exp_after:wN {#1}
17001     \int_incr:N \l__tl_build_index_int
17002     \if_int_compare:w \l__tl_build_index_int > \c_max_register_int
17003       \__tl_build_unpack:
17004       \l__tl_build_index_int \l__tl_build_start_index_int
17005     \fi:
17006   }
17007 \cs_new_protected:Npn \__tl_build_one:x #1
17008   { \use:x { \__tl_build_one:n {#1} } }
(End definition for \__tl_build_one:n.)

```

17009 ⟨/initex | package⟩

36 I3regex implementation

```

17010 (*initex | package)
17011 (@@=regex)

```

36.1 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 T_EX 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 $\langle position \rangle$, with $\text{min_pos} - 1 \leq \langle position \rangle \leq \text{max_pos}$. The lowest and highest positions correspond to imaginary begin and end markers (with inaccessible category code and character code).
- *Query*: the token list to which we apply the regular expression.
- *State*: each state of the NFA is labelled by an integer $\langle state \rangle$ with $\text{min_state} \leq \langle state \rangle < \text{max_state}$.
- *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 $\text{\l1_regex_step_int}$ is a unique id for all the steps of the matching algorithm.

We use `I3intarray` to manipulate arrays of integers (stored into some dimension registers in scaled points). 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 compiling, `\toks` registers are used under the hood by functions from the `I3tl-build` module. When building, $\text{\toks}\langle state \rangle$ holds the tests and actions to perform in the $\langle state \rangle$ of the NFA. When matching,

- `\g__regex_state_active_intarray` holds the last $\langle step \rangle$ in which each $\langle state \rangle$ was active.
- `\g__regex_thread_state_intarray` maps each $\langle thread \rangle$ (with $\text{min_active} \leq \langle thread \rangle < \text{max_active}$) to the $\langle state \rangle$ in which the $\langle thread \rangle$ currently is. The $\langle threads \rangle$ or ordered starting from the best to the least preferred.
- `\toks\langle thread \rangle` holds the submatch information for the $\langle thread \rangle$, as the contents of a property list.
- `\g__regex_charcode_intarray` and `\g__regex_catcode_intarray` hold the character codes and category codes of tokens at each $\langle position \rangle$ 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.
- `\toks\langle position \rangle` holds $\langle tokens \rangle$ which o- and x-expand to the $\langle position \rangle$ -th token in the query.

- `\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.

`\count` registers are not abused, which means that we can safely use named integers in this module. Note that `\box` registers are not abused either; maybe we could leverage those for some purpose.

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.

36.2 Helpers

`_regex_standard_escapechar`: Make the `\escapechar` into the standard backslash.

```
17012 \cs_new_protected:Npn \_regex_standard_escapechar:
17013   { \int_set:Nn \tex_escapechar:D { '\\ } }
```

(End definition for `_regex_standard_escapechar:..`)

`_regex_toks_use:w`: Unpack a `\toks` given its number.

```
17014 \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.

```
17015 \cs_new_protected:Npn \_regex_toks_clear:N #1
17016   { \tex_toks:D #1 { } }
17017 \cs_new_eq:NN \_regex_toks_set:Nn \tex_toks:D
17018 \cs_new_protected:Npn \_regex_toks_set:No #1
17019   { \_regex_toks_set:Nn #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`.

```
17020 \cs_new_protected:Npn \_regex_toks_memcpy:NNn #1#2#3
17021   {
17022     \prg_replicate:nn {#3}
17023       {
17024         \tex_toks:D #1 = \tex_toks:D #2
17025         \int_incr:N #1
17026         \int_incr:N #2
17027       }
17028   }
```

(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 or to the right. The expansion is done “by hand” for optimization (these operations are 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`.

```

17029 \cs_new_protected:Npn \__regex_toks_put_left:Nx #1#2
17030 {
17031   \cs_set:Npx \__regex_tmp:w { #2 }
17032   \tex_toks:D #1 \exp_after:wN \exp_after:wN \exp_after:wN
17033     { \exp_after:wN \__regex_tmp:w \tex_the:D \tex_toks:D #1 }
17034 }
17035 \cs_new_protected:Npn \__regex_toks_put_right:Nx #1#2
17036 {
17037   \cs_set:Npx \__regex_tmp:w {#2}
17038   \tex_toks:D #1 \exp_after:wN
17039     { \tex_the:D \tex_toks:D \exp_after:wN #1 \__regex_tmp:w }
17040 }
17041 \cs_new_protected:Npn \__regex_toks_put_right:Nn #1#2
17042   { \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_current_cs_to_str:` Expands to the string representation of the token (known to be a control sequence) at the current position `\l__regex_current_pos_int`. It should only be used in x-expansion to avoid losing a leading space.

```

17043 \cs_new:Npn \__regex_current_cs_to_str:
17044 {
17045   \exp_after:wN \exp_after:wN \exp_after:wN \cs_to_str:N
17046   \tex_the:D \tex_toks:D \l__regex_current_pos_int
17047 }

(End definition for \__regex_current_cs_to_str..)

```

36.2.1 Constants and variables

`__regex_tmp:w` Temporary function used for various short-term purposes.

```
17048 \cs_new:Npn \__regex_tmp:w { }
```

(End definition for `__regex_tmp:w`.)

`\l__regex_internal_a_tl` Temporary variables used for various purposes.

```

17049 \tl_new:N \l__regex_internal_a_tl
17050 \tl_new:N \l__regex_internal_b_tl
17051 \int_new:N \l__regex_internal_a_int
17052 \int_new:N \l__regex_internal_b_int
17053 \int_new:N \l__regex_internal_c_int
17054 \bool_new:N \l__regex_internal_bool
17055 \seq_new:N \l__regex_internal_seq
17056 \tl_new:N \g__regex_internal_tl

```

(End definition for `\l__regex_internal_a_tl` and others.)

\c__regex_no_match_regex 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 \regex_new:N.

```

17057 \tl_const:Nn \c__regex_no_match_regex
17058 {
17059   \__regex_branch:n
17060   { \__regex_class:NnnN \c_true_bool { } { 1 } { 0 } \c_true_bool }
17061 }
```

(End definition for \c__regex_no_match_regex.)

\g__regexCharCode_intarray
\g__regexCatcode_intarray
\g__regexBalance_intarray The first thing we do when matching is to go once through the query token list and store the information for each token into \g__regexCharCode_intarray, \g__regexCatcode_intarray and \toks registers. We also store the balance of begin-group/end-group characters into \g__regexBalance_intarray.

```

17062 \__intarray_new:Nn \g__regexCharCode_intarray { 65536 }
17063 \__intarray_new:Nn \g__regexCatcode_intarray { 65536 }
17064 \__intarray_new:Nn \g__regexBalance_intarray { 65536 }
```

(End definition for \g__regexCharCode_intarray, \g__regexCatcode_intarray, and \g__regexBalance_intarray.)

\l__regexBalance_int During this phase, \l__regexBalance_int 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.

```
17065 \int_new:N \l__regexBalance_int
```

(End definition for \l__regexBalance_int.)

\l__regexCsName_tl This variable is used in __regex_item_cs:n to store the csname of the currently-tested token when the regex contains a sub-regex for testing csnames.

```
17066 \tl_new:N \l__regexCsName_tl
```

(End definition for \l__regexCsName_tl.)

36.2.2 Testing characters

\c__regex_ascii_min_int
\c__regex_ascii_max_control_int
\c__regex_ascii_max_int 17067 \int_const:Nn \c__regex_ascii_min_int { 0 }
17068 \int_const:Nn \c__regex_ascii_max_control_int { 31 }
17069 \int_const:Nn \c__regex_ascii_max_int { 127 }

(End definition for \c__regex_ascii_min_int, \c__regex_ascii_max_control_int, and \c__regex_ascii_max_int.)

\c__regex_ascii_lower_int 17070 \int_const:Nn \c__regex_ascii_lower_int { 'a' - 'A' }

(End definition for \c__regex_ascii_lower_int.)

__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⟩ ... ⟨testn

```

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.

```

17071 \cs_new_protected:Npn \__regex_break_true:w
17072     #1 \__regex_break_point:TF #2 #3 {#2}
17073 \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 will thus match `\D` and other negated properties; this case is caught by another part of the code.

```

17074 \cs_new_protected:Npn \__regex_item_reverse:n #1
17075     {
17076         #1
17077         \__regex_break_point:TF { } \__regex_break_true:w
17078     }

```

(End definition for `__regex_item_reverse:n`.)

`__regex_item_caseful_equal:n` Simple comparisons triggering `__regex_break_true:w` when true.

```

17079 \cs_new_protected:Npn \__regex_item_caseful_equal:n #1
17080     {
17081         \if_int_compare:w #1 = \l__regex_current_char_int
17082             \exp_after:wN \__regex_break_true:w
17083         \fi:
17084     }
17085 \cs_new_protected:Npn \__regex_item_caseful_range:nn #1 #2
17086     {
17087         \reverse_if:N \if_int_compare:w #1 > \l__regex_current_char_int
17088             \reverse_if:N \if_int_compare:w #2 < \l__regex_current_char_int
17089                 \exp_after:wN \exp_after:wN \exp_after:wN \__regex_break_true:w
17090             \fi:
17091         \fi:
17092     }

```

(End definition for `__regex_item_caseful_equal:n` and `__regex_item_caseful_range:nn`.)

`__regex_item_caseless_equal:n` For caseless matching, we perform the test both on the `current_char` and on the `case_changed_char`. Before doing the second set of tests, we make sure that `case_changed_char` has been computed.

```

17093 \cs_new_protected:Npn \__regex_item_caseless_equal:n #1
17094     {
17095         \if_int_compare:w #1 = \l__regex_current_char_int
17096             \exp_after:wN \__regex_break_true:w
17097         \fi:
17098         \if_int_compare:w \l__regex_case_changed_char_int = \c_max_int
17099             \__regex_compute_case_changed_char:
17100         \fi:
17101         \if_int_compare:w #1 = \l__regex_case_changed_char_int
17102             \exp_after:wN \__regex_break_true:w

```

```

17103     \fi:
17104   }
17105 \cs_new_protected:Npn \__regex_item_caseless_range:nn #1 #2
17106   {
17107     \reverse_if:N \if_int_compare:w #1 > \l__regex_current_char_int
17108       \reverse_if:N \if_int_compare:w #2 < \l__regex_current_char_int
17109         \exp_after:wN \exp_after:wN \exp_after:wN \__regex_break_true:w
17110       \fi:
17111     \fi:
17112     \if_int_compare:w \l__regex_case_changed_char_int = \c_max_int
17113       \__regex_compute_case_changed_char:
17114     \fi:
17115     \reverse_if:N \if_int_compare:w #1 > \l__regex_case_changed_char_int
17116       \reverse_if:N \if_int_compare:w #2 < \l__regex_case_changed_char_int
17117         \exp_after:wN \exp_after:wN \exp_after:wN \__regex_break_true:w
17118       \fi:
17119     \fi:
17120   }

```

(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 (or rather, when it is set to the marker value `\c_max_int`). 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.

```

17121 \cs_new_protected:Npn \__regex_compute_case_changed_char:
17122   {
17123     \int_set_eq:NN \l__regex_case_changed_char_int \l__regex_current_char_int
17124     \if_int_compare:w \l__regex_current_char_int > 'Z \exp_stop_f:
17125       \if_int_compare:w \l__regex_current_char_int > 'z \exp_stop_f: \else:
17126         \if_int_compare:w \l__regex_current_char_int < 'a \exp_stop_f: \else:
17127           \int_sub:Nn \l__regex_case_changed_char_int { \c__regex_ascii_lower_int }
17128         \fi:
17129       \fi:
17130     \else:
17131       \if_int_compare:w \l__regex_current_char_int < 'A \exp_stop_f: \else:
17132         \int_add:Nn \l__regex_case_changed_char_int { \c__regex_ascii_lower_int }
17133       \fi:
17134     \fi:
17135   }

```

(End definition for `__regex_compute_case_changed_char:..`)

`__regex_item_equal:n` `__regex_item_range:nn`: 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.

```

17136 \cs_new_eq:NN \__regex_item_equal:n ?
17137 \cs_new_eq:NN \__regex_item_range:nn ?

```

(End definition for `__regex_item_equal:n` and `__regex_item_range:nn`.)

`__regex_item_catcode:nT` `__regex_item_catcode_reverse:nT` `__regex_item_catcode:`: 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.

```

17138 \cs_new_protected:Npn \__regex_item_catcode:
17139   {
17140     "
17141     \if_case:w \l__regex_current_catcode_int
17142       1          \or: 4          \or: 10          \or: 40
17143       \or: 100        \or:           \or: 1000        \or: 4000
17144       \or: 10000       \or:           \or: 100000      \or: 400000
17145       \or: 1000000    \or: 4000000 \else: 1*0
17146     \fi:
17147   }
17148 \cs_new_protected:Npn \__regex_item_catcode:nT #1
17149   {
17150     \if_int_odd:w \__int_eval:w #1 / \__regex_item_catcode: \__int_eval_end:
17151       \exp_after:wN \use:n
17152     \else:
17153       \exp_after:wN \use_none:n
17154     \fi:
17155   }
17156 \cs_new_protected:Npn \__regex_item_catcode_reverse:nT #1#2
17157   { \__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 `<category>-<character code>` pair, or an exact control sequence, more precisely one of several possible control sequences.

```

17158 \cs_new_protected:Npn \__regex_item_exact:nn #1#2
17159   {
17160     \if_int_compare:w #1 = \l__regex_current_catcode_int
17161       \if_int_compare:w #2 = \l__regex_current_char_int
17162         \exp_after:wN \exp_after:wN \exp_after:wN \__regex_break_true:w
17163       \fi:
17164     \fi:
17165   }
17166 \cs_new_protected:Npn \__regex_item_exact_cs:n #1
17167   {
17168     \int_compare:nNnTF \l__regex_current_catcode_int = 0
17169     {
17170       \tl_set:Nx \l__regex_internal_a_tl
17171         { \scan_stop: \__regex_current_cs_to_str: \scan_stop: }
17172       \tl_if_in:noTF { \scan_stop: #1 \scan_stop: } \l__regex_internal_a_tl
17173         { \__regex_break_true:w } { }
17174     }
17175   }
17176 }
```

(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. The three `\exp_after:wN` expand the contents of the `\toks<current position>` (of the form `\exp_not:n {<control sequence>}`) to `<control sequence>`. We store

the cs name before building states for the cs, as those states may overlap with toks registers storing the user's input.

```

17177 \cs_new_protected:Npn \__regex_item_cs:n #1
17178 {
17179   \int_compare:nNnT \l__regex_current_catcode_int = 0
17180   {
17181     \group_begin:
17182       \tl_set:Nx \l__regex_cs_name_tl { \__regex_current_cs_to_str: }
17183       \__regex_single_match:
17184       \__regex_disable_submatches:
17185       \__regex_build_for_cs:n {#1}
17186       \bool_set_eq:NN \l__regex_saved_success_bool \g__regex_success_bool
17187       \exp_args:NV \__regex_match:n \l__regex_cs_name_tl
17188       \if_meaning:w \c_true_bool \g__regex_success_bool
17189         \group_insert_after:N \__regex_break_true:w
17190       \fi:
17191       \bool_gset_eq:NN \g__regex_success_bool \l__regex_saved_success_bool
17192     \group_end:
17193   }
17194 }
```

(End definition for `__regex_item_cs:n`.)

36.2.3 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]`, `\w=[0-9A-Z_a-z]`, `\s=[\u\^\^I\^\^J\^\^L\^\^M]`, `\h=[\u\^\^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.

```

17195 \cs_new_protected:Npn \__regex_prop_d:
17196   { \__regex_item_caseful_range:nn { '0 } { '9 } }
17197 \cs_new_protected:Npn \__regex_prop_h:
17198   {
17199     \__regex_item_caseful_equal:n { '\ }
17200     \__regex_item_caseful_equal:n { '\^\^I }
17201   }
17202 \cs_new_protected:Npn \__regex_prop_s:
17203   {
17204     \__regex_item_caseful_equal:n { '\ }
17205     \__regex_item_caseful_equal:n { '\^\^I }
17206     \__regex_item_caseful_equal:n { '\^\^J }
17207     \__regex_item_caseful_equal:n { '\^\^L }
17208     \__regex_item_caseful_equal:n { '\^\^M }
17209   }
17210 \cs_new_protected:Npn \__regex_prop_v:
17211   { \__regex_item_caseful_range:nn { '\^\^J } { '\^\^M } } % lf, vtab, ff, cr
17212 \cs_new_protected:Npn \__regex_prop_w:
17213   {
17214     \__regex_item_caseful_range:nn { 'a } { 'z }
17215     \__regex_item_caseful_range:nn { 'A } { 'Z }
17216     \__regex_item_caseful_range:nn { '0 } { '9 }
17217     \__regex_item_caseful_equal:n { '_ }
```

```

17218     }
17219 \cs_new_protected:Npn \__regex_prop_N:
17220 {
17221     \__regex_item_reverse:n
17222     { \__regex_item_caseful_equal:n { '\^J' } }
17223 }

```

(End definition for `__regex_prop_d:` and others.)

`__regex_posix_alnum:` POSIX properties. No surprise.

```

\__regex_posix_alpha: 17224 \cs_new_protected:Npn \__regex_posix_alnum:
\__regex_posix_ascii: 17225 { \__regex_posix_alpha: \__regex_posix_digit: }
\__regex_posix_blank: 17226 \cs_new_protected:Npn \__regex_posix_alpha:
\__regex_posix_cntrl: 17227 { \__regex_posix_lower: \__regex_posix_upper: }
\__regex_posix_digit: 17228 \cs_new_protected:Npn \__regex_posix_ascii:
\__regex_posix_graph: 17229 {
\__regex_posix_lower: 17230     \__regex_item_caseful_range:nn
\__regex_posix_print: 17231     \c_regex_ascii_min_int
\__regex_posix_punct: 17232     \c_regex_ascii_max_int
\__regex_posix_space: 17233 }
17234 \cs_new_eq:NN \__regex_posix_blank: \__regex_prop_h:
\__regex_posix_upper: 17235 \cs_new_protected:Npn \__regex_posix_cntrl:
\__regex_posix_word: 17236 {
\__regex_posix_xdigit: 17237     \__regex_item_caseful_range:nn
17238     \c_regex_ascii_min_int
17239     \c_regex_ascii_max_control_int
17240     \__regex_item_caseful_equal:n \c_regex_ascii_max_int
17241 }
17242 \cs_new_eq:NN \__regex_posix_digit: \__regex_prop_d:
17243 \cs_new_protected:Npn \__regex_posix_graph:
17244 { \__regex_item_caseful_range:nn { '!' } { '\~{}' } }
17245 \cs_new_protected:Npn \__regex_posix_lower:
17246 { \__regex_item_caseful_range:nn { 'a' } { 'z' } }
17247 \cs_new_protected:Npn \__regex_posix_print:
17248 { \__regex_item_caseful_range:nn { '\`{}' } { '\~{}' } }
17249 \cs_new_protected:Npn \__regex_posix_punct:
17250 {
17251     \__regex_item_caseful_range:nn { '!' } { '/' }
17252     \__regex_item_caseful_range:nn { ':' } { 'ø' }
17253     \__regex_item_caseful_range:nn { '[' } { '[' }
17254     \__regex_item_caseful_range:nn { '\{ ' } { '\~{}' }
17255 }
17256 \cs_new_protected:Npn \__regex_posix_space:
17257 {
17258     \__regex_item_caseful_equal:n { '\`{}' }
17259     \__regex_item_caseful_range:nn { '\^I' } { '\^M' }
17260 }
17261 \cs_new_protected:Npn \__regex_posix_upper:
17262 { \__regex_item_caseful_range:nn { 'A' } { 'Z' } }
17263 \cs_new_eq:NN \__regex_posix_word: \__regex_prop_w:
17264 \cs_new_protected:Npn \__regex_posix_xdigit:
17265 {
17266     \__regex_posix_digit:
\__regex_item_caseful_range:nn { 'A' } { 'F' }

```

```

17268     \__regex_item_caseful_range:nn { 'a } { 'f }
17269 }
```

(End definition for `__regex_posix_alnum:` and others.)

36.2.4 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 mostly done within an `x`-expanding assignment, except for the `\x` escape sequence, which is not amenable to that in general. For this, we use the general framework of `__tl_build:Nw`.

`__regex_escape_use:nnnn` The result is built in `\l__regex_internal_a_t1`, which is then left in the input stream. Go through #4 once, applying #1, #2, or #3 as relevant to each character (after de-escaping it). Note that we cannot replace `\tl_set:Nx` and `__tl_build_one:o` by a single call to `__tl_build_one:x`, because the `x`-expanding assignment may be interrupted by `\x`.

```

17270 \cs_new_protected:Npn \__regex_escape_use:nnnn #1#2#3#4
17271 {
17272 (trace) \trace_push:nnn { regex } { 1 } { __regex_escape_use:nnnn }
17273 \__tl_build:Nw \l__regex_internal_a_t1
17274 \cs_set:Npn \__regex_escape_unescaped:N ##1 { #1 }
17275 \cs_set:Npn \__regex_escape_escaped:N ##1 { #2 }
17276 \cs_set:Npn \__regex_escape_raw:N ##1 { #3 }
17277 \__regex_standard_escapechar:
17278 \tl_gset:Nx \g__regex_internal_tl { \__str_to_other_fast:n {#4} }
17279 \tl_set:Nx \l__regex_internal_b_t1
17280 {
17281 \exp_after:wN \__regex_escape_loop:N \g__regex_internal_tl
17282 { break } \__prg_break_point:
17283 }
17284 \__tl_build_one:o \l__regex_internal_b_t1
17285 \__tl_build_end:
17286 (trace) \trace_pop:nnn { regex } { 1 } { __regex_escape_use:nnnn }
17287 \l__regex_internal_a_t1
17288 }
```

(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”.

```

17289 \cs_new:Npn \_\_regex_escape_loop:N #1
17290 {
17291     \cs_if_exist_use:cF { \_\_regex_escape_\token_to_str:N #1:w }
17292         { \_\_regex_escape_unescaped:N #1 }
17293     \_\_regex_escape_loop:N
17294 }
17295 \cs_new:cpn { \_\_regex_escape_ \c_backslash_str :w }
17296     \_\_regex_escape_loop:N #1
17297 {
17298     \cs_if_exist_use:cF { \_\_regex_escape_/\token_to_str:N #1:w }
17299         { \_\_regex_escape_escaped:N #1 }
17300     \_\_regex_escape_loop:N
17301 }
```

(End definition for __regex_escape_loop:N and __regex_escape_\:w.)

Those functions are never called before being given a new meaning, so their definitions here don’t matter.

```

17302 \cs_new_eq:NN \_\_regex_escape_unescaped:N ?
17303 \cs_new_eq:NN \_\_regex_escape_escaped:N ?
17304 \cs_new_eq:NN \_\_regex_escape_raw:N ?
```

(End definition for __regex_escape_unescaped:N, __regex_escape_escaped:N, and __regex_escape_raw:N.)

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.

```

17305 \cs_new_eq:NN \_\_regex_escape_break:w \_\_prg_break:
17306 \cs_new:cpn { \_\_regex_escape_/_break:w }
17307 {
17308     \if_false: { \fi: }
17309         \_\_msg_kernel_error:nn { regex } { trailing-backslash }
17310         \exp_after:wN \use_none:n \exp_after:wn { \if_false: } \fi:
17311     }
17312 \cs_new:cpn { \_\_regex_escape_\~:w } { }
17313 \cs_new:cpn { \_\_regex_escape_/_a:w }
17314     { \exp_not:N \_\_regex_escape_raw:N \iow_char:N \^\^G }
17315 \cs_new:cpn { \_\_regex_escape_/_t:w }
17316     { \exp_not:N \_\_regex_escape_raw:N \iow_char:N \^\^I }
17317 \cs_new:cpn { \_\_regex_escape_/_n:w }
17318     { \exp_not:N \_\_regex_escape_raw:N \iow_char:N \^\^J }
17319 \cs_new:cpn { \_\_regex_escape_/_f:w }
17320     { \exp_not:N \_\_regex_escape_raw:N \iow_char:N \^\^L }
17321 \cs_new:cpn { \_\_regex_escape_/_r:w }
17322     { \exp_not:N \_\_regex_escape_raw:N \iow_char:N \^\^M }
17323 \cs_new:cpn { \_\_regex_escape_/_e:w }
17324     { \exp_not:N \_\_regex_escape_raw:N \iow_char:N \^\^[] }
```

(End definition for __regex_escape_break: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.

```
17325 \cs_new:cpn { __regex_escape_x:w } __regex_escape_loop:N
17326 {
17327     \exp_after:wN __regex_escape_x_end:w
17328     __int_value:w "0 __regex_escape_x_test:N
17329 }
17330 \cs_new:Npn __regex_escape_x_end:w #1 ;
17331 {
17332     \int_compare:nNnTF {#1} > \c_max_char_int
17333 {
17334     \if_false: { \fi: }
17335     \tl_build_one:o \l__regex_internal_b_tl
17336     __msg_kernel_error:nnx { regex } { x-overflow } {#1}
17337     \tl_set:Nx \l__regex_internal_b_tl
17338         { \if_false: } \fi:
17339 }
17340 {
17341     \exp_last_unbraced:Nf __regex_escape_raw:N
17342         { \char_generate:nn {#1} { 12 } }
17343 }
17344 }
```

(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`.

```
17345 \cs_new:Npn __regex_escape_x_test:N #1
17346 {
17347     \str_if_eq_x:nnTF {#1} { break } { ; }
17348 {
17349     \if_charcode:w \c_space_token #1
17350         \exp_after:wN __regex_escape_x_test:N
17351     \else:
17352         \exp_after:wN __regex_escape_x_testii:N
17353         \exp_after:wN #1
17354     \fi:
17355 }
17356 }
17357 \cs_new:Npn __regex_escape_x_testii:N #1
17358 {
17359     \if_charcode:w \c_left_brace_str #1
17360         \exp_after:wN __regex_escape_x_loop:N
17361     \else:
17362         __regex_hexadecimal_use:NTF #1
17363             { \exp_after:wN __regex_escape_x:N }
17364             { ; \exp_after:wN __regex_escape_loop:N \exp_after:wN #1 }
17365     \fi:
17366 }
```

(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.

```
17367 \cs_new:Npn \_regex_escape_x:N #1
17368 {
17369     \str_if_eq_x:nnTF {#1} { break } { ; }
17370     {
17371         \_regex_hexadecimal_use:NTF #1
17372         { ; \_regex_escape_loop:N }
17373         { ; \_regex_escape_loop:N #1 }
17374     }
17375 }
```

(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, otherwise raise an error outside the assignment.

```
17376 \cs_new:Npn \_regex_escape_x_loop:N #1
17377 {
17378     \str_if_eq_x:nnTF {#1} { break }
17379     { ; \_regex_escape_x_loop_error:n { } {#1} }
17380     {
17381         \_regex_hexadecimal_use:NTF #1
17382         { \_regex_escape_x_loop:N }
17383         {
17384             \token_if_eq_charcode:NNTF \c_space_token #1
17385             { \_regex_escape_x_loop:N }
17386             {
17387                 ;
17388                 \exp_after:wN
17389                 \token_if_eq_charcode:NNTF \c_right_brace_str #1
17390                 { \_regex_escape_loop:N }
17391                 { \_regex_escape_x_loop_error:n {#1} }
17392             }
17393         }
17394     }
17395 }
17396 \cs_new:Npn \_regex_escape_x_loop_error:n #1
17397 {
17398     \if_false: { \fi: }
17399     \tl_build_one:o \l_regex_internal_b_tl
17400     \msg_kernel_error:nnx { regex } { x-missing-rbrace } {#1}
17401     \tl_set:Nx \l_regex_internal_b_tl
17402     { \if_false: } \fi: \_regex_escape_loop:N #1
17403 }
```

(End definition for `_regex_escape_x_loop:N` and `_regex_escape_x_loop_error:n`.)

`_regex_hexadecimal_use:NTF` TeX detects uppercase hexadecimal digits for us but not the lowercase letters, which we need to detect and replace by their uppercase counterpart.

```
17404 \prg_new_conditional:Npnn \_regex_hexadecimal_use:N #1 { TF }
17405 {
17406     \if_int_compare:w 1 < "1 \token_to_str:N #1 \exp_stop_f:
17407         #1 \prg_return_true:
```

```

17408     \else:
17409         \if_case:w \__int_eval:w
17410             \exp_after:wN ` \token_to_str:N #1 - `a
17411             \__int_eval_end:
17412                 A
17413                 \or: B
17414                 \or: C
17415                 \or: D
17416                 \or: E
17417                 \or: F
17418             \else:
17419                 \prg_return_false:
17420                 \exp_after:wN \use_none:n
17421             \fi:
17422             \prg_return_true:
17423         \fi:
17424     }

```

(End definition for `__regex_hexadecimal_use:NTF`.)

`__regex_char_if_alphanumeric:NTF`
`__regex_char_if_special:NTF`

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,

- alphanumerics 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).

```

17425     \prg_new_conditional:Npnn \__regex_char_if_special:N #1 { TF }
17426     {
17427         \if_int_compare:w '#1 > 'Z \exp_stop_f:
17428             \if_int_compare:w '#1 > 'z \exp_stop_f:
17429                 \if_int_compare:w '#1 < \c__regex_ascii_max_int
17430                     \prg_return_true: \else: \prg_return_false: \fi:
17431                 \else:
17432                     \if_int_compare:w '#1 < 'a \exp_stop_f:
17433                         \prg_return_true: \else: \prg_return_false: \fi:
17434                     \fi:
17435                 \else:
17436                     \if_int_compare:w '#1 > '9 \exp_stop_f:
17437                         \if_int_compare:w '#1 < 'A \exp_stop_f:
17438                             \prg_return_true: \else: \prg_return_false: \fi:
17439                         \else:
17440                             \if_int_compare:w '#1 < '0 \exp_stop_f:
17441                                 \if_int_compare:w '#1 < '\ \exp_stop_f:
17442                                     \prg_return_false: \else: \prg_return_true: \fi:
17443                                     \else: \prg_return_false: \fi:

```

```

17444     \fi:
17445     \fi:
17446   }
17447 \prg_new_conditional:Npnn \__regex_char_if_alphanumeric:N #1 { TF }
17448   {
17449     \if_int_compare:w '#1 > 'Z \exp_stop_f:
17450       \if_int_compare:w '#1 > 'z \exp_stop_f:
17451         \prg_return_false:
17452       \else:
17453         \if_int_compare:w '#1 < 'a \exp_stop_f:
17454           \prg_return_false: \else: \prg_return_true: \fi:
17455         \fi:
17456       \else:
17457         \if_int_compare:w '#1 > '9 \exp_stop_f:
17458           \if_int_compare:w '#1 < 'A \exp_stop_f:
17459             \prg_return_false: \else: \prg_return_true: \fi:
17460           \else:
17461             \if_int_compare:w '#1 < '0 \exp_stop_f:
17462               \prg_return_false: \else: \prg_return_true: \fi:
17463             \fi:
17464           \fi:
17465   }

```

(End definition for `__regex_char_if_alphanumeric:NTF` and `__regex_char_if_special:NTF`.)

36.3 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 $\{\langle\text{__regex_anchor:N}\langle\text{integer}\rangle\}$

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` {*catcode bitmap*} {*tests*}
- `_regex_item_reverse:n` {*tests*}
- `_regex_item_exact:nn` {*catcode*} {*char code*}
- `_regex_item_exact_cs:n` {*csnames*}, more precisely given as *csname* `\scan_stop:` *csname* `\scan_stop:` *csname* and so on in a brace group.
- `_regex_item_cs:n` {*compiled regex*}

36.3.1 Variables used when compiling

`\l_regex_group_level_int` We make sure to open the same number of groups as we close.

17466 `\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 $-63, -23, -6, -2, 0, 2, 3, 6, 23, 63$. See section 36.3.3. We only define some of these as constants.

17467 `\int_new:N \l_regex_mode_int`
 17468 `\int_const:Nn \c_regex_cs_in_class_mode_int { -6 }`
 17469 `\int_const:Nn \c_regex_cs_mode_int { -2 }`
 17470 `\int_const:Nn \c_regex_outer_mode_int { 0 }`
 17471 `\int_const:Nn \c_regex_catcode_mode_int { 2 }`
 17472 `\int_const:Nn \c_regex_class_mode_int { 3 }`
 17473 `\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]`.

17474 `\int_new:N \l_regex_catcodes_int`
 17475 `\int_new:N \l_regex_default_catcodes_int`
 17476 `\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.

17477 `\int_const:Nn \c_regex_catcode_C_int { "1" }`
 17478 `\int_const:Nn \c_regex_catcode_B_int { "4" }`
 17479 `\int_const:Nn \c_regex_catcode_E_int { "10" }`
 17480 `\int_const:Nn \c_regex_catcode_M_int { "40" }`
 17481 `\int_const:Nn \c_regex_catcode_T_int { "100" }`
 17482 `\int_const:Nn \c_regex_catcode_P_int { "1000" }`
 17483 `\int_const:Nn \c_regex_catcode_U_int { "4000" }`
 17484 `\int_const:Nn \c_regex_catcode_D_int { "10000" }`
 17485 `\int_const:Nn \c_regex_catcode_S_int { "100000" }`

```

17486 \int_const:Nn \c__regex_catcode_L_int { "400000 }
17487 \int_const:Nn \c__regex_catcode_O_int { "1000000 }
17488 \int_const:Nn \c__regex_catcode_A_int { "4000000 }
17489 \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.

```

17490 \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.

```

17491 \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.

```

17492 \int_new:N \l__regex_show_lines_int

(End definition for \l__regex_show_lines_int.)

```

36.3.2 Generic helpers used when compiling

__regex_get_digits:NTFw If followed by some raw digits, collect them one by one in the integer variable #1, and take the **true** branch. Otherwise, take the **false** branch.

```

17493 \cs_new_protected:Npn \_\_regex_get_digits:NTFw #1#2#3#4#5
17494 {
17495     \_\_regex_if_raw_digit:NNTF #4 #5
17496     { #1 = #5 \_\_regex_get_digits_loop:nw {#2} }
17497     { #3 #4 #5 }
17498 }
17499 \cs_new:Npn \_\_regex_get_digits_loop:nw #1#2#3
17500 {
17501     \_\_regex_if_raw_digit:NNTF #2 #3
17502     { #3 \_\_regex_get_digits_loop:nw {#1} }
17503     { \scan_stop: #1 #2 #3 }
17504 }

```

(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.

```

17505 \prg_new_conditional:Npnn \_\_regex_if_raw_digit:NN #1#2 { TF }
17506 {
17507     \if_meaning:w \_\_regex_compile_raw:N #1
17508     \if_int_compare:w 1 < 1 #2 \exp_stop_f:
17509         \prg_return_true:
17510     \else:
17511         \prg_return_false:
17512     \fi:
17513 \else:

```

```

17514     \prg_return_false:
17515     \fi:
17516 }
```

(End definition for `_regex_if_raw_digit:NNTF.`)

36.3.3 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.

```

17517 \cs_new:Npn \_\_regex_if_in_class:TF
17518 {
17519     \if_int_odd:w \l__regex_mode_int
17520         \exp_after:wN \use_i:nn
17521     \else:
17522         \exp_after:wN \use_ii:nn
17523     \fi:
17524 }
```

(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).

```

17525 \cs_new:Npn \_\_regex_if_in_cs:TF
17526 {
17527     \if_int_odd:w \l__regex_mode_int
17528         \exp_after:wN \use_ii:nn
17529     \else:
17530         \if_int_compare:w \l__regex_mode_int < \c__regex_outer_mode_int
17531             \exp_after:wN \exp_after:wN \exp_after:wN \use_i:nn
17532         \else:
17533             \exp_after:wN \exp_after:wN \exp_after:wN \use_ii:nn
17534         \fi:
17535     \fi:
17536 }
```

(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.

```

17537 \cs_new:Npn \_\_regex_if_in_class_or_catcode:TF
17538 {
17539     \if_int_odd:w \l__regex_mode_int
17540         \exp_after:wN \use_i:nn
17541     \else:
17542         \if_int_compare:w \l__regex_mode_int > \c__regex_outer_mode_int
17543             \exp_after:wN \exp_after:wN \exp_after:wN \use_i:nn
17544         \else:
17545             \exp_after:wN \exp_after:wN \exp_after:wN \use_ii:nn
17546         \fi:
17547     \fi:
17548 }
```

(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.

```

17549 \cs_new:Npn \_\_regex_if_within_catcode:TF
17550 {
17551     \if_int_compare:w \l__regex_mode_int > \c__regex_outer_mode_int
17552         \exp_after:wN \use_i:nn
```

```

17553     \else:
17554         \exp_after:wN \use_i:nn
17555     \fi:
17556 }

```

(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.

```

17557 \cs_new_protected:Npn \_regex_chk_c_allowed:T
17558 {
17559     \if_int_compare:w \l__regex_mode_int = \c__regex_outer_mode_int
17560         \exp_after:wN \use:n
17561     \else:
17562         \if_int_compare:w \l__regex_mode_int = \c__regex_class_mode_int
17563             \exp_after:wN \exp_after:wN \exp_after:wN \use:n
17564         \else:
17565             \_msg_kernel_error:nn { regex } { c-bad-mode }
17566             \exp_after:wN \exp_after:wN \exp_after:wN \use:none:n
17567         \fi:
17568     \fi:
17569 }

```

(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.

```

17570 \cs_new_protected:Npn \_regex_mode_quit_c:
17571 {
17572     \if_int_compare:w \l__regex_mode_int = \c__regex_catcode_mode_int
17573         \int_set_eq:NN \l__regex_mode_int \c__regex_outer_mode_int
17574     \else:
17575         \if_int_compare:w \l__regex_mode_int = \c__regex_catcode_in_class_mode_int
17576             \int_set_eq:NN \l__regex_mode_int \c__regex_class_mode_int
17577         \fi:
17578     \fi:
17579 }

```

(End definition for `_regex_mode_quit_c:.`)

36.3.4 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`.

```

17580 \cs_new_protected:Npn \_regex_compile:w
17581 {
17582     \_tl_build_x:Nw \l__regex_internal_regex
17583         \int_zero:N \l__regex_group_level_int
17584         \int_set_eq:NN \l__regex_default_catcodes_int \c__regex_all_catcodes_int
17585         \int_set_eq:NN \l__regex_catcodes_int \l__regex_default_catcodes_int
17586         \cs_set:Npn \_regex_item_equal:n { \_regex_item_caseful_equal:n }
17587         \cs_set:Npn \_regex_item_range:nn { \_regex_item_caseful_range:nn }

```

```

17588      \_\_tl\_build\_one:n { \_\_regex\_branch:n { \if\_false: } \fi: }
17589    }
17590 \cs\_new\_protected:Npn \_\_regex\_compile\_end:
17591  {
17592    \_\_regex\_if\_in\_class:TF
17593    {
17594      \_\_msg\_kernel\_error:nn { regex } { missing-rbrack }
17595      \use:c { \_\_regex\_compile\_l: }
17596      \prg\_do\_nothing: \prg\_do\_nothing:
17597    }
17598    {
17599      \if\_int\_compare:w \l\_regex\_group\_level\_int > 0 \exp_stop_f:
17600        \_\_msg\_kernel\_error:nnx { regex } { missing-rparen }
17601        { \int\_use:N \l\_regex\_group\_level\_int }
17602        \prg\_replicate:nn
17603        { \l\_regex\_group\_level\_int }
17604    }
17605    \_\_tl\_build\_one:n
17606    {
17607      \if\_false: { \fi: }
17608      \if\_false: { \fi: } { 1 } { 0 } \c\_true\_bool
17609    }
17610    \_\_tl\_build\_end:
17611    \_\_tl\_build\_one:o \l\_regex\_internal\_regex
17612  }
17613  \fi:
17614  \_\_tl\_build\_one:n { \if\_false: { \fi: } }
17615  \_\_tl\_build\_end:
17616}

```

(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.

```

17617 \cs\_new\_protected:Npn \_\_regex\_compile:n #1
17618  {
17619    \_\_regex\_compile:w
17620    \_\_regex\_standard\_escapechar:
17621    \int\_set\_eq:NN \l\_regex\_mode\_int \c\_regex\_outer\_mode\_int
17622    \_\_regex\_escape\_use:nnnn
17623    {
17624      \_\_regex\_char\_if\_special:NTF ##1
17625        \_\_regex\_compile\_special:N \_\_regex\_compile\_raw:N ##1
17626    }
17627    {
17628      \_\_regex\_char\_if\_alphanumeric:NTF ##1
17629        \_\_regex\_compile\_escaped:N \_\_regex\_compile\_raw:N ##1
17630    }
17631  { \_\_regex\_compile\_raw:N ##1 }

```

```

17632     { #1 }
17633     \prg_do_nothing: \prg_do_nothing:
17634     \prg_do_nothing: \prg_do_nothing:
17635     \int_compare:nNnT \l__regex_mode_int = \c__regex_catcode_mode_int
17636         { \__msg_kernel_error:nn { regex } { c-trailing } }
17637     \int_compare:nNnT \l__regex_mode_int < \c__regex_outer_mode_int
17638         {
17639             \__msg_kernel_error:nn { regex } { c-missing-rbrace }
17640             \__regex_compile_end_cs:
17641                 \prg_do_nothing: \prg_do_nothing:
17642                     \prg_do_nothing: \prg_do_nothing:
17643                 }
17644             \__regex_compile_end:
17645         }

```

(End definition for `__regex_compile:n`.)

- `__regex_compile_escaped:N` If the special character or escaped alphanumeric has a particular meaning in regexes,
`__regex_compile_special:N` 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.

```

17646 \cs_new_protected:Npn \__regex_compile_special:N #1
17647     {
17648         \cs_if_exist_use:cF { __regex_compile:#1: }
17649             { \__regex_compile_raw:N #1 }
17650     }
17651 \cs_new_protected:Npn \__regex_compile_escaped:N #1
17652     {
17653         \cs_if_exist_use:cF { __regex_compile_/#1: }
17654             { \__regex_compile_raw:N #1 }
17655     }

```

(End definition for `__regex_compile_escaped:N` and `__regex_compile_special:N`.)

- `__regex_compile_one:x` 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.

```

17656 \cs_new_protected:Npn \__regex_compile_one:x #1
17657     {
17658         \__regex_mode_quit_c:
17659         \__regex_if_in_class:TF { }
17660             {
17661                 \__tl_build_one:n
17662                     { \__regex_class:NnnnN \c_true_bool { \if_false: } \fi: }
17663             }
17664         \__tl_build_one:x
17665             {
17666                 \if_int_compare:w \l__regex_catcodes_int < \c__regex_all_catcodes_int
17667                     \__regex_item_catcode:nT { \int_use:N \l__regex_catcodes_int }
17668                         { \exp_not:N \exp_not:n {#1} }
17669                 \else:
17670                     \exp_not:N \exp_not:n {#1}
17671                 \fi:

```

```

17672      }
17673      \int_set_eq:NN \l__regex_catcodes_int \l__regex_default_catcodes_int
17674      \_regex_if_in_class:TF { } { \_regex_compile_quantifier:w }
17675  }

(End definition for \_regex_compile_one:x.)
```

_regex_compile_abort_tokens:n This function places the collected tokens back in the input stream, each as a raw character.
_regex_compile_abort_tokens:x Spaces are not preserved.

```

17676 \cs_new_protected:Npn \_regex_compile_abort_tokens:n #1
17677  {
17678    \use:x
17679    {
17680      \exp_args:No \tl_map_function:nN { \tl_to_str:n {#1} }
17681      \_regex_compile_raw:N
17682    }
17683  }
17684 \cs_generate_variant:Nn \_regex_compile_abort_tokens:n { x }

(End definition for \_regex_compile_abort_tokens:n.)
```

36.3.5 Quantifiers

_regex_compile_quantifier:w This looks ahead and finds any quantifier (special character equal to either of ?+*{}).

```

17685 \cs_new_protected:Npn \_regex_compile_quantifier:w #1#2
17686  {
17687    \token_if_eq_meaning:NNTF #1 \_regex_compile_special:N
17688    {
17689      \cs_if_exist_use:cF { \_regex_compile_quantifier:#2:w }
17690      { \_regex_compile_quantifier_none: #1 #2 }
17691    }
17692    { \_regex_compile_quantifier_none: #1 #2 }
17693  }

(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).

```

17694 \cs_new_protected:Npn \_regex_compile_quantifier_none:
17695  { \_tl_build_one:n { \if_false: { \fi: } { 1 } { 0 } \c_false_bool } }
17696 \cs_new_protected:Npn \_regex_compile_quantifier_abort:xNN #1#2#3
17697  {
17698    \_regex_compile_quantifier_none:
17699    \_msg_kernel_warning:nnxx { regex } { invalid-quantifier } {#1} {#3}
17700    \_regex_compile_abort_tokens:x {#1}
17701    #2 #3
17702  }

(End definition for \_regex_compile_quantifier_none: and \_regex_compile_quantifier_abort:xNN.)
```

_regex_compile_quantifier_laziness: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.

```

17703 \cs_new_protected:Npn \__regex_compile_quantifier_lazyness:nnNN #1#2#3#4
17704 {
17705   \str_if_eq:nnTF { #3 #4 } { \__regex_compile_special:N ? }
17706   { \__tl_build_one:n { \if_false: { \fi: } { #1 } { #2 } \c_true_bool } }
17707   {
17708     \__tl_build_one:n { \if_false: { \fi: } { #1 } { #2 } \c_false_bool }
17709     #3 #4
17710   }
17711 }
```

(End definition for `__regex_compile_quantifier_lazyness:nnNN`.)

For each “basic” quantifier, ?, *, +, feed the correct arguments to `__regex_compile_quantifier_lazyness:nnNN`, -1 means that there is no upper bound on the number of repetitions.

```

17712 \cs_new_protected:cpn { __regex_compile_quantifier_?:w }
17713   { __regex_compile_quantifier_lazyness:nnNN { 0 } { 1 } }
17714 \cs_new_protected:cpn { __regex_compile_quantifier_*:w }
17715   { __regex_compile_quantifier_lazyness:nnNN { 0 } { -1 } }
17716 \cs_new_protected:cpn { __regex_compile_quantifier_+:w }
17717   { __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: `{<int>}`, `{<int>,<int>}`, or `{<int>,<int>}`. 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 $\{a\}{b - a}$.

```

17718 \cs_new_protected:cpn { __regex_compile_quantifier_ \c_left_brace_str :w }
17719 {
17720   \__regex_get_digits:NTFw \l__regex_internal_a_int
17721   { __regex_compile_quantifier_braced_auxi:w }
17722   { __regex_compile_quantifier_abort:xNN { \c_left_brace_str } }
17723 }
17724 \cs_new_protected:Npn \__regex_compile_quantifier_braced_auxi:w #1#2
17725 {
17726   \str_case_x:nnF { #1 #2 }
17727   {
17728     { __regex_compile_special:N \c_right_brace_str }
17729     {
17730       \exp_args:No __regex_compile_quantifier_lazyness:nnNN
17731         { \int_use:N \l__regex_internal_a_int } { 0 }
17732     }
17733     { __regex_compile_special:N , }
17734     {
17735       \__regex_get_digits:NTFw \l__regex_internal_b_int
17736         { __regex_compile_quantifier_braced_auxii:w }
17737         { __regex_compile_quantifier_braced_auxi:w }
17738     }
17739   }
17740 }
```

```

17741     \_\_regex_compile_quantifier_abort:xNN
17742         { \c_left_brace_str \int_use:N \l\_regex_internal_a_int }
17743     #1 #2
17744 }
17745 }
17746 \cs_new_protected:Npn \_\_regex_compile_quantifier_braced_auxii:w #1#2
17747 {
17748     \str_if_eq_x:nnTF
17749         { #1 #2 } { \_\_regex_compile_special:N \c_right_brace_str }
17750     {
17751         \exp_args:No \_\_regex_compile_quantifier_lazyness:nnNN
17752             { \int_use:N \l\_regex_internal_a_int } { -1 }
17753     }
17754     {
17755         \_\_regex_compile_quantifier_abort:xNN
17756             { \c_left_brace_str \int_use:N \l\_regex_internal_a_int , }
17757         #1 #2
17758     }
17759 }
17760 \cs_new_protected:Npn \_\_regex_compile_quantifier_braced_auxiii:w #1#2
17761 {
17762     \str_if_eq_x:nnTF
17763         { #1 #2 } { \_\_regex_compile_special:N \c_right_brace_str }
17764     {
17765         \if_int_compare:w \l\_regex_internal_a_int > \l\_regex_internal_b_int
17766             \msg_kernel_error:nnxx { regex } { backwards-quantifier }
17767                 { \int_use:N \l\_regex_internal_a_int }
17768                 { \int_use:N \l\_regex_internal_b_int }
17769                 \int_zero:N \l\_regex_internal_b_int
17770         \else:
17771             \int_sub:Nn \l\_regex_internal_b_int \l\_regex_internal_a_int
17772         \fi:
17773         \exp_args:Nno \_\_regex_compile_quantifier_lazyness:nnNN
17774             { \int_use:N \l\_regex_internal_a_int }
17775             { \int_use:N \l\_regex_internal_b_int }
17776     }
17777     {
17778         \_\_regex_compile_quantifier_abort:xNN
17779             {
17780                 \c_left_brace_str
17781                 \int_use:N \l\_regex_internal_a_int ,
17782                 \int_use:N \l\_regex_internal_b_int
17783             }
17784             #1 #2
17785     }
17786 }
17787

```

(End definition for `__regex_compile_quantifier_{:w}` and others.)

36.3.6 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.

```

17787 \cs_new_protected:Npn \__regex_compile_raw_error:N #1
17788 {
17789     \__msg_kernel_error:nnx { regex } { bad-escape } {#1}
17790     \__regex_compile_raw:N #1
17791 }

```

(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.

```

17792 \cs_new_protected:Npn \__regex_compile_raw:N #1#2#3
17793 {
17794     \__regex_if_in_class:TF
17795     {
17796         \str_if_eq:nnTF {#2#3} { \__regex_compile_special:N - }
17797         { \__regex_compile_range:Nw #1 }
17798         {
17799             \__regex_compile_one:x
17800             { \__regex_item_equal:n { \__int_value:w '#1 ~ } }
17801             #2 #3
17802         }
17803     }
17804     {
17805         \__regex_compile_one:x
17806         { \__regex_item_equal:n { \__int_value:w '#1 ~ } }
17807         #2 #3
17808     }
17809 }

```

(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.

```

17810 \prg_new_protected_conditional:Npnn \__regex_if_end_range:NN #1#2 { TF }
17811 {
17812     \if_meaning:w \__regex_compile_raw:N #1
17813     \prg_return_true:
17814     \else:
17815         \if_meaning:w \__regex_compile_special:N #1
17816         \if_charcode:w ] #2
17817         \prg_return_false:
17818         \else:
17819             \prg_return_true:
17820             \fi:
17821         \else:
17822             \prg_return_false:
17823             \fi:
17824             \fi:
17825     }
17826 \cs_new_protected:Npn \__regex_compile_range:Nw #1#2#3
17827 {
17828     \__regex_if_end_range:NNTF #2 #3
17829     {

```

```

17830     \if_int_compare:w '#1 > '#3 \exp_stop_f:
17831         \_msg_kernel_error:nnxx { regex } { range-backwards } {#1} {#3}
17832     \else:
17833         \_tl_build_one:x
17834         {
17835             \if_int_compare:w '#1 = '#3 \exp_stop_f:
17836                 \_regex_item_equal:n
17837             \else:
17838                 \_regex_item_range:nn { \_int_value:w '#1 ~ }
17839             \fi:
17840                 { \_int_value:w '#3 ~ }
17841             }
17842         \fi:
17843     }
17844     {
17845         \_msg_kernel_warning:nnxx { regex } { range-missing-end }
17846             {#1} { \c_backslash_str #3 }
17847         \_tl_build_one:x
17848             {
17849                 \_regex_item_equal:n { \_int_value:w '#1 ~ }
17850                 \_regex_item_equal:n { \_int_value:w '- ~ }
17851             }
17852             #2#3
17853         }
17854     }

```

(End definition for `_regex_compile_range:Nw` and `_regex_if_end_range:NNTF`.)

36.3.7 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).

```

17855 \cs_new_protected:cpx { \_regex_compile_..: }
17856     {
17857         \exp_not:N \_regex_if_in_class:TF
17858             { \_regex_compile_raw:N . }
17859             { \_regex_compile_one:x \exp_not:c { \_regex_prop_..: } }
17860     }
17861 \cs_new_protected:cpn { \_regex_prop_..: }
17862     {
17863         \if_int_compare:w \l_\_regex_current_char_int > - 2 \exp_stop_f:
17864             \exp_after:wN \_regex_break_true:w
17865         \fi:
17866     }

```

(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.

```

17867 \cs_set_protected:Npn \_regex_tmp:w #1#2
17868     {
17869         \cs_new_protected:cpx { \_regex_compile_/#1: }
17870             { \_regex_compile_one:x \exp_not:c { \_regex_prop_#1: } }

```

```

17871     \cs_new_protected:cpn { __regex_compile_/#2: }
17872     {
17873         __regex_compile_one:x
17874         { __regex_item_reverse:n \exp_not:c { __regex_prop_#1: } }
17875     }
17876 }
17877 __regex_tmp:w d D
17878 __regex_tmp:w h H
17879 __regex_tmp:w s S
17880 __regex_tmp:w v V
17881 __regex_tmp:w w W
17882 \cs_new_protected:cpn { __regex_compile_/#N: }
17883 { __regex_compile_one:x __regex_prop_N: }

```

(End definition for `__regex_compile_/#d:` and others.)

36.3.8 Anchoring and simple assertions

`__regex_compile_anchor:NF`
`__regex_compile_~:`
`__regex_compile/_A:`
`__regex_compile/_G:`
`__regex_compile/$:`
`__regex_compile/_Z:`
`__regex_compile/_z:`

In modes where assertions are allowed, anchor to the start of the query, the start of the match, or the end of the query, depending on the integer #1. In other modes, #2 treats the character as raw, with an error for escaped letters (\$ is valid in a class, but \A is definitely a mistake on the user's part).

```

17884 \cs_new_protected:Npn __regex_compile_anchor:NF #1#2
17885 {
17886     __regex_if_in_class_or_catcode:TF {#2}
17887     {
17888         __tl_build_one:n
17889         { __regex_assertion:Nn \c_true_bool { __regex_anchor:N #1 } }
17890     }
17891 }
17892 \cs_set_protected:Npn __regex_tmp:w #1#2
17893 {
17894     \cs_new_protected:cpn { __regex_compile_/#1: }
17895     { __regex_compile_anchor:NF #2 { __regex_compile_raw_error:N #1 } }
17896 }
17897 __regex_tmp:w A \l__regex_min_pos_int
17898 __regex_tmp:w G \l__regex_start_pos_int
17899 __regex_tmp:w Z \l__regex_max_pos_int
17900 __regex_tmp:w z \l__regex_max_pos_int
17901 \cs_set_protected:Npn __regex_tmp:w #1#2
17902 {
17903     \cs_new_protected:cpn { __regex_compile_/#1: }
17904     { __regex_compile_anchor:NF #2 { __regex_compile_raw:N #1 } }
17905 }
17906 \exp_args:Nx __regex_tmp:w { \iow_char:N \^ } \l__regex_min_pos_int
17907 \exp_args:Nx __regex_tmp:w { \iow_char:N \$ } \l__regex_max_pos_int

```

(End definition for `__regex_compile_anchor:NF` and others.)

`__regex_compile/_b:` Contrarily to `~` and `$`, which could be implemented without really knowing what precedes in the token list, this requires more information, namely, the knowledge of the last character code.

```

17908 \cs_new_protected:cpn { __regex_compile/_b: }
17909 {

```

```

17910     \_\_regex_if_in_class_or_catcode:TF
17911     { \_\_regex_compile_raw_error:N b }
17912     {
17913         \_\_tl_build_one:n
17914         { \_\_regex_assertion:Nn \c_true_bool { \_\_regex_b_test: } }
17915     }
17916 }
17917 \cs_new_protected:cpn { \_\_regex_compile_/B: }
17918 {
17919     \_\_regex_if_in_class_or_catcode:TF
17920     { \_\_regex_compile_raw_error:N B }
17921     {
17922         \_\_tl_build_one:n
17923         { \_\_regex_assertion:Nn \c_false_bool { \_\_regex_b_test: } }
17924     }
17925 }

```

(End definition for `__regex_compile_/_b:` and `__regex_compile_/_B::`)

36.3.9 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.

```

17926 \cs_new_protected:cpn { \_\_regex_compile_[:} :
17927 {
17928     \_\_regex_if_in_class:TF
17929     {
17930         \if_int_compare:w \l_\_regex_mode_int > \c_\_regex_catcode_in_class_mode_int
17931             \_\_tl_build_one:n { \if_false: { \fi: } }
17932             \fi:
17933             \tex_advance:D \l_\_regex_mode_int - 15 \exp_stop_f:
17934             \tex_divide:D \l_\_regex_mode_int 13 \exp_stop_f:
17935             \if_int_odd:w \l_\_regex_mode_int \else:
17936                 \exp_after:wN \_\_regex_compile_quantifier:w
17937                 \fi:
17938             }
17939             { \_\_regex_compile_raw:N } }
17940 }

```

(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.

```

17941 \cs_new_protected:cpn { \_\_regex_compile_[:} :
17942 {
17943     \_\_regex_if_in_class:TF
17944     { \_\_regex_compile_class_posix_test:w }
17945     {
17946         \_\_regex_if_within_catcode:TF
17947         {
17948             \exp_after:wN \_\_regex_compile_class_catcode:w

```

```

17949           \int_use:N \l__regex_catcodes_int ;
17950       }
17951   { \__regex_compile_class_normal:w }
17952 }
17953 }
```

(End definition for `__regex_compile_[:]`)

`__regex_compile_class_normal:w` In the “normal” case, we will 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.

```

17954 \cs_new_protected:Npn \__regex_compile_class_normal:w
17955 {
17956     \__regex_compile_class:TFNN
17957     { \__regex_class:NnnnN \c_true_bool }
17958     { \__regex_class:NnnnN \c_false_bool }
17959 }
```

(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.

```

17960 \cs_new_protected:Npn \__regex_compile_class_catcode:w #1,
17961 {
17962     \if_int_compare:w \l__regex_mode_int = \c_regex_catcode_mode_int
17963         \tl_build_one:n
17964             { \__regex_class:NnnnN \c_true_bool { \if_false: } \fi: }
17965     \fi:
17966     \int_set_eq:NN \l__regex_catcodes_int \l__regex_default_catcodes_int
17967     \__regex_compile_class:TFNN
17968     { \__regex_item_catcode:nT {#1} }
17969     { \__regex_item_catcode_reverse:nT {#1} }
17970 }
```

(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 #1). If the next character is a right bracket, then it should be changed to a raw one.

```

17971 \cs_new_protected:Npn \__regex_compile_class:TFNN #1#2#3#4
17972 {
17973     \l__regex_mode_int = \int_value:w \l__regex_mode_int 3 \exp_stop_f:
17974     \str_if_eq:nnTF {#3 #4} { \__regex_compile_special:N ^ }
17975     {
17976         \tl_build_one:n {#2 { \if_false: } \fi: }
17977         \__regex_compile_class:NN
17978     }
17979     {
17980         \tl_build_one:n {#1 { \if_false: } \fi: }
17981         \__regex_compile_class:NN #3 #4
17982     }
```

```

17983     }
17984 \cs_new_protected:Npn \__regex_compile_class>NN #1#2
17985 {
17986     \token_if_eq_charcode:NNTF #2 ]
17987     { \__regex_compile_raw:N #2 }
17988     { #1 #2 }
17989 }

```

(End definition for `__regex_compile_class:TFNN` and `__regex_compile_class>NN`.)

```

\__regex_compile_class_posix_test:w
\__regex_compile_class_posix:NNNNw
\__regex_compile_class_posix_loop:w
\__regex_compile_class_posix_end:w

```

Here we check for a syntax such as `[:alpha:]`. We also detect [= and [. which have a meaning in POSIX regular expressions, but are not implemented in l3regex. In case we see [:, grab raw characters until hopefully reaching :]. If that's missing, or the POSIX 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.

```

17990 \cs_new_protected:Npn \__regex_compile_class_posix_test:w #1#2
17991 {
17992     \token_if_eq_meaning:NNT \__regex_compile_special:N #1
17993     {
17994         \str_case:nn { #2 }
17995         {
17996             : { \__regex_compile_class_posix:NNNNw }
17997             = { \__msg_kernel_warning:nnx { regex } { posix-unsupported } { = } }
17998             . { \__msg_kernel_warning:nnx { regex } { posix-unsupported } { . } }
17999         }
18000     }
18001     \__regex_compile_raw:N [ #1 #2
18002 }
18003 \cs_new_protected:Npn \__regex_compile_class_posix:NNNNw #1#2#3#4#5#6
18004 {
18005     \str_if_eq:nnTF { #5 #6 } { \__regex_compile_special:N ^ }
18006     {
18007         \bool_set_false:N \l__regex_internal_bool
18008         \tl_set:Nx \l__regex_internal_a_tl { \if_false: } \fi:
18009         \__regex_compile_class_posix_loop:w
18010     }
18011     {
18012         \bool_set_true:N \l__regex_internal_bool
18013         \tl_set:Nx \l__regex_internal_a_tl { \if_false: } \fi:
18014         \__regex_compile_class_posix_loop:w #5 #6
18015     }
18016 }
18017 \cs_new:Npn \__regex_compile_class_posix_loop:w #1#2
18018 {
18019     \token_if_eq_meaning:NNTF \__regex_compile_raw:N #1
18020     { #2 \__regex_compile_class_posix_loop:w }
18021     { \if_false: { \fi: } \__regex_compile_class_posix_end:w #1 #2 }
18022 }
18023 \cs_new_protected:Npn \__regex_compile_class_posix_end:w #1#2#3#4
18024 {
18025     \str_if_eq:nnTF { #1 #2 #3 #4 }
18026     { \__regex_compile_special:N : \__regex_compile_special:N }
18027     {
18028         \cs_if_exist:cTF { __regex_posix_ \l__regex_internal_a_tl : }

```

```

18029    {
18030        \__regex_compile_one:x
18031        {
18032            \bool_if:NF \l__regex_internal_bool \__regex_item_reverse:n
18033            \exp_not:c { __regex_posix_ \l__regex_internal_a_tl : }
18034        }
18035    }
18036    {
18037        \__msg_kernel_warning:nnx { regex } { posix-unknown }
18038        { \l__regex_internal_a_tl }
18039        \__regex_compile_abort_tokens:x
18040        {
18041            [: \bool_if:NF \l__regex_internal_bool { ^ }
18042            \l__regex_internal_a_tl :]
18043        }
18044    }
18045}
18046{
18047    \__msg_kernel_error:nnxx { regex } { posix-missing-close }
18048    { [: \l__regex_internal_a_tl ] { #2 #4 }
18049    \__regex_compile_abort_tokens:x { [: \l__regex_internal_a_tl ]
18050        #1 #2 #3 #4
18051    }
18052}

```

(End definition for `__regex_compile_class_posix_test:w` and others.)

36.3.10 Groups and alternations

The contents of a regex group are turned into compiled code in `\l__regex_internal_regex`, which ends up with items of the form `__regex_branch:n {<concatenation>}`. This construction is done using `\tl`-build within a `\TEX` 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.

```

18053 \cs_new_protected:Npn \__regex_compile_group_begin:N #1
18054 {
18055     \tl_build_one:n { #1 { \if_false: } \fi: }
18056     \__regex_mode_quit_c:
18057     \tl_build:Nw \l__regex_internal_regex
18058     \int_set_eq:NN \l__regex_default_catcodes_int \l__regex_catcodes_int
18059     \int_incr:N \l__regex_group_level_int
18060     \tl_build_one:n { \__regex_branch:n { \if_false: } \fi: }
18061 }
18062 \cs_new_protected:Npn \__regex_compile_group_end:
18063 {
18064     \if_int_compare:w \l__regex_group_level_int > 0 \exp_stop_f:
18065         \tl_build_one:n { \if_false: { \fi: } }
18066     \tl_build_end:
18067     \int_set_eq:NN \l__regex_catcodes_int \l__regex_default_catcodes_int
18068     \tl_build_one:o \l__regex_internal_regex
18069     \exp_after:wn \__regex_compile_quantifier:w

```

```

18070     \else:
18071         \_\_msg_kernel_warning:nn { regex } { extra-rparen }
18072         \exp_after:wN \_\_regex_compile_raw:N \exp_after:wN )
18073     \fi:
18074 }

```

(End definition for `__regex_compile_group_begin:N` and `__regex_compile_group_end:..`)

`__regex_compile_(:`: In a class, parentheses are not special. Outside, check for a ?, denoting special groups, and run the code for the corresponding special group.

```

18075 \cs_new_protected:cpn { \_\_regex_compile_(: }
18076 {
18077     \_\_regex_if_in_class:TF { \_\_regex_compile_raw:N ( }
18078     { \_\_regex_compile_lparen:w }
18079 }
18080 \cs_new_protected:Npn \_\_regex_compile_lparen:w #1#2#3#4
18081 {
18082     \str_if_eq:nnTF { #1 #2 } { \_\_regex_compile_special:N ? }
18083     {
18084         \cs_if_exist_use:cF
18085         { \_\_regex_compile_special_group_\token_to_str:N #4 :w }
18086         {
18087             \_\_msg_kernel_warning:nnx { regex } { special-group-unknown }
18088             { (? #4 }
18089             \_\_regex_compile_group_begin:N \_\_regex_group:nnnN
18090             \_\_regex_compile_raw:N ? #3 #4
18091         }
18092     }
18093     {
18094         \_\_regex_compile_group_begin:N \_\_regex_group:nnnN
18095         #1 #2 #3 #4
18096     }
18097 }

```

(End definition for `__regex_compile_(:..)`)

`__regex_compile_|:`: In a class, the pipe is not special. Otherwise, end the current branch and open another one.

```

18098 \cs_new_protected:cpn { \_\_regex_compile_|: }
18099 {
18100     \_\_regex_if_in_class:TF { \_\_regex_compile_raw:N | }
18101     {
18102         \_\_tl_build_one:n
18103         { \if_false: { \fi: } \_\_regex_branch:n { \if_false: } \fi: }
18104     }
18105 }

```

(End definition for `__regex_compile_|:..`)

`__regex_compile_(:`: Within a class, parentheses are not special. Outside, close a group.

```

18106 \cs_new_protected:cpn { \_\_regex_compile_(): }
18107 {
18108     \_\_regex_if_in_class:TF { \_\_regex_compile_raw:N ) }
18109     { \_\_regex_group_end: }
18110 }

```

(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 will come when building.

```
18111 \cs_new_protected:cpn { __regex_compile_special_group_::w }
18112   { \__regex_compile_group_begin:N \__regex_group_no_capture:nnN }
18113 \cs_new_protected:cpn { __regex_compile_special_group_|:w }
18114   { \__regex_compile_group_begin:N \__regex_group_resetting:nnN }
```

(End definition for `_regex_compile_special_group_::w` and `_regex_compile_special_group_|:w`.)

`_regex_compile_special_group_i:w` The match can be made case-insensitive by setting the option with `(?i)`; the original behaviour is restored by `(?-i)`. This is the only supported option.

```
18115 \cs_new_protected:Npn \__regex_compile_special_group_i:w #1#2
18116   {
18117     \str_if_eq:nnTF { #1 #2 } { \__regex_compile_special:N }
18118     {
18119       \cs_set:Npn \__regex_item_equal:n { \__regex_item_caseless_equal:n }
18120       \cs_set:Npn \__regex_item_range:nn { \__regex_item_caseless_range:nn }
18121     }
18122     {
18123       \__msg_kernel_warning:nnx { regex } { unknown-option } { (?i #2 }
18124       \__regex_compile_raw:N (
18125       \__regex_compile_raw:N ?
18126       \__regex_compile_raw:N i
18127       #1 #2
18128     }
18129   }
18130 \cs_new_protected:cpn { __regex_compile_special_group_-:w } #1#2#3#4
18131   {
18132     \str_if_eq:nnTF { #1 #2 #3 #4 }
18133     { \__regex_compile_raw:N i \__regex_compile_special:N }
18134     {
18135       \cs_set:Npn \__regex_item_equal:n { \__regex_item_caseful_equal:n }
18136       \cs_set:Npn \__regex_item_range:nn { \__regex_item_caseful_range:nn }
18137     }
18138     {
18139       \__msg_kernel_warning:nnx { regex } { unknown-option } { (?-#2#4 }
18140       \__regex_compile_raw:N (
18141       \__regex_compile_raw:N ?
18142       \__regex_compile_raw:N -
18143       #1 #2 #3 #4
18144     }
18145   }
```

(End definition for `_regex_compile_special_group_i:w` and `_regex_compile_special_group_-:w`.)

36.3.11 Catcodes and csnames

`_regex_compile/_c:` The `\c` escape sequence can be followed by a capital letter representing a character 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.

```
18146 \cs_new_protected:cpn { __regex_compile/_c: }
18147   { \__regex_chk_c_allowed:T { \__regex_compile_c_test>NN } }
```

```

18148 \cs_new_protected:Npn \__regex_compile_c_test:NN #1#2
18149  {
18150    \token_if_eq_meaning:NNTF #1 \__regex_compile_raw:N
18151    {
18152      \int_if_exist:cTF { c__regex_catcode_#2_int }
18153      {
18154        \int_set_eq:Nc \l__regex_catcodes_int { c__regex_catcode_#2_int }
18155        \l__regex_mode_int
18156        = \if_case:w \l__regex_mode_int
18157          \c__regex_catcode_mode_int
18158        \else:
18159          \c__regex_catcode_in_class_mode_int
18160        \fi:
18161      }
18162    }
18163    \cs_if_exist_use:cF { __regex_compile_c_#2:w } }
18164    {
18165      \__msg_kernel_error:nnx { regex } { c-missing-category } {#2}
18166      #1 #2
18167    }
18168  }

```

(End definition for `__regex_compile_c:` and `__regex_compile_c_test:NN`.)

`__regex_compile_c_[:w`
`__regex_compile_c_lbrack_loop:NN`

```

\__regex_compile_c_lbrack_add:N
\__regex_compile_c_lbrack_end:
18169 \cs_new_protected:cpn { __regex_compile_c_[:w } #1#2
18170  {
18171    \l__regex_mode_int
18172    = \if_case:w \l__regex_mode_int
18173      \c__regex_catcode_mode_int
18174    \else:
18175      \c__regex_catcode_in_class_mode_int
18176    \fi:
18177    \int_zero:N \l__regex_catcodes_int
18178    \str_if_eq:nnTF { #1 #2 } { \__regex_compile_special:N ^ }
18179    {
18180      \bool_set_false:N \l__regex_catcodes_bool
18181      \__regex_compile_c_lbrack_loop:NN
18182    }
18183    {
18184      \bool_set_true:N \l__regex_catcodes_bool
18185      \__regex_compile_c_lbrack_loop:NN
18186      #1 #2
18187    }
18188  }
18189 \cs_new_protected:Npn \__regex_compile_c_lbrack_loop:NN #1#2
18190  {
18191    \token_if_eq_meaning:NNTF #1 \__regex_compile_raw:N
18192    {
18193      \int_if_exist:cTF { c__regex_catcode_#2_int }
18194      {
18195        \exp_args:Nc \__regex_compile_c_lbrack_add:N
18196          { c__regex_catcode_#2_int }

```

```

18197           \_\_regex_compile_c_lbrack_loop:NN
18198       }
18199   }
18200   {
18201     \token_if_eq_charcode:NNTF #2 ]
18202     { \_\_regex_compile_c_lbrack_end: }
18203   }
18204   {
18205     \_\_msg_kernel_error:nnx { regex } { c-missing-rbrack } {#2}
18206     \_\_regex_compile_c_lbrack_end:
18207     #1 #2
18208   }
18209 }
18210 \cs_new_protected:Npn \_\_regex_compile_c_lbrack_add:N #1
18211   {
18212     \if_int_odd:w \_\_int_eval:w \l_\_regex_catcodes_int / #1 \_\_int_eval_end:
18213     \else:
18214       \int_add:Nn \l_\_regex_catcodes_int {#1}
18215     \fi:
18216   }
18217 \cs_new_protected:Npn \_\_regex_compile_c_lbrack_end:
18218   {
18219     \if_meaning:w \c_false_bool \l_\_regex_catcodes_bool
18220     \int_set:Nn \l_\_regex_catcodes_int
18221     { \c_\_regex_all_catcodes_int - \l_\_regex_catcodes_int }
18222   }
18223 }

```

(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{...}`.

```

18224 \cs_new_protected:cpn { \_\_regex_compile_c_ \c_left_brace_str :w }
18225   {
18226     \_\_regex_compile:w
18227     \_\_regex_disable_submatches:
18228     \l_\_regex_mode_int
18229     = \if_case:w \l_\_regex_mode_int
18230       \c_\_regex_cs_mode_int
18231     \else:
18232       \c_\_regex_cs_in_class_mode_int
18233     \fi:
18234   }

```

(End definition for `__regex_compile_c_{:..}`)

`__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:`.

```

18235 \flag_new:n { __regex_cs }
18236 \cs_new_protected:cpn { __regex_compile_ \c_right_brace_str : }
18237 {
18238     __regex_if_in_cs:TF
18239     { __regex_compile_end_cs: }
18240     { \exp_after:wN __regex_compile_raw:N \c_right_brace_str }
18241 }
18242 \cs_new_protected:Npn __regex_compile_end_cs:
18243 {
18244     __regex_compile_end:
18245     \flag_clear:n { __regex_cs }
18246     \tl_set:Nx \l__regex_internal_a_tl
18247     {
18248         \exp_after:wN __regex_compile_cs_aux:Nn \l__regex_internal_regex
18249         \q_nil \q_nil \q_recursion_stop
18250     }
18251     \exp_args:Nx __regex_compile_one:x
18252     {
18253         \flag_if_raised:nTF { __regex_cs }
18254         { __regex_item_cs:n { \exp_not:o \l__regex_internal_regex } }
18255         { __regex_item_exact_cs:n { \tl_tail:N \l__regex_internal_a_tl } }
18256     }
18257 }
18258 \cs_new:Npn __regex_compile_cs_aux:Nn #1#2
18259 {
18260     \cs_if_eq:NNTF #1 __regex_branch:n
18261     {
18262         \scan_stop:
18263         __regex_compile_cs_aux>NNnnnN #2
18264         \q_nil \q_nil \q_nil \q_nil \q_nil \q_nil \q_recursion_stop
18265         __regex_compile_cs_aux:Nn
18266     }
18267     {
18268         \quark_if_nil:NF #1 { \flag_raise:n { __regex_cs } }
18269         \use_none_delimit_by_q_recursion_stop:w
18270     }
18271 }
18272 \cs_new:Npn __regex_compile_cs_aux>NNnnnN #1#2#3#4#5#6
18273 {
18274     \bool_lazy_all:nTF
18275     {
18276         { \cs_if_eq_p:NN #1 __regex_class:NnnnN }
18277         {#2}
18278         { \tl_if_head_eq_meaning_p:nN {#3} __regex_item_caseful_equal:n }
18279         { \int_compare_p:nNn { \tl_count:n {#3} } = { 2 } }
18280         { \int_compare_p:nNn {#5} = { 0 } }
18281     }
18282     {
18283         \prg_replicate:nn {#4}
18284         { \char_generate:nn { \use_i:nn #3 } {12} }
18285         __regex_compile_cs_aux>NNnnnN
18286     }
18287     {
18288         \quark_if_nil:NF #1

```

```

18289     {
18290         \flag_raise:n { __regex_cs }
18291         \use_i_delimit_by_q_recursion_stop:nw
18292     }
18293     \use_none_delimit_by_q_recursion_stop:w
18294 }
18295 }
```

(End definition for `__regex_compile_`: and others.)

36.3.12 Raw token lists with \u

The `\u` escape is invalid in classes and directly following a catcode test. Otherwise, it must be followed by a left brace. We then 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 is missing, then we will 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.

```

18296 \cs_new_protected:cpn { __regex_compile_u: } #1#2
18297 {
18298     \__regex_if_in_class_or_catcode:TF
18299     { \__regex_compile_raw_error:N u #1 #2 }
18300     {
18301         \str_if_eq_x:nnTF {#1#2} { \__regex_compile_special:N \c_left_brace_str }
18302         {
18303             \tl_set:Nx \l__regex_internal_a_tl { \if_false: } \fi:
18304             \__regex_compile_u_loop:NN
18305         }
18306         {
18307             \__msg_kernel_error:nn { regex } { u-missing-lbrace }
18308             \__regex_compile_raw:N u #1 #2
18309         }
18310     }
18311 }
18312 \cs_new:Npn \__regex_compile_u_loop:NN #1#2
18313 {
18314     \token_if_eq_meaning:NNTF #1 \__regex_compile_raw:N
18315     { #2 \__regex_compile_u_loop:NN }
18316     {
18317         \token_if_eq_meaning:NNTF #1 \__regex_compile_special:N
18318         {
18319             \exp_after:wN \token_if_eqCharCode:NNTF \c_right_brace_str #2
18320             { \if_false: { \fi: } \__regex_compile_u_end: }
18321             { #2 \__regex_compile_u_loop:NN }
18322         }
18323         {
18324             \if_false: { \fi: }
18325             \__msg_kernel_error:nnx { regex } { u-missing-rbrace } {#2}
18326             \__regex_compile_u_end:
18327             #1 #2
18328         }
18329     }
18330 }
```

(End definition for `_regex_compile_u:` and `_regex_compile_u_loop>NN`.)

`_regex_compile_u_end:` Once we have extracted the variable's name, we store the contents of that variable in `\l_regex_internal_a_t1`. 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.

```
18331 \cs_new_protected:Npn \_regex_compile_u_end:
18332 {
18333     \tl_set:Nv \l_regex_internal_a_t1 { \l_regex_internal_a_t1 }
18334     \if_int_compare:w \l_regex_mode_int = \c_regex_outer_mode_int
18335         \_regex_compile_u_not_cs:
18336     \else:
18337         \_regex_compile_u_in_cs:
18338     \fi:
18339 }
```

(End definition for `_regex_compile_u_end`.)

`_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.

```
18340 \cs_new_protected:Npn \_regex_compile_u_in_cs:
18341 {
18342     \tl_gset:Nx \g_regex_internal_tl
18343         { \exp_args:No \str_to_other_fast:n { \l_regex_internal_a_t1 } }
18344     \_tl_build_one:x
18345     {
18346         \tl_map_function:NN \g_regex_internal_tl
18347             \_regex_compile_u_in_cs_aux:n
18348     }
18349 }
18350 \cs_new:Npn \_regex_compile_u_in_cs_aux:n #1
18351 {
18352     \_regex_class:NnnnN \c_true_bool
18353         { \_regex_item_caseful_equal:n { \_int_value:w '#1 } }
18354         { 1 } { 0 } \c_false_bool
18355 }
```

(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_t1`. 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.

```
18356 \cs_new_protected:Npn \_regex_compile_u_not_cs:
18357 {
18358     \exp_args:No \_tl_analysis_map_inline:nn { \l_regex_internal_a_t1 }
18359     {
18360         \_tl_build_one:n
18361         {
18362             \_regex_class:NnnnN \c_true_bool
18363             {
18364                 \if_int_compare:w "##2 = 0 \exp_stop_f:
18365                     \_regex_item_exact_cs:n { \exp_after:wN \cs_to_str:N ##1 }
18366                 \else:
18367                     \_regex_item_exact:nn { \_int_value:w "##2 } { ##3 }
```

```

18368          \fi:
18369      }
18370      { 1 } { 0 } \c_false_bool
18371  }
18372 }
18373 }
```

(End definition for `_regex_compile_u_not_cs`.)

36.3.13 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.

```

18374 \cs_new_protected:cpn { \_regex_compile_K: }
18375 {
18376     \int_compare:nNnTF \l__regex_mode_int = \c__regex_outer_mode_int
18377     { \tl_build_one:n { \_regex_command_K: } }
18378     { \_regex_compile_raw_error:N K }
18379 }
```

(End definition for `_regex_compile_K`.)

36.3.14 Showing regexes

`_regex_show:Nn`: Within a `_tl_build:Nw ... _tl_build_end:` group, we redefine all the functions that can appear in a compiled regex, then run the regex. The result is then shown.

```

18380 \cs_new_protected:Npn \_regex_show:Nn #1#2
18381 {
18382     \_tl_build:Nw \l__regex_internal_a_tl
18383     \cs_set_protected:Npn \_regex_branch:n
18384     {
18385         \seq_pop_right:NN \l__regex_show_prefix_seq \l__regex_internal_a_tl
18386         \_regex_show_one:n { ++branch }
18387         \seq_put_right:No \l__regex_show_prefix_seq \l__regex_internal_a_tl
18388         \use:n
18389     }
18390     \cs_set_protected:Npn \_regex_group:nnnN
18391     { \_regex_show_group_aux:nnnnN { } }
18392     \cs_set_protected:Npn \_regex_group_no_capture:nnnN
18393     { \_regex_show_group_aux:nnnnN { ~(no-capture) } }
18394     \cs_set_protected:Npn \_regex_group_resetting:nnnN
18395     { \_regex_show_group_aux:nnnnN { ~(resetting) } }
18396     \cs_set_eq:NN \_regex_class:NnnnN \_regex_show_class:NnnnN
18397     \cs_set_protected:Npn \_regex_command_K:
18398     { \_regex_show_one:n { reset~match~start~(\iow_char:N\K) } }
18399     \cs_set_protected:Npn \_regex_assertion:Nn ##1##2
18400     { \_regex_show_one:n { \bool_if:NF ##1 { negative~ } assertion:~##2 } }
18401     \cs_set:Npn \_regex_b_test: { word~boundary }
18402     \cs_set_eq:NN \_regex_anchor:N \_regex_show_anchor_to_str:N
18403     \cs_set_protected:Npn \_regex_item_caseful_equal:n ##1
18404     { \_regex_show_one:n { char~code~\int_eval:n{##1} } }
18405     \cs_set_protected:Npn \_regex_item_caseful_range:nn ##1##2
18406     { \_regex_show_one:n { range~[\int_eval:n{##1}, \int_eval:n{##2}] } }
```

```

18407 \cs_set_protected:Npn \__regex_item_caseless_equal:n ##1
18408   { \__regex_show_one:n { char-code~\int_eval:n{##1}~(caseless) } }
18409 \cs_set_protected:Npn \__regex_item_caseless_range:nn ##1##2
18410   {
18411     \__regex_show_one:n
18412       { Range~[\int_eval:n{##1}, \int_eval:n{##2}]~(caseless) }
18413   }
18414 \cs_set_protected:Npn \__regex_item_catcode:nT
18415   { \__regex_show_item_catcode:NnT \c_true_bool }
18416 \cs_set_protected:Npn \__regex_item_catcode_reverse:nT
18417   { \__regex_show_item_catcode:NnT \c_false_bool }
18418 \cs_set_protected:Npn \__regex_item_reverse:n
18419   { \__regex_show_scope:nn { Reversed-match } }
18420 \cs_set_protected:Npn \__regex_item_exact:nn ##1##2
18421   { \__regex_show_one:n { char~##2,~catcode~##1 } }
18422 \cs_set_eq:NN \__regex_item_exact_cs:n \__regex_show_item_exact_cs:n
18423 \cs_set_protected:Npn \__regex_item_cs:n
18424   { \__regex_show_scope:nn { control-sequence } }
18425 \cs_set:cpn { __regex_prop_.: } { \__regex_show_one:n { any-token } }
18426 \seq_clear:N \l__regex_show_prefix_seq
18427 \__regex_show_push:n { ~ }
18428 \cs_if_exist_use:N #1
18429 \__tl_build_end:
18430 \__msg_show_variable>NNNnn #1 \cs_if_exist:NTF ? { }
18431   { >~Compiled~regex~#2: \l__regex_internal_a_tl }
18432 }

```

(End definition for `__regex_show:Nn`.)

`__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.

```

18433 \cs_new_protected:Npn \__regex_show_one:n #1
18434   {
18435     \int_incr:N \l__regex_show_lines_int
18436     \__tl_build_one:x
18437     {
18438       \exp_not:N \\
18439       \seq_map_function:NN \l__regex_show_prefix_seq \use:n
18440       #1
18441     }
18442   }

```

(End definition for `__regex_show_one:n`.)

`__regex_show_push:n` `__regex_show_pop:` 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.

```

18443 \cs_new_protected:Npn \__regex_show_push:n #1
18444   { \seq_put_right:Nx \l__regex_show_prefix_seq { #1 ~ } }
18445 \cs_new_protected:Npn \__regex_show_pop:
18446   { \seq_pop_right:NN \l__regex_show_prefix_seq \l__regex_internal_a_tl }
18447 \cs_new_protected:Npn \__regex_show_scope:nn #1#2
18448   {
18449     \__regex_show_one:n {#1}
18450     \__regex_show_push:n { ~ }

```

```

18451      #2
18452      \_\_regex\_show\_pop:
18453  }

```

(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.

```

18454  \cs_new_protected:Npn \_\_regex\_show\_group\_aux:nnnnN #1#2#3#4#5
18455  {
18456      \_\_regex\_show\_one:n { ,-group~begin #1 }
18457      \_\_regex\_show\_push:n { | }
18458      \use_ii:nn #2
18459      \_\_regex\_show\_pop:
18460      \_\_regex\_show\_one:n
18461          { '-group-end \_\_regex\_msg_repeated:nnN {#3} {#4} #5 }
18462  }

```

(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.

```

18463  \cs_set:Npn \_\_regex\_show\_class:NnnnN #1#2#3#4#5
18464  {
18465      \_\_tl_build:Nw \l_\_regex_internal_a_tl
18466      \int_zero:N \l_\_regex_show_lines_int
18467      \_\_regex_show_push:n {~}
18468      #2
18469      \exp_last_unbraced:Nf
18470      \int_case:nnF { \l_\_regex_show_lines_int }
18471      {
18472          {0}
18473          {
18474              \_\_tl_build_end:
18475              \_\_regex_show_one:n { \bool_if:NTF #1 { Fail } { Pass } }
18476          }
18477          {1}
18478          {
18479              \_\_tl_build_end:
18480              \bool_if:NTF #1
18481                  {
18482                      #2
18483                      \_\_tl_build_one:n { \_\_regex_msg_repeated:nnN {#3} {#4} #5 }
18484                  }
18485                  {
18486                      \_\_regex_show_one:n
18487                          { Don't-match~\_\_regex_msg_repeated:nnN {#3} {#4} #5 }
18488                      \_\_tl_build_one:o \l_\_regex_internal_a_tl
18489                  }

```

```

18490     }
18491 }
18492 {
18493     \_tl_build_end:
18494     \_regex_show_one:n
18495     {
18496         \bool_if:NTF #1 { M } { Don't~m } atch
18497         \_regex_msg_repeated:nnN {#3} {#4} #5
18498     }
18499     \_tl_build_one:o \l__regex_internal_a_tl
18500 }
18501 }
```

(End definition for `_regex_show_class:NnnnN`.)

`_regex_show_anchor_to_str:N` The argument is an integer telling us where the anchor is. We convert that to the relevant info.

```

18502 \cs_new:Npn \_regex_show_anchor_to_str:N #1
18503 {
18504     anchor~at~
18505     \str_case:nnF { #1 }
18506     {
18507         { \l__regex_min_pos_int } { start~(\iow_char:N\\A) }
18508         { \l__regex_start_pos_int } { start~of~match~(\iow_char:N\\G) }
18509         { \l__regex_max_pos_int } { end~(\iow_char:N\\Z) }
18510     }
18511     { <error:~'#1'~not~recognized> }
18512 }
```

(End definition for `_regex_show_anchor_to_str:N`.)

`_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.

```

18513 \cs_new_protected:Npn \_regex_show_item_catcode:NnT #1#2
18514 {
18515     \seq_set_split:Nnn \l__regex_internal_seq { } { CBEMTPUDSLOA }
18516     \seq_set_filter:NNn \l__regex_internal_seq \l__regex_internal_seq
18517     { \int_if_odd_p:n { #2 / \int_use:c { c__regex_catcode_##1_int } } }
18518     \_regex_show_scope:nn
18519     {
18520         categories~
18521         \seq_map_function:NN \l__regex_internal_seq \use:n
18522         ,
18523         \bool_if:NF #1 { negative~ } class
18524     }
18525 }
```

(End definition for `_regex_show_item_catcode:NnT`.)

```

\_regex_show_item_exact_cs:n
18526 \cs_new_protected:Npn \_regex_show_item_exact_cs:n #1
18527 {
18528     \seq_set_split:Nnn \l__regex_internal_seq { \scan_stop: } {#1}
18529     \seq_set_map:NNn \l__regex_internal_seq
18530     \l__regex_internal_seq { \iow_char:N\\##1 }
```

```

18531     \_\_regex\_show\_one:n
18532         { control-sequence~ \seq_use:Nn \l\_regex\_internal_seq { ~or~ } }
18533     }

```

(End definition for `__regex_show_item_exact_cs:n`.)

36.4 Building

36.4.1 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, but is included to avoid hard-coding this value everywhere.

```

18534 \int_new:N \l\_regex_min_state_int
18535 \int_set:Nn \l\_regex_min_state_int { 1 }
18536 \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` 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.

```

18537 \int_new:N \l\_regex_left_state_int
18538 \int_new:N \l\_regex_right_state_int
18539 \seq_new:N \l\_regex_left_state_seq
18540 \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 ID number that will be assigned to a capturing group if one was opened now. 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.

```
18541 \int_new:N \l\_regex_capturing_group_int
```

(End definition for `\l_regex_capturing_group_int`.)

36.4.2 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`: inserted at the start of the regular expression to make it unanchored.
- `__regex_action_success`: marks the exit state of the NFA.
- `__regex_action_cost:n {\langle shift \rangle}` is a transition from the current `\langle state \rangle` to `\langle state \rangle + \langle 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` { $\langle shift \rangle$ }, and `_regex_action_free_group:n` { $\langle shift \rangle$ } are free transitions, which immediately perform the actions for the state $\langle state \rangle + \langle shift \rangle$ 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:n` { $\langle key \rangle$ } where the $\langle key \rangle$ is a group number followed by `<` or `>` for the beginning or end of group. This causes the current position in the query to be stored as the $\langle key \rangle$ submatch boundary.

We strive to preserve the following properties while building.

- The current capturing group is `capturing_group - 1`, and if a group is opened now, it will 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, which will be numbered 0 (current value of `capturing_group`). Finally, if the match reaches the last state, it is successful.

```

18542 \cs_new_protected:Npn \_regex_build:n #1
18543 {
18544     \_regex_compile:n [#1]
18545     \_regex_build:N \l__regex_internal_regex
18546 }
18547 \cs_new_protected:Npn \_regex_build:N #1
18548 {
18549 (trace)    \trace_push:nnn { regex } { 1 } { \_regex_build }
18550     \_regex_standard_escapechar:
18551     \int_zero:N \l__regex_capturing_group_int
18552     \int_set_eq:NN \l__regex_max_state_int \l__regex_min_state_int
18553     \_regex_build_new_state:
18554     \_regex_build_new_state:
18555     \_regex_toks_put_right:Nn \l__regex_left_state_int
18556         { \_regex_action_start_wildcard: }
18557     \_regex_group:nnnN [#1] { 1 } { 0 } \c_false_bool
18558     \_regex_toks_put_right:Nn \l__regex_right_state_int
18559         { \_regex_action_success: }
18560 (trace)    \_regex_trace_states:n { 2 }
18561 (trace)    \trace_pop:nnn { regex } { 1 } { \_regex_build }
18562 }
```

(End definition for `_regex_build:n` and `_regex_build:N`.)

__regex_build_for_cs:n 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.

```

18563 \cs_new_protected:Npn \_\_regex_build_for_cs:n #1
18564  {
18565  <trace>    \trace_push:nnn { regex } { 1 } { \_\_regex_build_for_cs }
18566  \int_set_eq:NN \l_\_regex_max_state_int \l_\_regex_min_state_int
18567  \_\_regex_build_new_state:
18568  \_\_regex_build_new_state:
18569  \_\_regex_push_lr_states:
18570  #1
18571  \_\_regex_pop_lr_states:
18572  \_\_regex_toks_put_right:Nn \l_\_regex_right_state_int
18573  {
18574  \if_int_compare:w \l_\_regex_current_pos_int = \l_\_regex_max_pos_int
18575  \exp_after:wn \_\_regex_action_success:
18576  \fi:
18577  }
18578 <trace>    \_\_regex_trace_states:n { 2 }
18579 <trace>    \trace_pop:nnn { regex } { 1 } { \_\_regex_build_for_cs }
18580 }
```

(End definition for __regex_build_for_cs:n.)

36.4.3 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.
__regex_pop_lr_states:

```

18581 \cs_new_protected:Npn \_\_regex_push_lr_states:
18582  {
18583  \seq_push:No \l_\_regex_left_state_seq
18584  { \int_use:N \l_\_regex_left_state_int }
18585  \seq_push:No \l_\_regex_right_state_seq
18586  { \int_use:N \l_\_regex_right_state_int }
18587  }
18588 \cs_new_protected:Npn \_\_regex_pop_lr_states:
18589  {
18590  \seq_pop:NN \l_\_regex_left_state_seq \l_\_regex_internal_a_t1
18591  \int_set:Nn \l_\_regex_left_state_int \l_\_regex_internal_a_t1
18592  \seq_pop:NN \l_\_regex_right_state_seq \l_\_regex_internal_a_t1
18593  \int_set:Nn \l_\_regex_right_state_int \l_\_regex_internal_a_t1
18594 }
```

(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.

```

18595 \cs_new_protected:Npn \_\_regex_build_transition_left:NNN #1#2#3
18596  { \_\_regex_toks_put_left:Nx #2 { #1 { \int_eval:n { #3 - #2 } } } }
18597 \cs_new_protected:Npn \_\_regex_build_transition_right:nNn #1#2#3
18598  { \_\_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.

```
18599 \cs_new_protected:Npn \_\_regex_build_new_state:
18600 {
18601 (*trace)
18602     \trace:nnx { regex } { 2 }
18603     {
18604         regex~new~state~
18605         L=\int_use:N \l_\_regex_left_state_int ~ -> ~
18606         R=\int_use:N \l_\_regex_right_state_int ~ -> ~
18607         M=\int_use:N \l_\_regex_max_state_int ~ -> ~
18608         \int_eval:n { \l_\_regex_max_state_int + 1 }
18609     }
18610 /trace>
18611     \_\_regex_toks_clear:N \l_\_regex_max_state_int
18612     \int_set_eq:NN \l_\_regex_left_state_int \l_\_regex_right_state_int
18613     \int_set_eq:NN \l_\_regex_right_state_int \l_\_regex_max_state_int
18614     \int_incr:N \l_\_regex_max_state_int
18615 }
```

(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.

```
18616 \cs_new_protected:Npn \_\_regex_build_transitions_lazyness:NNNNN #1#2#3#4#5
18617 {
18618     \_\_regex_build_new_state:
18619     \_\_regex_toks_put_right:Nx \l_\_regex_left_state_int
18620     {
18621         \if_meaning:w \c_true_bool #1
18622             #2 { \int_eval:n { #3 - \l_\_regex_left_state_int } }
18623             #4 { \int_eval:n { #5 - \l_\_regex_left_state_int } }
18624         \else:
18625             #4 { \int_eval:n { #5 - \l_\_regex_left_state_int } }
18626             #2 { \int_eval:n { #3 - \l_\_regex_left_state_int } }
18627         \fi:
18628     }
18629 }
```

(End definition for `__regex_build_transitions_lazyness:NNNNN`.)

36.4.4 Building classes

`__regex_class:NnnnN` 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.

```
18630 \cs_new_protected:Npn \_\_regex_class:NnnnN #1#2#3#4#5
```

```

18631    {
18632      \cs_set:Npx \__regex_tests_action_cost:n ##1
18633      {
18634        \exp_not:n { \exp_not:n {#2} }
18635        \bool_if:NTF #1
18636          { \__regex_break_point:TF { \__regex_action_cost:n {##1} } { } }
18637          { \__regex_break_point:TF { } { \__regex_action_cost:n {##1} } }
18638      }
18639      \if_case:w - #4 \exp_stop_f:
18640        \__regex_class_repeat:n {#3}
18641      \or: \__regex_class_repeat:nN {#3} #5
18642      \else: \__regex_class_repeat:nnN {#3} {#4} #5
18643      \fi:
18644    }
18645  \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.

```

18646  \cs_new_protected:Npn \__regex_class_repeat:n #1
18647  {
18648    \prg_replicate:nn {#1}
18649    {
18650      \__regex_build_new_state:
18651      \__regex_build_transition_right:nNn \__regex_tests_action_cost:n
18652        \l__regex_left_state_int \l__regex_right_state_int
18653    }
18654  }
```

(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.

```

18655  \cs_new_protected:Npn \__regex_class_repeat:nN #1#2
18656  {
18657    \if_int_compare:w #1 = 0 \exp_stop_f:
18658      \__regex_build_transitions_lazyness:NNNNN #2
18659      \__regex_action_free:n \l__regex_right_state_int
18660      \__regex_tests_action_cost:n \l__regex_left_state_int
18661  \else:
18662    \__regex_class_repeat:n {#1}
18663    \int_set_eq:NN \l__regex_internal_a_int \l__regex_left_state_int
18664    \__regex_build_transitions_lazyness:NNNNN #2
18665    \__regex_action_free:n \l__regex_right_state_int
18666    \__regex_action_free:n \l__regex_internal_a_int
18667  \fi:
18668 }
```

(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`.

```
18669 \cs_new_protected:Npn \_\_regex\_class\_repeat:nnN #1#2#3
18670 {
18671     \_\_regex\_class\_repeat:n {#1}
18672     \int_set:Nn \l_\_regex_internal_a_int
18673         { \l_\_regex_max_state_int + #2 - 1 }
18674     \prg_replicate:nn { #2 }
18675     {
18676         \_\_regex_build_transitions_lazyness:NNNNN #3
18677             \_\_regex_action_free:n \l_\_regex_internal_a_int
18678                 \_\_regex_tests_action_cost:n \l_\_regex_right_state_int
18679     }
18680 }
```

(End definition for `__regex_class_repeat:nnN`.)

36.4.5 Building groups

`__regex_group_aux:nnnn` Arguments: {`<label>`} {`<contents>`} {`<min>`} {`<more>`} `<lazyness>`. If `<min>` 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 will 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 `<label>` #1 is used for submatch tracking. Each of the three auxiliaries expects `left_state` and `right_state` to be set properly.

```
18681 \cs_new_protected:Npn \_\_regex_group_aux:nnnnN #1#2#3#4#5
18682 {
18683     \trace_push:nnn { regex } { 1 } { \_\_regex_group }
18684     \if_int_compare:w #3 = 0 \exp_stop_f:
18685         \_\_regex_build_new_state:
18686     \assert \assert_int:n { \l_\_regex_max_state_int = \l_\_regex_right_state_int + 1 }
18687         \_\_regex_build_transition_right:nNn \_\_regex_action_free_group:n
18688             \l_\_regex_left_state_int \l_\_regex_right_state_int
18689     \fi:
18690     \_\_regex_build_new_state:
18691     \_\_regex_push_lr_states:
18692     #2
18693     \_\_regex_pop_lr_states:
18694     \if_case:w - #4 \exp_stop_f:
18695         \_\_regex_group_repeat:nn {#1} {#3}
18696     \or: \_\_regex_group_repeat:nnN {#1} {#3} #5
18697     \else: \_\_regex_group_repeat:nnnN {#1} {#3} {#4} #5
18698     \fi:
18699     \trace_pop:nnn { regex } { 1 } { \_\_regex_group }
```

18700 }

(End definition for `__regex_group_aux:nnnnN`.)

`__regex_group:nnnN` Hand to `__regex_group_aux:nnnnnN` the label of that group (expanded), and the group itself, with some extra commands to perform.

```
18701 \cs_new_protected:Npn \_\_regex_group:nnnN #1
18702 {
18703     \exp_args:No \_\_regex_group_aux:nnnnN
18704     { \int_use:N \l_\_regex_capturing_group_int }
18705     {
18706         \int_incr:N \l_\_regex_capturing_group_int
18707         #1
18708     }
18709 }
18710 \cs_new_protected:Npn \_\_regex_group_no_capture:nnnN
18711 { \_\_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>}`.

```
18712 \cs_new_protected:Npn \_\_regex_group_resetting:nnnN #1
18713 {
18714     \_\_regex_group_aux:nnnnN { -1 }
18715     {
18716         \exp_args:Noo \_\_regex_group_resetting_loop:nnNn
18717         { \int_use:N \l_\_regex_capturing_group_int }
18718         { \int_use:N \l_\_regex_capturing_group_int }
18719         #1
18720         { ?? \_\_prg_break:n } { }
18721         \_\_prg_break_point:
18722     }
18723 }
18724 \cs_new_protected:Npn \_\_regex_group_resetting_loop:nnNn #1#2#3#4
18725 {
18726     \use_none:nn #3 { \int_set:Nn \l_\_regex_capturing_group_int {#1} }
18727     \int_set:Nn \l_\_regex_capturing_group_int {#2}
18728     #3 {#4}
18729     \exp_args:Nf \_\_regex_group_resetting_loop:nnNn
18730     { \int_max:nn {#1} { \l_\_regex_capturing_group_int } }
18731     {#2}
18732 }
```

(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.

```
18733 \cs_new_protected:Npn \_\_regex_branch:n #1
18734 {
```

```

18735   <trace>    \trace_push:nnn { regex } { 1 } { __regex_branch }
18736       __regex_build_new_state:
18737       \seq_get:NN \l__regex_left_state_seq \l__regex_internal_a_t1
18738       \int_set:Nn \l__regex_left_state_int \l__regex_internal_a_t1
18739       __regex_build_transition_right:nNn \__regex_action_free:n
18740           \l__regex_left_state_int \l__regex_right_state_int
18741       #1
18742       \seq_get:NN \l__regex_right_state_seq \l__regex_internal_a_t1
18743       __regex_build_transition_right:nNn \__regex_action_free:n
18744           \l__regex_right_state_int \l__regex_internal_a_t1
18745   <trace>    \trace_pop:nnn { regex } { 1 } { __regex_branch }
18746   }

```

(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.

```

18747   \cs_new_protected:Npn \__regex_group_repeat:nn #1#2
18748   {
18749       \if_int_compare:w #2 = 0 \exp_stop_f:
18750           \int_set:Nn \l__regex_max_state_int
18751               { \l__regex_left_state_int - 1 }
18752           __regex_build_new_state:
18753       \else:
18754           __regex_group_repeat_aux:n {#2}
18755           __regex_group_submatches:nNN {#1}
18756               \l__regex_internal_a_int \l__regex_right_state_int
18757           __regex_build_new_state:
18758       \fi:
18759   }

```

(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`.

```

18760   \cs_new_protected:Npn \__regex_group_submatches:nNN #1#2#3
18761   {
18762       \if_int_compare:w #1 > - 1 \exp_stop_f:
18763           __regex_toks_put_left:Nx #2 { \__regex_action_submatch:n { #1 < } }
18764           __regex_toks_put_left:Nx #3 { \__regex_action_submatch:n { #1 > } }
18765       \fi:
18766   }

```

(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 will "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.

```

18767 \cs_new_protected:Npn \__regex_group_repeat_aux:n #1
18768  {
18769      \__regex_build_transition_right:nNn \__regex_action_free:n
18770          \l__regex_right_state_int \l__regex_max_state_int
18771      \int_set_eq:NN \l__regex_internal_a_int \l__regex_left_state_int
18772      \int_set_eq:NN \l__regex_internal_b_int \l__regex_max_state_int
18773      \if_int_compare:w \__int_eval:w #1 > 1 \exp_stop_f:
18774          \int_set:Nn \l__regex_internal_c_int
18775          {
18776              ( #1 - 1 )
18777              * ( \l__regex_internal_b_int - \l__regex_internal_a_int )
18778          }
18779          \int_add:Nn \l__regex_right_state_int { \l__regex_internal_c_int }
18780          \int_add:Nn \l__regex_max_state_int { \l__regex_internal_c_int }
18781          \__regex_toks_memcpy:NNn
18782              \l__regex_internal_b_int
18783              \l__regex_internal_a_int
18784              \l__regex_internal_c_int
18785      \fi:
18786  }

```

(End definition for `__regex_group_repeat_aux:n`.)

`__regex_group_repeat:nN` 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`.

```

18787 \cs_new_protected:Npn \__regex_group_repeat:nnN #1#2#3
18788  {
18789      \if_int_compare:w #2 = 0 \exp_stop_f:
18790          \__regex_group_submatches:nNN {#1}
18791              \l__regex_left_state_int \l__regex_right_state_int
18792          \int_set:Nn \l__regex_internal_a_int
18793              { \l__regex_left_state_int - 1 }
18794          \__regex_build_transition_right:nNn \__regex_action_free:n
18795              \l__regex_right_state_int \l__regex_internal_a_int
18796          \__regex_build_new_state:
18797          \if_meaning:w \c_true_bool #3
18798              \__regex_build_transition_left:NNN \__regex_action_free:n
18799                  \l__regex_internal_a_int \l__regex_right_state_int
18800          \else:
18801              \__regex_build_transition_right:nNn \__regex_action_free:n
18802                  \l__regex_internal_a_int \l__regex_right_state_int
18803          \fi:
18804      \else:

```

```

18805   \_\_regex_group_repeat_aux:n {#2}
18806   \_\_regex_group_submatches:nNN {#1}
18807     \l\_\_regex\_internal_a\_int \l\_\_regex_right_state_int
18808 \if_meaning:w \c_true_bool #3
18809   \_\_regex_build_transition_right:nNn \_\_regex_action_free_group:n
18810     \l\_\_regex_right_state_int \l\_\_regex_internal_a_int
18811 \else:
18812   \_\_regex_build_transition_left:NNN \_\_regex_action_free_group:n
18813     \l\_\_regex_right_state_int \l\_\_regex_internal_a_int
18814 \fi:
18815   \_\_regex_build_new_state:
18816 \fi:
18817 }

```

(End definition for `__regex_group_repeat:nnN`.)

`__regex_group_repeat:nnN`

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.

```

18818 \cs_new_protected:Npn \_\_regex_group_repeat:nnN #1#2#3#4
18819 {
18820   \_\_regex_group_submatches:nNN {#1}
18821     \l\_\_regex_left_state_int \l\_\_regex_right_state_int
18822   \_\_regex_group_repeat_aux:n { #2 + #3 }
18823 \if_meaning:w \c_true_bool #4
18824   \int_set_eq:NN \l\_\_regex_left_state_int \l\_\_regex_max_state_int
18825   \prg_replicate:nn { #3 }
18826   {
18827     \int_sub:Nn \l\_\_regex_left_state_int
18828       { \l\_\_regex_internal_b_int - \l\_\_regex_internal_a_int }
18829     \_\_regex_build_transition_left:NNN \_\_regex_action_free:n
18830       \l\_\_regex_left_state_int \l\_\_regex_max_state_int
18831   }
18832 \else:
18833   \prg_replicate:nn { #3 - 1 }
18834   {
18835     \int_sub:Nn \l\_\_regex_right_state_int
18836       { \l\_\_regex_internal_b_int - \l\_\_regex_internal_a_int }
18837     \_\_regex_build_transition_right:nNn \_\_regex_action_free:n
18838       \l\_\_regex_right_state_int \l\_\_regex_max_state_int
18839   }
18840 \if_int_compare:w #2 = 0 \exp_stop_f:
18841   \int_set:Nn \l\_\_regex_right_state_int
18842     { \l\_\_regex_left_state_int - 1 }
18843 \else:

```

```

18844     \int_sub:Nn \l__regex_right_state_int
18845     { \l__regex_internal_b_int - \l__regex_internal_a_int }
18846 \fi:
18847     \_regex_build_transition_right:nNn \_regex_action_free:n
18848     \l__regex_right_state_int \l__regex_max_state_int
18849 \fi:
18850     \_regex_build_new_state:
18851 }

```

(End definition for `_regex_group_repeat:nnnN`.)

36.4.6 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.
- `_regex_anchor:N` 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 boundary-markers of the string are non-word characters for this purpose. Anchors at the start or end of match use `_regex_anchor:N`, with a position controlled by the integer #1.

```

18852 \cs_new_protected:Npn \_regex_assertion:Nn #1#2
18853 {
18854     \_regex_build_new_state:
18855     \_regex_toks_put_right:Nx \l__regex_left_state_int
18856     {
18857         \exp_not:n {#2}
18858         \_regex_break_point:TF
18859         \bool_if:NF #1 { { } }
18860         {
18861             \_regex_action_free:n
18862             {
18863                 \int_eval:n
18864                 { \l__regex_right_state_int - \l__regex_left_state_int }
18865             }
18866         }
18867         \bool_if:NT #1 { { } }
18868     }
18869 }
18870 \cs_new_protected:Npn \_regex_anchor:N #1
18871 {
18872     \if_int_compare:w #1 = \l__regex_current_pos_int
18873     \exp_after:wN \_regex_break_true:w
18874 \fi:
18875 }
18876 \cs_new_protected:Npn \_regex_b_test:
18877 {
18878     \group_begin:
18879     \int_set_eq:NN \l__regex_current_char_int \l__regex_last_char_int
18880     \_regex_prop_w:
18881     \_regex_break_point:TF
18882     { \group_end: \_regex_item_reverse:n \_regex_prop_w: }
18883     { \group_end: \_regex_prop_w: }
18884 }

```

(End definition for `_regex_assertion:Nn`, `_regex_b_test:`, and `_regex_anchor:N`.)

- __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.

```

18885 \cs_new_protected:Npn \_\_regex_command_K:
18886 {
18887     \_\_regex_build_new_state:
18888     \_\_regex_toks_put_right:Nx \l\_\_regex_left_state_int
18889     {
18890         \_\_regex_action_submatch:n { 0< }
18891         \bool_set_true:N \l\_\_regex_fresh_thread_bool
18892         \_\_regex_action_free:n
18893             { \int_eval:n { \l\_\_regex_right_state_int - \l\_\_regex_left_state_int } }
18894         \bool_set_false:N \l\_\_regex_fresh_thread_bool
18895     }
18896 }
```

(End definition for __regex_command_K:.)

36.5 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_state_intarray`: this thread will be 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 the future execution will 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 will keep 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.

36.5.1 Variables used when matching

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.

```
\l\_\_regex_min_pos_int
\l\_\_regex_max_pos_int
\l\_\_regex_current_pos_int
\l\_\_regex_start_pos_int
\l\_\_regex_success_pos_int
```

We don't start from 0 because the `\toks` registers with low numbers are used to hold the states of the NFA. We match without backtracking, keeping all threads in lockstep at the `current_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.

```
18897 \int_new:N \l__regex_min_pos_int
18898 \int_new:N \l__regex_max_pos_int
18899 \int_new:N \l__regex_current_pos_int
18900 \int_new:N \l__regex_start_pos_int
18901 \int_new:N \l__regex_success_pos_int
```

(End definition for `\l__regex_min_pos_int` and others.)

`\l__regex_current_char_int` The character and category codes of the token at the current position; the character code of the token at the previous position; and the character code of the result of changing the case of the current token ($A-Z \leftrightarrow a-z$). This last integer is only computed when necessary, and is otherwise `\c_max_int`. The `current_char` variable is also used in various other phases to hold a character code.

```
18902 \int_new:N \l__regex_current_char_int
18903 \int_new:N \l__regex_current_catcode_int
18904 \int_new:N \l__regex_last_char_int
18905 \int_new:N \l__regex_case_changed_char_int
```

(End definition for `\l__regex_current_char_int` and others.)

`\l__regex_current_state_int` For every character in the token list, each of the active states is considered in turn. The variable `\l__regex_current_state_int` holds the state of the NFA which is currently considered: transitions are then given as shifts relative to the current state.

```
18906 \int_new:N \l__regex_current_state_int
```

(End definition for `\l__regex_current_state_int`.)

`\l__regex_current_submatches_prop` The submatches for the thread which is currently active are stored in the `current_submatches` property list variable. This property list is stored by `_regex_action_cost:n` into the `\toks` register for the target state of the transition, to be retrieved when matching at the next position. When a thread succeeds, this property list is copied to `\l__regex_success_submatches_prop`: only the last successful thread will remain there.

```
18907 \prop_new:N \l__regex_current_submatches_prop
18908 \prop_new:N \l__regex_success_submatches_prop
```

(End definition for `\l__regex_current_submatches_prop` and `\l__regex_success_submatches_prop`.)

`\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 `<state>` 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(state)`, 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).

```
18909 \int_new:N \l__regex_step_int
```

(End definition for `\l_regex_step_int`.)

`\l_regex_min_active_int` All the currently active threads are kept in order of precedence in `\g_regex_thread_state_intarray`, and the corresponding submatches in the `\toks`. For our purposes, those serve as an array, indexed from `min_active` (inclusive) to `max_active` (excluded). At the start of every step, the whole array is unpacked, so that the space can immediately be reused, and `max_active` is reset to `min_active`, effectively clearing the array.

```
18910 \int_new:N \l_regex_min_active_int  
18911 \int_new:N \l_regex_max_active_int
```

(End definition for `\l_regex_min_active_int` and `\l_regex_max_active_int`.)

`\g_regex_state_active_intarray` `\g_regex_state_active_intarray` stores the last `<step>` in which each `<state>` was active. `\g_regex_thread_state_intarray` stores threads that will be considered in the next step, more precisely the states in which these threads are.

```
18912 \__intarray_new:Nn \g_regex_state_active_intarray { 65536 }  
18913 \__intarray_new:Nn \g_regex_thread_state_intarray { 65536 }
```

(End definition for `\g_regex_state_active_intarray` and `\g_regex_thread_state_intarray`.)

`\l_regex_every_match_t1` 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`.

```
18914 \tl_new:N \l_regex_every_match_t1
```

(End definition for `\l_regex_every_match_t1`.)

`\l_regex_fresh_thread_bool` 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`.

```
18915 \bool_new:N \l_regex_fresh_thread_bool  
18916 \bool_new:N \l_regex_empty_success_bool  
18917 \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` 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.

```
18918 \bool_new:N \g_regex_success_bool  
18919 \bool_new:N \l_regex_saved_success_bool  
18920 \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`.)

36.5.2 Matching: framework

- `__regex_match:n` First store the query into `\toks` registers and arrays (see `__regex_query_set:nnn`). Then 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.

```

18921 \cs_new_protected:Npn \__regex_match:n #1
18922   {
18923     \trace_push:nnx { regex } { 1 } { __regex_match }
18924     \trace:nnx { regex } { 1 } { analyzing~query~token~list }
18925       \int_zero:N \l__regex_balance_int
18926       \int_set:Nn \l__regex_current_pos_int { 2 * \l__regex_max_state_int }
18927       \__regex_query_set:nnn { } { -1 } { -2 }
18928       \int_set_eq:NN \l__regex_min_pos_int \l__regex_current_pos_int
18929       \__tl_analysis_map_inline:nn {#1}
18930         { \__regex_query_set:nnn {##1} {"##2} {"##3} }
18931         \int_set_eq:NN \l__regex_max_pos_int \l__regex_current_pos_int
18932         \__regex_query_set:nnn { } { -1 } { -2 }
18933     \trace:nnx { regex } { 1 } { initializing }
18934       \bool_gset_false:N \g__regex_success_bool
18935       \int_step_inline:nnnn
18936         \l__regex_min_state_int { 1 } { \l__regex_max_state_int - 1 }
18937         { \__intarray_gset_fast:Nnn \g__regex_state_active_intarray {##1} { 1 } }
18938       \int_set_eq:NN \l__regex_min_active_int \l__regex_max_state_int
18939       \int_zero:N \l__regex_step_int
18940       \int_set_eq:NN \l__regex_success_pos_int \l__regex_min_pos_int
18941       \int_set:Nn \l__regex_min_submatch_int
18942         { 2 * \l__regex_max_state_int }
18943       \int_set_eq:NN \l__regex_submatch_int \l__regex_min_submatch_int
18944       \bool_set_false:N \l__regex_empty_success_bool
18945       \__regex_match_once:
18946     \trace_pop:nnx { regex } { 1 } { __regex_match }
18947   }

```

(End definition for `__regex_match:n`.)

- `__regex_match_once:` This function finds one match, then does some action defined by the `every_match` token list, which may recursively call `__regex_match_once::`. 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, and get that token, so that the `last_char` will be set properly for word boundaries. Then call `__regex_match_loop::`, which runs through the query until the end or until a successful match breaks early.

```

18948 \cs_new_protected:Npn \__regex_match_once:
18949   {

```

```

18950      \if_meaning:w \c_true_bool \l__regex_empty_success_bool
18951          \cs_set:Npn \__regex_if_two_empty_matches:F
18952              { \int_compare:nNnF \l__regex_start_pos_int = \l__regex_current_pos_int }
18953      \else:
18954          \cs_set_eq:NN \__regex_if_two_empty_matches:F \use:n
18955      \fi:
18956          \int_set_eq:NN \l__regex_start_pos_int \l__regex_success_pos_int
18957          \bool_set_false:N \l__regex_match_success_bool
18958          \prop_clear:N \l__regex_current_submatches_prop
18959          \int_set_eq:NN \l__regex_max_active_int \l__regex_min_active_int
18960          \__regex_store_state:n { \l__regex_min_state_int }
18961          \int_set:Nn \l__regex_current_pos_int
18962              { \l__regex_start_pos_int - 1 }
18963          \__regex_query_get:
18964          \__regex_match_loop:
18965              \l__regex_every_match_tl
18966      }

```

(End definition for `__regex_match_once:..`)

`__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.

```

18967      \cs_new_protected:Npn \__regex_single_match:
18968          {
18969              \tl_set:Nn \l__regex_every_match_tl
18970                  { \bool_gset_eq:NN \g__regex_success_bool \l__regex_match_success_bool }
18971          }
18972      \cs_new_protected:Npn \__regex_multi_match:n #1
18973          {
18974              \tl_set:Nn \l__regex_every_match_tl
18975                  {
18976                      \if_meaning:w \c_true_bool \l__regex_match_success_bool
18977                          \bool_gset_true:N \g__regex_success_bool
18978                          #1
18979                          \exp_after:wn \__regex_match_once:
18980                      \fi:
18981                  }
18982          }

```

(End definition for `__regex_single_match: and __regex_multi_match:n.`)

`__regex_match_loop:` 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_active`). This results in a sequence of `__regex_use_state_and_submatches:nn` `{(state)}` `{(prop)}`, 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 what `__regex_match_once:` matches. We explain the `fresh_thread` business when describing `__regex_action_wildcard:`

```

18983      \cs_new_protected:Npn \__regex_match_loop:
18984          {
18985              \int_add:Nn \l__regex_step_int { 2 }
18986              \int_incr:N \l__regex_current_pos_int

```

```

18987 \int_set_eq:NN \l__regex_last_char_int \l__regex_current_char_int
18988 \int_set_eq:NN \l__regex_case_changed_char_int \c_max_int
18989 \__regex_query_get:
18990 \use:x
18991 {
18992     \int_set_eq:NN \l__regex_max_active_int \l__regex_min_active_int
18993     \int_step_function:nnN
18994     { \l__regex_min_active_int }
18995     { 1 }
18996     { \l__regex_max_active_int - 1 }
18997     \__regex_match_one_active:n
18998 }
18999 \__prg_break_point:
19000 \bool_set_false:N \l__regex_fresh_thread_bool %^^A was arg of break_point:n
19001 \if_int_compare:w \l__regex_max_active_int > \l__regex_min_active_int
19002     \if_int_compare:w \l__regex_current_pos_int < \l__regex_max_pos_int
19003         \exp_after:wN \exp_after:wN \exp_after:wN \__regex_match_loop:
19004     \fi:
19005 \fi:
19006 }
19007 \cs_new:Npn \__regex_match_one_active:n #1
19008 {
19009     \__regex_use_state_and_submatches:nn
19010     { \__intarray_item_fast:Nn \g__regex_thread_state_intarray {#1} }
19011     { \__regex_toks_use:w #1 }
19012 }

```

(End definition for `__regex_match_loop:` and `__regex_match_one_active:n.`)

`__regex_query_set:nnn`

The arguments are: tokens that o and x expand to one token of the query, the catcode, and the character code. Store those, and the current brace balance (used later to check for overall brace balance) in a `\toks` register and some arrays, then update the `balance`.

```

19013 \cs_new_protected:Npn \__regex_query_set:nnn #1#2#3
19014 {
19015     \__intarray_gset_fast:Nnn \g__regex_charcode_intarray
19016     { \l__regex_current_pos_int } {#3}
19017     \__intarray_gset_fast:Nnn \g__regex_catcode_intarray
19018     { \l__regex_current_pos_int } {#2}
19019     \__intarray_gset_fast:Nnn \g__regex_balance_intarray
19020     { \l__regex_current_pos_int } { \l__regex_balance_int }
19021     \__regex_toks_set:Nn \l__regex_current_pos_int {#1}
19022     \int_incr:N \l__regex_current_pos_int
19023     \if_case:w #2 \exp_stop_f:
19024     \or: \int_incr:N \l__regex_balance_int
19025     \or: \int_decr:N \l__regex_balance_int
19026     \fi:
19027 }

```

(End definition for `__regex_query_set:nnn.`)

`__regex_query_get:`

Extract the current character and category codes at the current position from the appropriate arrays.

```

19028 \cs_new_protected:Npn \__regex_query_get:
19029 {

```

```

19030     \l__regex_current_char_int
19031     = \_\_intarray_item_fast:Nn \g__regex_charcode_intarray
19032         { \l__regex_current_pos_int } \scan_stop:
19033     \l__regex_current_catcode_int
19034     = \_\_intarray_item_fast:Nn \g__regex_catcode_intarray
19035         { \l__regex_current_pos_int } \scan_stop:
19036 }

```

(End definition for `__regex_query_get:..`)

36.5.3 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.

```

19037 \cs_new_protected:Npn \_\_regex_use_state:
19038 {
19039 (*trace)
19040     \trace:nnx { regex } { 2 } { state~\int_use:N \l__regex_current_state_int }
19041 (/trace)
19042     \_\_intarray_gset_fast:Nnn \g__regex_state_active_intarray
19043         { \l__regex_current_state_int } { \l__regex_step_int }
19044     \_\_regex_toks_use:w \l__regex_current_state_int
19045     \_\_intarray_gset_fast:Nnn \g__regex_state_active_intarray
19046         { \l__regex_current_state_int } { \l__regex_step_int + 1 }
19047 }

```

(End definition for `__regex_use_state:..`)

- `__regex_use_state_and_submatches:nn` 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 `current_state` and `current_submatches` and use the state if it has not yet been encountered at this step.

```

19048 \cs_new_protected:Npn \_\_regex_use_state_and_submatches:nn #1 #2
19049 {
19050     \int_set:Nn \l__regex_current_state_int {#1}
19051     \if_int_compare:w
19052         \_\_intarray_item_fast:Nn \g__regex_state_active_intarray
19053             { \l__regex_current_state_int }
19054                 < \l__regex_step_int
19055         \tl_set:Nn \l__regex_current_submatches_prop {#2}
19056         \exp_after:wN \_\_regex_use_state:
19057     \fi:
19058     \scan_stop:
19059 }

```

(End definition for `__regex_use_state_and_submatches:nn.`)

36.5.4 Actions when matching

- `__regex_action_startWildcard:` 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_loop`: too.

```
19060 \cs_new_protected:Npn \_regex_action_start_wildcard:
19061 {
19062     \bool_set_true:N \l__regex_fresh_thread_bool
19063     \_regex_action_free:n {1}
19064     \bool_set_false:N \l__regex_fresh_thread_bool
19065     \_regex_action_cost:n {0}
19066 }
```

(End definition for `_regex_action_start_wildcard`.)

`_regex_action_free:n` 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_current_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 when within the thread itself.

```
19067 \cs_new_protected:Npn \_regex_action_free:n
19068     { \_regex_action_free_aux:nn { > \l__regex_step_int \else: } }
19069 \cs_new_protected:Npn \_regex_action_free_group:n
19070     { \_regex_action_free_aux:nn { < \l__regex_step_int } }
19071 \cs_new_protected:Npn \_regex_action_free_aux:nn #1#2
19072 {
19073     \use:x
19074     {
19075         \int_add:Nn \l__regex_current_state_int {#2}
19076         \exp_not:n
19077         {
19078             \if_int_compare:w
19079                 \__intarray_item_fast:Nn \g__regex_state_active_intarray
19080                 { \l__regex_current_state_int }
19081             #1
19082             \exp_after:wN \_regex_use_state:
19083             \fi:
19084         }
19085         \int_set:Nn \l__regex_current_state_int
19086             { \int_use:N \l__regex_current_state_int }
19087         \tl_set:Nn \exp_not:N \l__regex_current_submatches_prop
19088             { \exp_not:o \l__regex_current_submatches_prop }
19089     }
19090 }
```

(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.

```
19091 \cs_new_protected:Npn \_regex_action_cost:n #1
19092 {
19093     \exp_args:No \_regex_store_state:n
19094     { \__int_value:w \__int_eval:w \l__regex_current_state_int + #1 }
```

```
19095 }
```

(End definition for `__regex_action_cost:n.`)

- `__regex_store_state:n`: Put the given state in `\g__regex_thread_state_intarray`, and increment the length of the array. Also store the current submatch in the appropriate `\toks`.

```
19096 \cs_new_protected:Npn \_\_regex_store_state:n #1
19097 {
19098     \_\_regex_store_submatches:
19099     \_\_intarray_gset_fast:Nnn \g\_\_regex_thread_state_intarray
19100     { \l\_\_regex_max_active_int } {#1}
19101     \int_incr:N \l\_\_regex_max_active_int
19102 }
19103 \cs_new_protected:Npn \_\_regex_store_submatches:
19104 {
19105     \_\_regex_toks_set:No \l\_\_regex_max_active_int
19106     { \l\_\_regex_current_submatches_prop }
19107 }
```

(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.

```
19108 \cs_new_protected:Npn \_\_regex_disable_submatches:
19109 {
19110     \cs_set_protected:Npn \_\_regex_store_submatches: { }
19111     \cs_set_protected:Npn \_\_regex_action_submatch:n ##1 { }
19112 }
```

(End definition for `__regex_disable_submatches:.`)

- `__regex_action_submatch:n`: Update the current submatches with the information from the current position. Maybe a bottleneck.

```
19113 \cs_new_protected:Npn \_\_regex_action_submatch:n #1
19114 {
19115     \prop_put:Nno \l\_\_regex_current_submatches_prop {#1}
19116     { \int_use:N \l\_\_regex_current_pos_int }
19117 }
```

(End definition for `__regex_action_submatch:n.`)

- `__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.

```
19118 \cs_new_protected:Npn \_\_regex_action_success:
19119 {
19120     \_\_regex_if_two_empty_matches:F
19121     {
19122         \bool_set_true:N \l\_\_regex_match_success_bool
19123         \bool_set_eq:NN \l\_\_regex_empty_success_bool
```

```

19124     \l__regex_fresh_thread_bool
19125     \int_set_eq:NN \l__regex_success_pos_int \l__regex_current_pos_int
19126     \prop_set_eq:NN \l__regex_success_submatches_prop
19127         \l__regex_current_submatches_prop
19128     \__prg_break:
19129 }
19130 }
```

(End definition for `__regex_action_success:..`)

36.6 Replacement

36.6.1 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 `}.`

```
19131 \int_new:N \l__regex_replacement_csnames_int
```

(End definition for `\l__regex_replacement_csnames_int`.)

`\l__regex_replacement_category_t1` This sequence of letters is used to correctly restore categories in nested constructions such as `\cL(abc\cD(_))d`.

```

19132 \tl_new:N \l__regex_replacement_category_t1
19133 \seq_new:N \l__regex_replacement_category_seq
```

(End definition for `\l__regex_replacement_category_t1` and `\l__regex_replacement_category_seq`.)

`\l__regex_balance_t1` This token list holds the replacement text for `__regex_replacement_balance_one_match:n` while it is being built incrementally.

```
19134 \tl_new:N \l__regex_balance_t1
```

(End definition for `\l__regex_balance_t1`.)

`__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).

```

19135 \cs_new:Npn \__regex_replacement_balance_one_match:n #1
19136     { - \__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, will produce the fully replaced token list. The initialization does not matter, but (as an example) we set it as for an empty replacement.

```
19137 \cs_new:Npn \_\_regex_replacement_do_one_match:n #1
19138   {
19139     \_\_regex_query_range:nn
19140       { \_\_intarray_item_fast:Nn \g_\_\_regex_submatch_prev_intarray {#1} }
19141       { \_\_intarray_item_fast:Nn \g_\_\_regex_submatch_begin_intarray {#1} }
19142   }
```

(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 ##.

```
19143 \cs_new:Npn \_\_regex_replacement_exp_not:N #1 { \exp_not:n {#1} }
```

(End definition for `__regex_replacement_exp_not:N`)

36.6.2 Query and brace balance

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 will result 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.

```
19144 \cs_new:Npn \_\_regex_query_range:nn #1#2
19145   {
19146     \exp_after:wN \_\_regex_query_range_loop:ww
19147     \_\_int_value:w \_\_int_eval:w #1 \exp_after:wN ;
19148     \_\_int_value:w \_\_int_eval:w #2 ;
19149     \_\_prg_break_point:
19150   }
19151 \cs_new:Npn \_\_regex_query_range_loop:ww #1 ; #2 ;
19152   {
19153     \if_int_compare:w #1 < #2 \exp_stop_f:
19154     \else:
19155       \exp_after:wN \_\_prg_break:
19156     \fi:
19157     \_\_regex_toks_use:w #1 \exp_stop_f:
19158     \exp_after:wN \_\_regex_query_range_loop:ww
19159       \_\_int_value:w \_\_int_eval:w #1 + 1 ; #2 ;
19160   }
```

(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).

```
19161 \cs_new:Npn \_regex_query_submatch:n #1
19162 {
19163     \_regex_query_range:nn
19164     { \_intarray_item_fast:Nn \g_regex_submatch_begin_intarray {#1} }
19165     { \_intarray_item_fast:Nn \g_regex_submatch_end_intarray {#1} }
19166 }
```

(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 *<max pos>* and *<min pos>*. These two positions are found in the corresponding “submatch” arrays.

```
19167 \cs_new_protected:Npn \_regex_submatch_balance:n #1
19168 {
19169     \_int_eval:w
19170     \int_compare:nNnTF
19171     { \_intarray_item_fast:Nn \g_regex_submatch_end_intarray {#1} } = 0
19172     { 0 }
19173     {
19174         \_intarray_item_fast:Nn \g_regex_balance_intarray
19175         { \_intarray_item_fast:Nn \g_regex_submatch_end_intarray {#1} }
19176     }
19177     -
19178     \int_compare:nNnTF
19179     { \_intarray_item_fast:Nn \g_regex_submatch_begin_intarray {#1} } = 0
19180     { 0 }
19181     {
19182         \_intarray_item_fast:Nn \g_regex_balance_intarray
19183         { \_intarray_item_fast:Nn \g_regex_submatch_begin_intarray {#1} }
19184     }
19185     \_int_eval_end:
19186 }
```

(End definition for `_regex_submatch_balance:n`.)

36.6.3 Framework

`_regex_replacement:n` The replacement text is built incrementally by abusing `\toks` within a group (see `\l3tl-build`). We keep track in `\l1_regex_balance_int` of the balance of explicit begin- and end-group tokens and `\l1_regex_balance_tl` will consist of 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.

```
19187 \cs_new_protected:Npn \_regex_replacement:n #1
19188 {
19189     \trace_push:nnn { regex } { 1 } { _regex_replacement:n }
19190     \l3tl_build:Nw \l1_regex_internal_a_tl
19191     \int_zero:N \l1_regex_balance_int
19192     \l3tl_clear:N \l1_regex_balance_tl
```

```

19193  \_\_regex\_escape\_use:nnnn
19194  {
19195      \if_charcode:w \c_right_brace_str ##1
19196          \_\_regex\_replacement_rbrace:N
19197      \else:
19198          \_\_regex\_replacement_normal:n
19199      \fi:
19200          ##1
19201  }
19202  { \_\_regex\_replacement\_escaped:N ##1 }
19203  { \_\_regex\_replacement\_normal:n ##1 }
19204  {##1}
19205  \prg_do_nothing: \prg_do_nothing:
19206  \if_int_compare:w \l\_\_regex\_replacement\_csnames\_int > 0 \exp_stop_f:
19207      \_\_msg_kernel_error:nnx { regex } { replacement-missing-rbrace }
19208          { \int_use:N \l\_\_regex\_replacement\_csnames\_int }
19209      \_\_tl_build_one:x
19210          { \prg_replicate:nn \l\_\_regex\_replacement\_csnames\_int \cs_end: }
19211  \fi:
19212  \seq_if_empty:NF \l\_\_regex\_replacement\_category\_seq
19213  {
19214      \_\_msg_kernel_error:nnx { regex } { replacement-missing-rparen }
19215          { \seq_count:N \l\_\_regex\_replacement\_category\_seq }
19216      \seq_clear:N \l\_\_regex\_replacement\_category\_seq
19217  }
19218  \cs_gset:Npx \_\_regex\_replacement\_balance\_one\_match:n ##1
19219  {
19220      + \int_use:N \l\_\_regex\_balance\_int
19221      \l\_\_regex\_balance_tl
19222      - \_\_regex\_submatch\_balance:n {##1}
19223  }
19224  \_\_tl_build_end:
19225  \exp_args:No \_\_regex\_replacement\_aux:n \l\_\_regex\_internal_a_tl
19226  (trace)    \trace_pop:nnn { regex } { 1 } { \_\_regex\_replacement:n }
19227  }
19228  \cs_new_protected:Npn \_\_regex\_replacement\_aux:n #1
19229  {
19230      \cs_set:Npn \_\_regex\_replacement\_do\_one\_match:n ##1
19231  {
19232      \_\_regex\_query\_range:nn
19233          { \_\_intarray_item_fast:Nn \g\_\_regex\_submatch\_prev\_intarray {##1} }
19234          { \_\_intarray_item_fast:Nn \g\_\_regex\_submatch\_begin\_intarray {##1} }
19235          #1
19236  }
19237  }

```

(End definition for __regex_replacement:n and __regex_replacement_aux:n.)

__regex_replacement_normal:n Most characters are simply sent to the output by __tl_build_one:n, 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.

```
19238  \cs_new_protected:Npn \_\_regex\_replacement\_normal:n #1
```

```

19239  {
19240    \tl_if_empty:NNTF \l__regex_replacement_category_tl
19241    { \__tl_build_one:n {#1} }
19242    { %
19243      \token_if_eq_charcode:NNTF #1 )
19244      {
19245        \seq_pop:NN \l__regex_replacement_category_seq
19246        \l__regex_replacement_category_tl
19247      }
19248      {
19249        \use:c { __regex_replacement_c_ \l__regex_replacement_category_tl :w }
19250        \__regex_replacement_normal:n {#1}
19251      }
19252    }
19253  }

```

(End definition for `__regex_replacement_normal:n`.)

`__regex_replacementEscaped: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. We use `\token_to_str:N` to give spaces the right category code.

```

19254  \cs_new_protected:Npn \__regex_replacementEscaped:N #1
19255  {
19256    \cs_if_exist_use:cF { __regex_replacement_#1:w }
19257    {
19258      \if_int_compare:w 1 < 1#1 \exp_stop_f:
19259      \__regex_replacement_put_submatch:n {#1}
19260      \else:
19261        \exp_args:No \__regex_replacement_normal:n
19262        { \token_to_str:N #1 }
19263      \fi:
19264    }
19265  }

```

(End definition for `__regex_replacementEscaped:N`.)

36.6.4 Submatches

`__regex_replacementPutSubmatch: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. Here, `#1` will receive a pointer to the 0-th submatch for a given match. We cannot use `\int_eval:n` because it is expandable, and would be expanded too early (short of adding `\exp_not:N`, making the code messy again).

```

19266  \cs_new_protected:Npn \__regex_replacement_put_submatch:n #1
19267  {
19268    \if_int_compare:w #1 < \l__regex_capturing_group_int
19269    \__tl_build_one:n { \__regex_query_submatch:n { #1 + ##1 } }
19270    \if_int_compare:w \l__regex_replacement_csnames_int = 0 \exp_stop_f:
19271      \tl_put_right:Nn \l__regex_balance_tl
19272      { + \__regex_submatch_balance:n { \__int_eval:w #1+##1 \__int_eval_end: } }
19273    \fi:
19274  \fi:
19275  }

```

(End definition for `_regex_replacement_put_submatch:n`.)

`_regex_replacement_g:w` 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.

```
19276 \cs_new_protected:Npn \_regex_replacement_g:w #1#2
19277 {
19278     \str_if_eq_x:mnTF { #1#2 } { \_regex_replacement_normal:n \c_left_brace_str }
19279     { \l_\_regex_internal_a_int = \_regex_replacement_g_digits:NN }
19280     { \_regex_replacement_error:NNN g #1 #2 }
19281 }
19282 \cs_new:Npn \_regex_replacement_g_digits:NN #1#2
19283 {
19284     \token_if_eq_meaning:NNTF #1 \_regex_replacement_normal:n
19285     {
19286         \if_int_compare:w 1 < 1#2 \exp_stop_f:
19287             #2
19288             \exp_after:wN \use_i:nnn
19289             \exp_after:wN \_regex_replacement_g_digits:NN
19290         \else:
19291             \exp_stop_f:
19292             \exp_after:wN \_regex_replacement_error:NNN
19293             \exp_after:wN g
19294         \fi:
19295     }
19296 {
19297     \exp_stop_f:
19298     \if_meaning:w \_regex_replacement_rbrace:N #1
19299         \exp_args:No \_regex_replacement_put_submatch:n
19300             { \int_use:N \l_\_regex_internal_a_int }
19301             \exp_after:wN \use_none:nn
19302         \else:
19303             \exp_after:wN \_regex_replacement_error:NNN
19304             \exp_after:wN g
19305         \fi:
19306     }
19307     #1 #2
19308 }
```

(End definition for `_regex_replacement_g:w` and `_regex_replacement_g_digits:NN`.)

36.6.5 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.

```
19309 \cs_new_protected:Npn \_regex_replacement_c:w #1#2
19310 {
19311     \token_if_eq_meaning:NNTF #1 \_regex_replacement_normal:n
19312     {
19313         \exp_after:wN \token_if_eq_charcode:NNTF \c_left_brace_str #2
19314         { \_regex_replacement_cu_aux:Nw \_regex_replacement_exp_not:N }
19315         {
19316             \cs_if_exist:cTF { \_regex_replacement_c:#2:w }
```

```

19317      { \__regex_replacement_cat:NNN #2 }
19318      { \__regex_replacement_error:NNN c #1#2 }
19319    }
19320  }
19321  { \__regex_replacement_error:NNN c #1#2 }
19322 }

```

(End definition for `__regex_replacement_c:w.`)

`__regex_replacement_cu_aux:Nw` Start a control sequence with `\cs:w`, which will be 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.

```

19323 \cs_new_protected:Npn \__regex_replacement_cu_aux:Nw #1
19324  {
19325    \if_case:w \l__regex_replacement_csnames_int
19326      \tl_build_one:n { \exp_not:n { \exp_after:wN #1 \cs:w } }
19327    \else:
19328      \tl_build_one:n { \exp_not:n { \exp_after:wN \tl_to_str:V \cs:w } }
19329    \fi:
19330    \int_incr:N \l__regex_replacement_csnames_int
19331  }

```

(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.

```

19332 \cs_new_protected:Npn \__regex_replacement_u:w #1#2
19333  {
19334    \str_if_eq_x:nnTF { #1#2 } { \__regex_replacement_normal:n \c_left_brace_str }
19335    { \__regex_replacement_cu_aux:Nw \exp_not:V }
19336    { \__regex_replacement_error:NNN u #1#2 }
19337  }

```

(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.

```

19338 \cs_new_protected:Npn \__regex_replacement_rbrace:N #1
19339  {
19340    \if_int_compare:w \l__regex_replacement_csnames_int > 0 \exp_stop_f:
19341      \tl_build_one:n \cs_end:
19342      \int_decr:N \l__regex_replacement_csnames_int
19343    \else:
19344      \__regex_replacement_normal:n {#1}
19345    \fi:
19346  }

```

(End definition for `__regex_replacement_rbrace:N.`)

36.6.6 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.

```

19347 \cs_new_protected:Npn \_\_regex_replacement_cat:NNN #1#2#3
19348 {
19349     \token_if_eq_meaning:NNTF \prg_do_nothing: #3
19350     { \_\_msg_kernel_error:nn { regex } { replacement-catcode-end } }
19351     {
19352         \int_compare:nNnTF { \l_\_regex_replacement_csnames_int } > 0
19353         {
19354             \_\_msg_kernel_error:nnnn
19355             { regex } { replacement-catcode-in-cs } {#1} {#3}
19356             #2 #3
19357         }
19358     {
19359         \str_if_eq:nnTF { #2 #3 } { \_\_regex_replacement_normal:n ( ) % }
19360         {
19361             \seq_push:NV \l_\_regex_replacement_category_seq
19362                 \l_\_regex_replacement_category_tl
19363                 \tl_set:Nn \l_\_regex_replacement_category_tl {#1}
19364             }
19365             { \use:c { __regex_replacement_c_#1:w } #2 #3 }
19366         }
19367     }
19368 }
```

(End definition for `__regex_replacement_cat:NNN`.)

We will 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.

```
19369 \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).

```

19370 \cs_new_protected:Npn \_\_regex_replacement_char:nNN #1#2#3
19371 {
19372     \tex_lccode:D 0 = '#3 \scan_stop:
19373     \tex_lowercase:D { \_\_tl_build_one:n {#1} }
19374 }
```

(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.

```
19375  \char_set_catcode_active:N \^@\n19376  \cs_new_protected:Npn \__regex_replacement_c_A:w\n19377    { \__regex_replacement_char:nNN { \exp_not:n { \exp_not:N ^@\ } } }\n\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.

```
19378 \char_set_catcode_group_begin:N `^`@  
19379 \cs_new_protected:Npn \__regex Replacement_c_B:w  
19380 {  
19381     \if_int_compare:w \l__regex Replacement_csnames_int = 0 \exp_stop_f:  
19382         \int_incr:N \l__regex_balance_int  
19383     \fi:  
19384     \__regex Replacement_char:nNN  
19385     { \exp_not:n { \exp_after:wN `^`@ \if_false: } \fi: } }  
19386 }
```

(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.

```
19387 \cs_new_protected:Npn \__regex Replacement_c:C #1#2  
19388 { \__tl_build_one:n { \exp_not:N \exp_not:N \exp_not:c {#2} } }
```

(End definition for __regex_replacement_c_C:w.)

Subscripts fit the mould: \lowercase the null byte with the correct category.

```
19389 \char_set_catcode_math_subscript:N ^^@  
19390 \cs_new_protected:Npn \__regex_replacement_c:D:w  
19391 { \__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.

```
19392 \char_set_catcode_group_end:N `^`@  
19393 \cs_new_protected:Npn \__regex Replacement_c_E:w  
19394 {  
19395     \if_int_compare:w \l__regex Replacement_csnames_int = 0 \exp_stop_f:  
19396         \int_decr:N \l__regex_balance_int  
19397     \fi:  
19398     \__regex Replacement_char:nNN  
19399     { \exp_not:n { \if_false: { \fi: `^`@ } } }  
19400 }
```

(End definition for `_regex_replacement_c_E:w`.)

__regex_replacement_c_L:w Simply \lowercase a letter null byte to produce an arbitrary letter.

```
19401 \char_set_catcode_letter:N `^`@  
19402 \cs_new_protected:Npn \_regex_replacement_c:L:w  
19403 { \_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.

```
19404 \char_set_catcode_math_toggle:N \^@\n19405 \cs_new_protected:Npn \_regex_replacement_c_M:w\n19406 { \_regex_replacement_char:nNN { ^@ } }
```

(End definition for `_regex_replacement_c_M:w`.)

`_regex_replacement_c_0:w` Lowercase an other null byte.

```
19407 \char_set_catcode_other:N \^@\n19408 \cs_new_protected:Npn \_regex_replacement_c_0:w\n19409 { \_regex_replacement_char:nNN { ^@ } }
```

(End definition for `_regex_replacement_c_0: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.

```
19410 \char_set_catcode_parameter:N \^@\n19411 \cs_new_protected:Npn \_regex_replacement_c_P:w\n19412 {\n19413 \_regex_replacement_char:nNN\n19414 { \exp_not:n { \exp_not:n { ^@^@^@^@^@ } } }\n19415 }
```

(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.

```
19416 \cs_new_protected:Npn \_regex_replacement_c_S:w #1#2\n19417 {\n19418 \if_int_compare:w '#2 = 0 \exp_stop_f:\n19419 \_msg_kernel_error:nn { regex } { replacement-null-space }\n19420 \fi:\n19421 \tex_lccode:D '\ = '#2 \scan_stop:\n19422 \tex_lowercase:D { \_tl_build_one:n {~} }\n19423 }
```

(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.

```
19424 \char_set_catcode_alignment:N \^@\n19425 \cs_new_protected:Npn \_regex_replacement_c_T:w\n19426 { \_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
^@\_.  

19427   \char_set_catcode_math_superscript:N ^@\_
19428   \cs_new_protected:Npn \_\_regex\_replacement\_c\_U:w
19429     { \_\_regex\_replacement\_char:nNN { ^@\_ } }
```

(End definition for __regex_replacement_c_U:w.)
 Restore the catcode of the null byte.

19430 \group_end:

36.6.7 An error

__regex_replacement_error:NNN Simple error reporting by calling one of the messages replacement-c, replacement-g, or replacement-u.

```
19431 \cs_new_protected:Npn \_\_regex\_replacement_error:NNN #1#2#3
19432   {
19433     \_\_msg_kernel_error:nnx { regex } { replacement-#1 } { #3}
19434     #2 #3
19435   }
```

(End definition for __regex_replacement_error:NNN.)

36.7 User functions

\regex_new:N Before being assigned a sensible value, a regex variable matches nothing.

```
19436 \cs_new_protected:Npn \regex_new:N #1
19437   { \cs_new_eq:NN #1 \c__regex_no_match_regex }
```

(End definition for \regex_new:N. This function is documented on page 203.)

\regex_set:Nn Compile, then store the result in the user variable with the appropriate assignment function.
\regex_gset:Nn

```
19438 \cs_new_protected:Npn \regex_set:Nn #1#2
19439   {
19440     \_\_regex_compile:n {#2}
19441     \tl_set_eq:NN #1 \l__regex_internal_regex
19442   }
19443 \cs_new_protected:Npn \regex_gset:Nn #1#2
19444   {
19445     \_\_regex_compile:n {#2}
19446     \tl_gset_eq:NN #1 \l__regex_internal_regex
19447   }
19448 \cs_new_protected:Npn \regex_const:Nn #1#2
19449   {
19450     \_\_regex_compile:n {#2}
19451     \tl_const:Nx #1 { \exp_not:o \l__regex_internal_regex }
```

(End definition for \regex_set:Nn, \regex_gset:Nn, and \regex_const:Nn. These functions are documented on page 203.)

`\regex_show:N` User functions: the `n` variant requires compilation first. Then show the variable with some appropriate text. The auxiliary `_regex_show:Nx` is defined in a different section.

```
19453 \cs_new_protected:Npn \regex_show:n #1
19454 {
19455     __regex_compile:n {#1}
19456     __regex_show:Nn \l__regex_internal_regex
19457     { { \tl_to_str:n {#1} } }
19458 }
19459 \cs_new_protected:Npn \regex_show:N #1
19460 { __regex_show:Nn #1 { variable~\token_to_str:N #1 } }
```

(End definition for `\regex_show:N` and `\regex_show:n`. These functions are documented on page [203](#).)

`\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`.

```
19461 \prg_new_protected_conditional:Npnn \regex_match:nn #1#2 { T , F , TF }
19462 {
19463     \__regex_if_match:nn { \__regex_build:n {#1} } {#2}
19464     \__regex_return:
19465 }
19466 \prg_new_protected_conditional:Npnn \regex_match:Nn #1#2 { T , F , TF }
19467 {
19468     \__regex_if_match:nn { \__regex_build:N #1 } {#2}
19469     \__regex_return:
19470 }
```

(End definition for `\regex_match:nnTF` and `\regex_match:NnTF`. These functions are documented on page [203](#).)

`\regex_count:nnN` Again, use an auxiliary whose first argument builds the NFA.

```
\regex_count:NnN 19471 \cs_new_protected:Npn \regex_count:nnN #1
                  { \__regex_count:nnN { \__regex_build:n {#1} } }
19472 \cs_new_protected:Npn \regex_count:NnN #1
19473 { \__regex_count:nnN { \__regex_build:N {#1} } }
```

(End definition for `\regex_count:nnN` and `\regex_count:NnN`. These functions are documented on page 204.)

`\regex_extract_once: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.

```
19475 \cs_set_protected:Npn \__regex_tmp:w #1#2#3
19476 {
19477     \cs_new_protected:Npn #2 ##1 { #1 { \__regex_build:n {##1} } }
19478     \cs_new_protected:Npn #3 ##1 { #1 { \__regex_build:N ##1 } }
19479     \prg_new_protected_conditional:Npnn #2 ##1##2##3 { T , F , TF }
19480         { #1 { \__regex_build:n {##1} } ##2 } ##3 \__regex_return: }
19481     \prg_new_protected_conditional:Npnn #3 ##1##2##3 { T , F , TF }
19482         { #1 { \__regex_build:N ##1 } ##2 } ##3 \__regex_return: }
19483 }
```

```

19484 \_\_regex\_tmp:w \_\_regex\_extract\_once:nnN
19485   \_\_regex\_extract\_once:nnN \_\_regex\_extract\_once:NnN
19486 \_\_regex\_tmp:w \_\_regex\_extract\_all:nnN
19487   \_\_regex\_extract\_all:nnN \_\_regex\_extract\_all:NnN
19488 \_\_regex\_tmp:w \_\_regex\_replace\_once:nnN
19489   \_\_regex\_replace\_once:nnN \_\_regex\_replace\_once:NnN
19490 \_\_regex\_tmp:w \_\_regex\_replace\_all:nnN
19491   \_\_regex\_replace\_all:nnN \_\_regex\_replace\_all:NnN
19492 \_\_regex\_tmp:w \_\_regex\_split:nnN \_\_regex\_split:nnN \_\_regex\_split:NnN

```

(End definition for `__regex_extract_once:nnN` and others. These functions are documented on page ??.)

36.7.1 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.

```
19493 \int_new:N \l\_\_regex\_match\_count\_int
```

(End definition for `\l__regex_match_count_int`.)

`__regex_begin` Those flags are raised to indicate extra begin-group or end-group tokens when extracting submatches.
`__regex_end`

```

19494 \flag_new:n { __regex_begin }
19495 \flag_new:n { __regex_end }
```

(End definition for `__regex_begin` and `__regex_end`.)

`\l__regex_min_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.
`\l__regex_submatch_int`
`\l__regex_zeroth_submatch_int`

```

19496 \int_new:N \l\_\_regex\_min\_submatch\_int
19497 \int_new:N \l\_\_regex\_submatch\_int
19498 \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` Hold the place where the match attempt begun and the end-points of each submatch.

```

19499 \__intarray_new:Nn \g\_\_regex\_submatch\_prev\_intarray { 65536 }
19500 \__intarray_new:Nn \g\_\_regex\_submatch\_begin\_intarray { 65536 }
19501 \__intarray_new:Nn \g\_\_regex\_submatch\_end\_intarray { 65536 }
```

(End definition for `\g__regex_submatch_prev_intarray`, `\g__regex_submatch_begin_intarray`, and `\g__regex_submatch_end_intarray`.)

`__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.

```

19502 \cs_new_protected:Npn \_\_regex\_return:
19503 {
19504   \if_meaning:w \c_true_bool \g\_\_regex\_success\_bool
```

```

19505      \prg_return_true:
19506      \else:
19507          \prg_return_false:
19508      \fi:
19509  }

```

(End definition for `_regex_return::`)

36.7.2 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.

```

19510  \cs_new_protected:Npn \_regex_if_match:nn #1#2
19511  {
19512      \group_begin:
19513          \_regex_disable_submatches:
19514          \_regex_single_match:
19515              #1
19516          \_regex_match:n {#2}
19517      \group_end:
19518  }

```

(End definition for `_regex_if_match: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.

```

19519  \cs_new_protected:Npn \_regex_count:nnN #1#2#3
19520  {
19521      \group_begin:
19522          \_regex_disable_submatches:
19523          \int_zero:N \l_regex_match_count_int
19524          \_regex_multi_match:n { \int_incr:N \l_regex_match_count_int }
19525              #1
19526          \_regex_match:n {#2}
19527          \exp_args:NNNo
19528      \group_end:
19529          \int_set:Nn #3 { \int_use:N \l_regex_match_count_int }
19530  }

```

(End definition for `_regex_count:nnN`.)

36.7.3 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.

```

19531  \cs_new_protected:Npn \_regex_extract_once:nnN #1#2#3
19532  {
19533      \group_begin:
19534          \_regex_single_match:
19535              #1
19536          \_regex_match:n {#2}

```

```

19537      \_\_regex\_extract:
19538      \_\_regex_group_end_extract_seq:N #3
19539  }
19540 \cs_new_protected:Npn \_\_regex_extract_all:nnN #1#2#3
19541  {
19542      \group_begin:
19543      \_\_regex_multi_match:n { \_\_regex_extract: }
19544      #1
19545      \_\_regex_match:n {#2}
19546      \_\_regex_group_end_extract_seq:N #3
19547  }

```

(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.

```

19548 \cs_new_protected:Npn \_\_regex_split:nnN #1#2#3
19549  {
19550      \group_begin:
19551      \_\_regex_multi_match:
19552  {
19553      \if_int_compare:w \l\_\_regex_start_pos_int < \l\_\_regex_success_pos_int
19554          \_\_regex_extract:
19555          \_\_intarray_gset_fast:Nnn \g\_\_regex_submatch_prev_intarray
19556          { \l\_\_regex_zeroth_submatch_int } { 0 }
19557          \_\_intarray_gset_fast:Nnn \g\_\_regex_submatch_end_intarray
19558          { \l\_\_regex_zeroth_submatch_int }
19559          {
19560              \_\_intarray_item_fast:Nn \g\_\_regex_submatch_begin_intarray
19561              { \l\_\_regex_zeroth_submatch_int }
19562          }
19563          \_\_intarray_gset_fast:Nnn \g\_\_regex_submatch_begin_intarray
19564          { \l\_\_regex_zeroth_submatch_int }
19565          { \l\_\_regex_start_pos_int }
19566      \fi:
19567  }
19568  #1
19569  \_\_regex_match:n {#2}
19570  \assert{\assert_int:n { \l\_\_regex_current_pos_int = \l\_\_regex_max_pos_int }}
19571  \_\_intarray_gset_fast:Nnn \g\_\_regex_submatch_prev_intarray
19572  { \l\_\_regex_submatch_int } { 0 }
19573  \_\_intarray_gset_fast:Nnn \g\_\_regex_submatch_end_intarray
19574  { \l\_\_regex_submatch_int }
19575  { \l\_\_regex_max_pos_int }
19576  \_\_intarray_gset_fast:Nnn \g\_\_regex_submatch_begin_intarray
19577  { \l\_\_regex_submatch_int }
19578  { \l\_\_regex_start_pos_int }
19579  \int_incr:N \l\_\_regex_submatch_int
19580  \if_meaning:w \c_true_bool \l\_\_regex_empty_success_bool

```

```

19581     \if_int_compare:w \l__regex_start_pos_int = \l__regex_max_pos_int
19582         \int_decr:N \l__regex_submatch_int
19583     \fi:
19584     \fi:
19585     \__regex_group_end_extract_seq:N #3
19586 }

```

(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 `\l__regex_internal_a_tl`. We detect unbalanced results using the two flags `\@_begin` and `\@_end`, raised whenever we see too many begin-group or end-group tokens in a submatch. We disable `__seq_item:n` to prevent two x-expansions.

```

19587 \cs_new_protected:Npn \__regex_group_end_extract_seq:N #1
19588 {
19589     \cs_set_eq:NN \__seq_item:n \scan_stop:
19590     \flag_clear:n { __regex_begin }
19591     \flag_clear:n { __regex_end }
19592     \tl_set:Nx \l__regex_internal_a_tl
19593     {
19594         \s__seq
19595         \int_step_function:nnN
19596             { \l__regex_min_submatch_int }
19597             { 1 }
19598             { \l__regex_submatch_int - 1 }
19599             \__regex_extract_seq_aux:n
19600     }
19601     \int_compare:nNnF
19602         { \flag_height:n { __regex_begin } + \flag_height:n { __regex_end } }
19603         = 0
19604     {
19605         \__msg_kernel_error:nnxxx { regex } { result-unbalanced }
19606         { splitting-or-extracting-submatches }
19607         { \flag_height:n { __regex_end } }
19608         { \flag_height:n { __regex_begin } }
19609     }
19610     \use:x
19611     {
19612         \group_end:
19613         \tl_set:Nn \exp_not:N #1 { \l__regex_internal_a_tl }
19614     }
19615 }

```

(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.

```

19616 \cs_new:Npn \__regex_extract_seq_aux:n #1
19617 {
19618     \__seq_item:n
19619     {
19620         \exp_after:wN \__regex_extract_seq_aux:ww

```

```

19621         \_\_int_value:w \_\_regex_submatch_balance:n {#1} ; #1;
19622     }
19623   }
19624 \cs_new:Npn \_\_regex_extract_seq_aux:ww #1; #2;
19625   {
19626     \if_int_compare:w #1 < 0 \exp_stop_f:
19627       \flag_raise:n { __regex_end }
19628       \prg_replicate:nn {#1} { \exp_not:n { { \if_false: } \fi: } }
19629     \fi:
19630     \_\_regex_query_submatch:n {#2}
19631     \if_int_compare:w #1 > 0 \exp_stop_f:
19632       \flag_raise:n { __regex_begin }
19633       \prg_replicate:nn {#1} { \exp_not:n { \if_false: { \fi: } } }
19634     \fi:
19635   }

```

(End definition for `__regex_extract_seq_aux:n` and `__regex_extract_seq_aux:ww`.)

```

\_\_regex_extract:
\_\_regex_extract_b:wn
\_\_regex_extract_e:wn

```

Our task here is to extract from the property list `\l___regex_success_submatches_prop` the list of end-points of submatches, and store them in appropriate array entries, from `\l___regex_zeroth_submatch_int` upwards. We begin by emptying those entries. Then for each `<key>`-`<value>` pair in the property list update the appropriate entry. This is somewhat a hack: the `<key>` is a non-negative integer followed by `<` or `>`, which we use in a comparison to `-1`. At the end, store the information about the position at which the match attempt started, in `\g___regex_submatch_prev_intarray`.

```

19636 \cs_new_protected:Npn \_\_regex_extract:
19637   {
19638     \if_meaning:w \c_true_bool \g_\_\_regex\_success\_bool
19639       \int_set_eq:NN \l_\_\_regex_zeroth_submatch_int \l_\_\_regex_submatch_int
19640       \prg_replicate:nn \l_\_\_regex_capturing_group_int
19641         {
19642           \_\_intarray_gset_fast:Nnn \g_\_\_regex_submatch_begin_intarray
19643             { \l_\_\_regex_submatch_int } { 0 }
19644           \_\_intarray_gset_fast:Nnn \g_\_\_regex_submatch_end_intarray
19645             { \l_\_\_regex_submatch_int } { 0 }
19646           \_\_intarray_gset_fast:Nnn \g_\_\_regex_submatch_prev_intarray
19647             { \l_\_\_regex_submatch_int } { 0 }
19648           \int_incr:N \l_\_\_regex_submatch_int
19649         }
19650       \prop_map_inline:Nn \l_\_\_regex_success_submatches_prop
19651         {
19652           \if_int_compare:w ##1 - 1 \exp_stop_f:
19653             \exp_after:wN \_\_regex_extract_e:wn \_\_int_value:w
19654           \else:
19655             \exp_after:wN \_\_regex_extract_b:wn \_\_int_value:w
19656           \fi:
19657             \_\_int_eval:w \l_\_\_regex_zeroth_submatch_int + ##1 {##2}
19658         }
19659       \_\_intarray_gset_fast:Nnn \g_\_\_regex_submatch_prev_intarray
19660         { \l_\_\_regex_zeroth_submatch_int } { \l_\_\_regex_start_pos_int }
19661       \fi:
19662     }
19663   \cs_new_protected:Npn \_\_regex_extract_b:wn #1 < #2
19664     { \_\_intarray_gset_fast:Nnn \g_\_\_regex_submatch_begin_intarray {#1} {#2} }

```

```

19665 \cs_new_protected:Npn \__regex_extract_e:wn #1 > #2
19666   { \__intarray_gset_fast:Nnn \g__regex_submatch_end_intarray {#1} {#2} }

```

(End definition for `__regex_extract:`, `__regex_extract_b:wn`, and `__regex_extract_e:wn`.)

36.7.4 Replacement

`__regex_replace_once: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.

```

19667 \cs_new_protected:Npn \__regex_replace_once:nnN #1#2#3
19668   {
19669     \group_begin:
19670       \__regex_single_match:
19671         #1
19672         \__regex_replacement:n {#2}
19673         \exp_args:No \__regex_match:n { #3 }
19674         \if_meaning:w \c_false_bool \g__regex_success_bool
19675           \group_end:
19676         \else:
19677           \__regex_extract:
19678             \int_set:Nn \l__regex_balance_int
19679               {
19680                 \__regex_replacement_balance_one_match:n
19681                   { \l__regex_zeroth_submatch_int }
19682               }
19683             \tl_set:Nx \l__regex_internal_a_tl
19684               {
19685                 \__regex_replacement_do_one_match:n { \l__regex_zeroth_submatch_int }
19686                 \__regex_query_range:nn
19687                   {
19688                     \__intarray_item_fast:Nn \g__regex_submatch_end_intarray
19689                       { \l__regex_zeroth_submatch_int }
19690                   }
19691                   { \l__regex_max_pos_int }
19692               }
19693             \__regex_group_end_replace:N #3
19694           \fi:
19695     }

```

(End definition for `__regex_replace_once: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.

```

19696 \cs_new_protected:Npn \__regex_replace_all:nnN #1#2#3
19697 {
19698     \group_begin:
19699         \__regex_multi_match:n { \__regex_extract: } #1
19700         \__regex_replacement:n {#2}
19701         \exp_args:No \__regex_match:n {#3}
19702         \int_set:Nn \l__regex_balance_int
19703             {
19704                 0
19705                 \int_step_function:nnN
19706                     { \l__regex_min_submatch_int }
19707                     \l__regex_capturing_group_int
19708                     { \l__regex_submatch_int - 1 }
19709                     \__regex_replacement_balance_one_match:n
19710             }
19711         }
19712         \tl_set:Nx \l__regex_internal_a_tl
19713             {
19714                 \int_step_function:nnN
19715                     { \l__regex_min_submatch_int }
19716                     \l__regex_capturing_group_int
19717                     { \l__regex_submatch_int - 1 }
19718                     \__regex_replacement_do_one_match:n
19719                     \__regex_query_range:nn
19720                         \l__regex_start_pos_int \l__regex_max_pos_int
19721             }
19722         \__regex_group_end_replace:N #3
19723     }

```

(End definition for `__regex_replace_all:nnN`.)

`__regex_group_end_replace:N` If the brace balance is not 0, raise an error. Then set the user's variable `#1` to the x-expansion of `\l__regex_internal_a_tl`, adding the appropriate braces to produce a balanced result. And end the group.

```

19724 \cs_new_protected:Npn \__regex_group_end_replace:N #1
19725 {
19726     \if_int_compare:w \l__regex_balance_int = 0 \exp_stop_f:
19727     \else:
19728         \__msg_kernel_error:nxxxx { regex } { result-unbalanced }
19729             { replacing }
19730             { \int_max:nn { - \l__regex_balance_int } { 0 } }
19731             { \int_max:nn { \l__regex_balance_int } { 0 } }
19732     \fi:
19733     \use:x
19734         {
19735             \group_end:
19736             \tl_set:Nn \exp_not:N #1
19737             {
19738                 \if_int_compare:w \l__regex_balance_int < 0 \exp_stop_f:
19739                     \prg_replicate:nn { - \l__regex_balance_int }
19740                         { { \if_false: } \fi: }
19741             \fi:

```

```

19742           \l__regex_internal_a_t1
19743           \if_int_compare:w \l__regex_balance_int > 0 \exp_stop_f:
19744             \prg_replicate:nn { \l__regex_balance_int }
19745               { \if_false: { \fi: } }
19746             \fi:
19747           }
19748         }
19749     }

```

(End definition for `_regex_group_end_replace:N`.)

36.7.5 Storing and showing compiled patterns

36.8 Messages

Messages for the preparsing phase.

```

19750 \__msg_kernel_new:nnnn { regex } { trailing-backslash }
19751   { Trailing~escape~character~'\iow_char:N\\'. }
19752   {
19753     A~regular~expression~or~its~replacement~text~ends~with~
19754     the~escape~character~'\iow_char:N\\'.~It~will~be~ignored.
19755   }
19756 \__msg_kernel_new:nnnn { regex } { x-missing-rbrace }
19757   { Missing~closing~brace~in~'\iow_char:N\\x'~hexadecimal~sequence. }
19758   {
19759     You~wrote~something~like~
19760     '\iow_char:N\\x\{\...#1'.~
19761     The~closing~brace~is~missing.
19762   }
19763 \__msg_kernel_new:nnnn { regex } { x-overflow }
19764   { Character~code~'#1'~too~large~in~'\iow_char:N\\x'~hexadecimal~sequence. }
19765   {
19766     You~wrote~something~like~
19767     '\iow_char:N\\x\{\int_to_Hex:n{#1}\}'.~
19768     The~character~code~#1~is~larger~than~
19769     the~maximum~value~\int_use:N \c_max_char_int.
19770   }

```

Invalid quantifier.

```

19771 \__msg_kernel_new:nnnn { regex } { invalid-quantifier }
19772   { Braced~quantifier~'#1'~may~not~be~followed~by~'#2'. }
19773   {
19774     The~character~'#2'~is~invalid~in~the~braced~quantifier~'#1'.~
19775     The~only~valid~quantifiers~are~'*',~'?',~'+',~'{<int>}',~
19776     '{<min>},~and~'{<min>,<max>}',~optionally~followed~by~'?'.
19777   }

```

Messages for missing or extra closing brackets and parentheses, with some fancy singular/plural handling for the case of parentheses.

```

19778 \__msg_kernel_new:nnnn { regex } { missing-rbrack }
19779   { Missing~right~bracket~inserted~in~regular~expression. }
19780   {
19781     LaTeX~was~given~a~regular~expression~where~a~character~class~
19782     was~started~with~'[',~but~the~matching~']'~is~missing.
19783   }

```

```

19784 \__msg_kernel_new:nnnn { regex } { missing-rparen }
19785 {
19786     Missing-right-
19787     \int_compare:nTF { #1 = 1 } { parenthesis } { parentheses } ~
19788     inserted-in-regular-expression.
19789 }
19790 {
19791     LaTeX-was-given-a-regular-expression-with-\int_eval:n {#1} ~
19792     more-left-parentheses-than-right-parentheses.
19793 }
19794 \__msg_kernel_new:nnnn { regex } { extra-rparen }
19795 {
19796     Extra-right-parenthesis-ignored-in-regular-expression. }
19797 {
19798     LaTeX-came-across-a-closing-parenthesis-when-no-submatch-group-
19799     was-open.-The-parenthesis-will-be-ignored.
19800 }
```

Some escaped alphanumerics are not allowed everywhere.

```

19800 \__msg_kernel_new:nnnn { regex } { bad-escape }
19801 {
19802     Invalid-escape-'\\iow_char:N\\#1'-
19803     \__regex_if_in_cs:TF { within-a-control-sequence. }
19804 {
19805     \__regex_if_in_class:TF
19806         { in-a-character-class. }
19807         { following-a-category-test. }
19808 }
19809 }
19810 {
19811     The-escape-sequence-'\\iow_char:N\\#1'-may-not-appear-
19812     \__regex_if_in_cs:TF
19813 {
19814     within-a-control-sequence-test-introduced-by-
19815     '\\iow_char:N\\c\\iow_char:N\\{'.
19816 }
19817 {
19818     \__regex_if_in_class:TF
19819         { within-a-character-class~ }
19820         { following-a-category-test-such-as-'\\iow_char:N\\cL'~ }
19821         because-it-does-not-match-exactly-one-character.
19822 }
19823 }
```

Range errors.

```

19824 \__msg_kernel_new:nnnn { regex } { range-missing-end }
19825 {
19826     Invalid-end-point-for-range-'#1-#2'-in-character-class. }
19827 {
19828     The-end-point-'#2'-of-the-range-'#1-#2'-may-not-serve-as-an-
19829     end-point-for-a-range:-alphanumeric-characters-should-not-be-
19830     escaped,-and-non-alphanumeric-characters-should-be-escaped.
19831 }
19832 \__msg_kernel_new:nnnn { regex } { range-backwards }
19833 {
19834     In-ranges-of-characters-'[x-y]'-appearing-in-character-classes,-
```

```

19835     the~first~character~code~must~not~be~larger~than~the~second.~
19836     Here,~'#1'~has~character~code~\int_eval:n {'#1},~while~
19837     '#2'~has~character~code~\int_eval:n {'#2}.
19838 }

Errors related to \c and \u.
19839 \__msg_kernel_new:nnnn { regex } { c-bad-mode }
19840   { Invalid~nested~'\iow_char:N\\c'~escape~in~regular~expression. }
19841   {
19842     The~'\iow_char:N\\c'~escape~cannot~be~used~within~
19843     a~control~sequence~test~'\iow_char:N\\c{...}'.~
19844     To~combine~several~category~tests,~use~'\iow_char:N\\c[...]'.
19845   }
19846 \__msg_kernel_new:nnnn { regex } { c-missing-rbrace }
19847   { Missing~right~brace~inserted~for~'\iow_char:N\\c'~escape. }
19848   {
19849     LaTeX~was~given~a~regular~expression~where~a~
19850     '\iow_char:N\\c\iow_char:N\\c{...}'~construction~was~not~ended~
19851     with~a~closing~brace~'\iow_char:N\\c}'.
19852   }
19853 \__msg_kernel_new:nnnn { regex } { c-missing-rbrack }
19854   { Missing~right~bracket~inserted~for~'\iow_char:N\\c'~escape. }
19855   {
19856     A~construction~'\iow_char:N\\c[...]'~appears~in~a~
19857     regular~expression,~but~the~closing~']'~is~not~present.
19858   }
19859 \__msg_kernel_new:nnnn { regex } { c-missing-category }
19860   { Invalid~character~'#1'~following~'\iow_char:N\\c'~escape. }
19861   {
19862     In~regular~expressions,~the~'\iow_char:N\\c'~escape~sequence~
19863     may~only~be~followed~by~a~left~brace,~a~left~bracket,~or~a~
19864     capital~letter~representing~a~character~category,~namely~
19865     one~of~'ABCDELMOPSTU'.
19866   }
19867 \__msg_kernel_new:nnnn { regex } { c-trailing }
19868   { Trailing~category~code~escape~'\iow_char:N\\c'... }
19869   {
19870     A~regular~expression~ends~with~'\iow_char:N\\c'~followed~
19871     by~a~letter.~It~will~be~ignored.
19872   }
19873 \__msg_kernel_new:nnnn { regex } { u-missing-lbrace }
19874   { Missing~left~brace~following~'\iow_char:N\\u'~escape. }
19875   {
19876     The~'\iow_char:N\\u'~escape~sequence~must~be~followed~by~
19877     a~brace~group~with~the~name~of~the~variable~to~use.
19878   }
19879 \__msg_kernel_new:nnnn { regex } { u-missing-rbrace }
19880   { Missing~right~brace~inserted~for~'\iow_char:N\\u'~escape. }
19881   {
19882     LaTeX~
19883     \str_if_eq_x:mnTF { } {#2}
19884       { reached~the~end~of~the~string~}
19885       { encountered~an~escaped~alphanumeric~character~'\iow_char:N\\#2'~}
19886       when~parsing~the~argument~of~an~'\iow_char:N\\u\iow_char:N\\{...\\}'~escape.
19887   }

```

Errors when encountering the POSIX syntax [:....].

```
19888 \__msg_kernel_new:nnnn { regex } { posix-unsupported }
19889   { POSIX~collating~element~'[#1 ~ #1]'~not~supported. }
19890   {
19891     The~' [.foo.]'~and~' [=bar=]'~syntaxes~have~a~special~meaning~
19892     in~POSIX~regular~expressions.~This~is~not~supported~by~LaTeX.~
19893     Maybe~you~forgot~to~escape~a~left~bracket~in~a~character~class?
19894   }
19895 \__msg_kernel_new:nnnn { regex } { posix-unknown }
19896   { POSIX~class~'[:#1:]'~unknown. }
19897   {
19898     '[:#1:]'~is~not~among~the~known~POSIX~classes~
19899     '[:alnum:]',~'[:alpha:]',~'[:ascii:]',~'[:blank:]',~
19900     '[:cntrl:]',~'[:digit:]',~'[:graph:]',~'[:lower:]',~
19901     '[:print:]',~'[:punct:]',~'[:space:]',~'[:upper:]',~
19902     '[:word:]',~and~'[:xdigit:]'.
19903   }
19904 \__msg_kernel_new:nnnn { regex } { posix-missing-close }
19905   { Missing~closing~':}'~for~POSIX~class. }
19906   { 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.

```
19907 \__msg_kernel_new:nnnn { regex } { result-unbalanced }
19908   { Missing~brace~inserted~when~'#1. }
19909   {
19910     LaTeX~was~asked~to~do~some~regular~expression~operation,~
19911     and~the~resulting~token~list~would~not~have~the~same~number~
19912     of~begin~group~and~end~group~tokens.~Braces~were~inserted:~
19913     '#2~left,~#3~right.
19914 }
```

Error message for unknown options.

```
19915 \__msg_kernel_new:nnnn { regex } { unknown-option }
19916   { Unknown~option~'#1'~for~regular~expressions. }
19917   {
19918     The~only~available~option~is~'case-insensitive',~toggled~by~
19919     '(?i)'~and~'(?-i)'.
19920   }
19921 \__msg_kernel_new:nnnn { regex } { special-group-unknown }
19922   { Unknown~special~group~'#1~...~'~in~a~regular~expression. }
19923   {
19924     The~only~valid~constructions~starting~with~'(?'~are~
19925     '(?:~...~)',~'(?|~...~)',~'(?i)',~and~'(?-i)'.
19926 }
```

Errors in the replacement text.

```
19927 \__msg_kernel_new:nnnn { regex } { replacement-c }
19928   { Misused~'\iow_char:N\\c'~command~in~a~replacement~text. }
19929   {
19930     In~a~replacement~text,~the~'\iow_char:N\\c'~escape~sequence~
19931     can~be~followed~by~one~of~the~letters~'ABCDELMOPSTU'~
19932     or~a~brace~group,~not~by~'#1'.
19933 }
```

```

19934 \__msg_kernel_new:nnnn { regex } { replacement-u }
19935   { Misused~'\iow_char:N\\u'~command~in~a~replacement~text. }
19936   {
19937     In~a~replacement~text,~the~'\iow_char:N\\u'~escape~sequence~
19938     must~be~followed~by~a~brace~group~holding~the~name~of~the~
19939     variable~to~use.
19940   }
19941 \__msg_kernel_new:nnnn { regex } { replacement-g }
19942   {
19943     Missing~brace~for~the~'\iow_char:N\\g'~construction~
19944     in~a~replacement~text.
19945   }
19946   {
19947     In~the~replacement~text~for~a~regular~expression~search,~
19948     submatches~are~represented~either~as~'\iow_char:N \\g{dd..d}',~
19949     or~'\d',~where~'d'~are~single~digits.~Here,~a~brace~is~missing.
19950   }
19951 \__msg_kernel_new:nnnn { regex } { replacement-catcode-end }
19952   {
19953     Missing~character~for~the~'\iow_char:N\\c<category><character>'~
19954     construction~in~a~replacement~text.
19955   }
19956   {
19957     In~a~replacement~text,~the~'\iow_char:N\\c'~escape~sequence~
19958     can~be~followed~by~one~of~the~letters~'ABCDELMOPTSU'~representing~
19959     the~character~category.~Then,~a~character~must~follow.~LaTeX~
19960     reached~the~end~of~the~replacement~when~looking~for~that.
19961   }
19962 \__msg_kernel_new:nnnn { regex } { replacement-catcode-in-cs }
19963   {
19964     Category~code~'\iow_char:N\\c#1#3'~ignored~inside~
19965     '\iow_char:N\\c\{...\}'~in~a~replacement~text.
19966   }
19967   {
19968     In~a~replacement~text,~the~category~codes~of~the~argument~of~
19969     '\iow_char:N\\c\{...\}'~are~ignored~when~building~the~control~
19970     sequence~name.
19971   }
19972 \__msg_kernel_new:nnnn { regex } { replacement-null-space }
19973   { TeX~cannot~build~a~space~token~with~character~code~0. }
19974   {
19975     You~asked~for~a~character~token~with~category~space,~
19976     and~character~code~0,~for~instance~through~
19977     '\iow_char:N\\cS\iow_char:N\\x00'.~
19978     This~specific~case~is~impossible~and~will~be~replaced~
19979     by~a~normal~space.
19980   }
19981 \__msg_kernel_new:nnnn { regex } { replacement-missing-rbrace }
19982   { Missing~right~brace~inserted~in~replacement~text. }
19983   {
19984     There~\int_compare:nTF { #1 = 1 } { was } { were } ~#1~
19985     missing~right~\int_compare:nTF { #1 = 1 } { brace } { braces } .
19986   }
19987 \__msg_kernel_new:nnnn { regex } { replacement-missing-rparen }

```

```

19988 { Missing-right-parenthesis-inserted-in-replacement-text. }
19989 {
19990   There~ \int_compare:nTF { #1 = 1 } { was } { were } ~ #1~
19991   missing-right-\int_compare:nTF { #1 = 1 } { parenthesis } { parentheses } .
19992 }
```

__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 lazyness.

```

19993 \cs_new:Npn \_\_regex\_msg\_repeated:nnN #1#2#3
19994 {
19995   \str_if_eq_x:nnF { #1 #2 } { 1 0 }
19996   {
19997     , ~ repeated ~
19998     \int_case:nnF {#2}
19999     {
20000       { -1 } { #1-or-more-times,~\bool_if:NTF #3 { lazy } { greedy } }
20001       { 0 } { #1-times }
20002     }
20003     {
20004       between~#1~and~\int_eval:n {#1+#2}-times,~
20005       \bool_if:NTF #3 { lazy } { greedy }
20006     }
20007   }
20008 }
```

(End definition for __regex_msg_repeated:nnN.)

36.9 Code for tracing

The tracing code is still very experimental, and is meant to be used with the l3trace package, currently in l3trial.

__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).

```

20009 (*trace)
2010 \cs_new_protected:Npn \_\_regex_trace_states:n #1
2011 {
2012   \int_step_inline:nnnn
2013     \l\_\_regex_min_state_int
2014     { 1 }
2015     { \l\_\_regex_max_state_int - 1 }
2016     {
2017       \trace:nnx { regex } { #1 }
2018       { \iow_char:N \toks ##1 = { \_\_regex_toks_use:w ##1 } }
2019     }
2020   }
2021 }
```

(End definition for __regex_trace_states:n.)

2022 ⟨/initex | package⟩

37 I3box implementation

```
20023 <*initex | package>
20024 <@@=box>
```

The code in this module is very straight forward so I'm not going to comment it very extensively.

37.1 Creating and initialising boxes

The following test files are used for this code: m3box001.lvt.

\box_new:N Defining a new $\langle box \rangle$ register: remember that box 255 is not generally available.

```
20025 <*package>
20026 \cs_new_protected:Npn \box_new:N #1
20027 {
20028     \__chk_if_free_cs:N #1
20029     \cs:w newbox \cs_end: #1
20030 }
20031 </package>
20032 \cs_generate_variant:Nn \box_new:N { c }
```

Clear a $\langle box \rangle$ register.

```
20033 \cs_new_protected:Npn \box_clear:N #1
20034 { \box_set_eq:NN #1 \c_empty_box }
20035 \cs_new_protected:Npn \box_gclear:N #1
20036 { \box_gset_eq:NN #1 \c_empty_box }
20037 \cs_generate_variant:Nn \box_clear:N { c }
20038 \cs_generate_variant:Nn \box_gclear:N { c }
```

Clear or new.

```
20039 \cs_new_protected:Npn \box_clear_new:N #1
20040 { \box_if_exist:NTF #1 { \box_clear:N #1 } { \box_new:N #1 } }
20041 \cs_new_protected:Npn \box_gclear_new:N #1
20042 { \box_if_exist:NTF #1 { \box_gclear:N #1 } { \box_new:N #1 } }
20043 \cs_generate_variant:Nn \box_clear_new:N { c }
20044 \cs_generate_variant:Nn \box_gclear_new:N { c }
```

Assigning the contents of a box to be another box.

```
20045 \cs_new_protected:Npn \box_set_eq:NN #1#2
20046 { \tex_setbox:D #1 \tex_copy:D #2 }
20047 \cs_new_protected:Npn \box_gset_eq:NN
20048 { \tex_global:D \box_set_eq:NN }
20049 \cs_generate_variant:Nn \box_set_eq:NN { c , Nc , cc }
20050 \cs_generate_variant:Nn \box_gset_eq:NN { c , Nc , cc }
```

Assigning the contents of a box to be another box. This clears the second box globally (that's how TeX does it).

```
20051 \cs_new_protected:Npn \box_set_eq_clear:NN #1#2
20052 { \tex_setbox:D #1 \tex_box:D #2 }
20053 \cs_new_protected:Npn \box_gset_eq_clear:NN
20054 { \tex_global:D \box_set_eq_clear:NN }
20055 \cs_generate_variant:Nn \box_set_eq_clear:NN { c , Nc , cc }
20056 \cs_generate_variant:Nn \box_gset_eq_clear:NN { c , Nc , cc }
```

Copies of the `cs` functions defined in `l3basics`.

```
20057 \prg_new_eq_conditional:NNN \box_if_exist:N \cs_if_exist:N
20058   { TF , T , F , p }
20059 \prg_new_eq_conditional:NNN \box_if_exist:c \cs_if_exist:c
20060   { TF , T , F , p }
\box_if_exist:cTF
\box_if_exist:cTF
```

37.2 Measuring and setting box dimensions

Accessing the height, depth, and width of a `<box>` register.

```
20061 \cs_new_eq:NN \box_ht:N \tex_ht:D
20062 \cs_new_eq:NN \box_dp:N \tex_dp:D
20063 \cs_new_eq:NN \box_wd:N \tex_wd:D
20064 \cs_generate_variant:Nn \box_ht:N { c }
20065 \cs_generate_variant:Nn \box_dp:N { c }
20066 \cs_generate_variant:Nn \box_wd:N { c }
\box_wd:c
```

Measuring is easy: all primitive work. These primitives are not expandable, so the derived functions are not either.

```
20067 \cs_new_protected:Npn \box_set_dp:Nn #1#2
20068   { \box_dp:N #1 \__dim_eval:w #2 \__dim_eval_end: }
20069 \cs_new_protected:Npn \box_set_ht:Nn #1#2
20070   { \box_ht:N #1 \__dim_eval:w #2 \__dim_eval_end: }
20071 \cs_new_protected:Npn \box_set_wd:Nn #1#2
20072   { \box_wd:N #1 \__dim_eval:w #2 \__dim_eval_end: }
20073 \cs_generate_variant:Nn \box_set_ht:Nn { c }
20074 \cs_generate_variant:Nn \box_set_dp:Nn { c }
20075 \cs_generate_variant:Nn \box_set_wd:Nn { c }
```

37.3 Using boxes

Using a `<box>`. These are just TeX primitives with meaningful names.

```
20076 \cs_new_eq:NN \box_use_clear:N \tex_box:D
20077 \cs_new_eq:NN \box_use:N \tex_copy:D
20078 \cs_generate_variant:Nn \box_use_clear:N { c }
20079 \cs_generate_variant:Nn \box_use:N { c }
```

Move box material in different directions.

```
20080 \cs_new_protected:Npn \box_move_left:nn #1#2
20081   { \tex_moveleft:D \__dim_eval:w #1 \__dim_eval_end: #2 }
20082 \cs_new_protected:Npn \box_move_right:nn #1#2
20083   { \tex_moveright:D \__dim_eval:w #1 \__dim_eval_end: #2 }
20084 \cs_new_protected:Npn \box_move_up:nn #1#2
20085   { \tex_raise:D \__dim_eval:w #1 \__dim_eval_end: #2 }
20086 \cs_new_protected:Npn \box_move_down:nn #1#2
20087   { \tex_lower:D \__dim_eval:w #1 \__dim_eval_end: #2 }
```

37.4 Box conditionals

The primitives for testing if a `<box>` is empty/void or which type of box it is.

```
20088 \cs_new_eq:NN \if_hbox:N      \tex_ifhbox:D
20089 \cs_new_eq:NN \if_vbox:N      \tex_ifvbox:D
20090 \cs_new_eq:NN \if_box_empty:N \tex_ifvoid:D
\if_box_empty:N
```

```

20091 \prg_new_conditional:Npnn \box_if_horizontal:N #1 { p , T , F , TF }
20092   { \if_hbox:N #1 \prg_return_true: \else: \prg_return_false: \fi: }
20093 \prg_new_conditional:Npnn \box_if_vertical:N #1 { p , T , F , TF }
20094   { \if_vbox:N #1 \prg_return_true: \else: \prg_return_false: \fi: }
20095 \cs_generate_variant:Nn \box_if_horizontal_p:N { c }
20096 \cs_generate_variant:Nn \box_if_horizontal:NT { c }
20097 \cs_generate_variant:Nn \box_if_horizontal:NF { c }
20098 \cs_generate_variant:Nn \box_if_horizontal:NTF { c }
20099 \cs_generate_variant:Nn \box_if_vertical_p:N { c }
20100 \cs_generate_variant:Nn \box_if_vertical:NT { c }
20101 \cs_generate_variant:Nn \box_if_vertical:NF { c }
20102 \cs_generate_variant:Nn \box_if_vertical:NTF { c }

```

Testing if a $\langle box \rangle$ is empty/void.

```

20103 \prg_new_conditional:Npnn \box_if_empty:N #1 { p , T , F , TF }
20104   { \if_box_empty:N #1 \prg_return_true: \else: \prg_return_false: \fi: }
20105 \cs_generate_variant:Nn \box_if_empty_p:N { c }
20106 \cs_generate_variant:Nn \box_if_empty:NT { c }
20107 \cs_generate_variant:Nn \box_if_empty:NF { c }
20108 \cs_generate_variant:Nn \box_if_empty:NTF { c }

```

(End definition for \box_new:N and others. These functions are documented on page 208.)

37.5 The last box inserted

$\text{\box_set_to_last:N}$ Set a box to the previous box.

```

20109 \cs_new_protected:Npn \box_set_to_last:N #1
20110   { \tex_setbox:D #1 \tex_lastbox:D }
20111 \cs_new_protected:Npn \box_gset_to_last:N
20112   { \tex_global:D \box_set_to_last:N }
20113 \cs_generate_variant:Nn \box_set_to_last:N { c }
20114 \cs_generate_variant:Nn \box_gset_to_last:N { c }

```

(End definition for $\text{\box_set_to_last:N}$ and $\text{\box_gset_to_last:N}$. These functions are documented on page 210.)

37.6 Constant boxes

\c_empty_box A box we never use.

```

20115 \box_new:N \c_empty_box

```

(End definition for \c_empty_box . This variable is documented on page 211.)

37.7 Scratch boxes

\l_tmpa_box Scratch boxes.

```

20116 \box_new:N \l_tmpa_box
20117 \box_new:N \l_tmpb_box
20118 \box_new:N \g_tmpa_box
20119 \box_new:N \g_tmpb_box

```

(End definition for \l_tmpa_box and others. These variables are documented on page 211.)

37.8 Viewing box contents

\TeX 's \showbox is not really that helpful in many cases, and it is also inconsistent with other $\text{L}\text{\TeX}3$ 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 arguments now outside the group.

```

 $\text{\box\_show:Nnn}$  20120 \cs_new_protected:Npn \box_show:N #1
 $\text{\box\_show:cnn}$  20121 { \box_show:Nnn #1 \c_max_int \c_max_int }
20122 \cs_generate_variant:Nn \box_show:N { c }
20123 \cs_new_protected:Npn \box_show:Nnn #1#2#3
20124 { \__box_show:NNff 1 #1 { \int_eval:n {#2} } { \int_eval:n {#3} } }
20125 \cs_generate_variant:Nn \box_show:Nnn { c }

```

(End definition for \box_show:N and \box_show:Nnn . These functions are documented on page 211.)

\box_log:N Getting \TeX to write to the log without interruption the run is done by altering the interaction mode. For that, the $\varepsilon\text{-}\text{\TeX}$ extensions are needed.

```

 $\text{\box\_log:c}$  20126 \cs_new_protected:Npn \box_log:N #1
20127 { \box_log:Nnn #1 \c_max_int \c_max_int }
 $\text{\box\_log:cnn}$  20128 \cs_generate_variant:Nn \box_log:N { c }
20129 \cs_new_protected:Npn \box_log:Nnn
20130 { \exp_args:No \__box_log:nNnn { \tex_the:D \etex_interactionmode:D } }
20131 \cs_new_protected:Npn \__box_log:nNnn #1#2#3#4
20132 {
20133     \int_set:Nn \etex_interactionmode:D { 0 }
20134     \__box_show:NNff 0 #2 { \int_eval:n {#3} } { \int_eval:n {#4} }
20135     \int_set:Nn \etex_interactionmode:D {#1}
20136 }
20137 \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 211.)

__box_show:NNnn The internal auxiliary to actually do the output uses a group to deal with breadth and depth values. The \use:n here gives better output appearance. Setting \tracingonline and $\text{\errorcontextlines}$ is used to control what appears in the terminal.

```

20138 \cs_new_protected:Npn \__box_show:NNnn #1#2#3#4
20139 {
20140     \box_if_exist:NTF #2
20141     {
20142         \group_begin:
20143             \int_set:Nn \tex_showboxbreadth:D {#3}
20144             \int_set:Nn \tex_showboxdepth:D {#4}
20145             \int_set:Nn \tex_tracingonline:D {#1}
20146             \int_set:Nn \tex_errorcontextlines:D { -1 }
20147             \tex_showbox:D \use:n {#2}
20148         \group_end:
20149     }
20150     {
20151         \__msg_kernel_error:nnx { kernel } { variable-not-defined }
20152             { \token_to_str:N #2 }
20153     }

```

```

20154     }
20155 \cs_generate_variant:Nn \__box_show:NNnn { NNff }
```

(End definition for `__box_show:NNnn`.)

37.9 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.

```

20156 \cs_new_protected:Npn \hbox:n #1
20157   { \tex_hbox:D \scan_stop: { \group_begin: #1 \group_end: } }
```

(End definition for `\hbox:n`. This function is documented on page 212.)

```

\hbox_set:Nn
\hbox_set:cn
\hbox_gset:Nn
\hbox_gset:cn
20158 \cs_new_protected:Npn \hbox_set:Nn #1#2
20159   { \tex_setbox:D #1 \tex_hbox:D { \group_begin: #2 \group_end: } }
20160 \cs_new_protected:Npn \hbox_gset:Nn { \tex_global:D \hbox_set:Nn }
20161 \cs_generate_variant:Nn \hbox_set:Nn { c }
20162 \cs_generate_variant:Nn \hbox_gset:Nn { c }
```

(End definition for `\hbox_set:Nn` and `\hbox_gset:Nn`. These functions are documented on page 212.)

\hbox_set_to_wd:Nnn Storing material in a horizontal box with a specified width.

```

\hbox_set_to_wd:cnn
\hbox_gset_to_wd:Nnn
\hbox_gset_to_wd:cnn
20163 \cs_new_protected:Npn \hbox_set_to_wd:Nnn #1#2#3
20164   {
20165     \tex_setbox:D #1 \tex_hbox:D to \__dim_eval:w #2 \__dim_eval_end:
20166     { \group_begin: #3 \group_end: }
20167   }
20168 \cs_new_protected:Npn \hbox_gset_to_wd:Nnn
20169   { \tex_global:D \hbox_set_to_wd:Nnn }
20170 \cs_generate_variant:Nn \hbox_set_to_wd:Nnn { c }
20171 \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 212.)

\hbox_set:Nw Storing material in a horizontal box. This type is useful in environment definitions.

```

\hbox_set:cw
\hbox_gset:Nw
\hbox_gset:cw
\hbox_set_end:
\hbox_gset_end:
20172 \cs_new_protected:Npn \hbox_set:Nw #1
20173   {
20174     \tex_setbox:D #1 \tex_hbox:D
20175     \c_group_begin_token
20176     \group_begin:
20177   }
20178 \cs_new_protected:Npn \hbox_gset:Nw
20179   { \tex_global:D \hbox_set:Nw }
20180 \cs_generate_variant:Nn \hbox_set:Nw { c }
20181 \cs_generate_variant:Nn \hbox_gset:Nw { c }
20182 \cs_new_protected:Npn \hbox_set_end:
20183   {
20184     \group_end:
20185     \c_group_end_token
20186   }
20187 \cs_new_eq:NN \hbox_gset_end: \hbox_set_end:
```

(End definition for `\hbox_set:Nw` and others. These functions are documented on page 212.)

`\hbox_to_wd:nn` Put a horizontal box directly into the input stream.

```
20188 \cs_new_protected:Npn \hbox_to_wd:nn #1#2
20189     {
20190         \tex_hbox:D to \__dim_eval:w #1 \__dim_eval_end:
20191             { \group_begin: #2 \group_end: }
20192     }
20193 \cs_new_protected:Npn \hbox_to_zero:n #1
20194     { \tex_hbox:D to \c_zero_dim { \group_begin: #1 \group_end: } }
```

(End definition for `\hbox_to_wd:nn` and `\hbox_to_zero:n`. These functions are documented on page 212.)

`\hbox_overlap_left: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.

```
20195 \cs_new_protected:Npn \hbox_overlap_left:n #1
20196     { \hbox_to_zero:n { \tex_hss:D #1 } }
20197 \cs_new_protected:Npn \hbox_overlap_right:n #1
20198     { \hbox_to_zero:n { #1 \tex_hss:D } }
```

(End definition for `\hbox_overlap_left:n` and `\hbox_overlap_right:n`. These functions are documented on page 212.)

`\hbox_unpack:N` Unpacking a box and if requested also clear it.

```
20199 \cs_new_eq:NN \hbox_unpack:N \tex_unhcopy:D
20200 \cs_new_eq:NN \hbox_unpack_clear:N \tex_unhbox:D
20201 \cs_generate_variant:Nn \hbox_unpack:N { c }
20202 \cs_generate_variant:Nn \hbox_unpack_clear:N { c }
```

(End definition for `\hbox_unpack:N` and `\hbox_unpack_clear:N`. These functions are documented on page 213.)

37.10 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.

`\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.

```
20203 \cs_new_protected:Npn \vbox:n #1
20204     { \tex_vbox:D { \group_begin: #1 \par \group_end: } }
20205 \cs_new_protected:Npn \vbox_top:n #1
20206     { \tex_vtop:D { \group_begin: #1 \par \group_end: } }
```

(End definition for `\vbox:n` and `\vbox_top:n`. These functions are documented on page 213.)

```

\ vbox_to_ht:nn Put a vertical box directly into the input stream.
\ vbox_to_zero:n
\ vbox_to_ht:nn
\ vbox_to_zero:n
20207 \cs_new_protected:Npn \vbox_to_ht:nn #1#2
20208 {
20209     \tex_vbox:D to \__dim_eval:w #1 \__dim_eval_end:
20210     { \group_begin: #2 \par \group_end: }
20211 }
20212 \cs_new_protected:Npn \vbox_to_zero:n #1
20213 {
20214     \tex_vbox:D to \c_zero_dim
20215     { \group_begin: #1 \par \group_end: }
20216 }

```

(End definition for `\vbox_to_ht:nn` and others. These functions are documented on page 213.)

```

\ vbox_set:Nn Storing material in a vertical box with a natural height.
\ vbox_set:cn
\ vbox_gset:Nn
\ vbox_gset:cn
20217 \cs_new_protected:Npn \vbox_set:Nn #1#2
20218 {
20219     \tex_setbox:D #1 \tex_vbox:D
20220     { \group_begin: #2 \par \group_end: }
20221 }
20222 \cs_new_protected:Npn \vbox_gset:Nn { \tex_global:D \vbox_set:Nn }
20223 \cs_generate_variant:Nn \vbox_set:Nn { c }
20224 \cs_generate_variant:Nn \vbox_gset:Nn { c }

```

(End definition for `\vbox_set:Nn` and `\vbox_gset:Nn`. These functions are documented on page 213.)

```

\ 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
\ vbox_gset_top:cn
20225 \cs_new_protected:Npn \vbox_set_top:Nn #1#2
20226 {
20227     \tex_setbox:D #1 \tex_vtop:D
20228     { \group_begin: #2 \par \group_end: }
20229 }
2030 \cs_new_protected:Npn \vbox_gset_top:Nn
2031 { \tex_global:D \vbox_set_top:Nn }
2032 \cs_generate_variant:Nn \vbox_set_top:Nn { c }
2033 \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 214.)

```

\ vbox_set_to_ht:Nnn Storing material in a vertical box with a specified height.
\ vbox_set_to_ht:cnn
\ vbox_gset_to_ht:Nnn
\ vbox_gset_to_ht:cnn
2034 \cs_new_protected:Npn \vbox_set_to_ht:Nnn #1#2#3
2035 {
2036     \tex_setbox:D #1 \tex_vbox:D to \__dim_eval:w #2 \__dim_eval_end:
2037     { \group_begin: #3 \par \group_end: }
2038 }
2039 \cs_new_protected:Npn \vbox_gset_to_ht:Nnn
2040 { \tex_global:D \vbox_set_to_ht:Nnn }
2041 \cs_generate_variant:Nn \vbox_set_to_ht:Nnn { c }
2042 \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 214.)

```

\ 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:
20243 \cs_new_protected:Npn \vbox_set:Nw #1
20244 {
20245     \tex_setbox:D #1 \tex_vbox:D
20246     \c_group_begin_token
20247     \group_begin:
20248 }
20249 \cs_new_protected:Npn \vbox_gset:Nw
20250 { \tex_global:D \vbox_set:Nw }
20251 \cs_generate_variant:Nn \vbox_set:Nw { c }
20252 \cs_generate_variant:Nn \vbox_gset:Nw { c }
20253 \cs_new_protected:Npn \vbox_set_end:
20254 {
20255     \par
20256     \group_end:
20257     \c_group_end_token
20258 }
20259 \cs_new_eq:NN \vbox_gset_end: \vbox_set_end:

```

(End definition for `\vbox_set:Nw` and others. These functions are documented on page 214.)

```

\ vbox_unpack:N Unpacking a box and if requested also clear it.
\ vbox_unpack:c
\ vbox_unpack_clear:N
\ vbox_unpack_clear:c
20260 \cs_new_eq:NN \vbox_unpack:N \tex_unvcopy:D
20261 \cs_new_eq:NN \vbox_unpack_clear:N \tex_unvbox:D
20262 \cs_generate_variant:Nn \vbox_unpack:N { c }
20263 \cs_generate_variant:Nn \vbox_unpack_clear:N { c }

```

(End definition for `\vbox_unpack:N` and `\vbox_unpack_clear:N`. These functions are documented on page 214.)

`\vbox_set_split_to_ht:NNn` Splitting a vertical box in two.

```

20264 \cs_new_protected:Npn \vbox_set_split_to_ht:NNn #1#2#3
20265 { \tex_setbox:D #1 \tex_vsplit:D #2 to \__dim_eval:w #3 \__dim_eval_end: }

```

(End definition for `\vbox_set_split_to_ht:NNn`. This function is documented on page 214.)

37.11 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.

```
20266 \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.

```

20267 \fp_new:N \l_box_cos_fp
20268 \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.

```

20269 \dim_new:N \l_box_top_dim
20270 \dim_new:N \l_box_bottom_dim
20271 \dim_new:N \l_box_left_dim
20272 \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.

```
20273 \dim_new:N \l_box_top_new_dim
20274 \dim_new:N \l_box_bottom_new_dim
20275 \dim_new:N \l_box_left_new_dim
20276 \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.

```
20277 \box_new:N \l_box_internal_box
```

(End definition for `\l_box_internal_box`.)

`\box_rotate:Nn` Rotation of a box starts with working out the relevant sine and cosine. The actual rotation is in an auxiliary to keep the flow slightly clearer

```
20278 \cs_new_protected:Npn \box_rotate:Nn #1#2
20279 {
20280     \hbox_set:Nn #1
20281     {
20282         \fp_set:Nn \l_box_angle_fp {#2}
20283         \fp_set:Nn \l_box_sin_fp { sind ( \l_box_angle_fp ) }
20284         \fp_set:Nn \l_box_cos_fp { cosd ( \l_box_angle_fp ) }
20285         \__box_rotate:N #1
20286     }
20287 }
```

The edges of the box are then recorded: the left edge will always be at zero. Rotation of the four edges then takes place: this is most efficiently done on a quadrant by quadrant basis.

```
20288 \cs_new_protected:Npn \__box_rotate:N #1
20289 {
20290     \dim_set:Nn \l_box_top_dim { \box_ht:N #1 }
20291     \dim_set:Nn \l_box_bottom_dim { -\box_dp:N #1 }
20292     \dim_set:Nn \l_box_right_dim { \box_wd:N #1 }
20293     \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.)

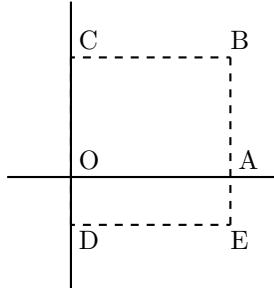


Figure 1: Co-ordinates of a box prior to rotation.

```

20294 \fp_compare:nNnTF \l__box_sin_fp > \c_zero_fp
20295 {
20296   \fp_compare:nNnTF \l__box_cos_fp > \c_zero_fp
20297     { \__box_rotate_quadrant_one: }
20298     { \__box_rotate_quadrant_two: }
20299   }
20300   {
20301     \fp_compare:nNnTF \l__box_cos_fp < \c_zero_fp
20302       { \__box_rotate_quadrant_three: }
20303       { \__box_rotate_quadrant_four: }
20304   }

```

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 will be in the same place as the original.

```

20305 \hbox_set:Nn \l__box_internal_box { \box_use:N #1 }
20306 \hbox_set:Nn \l__box_internal_box
20307 {
20308   \tex_kern:D -\l__box_left_new_dim
20309   \hbox:n
20310   {
20311     \__driver_box_use_rotate:Nn
20312       \l__box_internal_box
20313       \l__box_angle_fp
20314   }
20315 }

```

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.

```

20316 \box_set_ht:Nn \l__box_internal_box { \l__box_top_new_dim }
20317 \box_set_dp:Nn \l__box_internal_box { -\l__box_bottom_new_dim }
20318 \box_set_wd:Nn \l__box_internal_box
20319   { \l__box_right_new_dim - \l__box_left_new_dim }
20320   \box_use:N \l__box_internal_box
20321 }

```

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.

```

20322 \cs_new_protected:Npn \__box_rotate_x:nnN #1#2#3
20323 {
20324     \dim_set:Nn #3
20325     {
20326         \fp_to_dim:n
20327         {
20328             \l__box_cos_fp * \dim_to_fp:n {#1}
20329             - \l__box_sin_fp * \dim_to_fp:n {#2}
20330         }
20331     }
20332 }
20333 \cs_new_protected:Npn \__box_rotate_y:nnN #1#2#3
20334 {
20335     \dim_set:Nn #3
20336     {
20337         \fp_to_dim:n
20338         {
20339             \l__box_sin_fp * \dim_to_fp:n {#1}
20340             + \l__box_cos_fp * \dim_to_fp:n {#2}
20341         }
20342     }
20343 }
```

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.

```

20344 \cs_new_protected:Npn \__box_rotate_quadrant_one:
20345 {
20346     \__box_rotate_y:nnN \l__box_right_dim \l__box_top_dim
20347     \l__box_top_new_dim
20348     \__box_rotate_y:nnN \l__box_left_dim \l__box_bottom_dim
20349     \l__box_bottom_new_dim
20350     \__box_rotate_x:nnN \l__box_left_dim \l__box_top_dim
20351     \l__box_left_new_dim
20352     \__box_rotate_x:nnN \l__box_right_dim \l__box_bottom_dim
20353     \l__box_right_new_dim
20354 }
20355 \cs_new_protected:Npn \__box_rotate_quadrant_two:
20356 {
20357     \__box_rotate_y:nnN \l__box_right_dim \l__box_bottom_dim
20358     \l__box_top_new_dim
20359     \__box_rotate_y:nnN \l__box_left_dim \l__box_top_dim
20360     \l__box_bottom_new_dim
20361     \__box_rotate_x:nnN \l__box_right_dim \l__box_top_dim
20362     \l__box_left_new_dim
20363     \__box_rotate_x:nnN \l__box_left_dim \l__box_bottom_dim
20364     \l__box_right_new_dim
20365 }
20366 \cs_new_protected:Npn \__box_rotate_quadrant_three:
20367 {
```

```

20368     \__box_rotate_y:nnN \l_box_left_dim \l_box_bottom_dim
20369         \l_box_top_new_dim
20370     \__box_rotate_y:nnN \l_box_right_dim \l_box_top_dim
20371         \l_box_bottom_new_dim
20372     \__box_rotate_x:nnN \l_box_right_dim \l_box_bottom_dim
20373         \l_box_left_new_dim
20374     \__box_rotate_x:nnN \l_box_left_dim \l_box_top_dim
20375         \l_box_right_new_dim
20376     }
20377 \cs_new_protected:Npn \__box_rotate_quadrant_four:
20378 {
20379     \__box_rotate_y:nnN \l_box_left_dim \l_box_top_dim
20380         \l_box_top_new_dim
20381     \__box_rotate_y:nnN \l_box_right_dim \l_box_bottom_dim
20382         \l_box_bottom_new_dim
20383     \__box_rotate_x:nnN \l_box_left_dim \l_box_bottom_dim
20384         \l_box_left_new_dim
20385     \__box_rotate_x:nnN \l_box_right_dim \l_box_top_dim
20386         \l_box_right_new_dim
20387 }

```

(End definition for `\box_rotate:Nn` and others. These functions are documented on page 216.)

`\l_box_scale_x_fp` Scaling is potentially-different in the two axes.

```

20388 \fp_new:N \l_box_scale_x_fp
20389 \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.

```

20390 \cs_new_protected:Npn \box_resize_to_wd_and_ht_plus_dp:Nnn #1#2#3
20391 {
20392     \hbox_set:Nn #1
20393     {
20394         \__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.

```

20395     \fp_set:Nn \l_box_scale_x_fp
20396         { \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.

```

20397     \fp_set:Nn \l_box_scale_y_fp
20398     {
20399         \dim_to_fp:n {#3}
20400         / \dim_to_fp:n { \l_box_top_dim - \l_box_bottom_dim }
20401     }

```

Hand off to the auxiliary which does the rest of the work.

```

20402         \__box_resize:N #1
20403     }
20404     }
20405 \cs_generate_variant:Nn \box_resize_to_wd_and_ht_plus_dp:Nnn { c }
20406 \cs_new_protected:Npn \__box_resize_set_corners:N #1
20407 {
20408     \dim_set:Nn \l_box_top_dim { \box_ht:N #1 }

```

```

20409     \dim_set:Nn \l__box_bottom_dim { -\box_dp:N #1 }
20410     \dim_set:Nn \l__box_right_dim { \box_wd:N #1 }
20411     \dim_zero:N \l__box_left_dim
20412 }
```

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.

```

20413 \cs_new_protected:Npn \__box_resize:N #1
20414 {
20415     \__box_resize>NNN \l__box_right_new_dim
20416         \l__box_scale_x_fp \l__box_right_dim
20417     \__box_resize>NNN \l__box_bottom_new_dim
20418         \l__box_scale_y_fp \l__box_bottom_dim
20419     \__box_resize>NNN \l__box_top_new_dim
20420         \l__box_scale_y_fp \l__box_top_dim
20421     \__box_resize_common:N #1
20422 }
20423 \cs_new_protected:Npn \__box_resize>NNN #1#2#3
20424 {
20425     \dim_set:Nn #1
20426     { \fp_to_dim:n { \fp_abs:n { #2 } * \dim_to_fp:n { #3 } } } }
20427 }
```

(End definition for `\box_resize_to_wd_and_ht_plus_dp:Nnn` and others. These functions are documented on page 216.)

`\box_resize_to_ht:Nn`

`\box_resize_to_ht:cn`

`\box_resize_to_ht_plus_dp:Nn`

`\box_resize_to_ht_plus_dp:cn`

`\box_resize_to_wd:Nn`

`\box_resize_to_wd:cn`

`\box_resize_to_wd_and_ht:Nnn`

`\box_resize_to_wd_and_ht:cnn`

Scaling to a (total) height or to a width is a simplified version of the main resizing operation, with the scale simply copied between the two parts. The internal auxiliary is called using the scaling value twice, as the sign for both parts is needed (as this allows the same internal code to be used as for the general case).

```

20428 \cs_new_protected:Npn \box_resize_to_ht:Nn #1#2
20429 {
20430     \hbox_set:Nn #1
20431     {
20432         \__box_resize_set_corners:N #1
20433         \fp_set:Nn \l__box_scale_y_fp
20434             {
20435                 \dim_to_fp:n {#2}
20436                 / \dim_to_fp:n { \l__box_top_dim }
20437             }
20438         \fp_set_eq:NN \l__box_scale_x_fp \l__box_scale_y_fp
20439         \__box_resize:N #1
20440     }
20441 }
20442 \cs_generate_variant:Nn \box_resize_to_ht:Nn { c }
20443 \cs_new_protected:Npn \box_resize_to_ht_plus_dp:Nn #1#2
20444 {
20445     \hbox_set:Nn #1
20446     {
20447         \__box_resize_set_corners:N #1
20448         \fp_set:Nn \l__box_scale_y_fp
20449             {
```

```

20450           \dim_to_fp:n {#2}
20451           / \dim_to_fp:n { \l_box_top_dim - \l_box_bottom_dim }
20452       }
20453   \fp_set_eq:NN \l_box_scale_x_fp \l_box_scale_y_fp
20454   \__box_resize:N #1
20455 }
20456 }
20457 \cs_generate_variant:Nn \box_resize_to_ht_plus_dp:Nn { c }
20458 \cs_new_protected:Npn \box_resize_to_wd:Nn #1#2
20459 {
20460     \hbox_set:Nn #1
20461     {
20462         \__box_resize_set_corners:N #1
20463         \fp_set:Nn \l_box_scale_x_fp
20464             { \dim_to_fp:n {#2} / \dim_to_fp:n { \l_box_right_dim } }
20465         \fp_set_eq:NN \l_box_scale_y_fp \l_box_scale_x_fp
20466         \__box_resize:N #1
20467     }
20468 }
20469 \cs_generate_variant:Nn \box_resize_to_wd:Nn { c }
20470 \cs_new_protected:Npn \box_resize_to_wd_and_ht:Nnn #1#2#3
20471 {
20472     \hbox_set:Nn #1
20473     {
20474         \__box_resize_set_corners:N #1
20475         \fp_set:Nn \l_box_scale_x_fp
20476             { \dim_to_fp:n {#2} / \dim_to_fp:n { \l_box_right_dim } }
20477         \fp_set:Nn \l_box_scale_y_fp
20478             {
20479                 \dim_to_fp:n {#3}
20480                 / \dim_to_fp:n { \l_box_top_dim }
20481             }
20482         \__box_resize:N #1
20483     }
20484 }
20485 \cs_generate_variant:Nn \box_resize_to_wd_and_ht:Nnn { c }

```

(End definition for `\box_resize_to_ht:Nn` and others. These functions are documented on page 215.)

`\box_scale:Nnn`
`\box_scale:cnn`
`__box_scale_aux:N`

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. Once that is done then after a check for the trivial scaling a hand-off can be made to the common code. The code here is split into two as this allows sharing with the auto-resizing functions.

```

20486 \cs_new_protected:Npn \box_scale:Nnn #1#2#3
20487 {
20488     \hbox_set:Nn #1
20489     {
20490         \fp_set:Nn \l_box_scale_x_fp {#2}
20491         \fp_set:Nn \l_box_scale_y_fp {#3}
20492         \__box_scale_aux:N #1
20493     }
20494 }
20495 \cs_generate_variant:Nn \box_scale:Nnn { c }

```

```

20496 \cs_new_protected:Npn \__box_scale_aux:N #1
20497  {
20498      \dim_set:Nn \l__box_top_dim { \box_ht:N #1 }
20499      \dim_set:Nn \l__box_bottom_dim { -\box_dp:N #1 }
20500      \dim_set:Nn \l__box_right_dim { \box_wd:N #1 }
20501      \dim_zero:N \l__box_left_dim
20502      \dim_set:Nn \l__box_top_new_dim
20503          { \fp_abs:n { \l__box_scale_y_fp } \l__box_top_dim }
20504      \dim_set:Nn \l__box_bottom_new_dim
20505          { \fp_abs:n { \l__box_scale_y_fp } \l__box_bottom_dim }
20506      \dim_set:Nn \l__box_right_new_dim
20507          { \fp_abs:n { \l__box_scale_x_fp } \l__box_right_dim }
20508      \__box_resize_common:N #1
20509  }

```

(End definition for `\box_scale:Nnn` and `__box_scale_aux:N`. These functions are documented on page 216.)

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.

```

20510 \cs_new_protected:Npn \box_autosize_to_wd_and_ht:Nnn #1#2#3
20511  { \__box_autosize:Nnnn #1 {#2} {#3} { \box_ht:N #1 } }
20512 \cs_generate_variant:Nn \box_autosize_to_wd_and_ht:Nnn { c }
20513 \cs_new_protected:Npn \box_autosize_to_wd_and_ht_plus_dp:Nnn #1#2#3
20514  { \__box_autosize:Nnnn #1 {#2} {#3} { \box_ht:N #1 + \box_dp:N #1 } }
20515 \cs_generate_variant:Nn \box_autosize_to_wd_and_ht_plus_dp:Nnn { c }
20516 \cs_new_protected:Npn \__box_autosize:Nnnn #1#2#3#4
20517  {
20518      \hbox_set:Nn #1
20519  {
20520      \fp_set:Nn \l__box_scale_x_fp { ( #2 ) / \box_wd:N #1 }
20521      \fp_set:Nn \l__box_scale_y_fp { ( #3 ) / ( #4 ) }
20522      \fp_compare:nNnTF \l__box_scale_x_fp > \l__box_scale_y_fp
20523          { \fp_set_eq:NN \l__box_scale_x_fp \l__box_scale_y_fp }
20524          { \fp_set_eq:NN \l__box_scale_y_fp \l__box_scale_x_fp }
20525      \__box_scale_aux:N #1
20526  }
20527  }

```

(End definition for `\box_autosize_to_wd_and_ht:Nnn`, `\box_autosize_to_wd_and_ht_plus_dp:cnn`, and `__box_autosize:Nnnn`. These functions are documented on page 215.)

`__box_resize_common:N` The main resize function places in input into a box which will start of with zero width, and includes the handles for engine rescaling.

```

20528 \cs_new_protected:Npn \__box_resize_common:N #1
20529  {
20530      \hbox_set:Nn \l__box_internal_box
20531  {
20532      \__driver_box_use_scale:Nnn
20533          #1
20534          \l__box_scale_x_fp
20535          \l__box_scale_y_fp
20536  }

```

The new height and depth can be applied directly.

```

20537   \fp_compare:nNnTF \l__box_scale_y_fp > \c_zero_fp
20538   {
20539     \box_set_ht:Nn \l__box_internal_box { \l__box_top_new_dim }
20540     \box_set_dp:Nn \l__box_internal_box { -\l__box_bottom_new_dim }
20541   }
20542   {
20543     \box_set_dp:Nn \l__box_internal_box { \l__box_top_new_dim }
20544     \box_set_ht:Nn \l__box_internal_box { -\l__box_bottom_new_dim }
20545   }

```

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.

```

20546   \fp_compare:nNnTF \l__box_scale_x_fp < \c_zero_fp
20547   {
20548     \hbox_to_wd:nn { \l__box_right_new_dim }
20549     {
20550       \tex_kern:D \l__box_right_new_dim
20551       \box_use:N \l__box_internal_box
20552       \tex_hss:D
20553     }
20554   }
20555   {
20556     \box_set_wd:Nn \l__box_internal_box { \l__box_right_new_dim }
20557     \hbox:n
20558     {
20559       \tex_kern:D \c_zero_dim
20560       \box_use:N \l__box_internal_box
20561       \tex_hss:D
20562     }
20563   }
20564 }

```

(End definition for `_box_resize_common:N.`)

37.12 Deprecated functions

```

\box_resize:Nnn
\box_resize:cnn
20565 \cs_new_eq:NN \box_resize:Nnn \box_resize_to_wd_and_ht_plus_dp:Nnn
20566 \cs_new_eq:NN \box_resize:cnn \box_resize_to_wd_and_ht_plus_dp:cnn

(End definition for \box_resize:Nnn. This function is documented on page ??.)
```

20567 ⟨/initex | package⟩

38 I3coffins Implementation

```

20568 {*initex | package}
20569 (@@=coffin)
```

38.1 Coffins: data structures and general variables

\l__coffin_internal_box	Scratch variables.
\l__coffin_internal_dim	20570 \box_new:N \l__coffin_internal_box
\l__coffin_internal_tl	20571 \dim_new:N \l__coffin_internal_dim 20572 \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.
	20573 \prop_new:N \c__coffin_corners_prop 20574 \prop_put:Nnn \c__coffin_corners_prop { tl } { { Opt } { Opt } } 20575 \prop_put:Nnn \c__coffin_corners_prop { tr } { { Opt } { Opt } } 20576 \prop_put:Nnn \c__coffin_corners_prop { bl } { { Opt } { Opt } } 20577 \prop_put:Nnn \c__coffin_corners_prop { br } { { Opt } { Opt } }
	(End definition for \c__coffin_corners_prop.)
\c__coffin_poles_prop	Pole positions are given for horizontal, vertical and reference-point based values.
	20578 \prop_new:N \c__coffin_poles_prop 20579 \tl_set:Nn \l__coffin_internal_tl { { Opt } { Opt } { Opt } { 1000pt } } 20580 \prop_put:Nno \c__coffin_poles_prop { 1 } { \l__coffin_internal_tl } 20581 \prop_put:Nno \c__coffin_poles_prop { hc } { \l__coffin_internal_tl } 20582 \prop_put:Nno \c__coffin_poles_prop { r } { \l__coffin_internal_tl } 20583 \tl_set:Nn \l__coffin_internal_tl { { Opt } { Opt } { 1000pt } { Opt } } 20584 \prop_put:Nno \c__coffin_poles_prop { b } { \l__coffin_internal_tl } 20585 \prop_put:Nno \c__coffin_poles_prop { vc } { \l__coffin_internal_tl } 20586 \prop_put:Nno \c__coffin_poles_prop { t } { \l__coffin_internal_tl } 20587 \prop_put:Nno \c__coffin_poles_prop { B } { \l__coffin_internal_tl } 20588 \prop_put:Nno \c__coffin_poles_prop { H } { \l__coffin_internal_tl } 20589 \prop_put:Nno \c__coffin_poles_prop { T } { \l__coffin_internal_tl }
	(End definition for \c__coffin_poles_prop.)
\l__coffin_slope_x_fp	Used for calculations of intersections.
\l__coffin_slope_y_fp	20590 \fp_new:N \l__coffin_slope_x_fp 20591 \fp_new:N \l__coffin_slope_y_fp
	(End definition for \l__coffin_slope_x_fp and \l__coffin_slope_y_fp.)
\l__coffin_error_bool	For propagating errors so that parts of the code can work around them.
	20592 \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	20593 \dim_new:N \l__coffin_offset_x_dim 20594 \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
20595 \tl_new:N \l__coffin_pole_a_tl
20596 \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
\l__coffin_x_prime_dim
\l__coffin_y_prime_dim
20597 \dim_new:N \l__coffin_x_dim
20598 \dim_new:N \l__coffin_y_dim
20599 \dim_new:N \l__coffin_x_prime_dim
20600 \dim_new:N \l__coffin_y_prime_dim

(End definition for \l__coffin_x_dim and others.)
```

38.2 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_if_exist:p:N Several of the higher-level coffin functions will give multiple errors if the coffin does not
\coffin_if_exist:p:c exist. A cleaner way to handle this is provided here: both the box and the coffin structure
\coffin_if_exist:NTF are checked.
\coffin_if_exist:cTF
20601 \prg_new_conditional:Npnn \coffin_if_exist:N #1 { p , T , F , TF }
20602 {
20603   \cs_if_exist:NTF #1
20604   {
20605     \cs_if_exist:cTF { l__coffin_poles_ \__int_value:w #1 _prop }
20606     { \prg_return_true: }
20607     { \prg_return_false: }
20608   }
20609   { \prg_return_false: }
20610 }
```

```
20611 \cs_generate_variant:Nn \coffin_if_exist_p:N { c }
20612 \cs_generate_variant:Nn \coffin_if_exist:NT { c }
20613 \cs_generate_variant:Nn \coffin_if_exist:NF { c }
20614 \cs_generate_variant:Nn \coffin_if_exist:NTF { c }

(End definition for \coffin_if_exist:NTF. This function is documented on page 218.)
```

```
\__coffin_if_exist:NT Several of the higher-level coffin functions will 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.
```

```
20615 \cs_new_protected:Npn \__coffin_if_exist:NT #1#2
20616 {
20617   \coffin_if_exist:NTF #1
20618   { #2 }
20619   {
20620     \__msg_kernel_error:nnx { kernel } { unknown-coffin }
20621     { \token_to_str:N #1 }
20622   }
20623 }
```

(End definition for __coffin_if_exist:NT.)

\coffin_clear:N Clearing coffins means emptying the box and resetting all of the structures.

```
20624 \cs_new_protected:Npn \coffin_clear:N #1
20625 {
20626     \__coffin_if_exist:NT #1
20627     {
20628         \box_clear:N #1
20629         \__coffin_reset_structure:N #1
20630     }
20631 }
20632 \cs_generate_variant:Nn \coffin_clear:N { c }
```

(End definition for `\coffin_clear:N`. This function is documented on page 218.)

\coffin_new:N Creating a new coffin means making the underlying box and adding the data structures.

\coffin_new:c These are created globally, as there is a need to avoid any strange effects if the coffin is created inside a group. This means that the usual rule about `\l_...` variables has to be broken.

```
20633 \cs_new_protected:Npn \coffin_new:N #1
20634 {
20635     \box_new:N #1
20636     \__chk_suspend_log:
20637     \prop_clear_new:c { l__coffin_corners_ \__int_value:w #1 _prop }
20638     \prop_clear_new:c { l__coffin_poles_ \__int_value:w #1 _prop }
20639     \prop_gset_eq:cN { l__coffin_corners_ \__int_value:w #1 _prop }
20640         \c_coffin_corners_prop
20641     \prop_gset_eq:cN { l__coffin_poles_ \__int_value:w #1 _prop }
20642         \c_coffin_poles_prop
20643     \__chk_resume_log:
20644 }
20645 \cs_generate_variant:Nn \coffin_new:N { c }
```

(End definition for `\coffin_new:N`. This function is documented on page 218.)

\hcoffin_set:Nn Horizontal coffins are relatively easy: set the appropriate box, reset the structures then update the handle positions.

```
20646 \cs_new_protected:Npn \hcoffin_set:Nn #1#2
20647 {
20648     \__coffin_if_exist:NT #1
20649     {
20650         \hbox_set:Nn #1
20651         {
20652             \color_ensure_current:
20653             #2
20654         }
20655         \__coffin_reset_structure:N #1
20656         \__coffin_update_poles:N #1
20657         \__coffin_update_corners:N #1
20658     }
20659 }
20660 \cs_generate_variant:Nn \hcoffin_set:Nn { c }
```

(End definition for `\hcoffin_set:Nn`. This function is documented on page 218.)

```
\vcoffin_set:Nnn  
\vcoffin_set:cnn
```

Setting vertical coffins is more complex. First, the material is typeset with a given width. The default handles and poles are set as for a horizontal coffin, before finding the top baseline using a temporary box. No `\color_ensure_current:` here as that would add a 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).

```
20661 \cs_new_protected:Npn \vcoffin_set:Nnn #1#2#3  
20662 {  
20663   \__coffin_if_exist:NT #1  
20664   {  
20665     \vbox_set:Nn #1  
20666     {  
20667       \dim_set:Nn \tex_hsize:D {#2}  
20668   (*package)  
20669     \dim_set_eq:NN \linewidth \tex_hsize:D  
20670     \dim_set_eq:NN \columnwidth \tex_hsize:D  
20671   (/package)  
20672     #3  
20673   }  
20674   \__coffin_reset_structure:N #1  
20675   \__coffin_update_poles:N #1  
20676   \__coffin_update_corners:N #1  
20677   \vbox_set_top:Nn \l__coffin_internal_box { \vbox_unpack:N #1 }  
20678   \__coffin_set_pole:Nnx #1 { T }  
20679   {  
20680     { Opt }  
20681     {  
20682       \dim_eval:n  
20683         { \box_ht:N #1 - \box_ht:N \l__coffin_internal_box }  
20684     }  
20685     { 1000pt }  
20686     { Opt }  
20687   }  
20688   \box_clear:N \l__coffin_internal_box  
20689 }  
20690 }  
20691 \cs_generate_variant:Nn \vcoffin_set:Nnn { c }
```

(End definition for `\vcoffin_set:Nnn`. This function is documented on page 219.)

```
\hcoffin_set:Nw  
\hcoffin_set:cw  
\hcoffin_set_end:
```

These are the “begin”/“end” versions of the above: watch the grouping!

```
20692 \cs_new_protected:Npn \hcoffin_set:Nw #1  
20693 {  
20694   \__coffin_if_exist:NT #1  
20695   {  
20696     \hbox_set:Nw #1 \color_ensure_current:  
20697     \cs_set_protected:Npn \hcoffin_set_end:  
20698     {  
20699       \hbox_set_end:  
20700       \__coffin_reset_structure:N #1  
20701       \__coffin_update_poles:N #1  
20702       \__coffin_update_corners:N #1  
20703     }  
20704   }  
20705 }
```

```

20706 \cs_new_protected:Npn \hcoffin_set_end: { }
20707 \cs_generate_variant:Nn \hcoffin_set:Nw { c }

```

(End definition for `\hcoffin_set:Nw` and `\hcoffin_set_end:`. These functions are documented on page 218.)

\vcoffin_set:Nnw The same for vertical coffins.

```

\hcoffin_set:cnw
\vcoffin_set_end:
20708 \cs_new_protected:Npn \vcoffin_set:Nnw #1#2
20709 {
20710     \__coffin_if_exist:NT #1
20711     {
20712         \vbox_set:Nw #1
20713         \dim_set:Nn \tex_hsize:D {#2}
20714     (*package)
20715         \dim_set_eq:NN \linewidth \tex_hsize:D
20716         \dim_set_eq:NN \columnwidth \tex_hsize:D
20717 
```

```

    />package)
20718 \cs_set_protected:Npn \vcoffin_set_end:
20719 {
20720     \vbox_set_end:
20721     \__coffin_reset_structure:N #1
20722     \__coffin_update_poles:N #1
20723     \__coffin_update_corners:N #1
20724     \vbox_set_top:Nn \l__coffin_internal_box { \vbox_unpack:N #1 }
20725     \__coffin_set_pole:Nnx #1 { T }
20726     {
20727         { Opt }
20728         {
20729             \dim_eval:n
20730             { \box_ht:N #1 - \box_ht:N \l__coffin_internal_box }
20731         }
20732         { 1000pt }
20733         { Opt }
20734     }
20735     \box_clear:N \l__coffin_internal_box
20736 }
20737 }
20738 }
20739 \cs_new_protected:Npn \vcoffin_set_end: { }
20740 \cs_generate_variant:Nn \vcoffin_set:Nnw { c }

```

(End definition for `\vcoffin_set:Nnw` and `\vcoffin_set_end:`. These functions are documented on page 219.)

\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
20741 \cs_new_protected:Npn \coffin_set_eq:NN #1#2
20742 {
20743     \__coffin_if_exist:NT #1
20744     {
20745         \box_set_eq:NN #1 #2
20746         \__coffin_set_eq_structure:NN #1 #2
20747     }
20748 }
20749 \cs_generate_variant:Nn \coffin_set_eq:NN { c , Nc , cc }

```

(End definition for `\coffin_set_eq:NN`. This function is documented on page 218.)

`\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.

```
20750 \coffin_new:N \c_empty_coffin
20751 \hbox_set:Nn \c_empty_coffin { }
20752 \coffin_new:N \l__coffin_aligned_coffin
20753 \coffin_new:N \l__coffin_aligned_internal_coffin
```

(End definition for `\c_empty_coffin`, `\l__coffin_aligned_coffin`, and `\l__coffin_aligned_internal_coffin`. These variables are documented on page 221.)

`\l_tmpa_coffin` The usual scratch space.

`\l_tmpb_coffin`

```
20754 \coffin_new:N \l_tmpa_coffin
20755 \coffin_new:N \l_tmpb_coffin
```

(End definition for `\l_tmpa_coffin` and `\l_tmpb_coffin`. These variables are documented on page 221.)

38.3 Measuring coffins

`\coffin_dp:N` Coffins are just boxes when it comes to measurement. However, semantically a separate set of functions are required.

`\coffin_ht:N`

```
20756 \cs_new_eq:NN \coffin_dp:N \box_dp:N
```

`\coffin_ht:c`

```
20757 \cs_new_eq:NN \coffin_dp:c \box_dp:c
```

`\coffin_wd:N`

```
20758 \cs_new_eq:NN \coffin_ht:N \box_ht:N
```

`\coffin_wd:c`

```
20759 \cs_new_eq:NN \coffin_ht:c \box_ht:c
```

```
20760 \cs_new_eq:NN \coffin_wd:N \box_wd:N
```

```
20761 \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 220.)

38.4 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.

```
20762 \cs_new_protected:Npn \_coffin_get_pole:NnN #1#2#3
20763 {
20764     \prop_get:cnNF
20765         { l__coffin_poles_ \_int_value:w #1 _prop } {#2} #3
20766     {
20767         \_msg_kernel_error:nxxx { kernel } { unknown-coffin-pole }
20768             {#2} { \token_to_str:N #1 }
20769         \tl_set:Nn #3 { { Opt } { Opt } { Opt } { Opt } }
20770     }
20771 }
```

(End definition for `_coffin_get_pole:NnN`.)

`__coffin_reset_structure:N` Resetting the structure is a simple copy job.

```
20772 \cs_new_protected:Npn \__coffin_reset_structure:N #1
20773 {
20774     \prop_set_eq:cN { l__coffin_corners_ } { \__int_value:w #1 _prop }
20775     \c_coffin_corners_prop
20776     \prop_set_eq:cN { l__coffin_poles_ } { \__int_value:w #1 _prop }
20777     \c_coffin_poles_prop
20778 }
```

(End definition for `__coffin_reset_structure:N`.)

`__coffin_set_eq_structure>NN` Setting coffin structures equal simply means copying the property list.

```
20779 \cs_new_protected:Npn \__coffin_set_eq_structure:NN #1#2
20800 {
20801     \prop_set_eq:cc { l__coffin_corners_ } { \__int_value:w #1 _prop }
20802     { l__coffin_corners_ } { \__int_value:w #2 _prop }
20803     \prop_set_eq:cc { l__coffin_poles_ } { \__int_value:w #1 _prop }
20804     { l__coffin_poles_ } { \__int_value:w #2 _prop }
20805 }
20806 \cs_new_protected:Npn \__coffin_gset_eq_structure:NN #1#2
20807 {
20808     \prop_gset_eq:cc { l__coffin_corners_ } { \__int_value:w #1 _prop }
20809     { l__coffin_corners_ } { \__int_value:w #2 _prop }
20810     \prop_gset_eq:cc { l__coffin_poles_ } { \__int_value:w #1 _prop }
20811     { l__coffin_poles_ } { \__int_value:w #2 _prop }
20812 }
```

(End definition for `__coffin_set_eq_structure>NN` and `__coffin_gset_eq_structure>NN`.)

`\coffin_set_horizontal_pole:Nnn` 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.

```
20813 \cs_new_protected:Npn \coffin_set_horizontal_pole:Nnn #1#2#3
20814 {
20815     \__coffin_if_exist:NT #1
20816     {
20817         \__coffin_set_pole:Nnx #1 {#2}
20818         {
20819             { Opt } { \dim_eval:n {#3} }
20820             { 1000pt } { Opt }
20821         }
20822     }
20823 }
20824 \cs_new_protected:Npn \coffin_set_vertical_pole:Nnn #1#2#3
20825 {
20826     \__coffin_if_exist:NT #1
20827     {
20828         \__coffin_set_pole:Nnx #1 {#2}
20829         {
20830             { \dim_eval:n {#3} } { Opt }
20831             { Opt } { 1000pt }
20832         }
20833     }
20834 }
```

```

20815 \cs_new_protected:Npn \__coffin_set_pole:Nnn #1#2#3
20816   { \prop_put:cnn { l__coffin_poles_ \__int_value:w #1 _prop } {#2} {#3} }
20817 \cs_generate_variant:Nn \coffin_set_horizontal_pole:Nnn { c }
20818 \cs_generate_variant:Nn \coffin_set_vertical_pole:Nnn { c }
20819 \cs_generate_variant:Nn \__coffin_set_pole:Nnn { Nnx }

```

(End definition for `\coffin_set_horizontal_pole:Nnn`, `\coffin_set_vertical_pole:Nnn`, and `__coffin_set_pole:Nnn`. These functions are documented on page 219.)

`__coffin_update_corners:N` Updating the corners of a coffin is straight-forward as at this stage there can be no rotation. So the corners of the content are just those of the underlying TeX box.

```

20820 \cs_new_protected:Npn \__coffin_update_corners:N #1
20821   {
20822     \prop_put:cnx { l__coffin_corners_ \__int_value:w #1 _prop } { tl }
20823       { { Opt } { \dim_eval:n { \box_ht:N #1 } } }
20824     \prop_put:cnx { l__coffin_corners_ \__int_value:w #1 _prop } { tr }
20825       { { \dim_eval:n { \box_wd:N #1 } } { \dim_eval:n { \box_ht:N #1 } } }
20826     \prop_put:cnx { l__coffin_corners_ \__int_value:w #1 _prop } { bl }
20827       { { Opt } { \dim_eval:n { -\box_dp:N #1 } } }
20828     \prop_put:cnx { l__coffin_corners_ \__int_value:w #1 _prop } { br }
20829       { { \dim_eval:n { \box_wd:N #1 } } { \dim_eval:n { -\box_dp:N #1 } } }
20830   }

```

(End definition for `__coffin_update_corners:N`.)

`__coffin_update_poles:N` 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.

```

20831 \cs_new_protected:Npn \__coffin_update_poles:N #1
20832   {
20833     \prop_put:cnx { l__coffin_poles_ \__int_value:w #1 _prop } { hc }
20834     {
20835       { \dim_eval:n { 0.5 \box_wd:N #1 } }
20836       { Opt } { Opt } { 1000pt }
20837     }
20838     \prop_put:cnx { l__coffin_poles_ \__int_value:w #1 _prop } { r }
20839     {
20840       { \dim_eval:n { \box_wd:N #1 } }
20841       { Opt } { Opt } { 1000pt }
20842     }
20843     \prop_put:cnx { l__coffin_poles_ \__int_value:w #1 _prop } { vc }
20844     {
20845       { Opt }
20846       { \dim_eval:n { ( \box_ht:N #1 - \box_dp:N #1 ) / 2 } }
20847       { 1000pt }
20848       { Opt }
20849     }
20850     \prop_put:cnx { l__coffin_poles_ \__int_value:w #1 _prop } { t }
20851     {
20852       { Opt }
20853       { \dim_eval:n { \box_ht:N #1 } }
20854       { 1000pt }
20855       { Opt }

```

```

20856 }
20857 \prop_put:cnx { l__coffin_poles_ \__int_value:w #1 _prop } { b }
20858 {
20859   { Opt }
20860   { \dim_eval:n { -\box_dp:N #1 } }
20861   { 1000pt }
20862   { Opt }
20863 }
20864 }
```

(End definition for `__coffin_update_poles:N`.)

38.5 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.

```

20865 \cs_new_protected:Npn \__coffin_calculate_intersection:Nnn #1#2#3
20866 {
20867   \__coffin_get_pole:NnN #1 {#2} \l__coffin_pole_a_tl
20868   \__coffin_get_pole:NnN #1 {#3} \l__coffin_pole_b_tl
20869   \bool_set_false:N \l__coffin_error_bool
20870   \exp_last_two_unbraced:Nno
20871     \__coffin_calculate_intersection:nnnnnnnn
20872       \l__coffin_pole_a_tl \l__coffin_pole_b_tl
20873   \bool_if:NT \l__coffin_error_bool
20874   {
20875     \__msg_kernel_error:nn { kernel } { no-pole-intersection }
20876     \dim_zero:N \l__coffin_x_dim
20877     \dim_zero:N \l__coffin_y_dim
20878   }
20879 }
```

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' will be zero and a special case is needed.

```

20880 \cs_new_protected:Npn \__coffin_calculate_intersection:nnnnnnnn
20881   #1#2#3#4#5#6#7#8
20882   {
20883     \dim_compare:nNnTF {#3} = { \c_zero_dim }
```

The case where the first pole is vertical. So the x -component of the interaction will be at a . There is then a test on the second pole: if it is also vertical then there is an error.

```

20884 {
20885   \dim_set:Nn \l__coffin_x_dim {#1}
20886   \dim_compare:nNnTF {#7} = \c_zero_dim
20887     { \bool_set_true:N \l__coffin_error_bool }
```

The second pole may still be horizontal, in which case the y -component of the intersection will be b' . If not,

$$y = \frac{d'}{c'}(x - a') + b'$$

with the x -component already known to be #1. This calculation is done as a generalised auxiliary.

```

20888    {
20889        \dim_compare:nNnTF {#8} = \c_zero_dim
20890            { \dim_set:Nn \l_coffin_y_dim {#6} }
20891            {
20892                \__coffin_calculate_intersection_aux:nnnnN
20893                    {#1} {#5} {#6} {#7} {#8} \l_coffin_y_dim
20894            }
20895        }
20896    }

```

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.

```

20897    {
20898        \dim_compare:nNnTF {#4} = \c_zero_dim
20899            {
20900                \dim_set:Nn \l_coffin_y_dim {#2}
20901                \dim_compare:nNnTF {#8} = { \c_zero_dim }
20902                    { \bool_set_true:N \l_coffin_error_bool }
20903            {
20904                \dim_compare:nNnTF {#7} = \c_zero_dim
20905                    { \dim_set:Nn \l_coffin_x_dim {#5} }

```

The formula for the case where the second pole is neither horizontal nor vertical is

$$x = \frac{c'}{d'}(y - b') + a'$$

which is again handled by the same auxiliary.

```

20906    {
20907        \__coffin_calculate_intersection_aux:nnnnN
20908            {#2} {#6} {#5} {#8} {#7} \l_coffin_x_dim
20909        }
20910    }
20911 }

```

The first pole is neither horizontal nor vertical. This still leaves the second pole, which may be a special case. For those possibilities, the calculations are the same as above with the first and second poles interchanged.

```

20912    {
20913        \dim_compare:nNnTF {#7} = \c_zero_dim
20914            {
20915                \dim_set:Nn \l_coffin_x_dim {#5}
20916                \__coffin_calculate_intersection_aux:nnnnN
20917                    {#5} {#1} {#2} {#3} {#4} \l_coffin_y_dim
20918            }
20919    {
20920        \dim_compare:nNnTF {#8} = \c_zero_dim
20921            {
20922                \dim_set:Nn \l_coffin_y_dim {#6}
20923                \__coffin_calculate_intersection_aux:nnnnN
20924                    {#6} {#2} {#1} {#4} {#3} \l_coffin_x_dim
20925            }

```

If none of the special cases apply then there is still a need to check that there is a unique intersection between the two pole. This is the case if they have different slopes.

```

20926      {
20927          \fp_set:Nn \l__coffin_slope_x_fp
20928              { \dim_to_fp:n {#4} / \dim_to_fp:n {#3} }
20929          \fp_set:Nn \l__coffin_slope_y_fp
20930              { \dim_to_fp:n {#8} / \dim_to_fp:n {#7} }
20931          \fp_compare:nNnTF
20932              \l__coffin_slope_x_fp = \l__coffin_slope_y_fp
20933                  { \bool_set_true:N \l__coffin_error_bool }

```

All of the tests pass, so there is the full complexity of the calculation:

$$x = \frac{a(d/c) - a'(d'/c') - b + b'}{(d/c) - (d'/c')}$$

and noting that the two ratios are already worked out from the test just performed. There is quite a bit of shuffling from dimensions to floating points in order to do the work. The y -values is then worked out using the standard auxiliary starting from the x -position.

```

20934      {
20935          \dim_set:Nn \l__coffin_x_dim
20936              {
20937                  \fp_to_dim:n
20938                      {
20939                          (
20940                              \dim_to_fp:n {#1} * \l__coffin_slope_x_fp
20941                              - ( \dim_to_fp:n {#5} * \l__coffin_slope_y_fp )
20942                                  - \dim_to_fp:n {#2}
20943                                  + \dim_to_fp:n {#6}
20944                          )
20945                          /
20946                          ( \l__coffin_slope_x_fp - \l__coffin_slope_y_fp )
20947                      }
20948                  }
20949                  \__coffin_calculate_intersection_aux:nnnnnN
20950                      { \l__coffin_x_dim }
20951                      {#5} {#6} {#8} {#7} \l__coffin_y_dim
20952                  }
20953              }
20954          }
20955      }
20956  }
20957 }

```

The formula for finding the intersection point is in most cases the same. The formula here is

$$\#6 = \#4 \cdot \left(\frac{\#1 - \#2}{\#5} \right) \#3$$

Thus #4 and #5 should be the directions of the pole while #2 and #3 are co-ordinates.

```

20958 \cs_new_protected:Npn \__coffin_calculate_intersection_aux:nnnnN
20959     #1#2#3#4#5#6
20960     {
20961         \dim_set:Nn #6

```

```

20962 {
20963   \fp_to_dim:n
20964   {
20965     \dim_to_fp:n {#4} *
20966     ( \dim_to_fp:n {#1} - \dim_to_fp:n {#2} ) /
20967     \dim_to_fp:n {#5}
20968     + \dim_to_fp:n {#3}
20969   }
20970 }
20971 }
```

(End definition for `_coffin_calculate_intersection:Nnn`, `_coffin_calculate_intersection:nnnnnnnn`, and `_coffin_calculate_intersection_aux:nnnnnN`.)

38.6 Aligning and typesetting of coffins

`\coffin_join:NnnNnnnn`
`\coffin_join:cnnNnnnn`
`\coffin_join:Nnnccnnnn`
`\coffin_join:cnnccnnnn`

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 will have all of its handles reset to standard values. First, the more basic alignment function is used to get things started.

```

20972 \cs_new_protected:Npn \coffin_join:NnnNnnnn #1#2#3#4#5#6#7#8
20973 {
20974   \_coffin_align:NnnNnnnnN
20975     #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 will 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.

```

20976 \hbox_set:Nn \l__coffin_aligned_coffin
20977 {
20978   \dim_compare:nNnT { \l__coffin_offset_x_dim } < \c_zero_dim
20979     { \tex_kern:D -\l__coffin_offset_x_dim }
20980   \hbox_unpack:N \l__coffin_aligned_coffin
20981   \dim_set:Nn \l__coffin_internal_dim
20982     { \l__coffin_offset_x_dim - \box_wd:N #1 + \box_wd:N #4 }
20983   \dim_compare:nNnT \l__coffin_internal_dim < \c_zero_dim
20984     { \tex_kern:D -\l__coffin_internal_dim }
20985 }
```

The coffin structure is reset, and the corners are cleared: only those from the two parent coffins are needed.

```

20986 \_coffin_reset_structure:N \l__coffin_aligned_coffin
20987 \prop_clear:c
20988   { \l__coffin_corners_ \_int_value:w \l__coffin_aligned_coffin _ prop }
20989 \_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 will then depend on whether any shift was needed.

```

20990 \dim_compare:nNnTF \l__coffin_offset_x_dim < \c_zero_dim
20991   {
20992     \_coffin_offset_poles:Nnn #1 { -\l__coffin_offset_x_dim } { Opt }
20993     \_coffin_offset_poles:Nnn #4 { Opt } { \l__coffin_offset_y_dim }
```

```

20994     \__coffin_offset_corners:Nnn #1 { -\l__coffin_offset_x_dim } { Opt }
20995     \__coffin_offset_corners:Nnn #4 { Opt } { \l__coffin_offset_y_dim }
20996   }
20997   {
20998     \__coffin_offset_poles:Nnn #1 { Opt } { Opt }
20999     \__coffin_offset_poles:Nnn #4
21000       { \l__coffin_offset_x_dim } { \l__coffin_offset_y_dim }
21001     \__coffin_offset_corners:Nnn #1 { Opt } { Opt }
21002     \__coffin_offset_corners:Nnn #4
21003       { \l__coffin_offset_x_dim } { \l__coffin_offset_y_dim }
21004   }
21005   \__coffin_update_vertical_poles:NNN #1 #4 \l__coffin_aligned_coffin
21006   \coffin_set_eq:NN #1 \l__coffin_aligned_coffin
21007 }
21008 \cs_generate_variant:Nn \coffin_join:NnnNnnnn { c , Nnnc , cnnc }
```

(End definition for `\coffin_join:NnnNnnnn`. This function is documented on page 220.)

`\coffin_attach:NnnNnnnn`
`\coffin_attach:cnnNnnnn`
`\coffin_attach:Nnnccnnnn`
`\coffin_attach:cnnccnnnn`

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.

```

21009 \cs_new_protected:Npn \coffin_attach:NnnNnnnn #1#2#3#4#5#6#7#8
21010   {
21011     \__coffin_align:NnnNnnnnN
21012       #1 {#2} {#3} #4 {#5} {#6} {#7} {#8} \l__coffin_aligned_coffin
21013     \box_set_ht:Nn \l__coffin_aligned_coffin { \box_ht:N #1 }
21014     \box_set_dp:Nn \l__coffin_aligned_coffin { \box_dp:N #1 }
21015     \box_set_wd:Nn \l__coffin_aligned_coffin { \box_wd:N #1 }
21016     \__coffin_reset_structure:N \l__coffin_aligned_coffin
21017     \prop_set_eq:cc
21018       { \__coffin_corners_ \__int_value:w \l__coffin_aligned_coffin _prop }
21019       { \__coffin_corners_ \__int_value:w #1 _prop }
21020     \__coffin_update_poles:N \l__coffin_aligned_coffin
21021     \__coffin_offset_poles:Nnn #1 { Opt } { Opt }
21022     \__coffin_offset_poles:Nnn #4
21023       { \l__coffin_offset_x_dim } { \l__coffin_offset_y_dim }
21024     \__coffin_update_vertical_poles:NNN #1 #4 \l__coffin_aligned_coffin
21025     \coffin_set_eq:NN #1 \l__coffin_aligned_coffin
21026   }
21027 \cs_new_protected:Npn \coffin_attach_mark:NnnNnnnn #1#2#3#4#5#6#7#8
21028   {
21029     \__coffin_align:NnnNnnnnN
21030       #1 {#2} {#3} #4 {#5} {#6} {#7} {#8} \l__coffin_aligned_coffin
21031     \box_set_ht:Nn \l__coffin_aligned_coffin { \box_ht:N #1 }
21032     \box_set_dp:Nn \l__coffin_aligned_coffin { \box_dp:N #1 }
21033     \box_set_wd:Nn \l__coffin_aligned_coffin { \box_wd:N #1 }
21034     \box_set_eq:NN #1 \l__coffin_aligned_coffin
21035   }
21036 \cs_generate_variant:Nn \coffin_attach:NnnNnnnn { c , Nnnc , cnnc }
```

(End definition for `\coffin_attach:NnnNnnnn` and `\coffin_attach_mark:NnnNnnnn`. These functions are documented on page 219.)

__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 will depend on how the bounding box is being handled.

```

21037 \cs_new_protected:Npn \_\_coffin_align:NnnNnnnnN #1#2#3#4#5#6#7#8#9
21038 {
21039     \_\_coffin_calculate_intersection:Nnn #4 {#5} {#6}
21040     \dim_set:Nn \l\_\_coffin_x_prime_dim { \l\_\_coffin_x_dim }
21041     \dim_set:Nn \l\_\_coffin_y_prime_dim { \l\_\_coffin_y_dim }
21042     \_\_coffin_calculate_intersection:Nnn #1 {#2} {#3}
21043     \dim_set:Nn \l\_\_coffin_offset_x_dim
21044         { \l\_\_coffin_x_dim - \l\_\_coffin_x_prime_dim + #7 }
21045     \dim_set:Nn \l\_\_coffin_offset_y_dim
21046         { \l\_\_coffin_y_dim - \l\_\_coffin_y_prime_dim + #8 }
21047     \hbox_set:Nn \l\_\_coffin_aligned_internal_coffin
21048     {
21049         \box_use:N #1
21050         \tex_kern:D -\box_wd:N #1
21051         \tex_kern:D \l\_\_coffin_offset_x_dim
21052             \box_move_up:nn { \l\_\_coffin_offset_y_dim } { \box_use:N #4 }
21053     }
21054     \coffin_set_eq:NN #9 \l\_\_coffin_aligned_internal_coffin
21055 }
```

(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 to 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.

```

21056 \cs_new_protected:Npn \_\_coffin_offset_poles:Nnn #1#2#3
21057 {
21058     \prop_map_inline:cn { l\_\_coffin_poles_ \_\_int_value:w #1 _prop }
21059         { \_\_coffin_offset_pole:Nnnnnnn #1 {##1} ##2 {##2} {##3} }
21060 }
21061 \cs_new_protected:Npn \_\_coffin_offset_pole:Nnnnnnn #1#2#3#4#5#6#7#8
21062 {
21063     \dim_set:Nn \l\_\_coffin_x_dim { #3 + #7 }
21064     \dim_set:Nn \l\_\_coffin_y_dim { #4 + #8 }
21065     \tl_if_in:nnTF {##2} { - }
21066         { \tl_set:Nn \l\_\_coffin_internal_tl { {##2} } }
21067         { \tl_set:Nn \l\_\_coffin_internal_tl { { #1 - #2 } } }
21068     \exp_last_unbraced:NNo \_\_coffin_set_pole:Nnx \l\_\_coffin_aligned_coffin
21069         { \l\_\_coffin_internal_tl }
21070     {
21071         { \dim_use:N \l\_\_coffin_x_dim } { \dim_use:N \l\_\_coffin_y_dim }
21072             {##5} {##6}
```

```

21073     }
21074   }

```

(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.

```

21075 \cs_new_protected:Npn \__coffin_offset_corners:Nnn #1#2#3
21076   {
21077     \prop_map_inline:cn { l__coffin_corners_ \__int_value:w #1 _prop }
21078       { \__coffin_offset_corner:Nnnnn #1 {##1} ##2 {#2} {#3} }
21079   }
21080 \cs_new_protected:Npn \__coffin_offset_corner:Nnnnn #1#2#3#4#5#6
21081   {
21082     \prop_put:cnx
21083       { l__coffin_corners_ \__int_value:w \l__coffin_aligned_coffin _prop }
21084         { #1 - #2 }
21085         {
21086           { \dim_eval:n { #3 + #5 } }
21087           { \dim_eval:n { #4 + #6 } }
21088         }
21089   }

```

(End definition for `__coffin_offset_corners:Nnn` and `__coffin_offset_corner:Nnnnn`.)

`__coffin_update_vertical_poles:NNN`
`__coffin_update_T:nnnnnnnnnN`
`__coffin_update_B:nnnnnnnnN`

The T and B poles will 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.

```

21090 \cs_new_protected:Npn \__coffin_update_vertical_poles:NNN #1#2#3
21091   {
21092     \__coffin_get_pole:NnN #3 { #1 -T } \l__coffin_pole_a_t1
21093     \__coffin_get_pole:NnN #3 { #2 -T } \l__coffin_pole_b_t1
21094     \exp_last_two_unbraced:Noo \__coffin_update_T:nnnnnnnnnN
21095       \l__coffin_pole_a_t1 \l__coffin_pole_b_t1 #3
21096     \__coffin_get_pole:NnN #3 { #1 -B } \l__coffin_pole_a_t1
21097     \__coffin_get_pole:NnN #3 { #2 -B } \l__coffin_pole_b_t1
21098     \exp_last_two_unbraced:Noo \__coffin_update_B:nnnnnnnnN
21099       \l__coffin_pole_a_t1 \l__coffin_pole_b_t1 #3
21100   }
21101 \cs_new_protected:Npn \__coffin_update_T:nnnnnnnnnN #1#2#3#4#5#6#7#8#9
21102   {
21103     \dim_compare:nNnTF {#2} < {#6}
21104     {
21105       \__coffin_set_pole:Nnx #9 { T }
21106         { { Opt } {#6} { 1000pt } { Opt } }
21107     }
21108     {
21109       \__coffin_set_pole:Nnx #9 { T }
21110         { { Opt } {#2} { 1000pt } { Opt } }
21111     }
21112   }
21113 \cs_new_protected:Npn \__coffin_update_B:nnnnnnnnN #1#2#3#4#5#6#7#8#9
21114   {
21115     \dim_compare:nNnTF {#2} < {#6}

```

```

21116   {
21117     \__coffin_set_pole:Nnx #9 { B }
21118       { { Opt } {#2} { 1000pt } { Opt } }
21119   }
21120   {
21121     \__coffin_set_pole:Nnx #9 { B }
21122       { { Opt } {#6} { 1000pt } { Opt } }
21123   }
21124 }
```

(End definition for `__coffin_update_vertical_poles:NNN`, `__coffin_update_T:nnnnnnnnN`, and `__coffin_update_B:nnnnnnnnN`.)

\coffin_typeset:Nnnnn
\coffin_typeset:cnnnn

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.

```

21125 \cs_new_protected:Npn \coffin_typeset:Nnnnn #1#2#3#4#5
21126   {
21127     \hbox_unpack:N \c_empty_box
21128     \__coffin_align:NnnNnnnnN \c_empty_coffin { H } { 1 }
21129       #1 {#2} {#3} {#4} {#5} \l__coffin_aligned_coffin
21130     \box_use:N \l__coffin_aligned_coffin
21131   }
21132 \cs_generate_variant:Nn \coffin_typeset:Nnnnn { c }
```

(End definition for `\coffin_typeset:Nnnnn`. This function is documented on page 220.)

38.7 Coffin diagnostics

\l__coffin_display_coffin
\l__coffin_display_coord_coffin
\l__coffin_display_pole_coffin

Used for printing coffins with data structures attached.

```

21133 \coffin_new:N \l__coffin_display_coffin
21134 \coffin_new:N \l__coffin_display_coord_coffin
21135 \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.

```

21136 \prop_new:N \l__coffin_display_handles_prop
21137 \prop_put:Nnn \l__coffin_display_handles_prop { tl }
21138   { { b } { r } { -1 } { 1 } }
21139 \prop_put:Nnn \l__coffin_display_handles_prop { thc }
21140   { { b } { hc } { 0 } { 1 } }
21141 \prop_put:Nnn \l__coffin_display_handles_prop { tr }
21142   { { b } { 1 } { 1 } { 1 } }
21143 \prop_put:Nnn \l__coffin_display_handles_prop { vcl }
21144   { { vc } { r } { -1 } { 0 } }
21145 \prop_put:Nnn \l__coffin_display_handles_prop { vhc }
21146   { { vc } { hc } { 0 } { 0 } }
21147 \prop_put:Nnn \l__coffin_display_handles_prop { vcr }
21148   { { vc } { 1 } { 1 } { 0 } }
21149 \prop_put:Nnn \l__coffin_display_handles_prop { bl }
21150   { { t } { r } { -1 } { -1 } }
```

```

21151 \prop_put:Nnn \l__coffin_display_handles_prop { bhc }
21152   { { t } { hc } { 0 } { -1 } }
21153 \prop_put:Nnn \l__coffin_display_handles_prop { br }
21154   { { t } { 1 } { 1 } { -1 } }
21155 \prop_put:Nnn \l__coffin_display_handles_prop { Tl }
21156   { { t } { r } { -1 } { -1 } }
21157 \prop_put:Nnn \l__coffin_display_handles_prop { Thc }
21158   { { t } { hc } { 0 } { -1 } }
21159 \prop_put:Nnn \l__coffin_display_handles_prop { Tr }
21160   { { t } { 1 } { 1 } { -1 } }
21161 \prop_put:Nnn \l__coffin_display_handles_prop { Hl }
21162   { { vc } { r } { -1 } { 1 } }
21163 \prop_put:Nnn \l__coffin_display_handles_prop { Hhc }
21164   { { vc } { hc } { 0 } { 1 } }
21165 \prop_put:Nnn \l__coffin_display_handles_prop { Hr }
21166   { { vc } { 1 } { 1 } { 1 } }
21167 \prop_put:Nnn \l__coffin_display_handles_prop { Bl }
21168   { { b } { r } { -1 } { -1 } }
21169 \prop_put:Nnn \l__coffin_display_handles_prop { Bhc }
21170   { { b } { hc } { 0 } { -1 } }
21171 \prop_put:Nnn \l__coffin_display_handles_prop { Br }
21172   { { b } { 1 } { 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.

```

21173 \dim_new:N \l__coffin_display_offset_dim
21174 \dim_set:Nn \l__coffin_display_offset_dim { 2pt }

```

(End definition for `\l__coffin_display_offset_dim`.)

`\l__coffin_display_x_dim` As the intersections of poles have to be calculated to find which ones to print, there is
`\l__coffin_display_y_dim` a need to avoid repetition. This is done by saving the intersection into two dedicated values.

```

21175 \dim_new:N \l__coffin_display_x_dim
21176 \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.

```

21177 \prop_new:N \l__coffin_display_poles_prop

```

(End definition for `\l__coffin_display_poles_prop`.)

`\l__coffin_display_font_t1` Stores the settings used to print coffin data: this keeps things flexible.

```

21178 \tl_new:N \l__coffin_display_font_t1
21179 (*initex)
21180 \tl_set:Nn \l__coffin_display_font_t1 { } % TODO
21181 (/initex)
21182 (*package)
21183 \tl_set:Nn \l__coffin_display_font_t1 { \sffamily \tiny }
21184 (/package)

```

(End definition for `\l__coffin_display_font_t1`.)

```
\coffin_mark_handle:Nnnn
\coffin_mark_handle:cnnn
\_\coffin_mark_handle_aux:nnnnNnn
```

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.

```
21185 \cs_new_protected:Npn \coffin_mark_handle:Nnnn #1#2#3#4
21186 {
21187     \hcoffin_set:Nn \l__coffin_display_pole_coffin
21188     {
21189         (*initex)
21190         \hbox:n { \tex_vrule:D width 1pt height 1pt \scan_stop: } % TODO
21191     
```

$$\langle /initex \rangle$$

$$\langle *package \rangle$$

$$\color{#4} \\rule { 1pt } { 1pt }$$

$$\langle /package \rangle$$

$$\langle /initex \rangle$$

$$\langle *package \rangle$$

$$\color{#4}$$

$$\langle /package \rangle$$

$$\l__coffin_display_font_tl$$

$$(\tl_to_str:n { #2 , #3 })$$

$$\langle /initex \rangle$$

$$\langle *package \rangle$$

$$\color{#4}$$

$$\langle /package \rangle$$

$$\l__coffin_display_handles_prop$$

$${ #2 #3 } \l__coffin_internal_tl$$

$$\quark_if_no_value:NTF \l__coffin_internal_tl$$

$$\langle /initex \rangle$$

$$\langle *package \rangle$$

$$\color{#4}$$

$$\langle /package \rangle$$

$$\l__coffin_display_handles_prop$$

$${ #3 #2 } \l__coffin_internal_tl$$

$$\quark_if_no_value:NTF \l__coffin_internal_tl$$

$$\langle /initex \rangle$$

$$\langle *package \rangle$$

$$\color{#4}$$

$$\langle /package \rangle$$

$$\l__coffin_display_handles_prop$$

$${ #2 #3 } \l__coffin_internal_tl$$

$$\quark_if_no_value:NTF \l__coffin_internal_tl$$

$$\langle /initex \rangle$$

$$\langle *package \rangle$$

$$\color{#4}$$

$$\langle /package \rangle$$

$$\l__coffin_mark_handle_aux:nnnnNnn$$

$$\l__coffin_internal_tl #1 {#2} {#3}$$

$$\langle /initex \rangle$$

$$\langle *package \rangle$$

$$\color{#4}$$

$$\langle /package \rangle$$

$$\exp_last_unbraced:No \l__coffin_mark_handle_aux:nnnnNnn$$

$$\l__coffin_internal_tl #1 {#2} {#3}$$

$$\langle /initex \rangle$$

$$\langle *package \rangle$$

$$\color{#4}$$

$$\langle /package \rangle$$

$$\l__coffin_mark_handle_aux:nnnnNnn #1#2#3#4#5#6#7$$

$$\langle /initex \rangle$$

$$\langle *package \rangle$$

$$\color{#4}$$

$$\langle /package \rangle$$

$$\l__coffin_mark_handle_aux:nnnnNnn #5 {#6} {#7}$$

```

21235   \l__coffin_display_coord_coffin {#1} {#2}
21236   { #3 \l__coffin_display_offset_dim }
21237   { #4 \l__coffin_display_offset_dim }
21238 }
21239 \cs_generate_variant:Nn \coffin_mark_handle:Nnnn { c }

(End definition for \coffin_mark_handle:Nnnn and \__coffin_mark_handle_aux:nnnnNnn. These functions are documented on page 221.)
```

\coffin_display_handles:Nn
\coffin_display_handles:cn
\coffin_display_handles_aux:nnnn
\coffin_display_handles_aux:nnnn
\coffin_display_attach:NNNN

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.

```

21240 \cs_new_protected:Npn \coffin_display_handles:Nn #1#2
21241 {
21242   \hcoffin_set:Nn \l__coffin_display_pole_coffin
21243   {
21244     (*initex)
21245     \hbox:n { \tex_vrule:D width 1pt height 1pt \scan_stop: } % TODO
21246   (/initex)
21247   (*package)
21248     \color {#2}
21249     \rule { 1pt } { 1pt }
21250   (/package)
21251   }
21252   \prop_set_eq:Nc \l__coffin_display_poles_prop
21253   { l__coffin_poles_ \int_value:w #1 _prop }
21254   \__coffin_get_pole:NnN #1 { H } \l__coffin_pole_a_t1
21255   \__coffin_get_pole:NnN #1 { T } \l__coffin_pole_b_t1
21256   \tl_if_eq:NNT \l__coffin_pole_a_t1 \l__coffin_pole_b_t1
21257   { \prop_remove:Nn \l__coffin_display_poles_prop { T } }
21258   \__coffin_get_pole:NnN #1 { B } \l__coffin_pole_b_t1
21259   \tl_if_eq:NNT \l__coffin_pole_a_t1 \l__coffin_pole_b_t1
21260   { \prop_remove:Nn \l__coffin_display_poles_prop { B } }
21261   \coffin_set_eq:NN \l__coffin_display_coffin #1
21262   \prop_map_inline:Nn \l__coffin_display_poles_prop
21263   {
21264     \prop_remove:Nn \l__coffin_display_poles_prop {##1}
21265     \__coffin_display_handles_aux:nnnnnn {##1} ##2 {##2}
21266   }
21267   \box_use:N \l__coffin_display_coffin
21268 }
```

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.

```

21269 \cs_new_protected:Npn \__coffin_display_handles_aux:nnnnnn #1#2#3#4#5#6
21270 {
21271   \prop_map_inline:Nn \l__coffin_display_poles_prop
21272   {
21273     \bool_set_false:N \l__coffin_error_bool
21274     \__coffin_calculate_intersection:nnnnnnnn {##2} {##3} {##4} {##5} ##2
21275     \bool_if:NF \l__coffin_error_bool
21276     {
```

```

21277   \dim_set:Nn \l__coffin_display_x_dim { \l__coffin_x_dim }
21278   \dim_set:Nn \l__coffin_display_y_dim { \l__coffin_y_dim }
21279   \__coffin_display_attach:Nnnnn
21280     \l__coffin_display_pole_coffin { hc } { vc }
21281     { Opt } { Opt }
21282   \hcoffin_set:Nn \l__coffin_display_coord_coffin
21283   {
21284     /*initex}
21285     % TODO
21286   
```

```

21287   <*package>
21288   
```

```

21289   </package>
21290   
```

```

21291   
```

```

21292   
```

```

21293   
```

```

21294   
```

```

21295   
```

```

21296   
```

```

21297   
```

```

21298   
```

```

21299   
```

```

21300   
```

```

21301   
```

```

21302   
```

```

21303   
```

```

21304   
```

```

21305   
```

```

21306   
```

```

21307   
```

```

21308   
```

```

21309   
```

```

21310   
```

```

21311   
```

```

21312   
```

```

21313   
```

```

21314   
```

```

21315   
```

```

21316   
```

```

21317   
```

```

21318 \cs_new_protected:Npn \__coffin_display_handles_aux:nnnn #1#2#3#4
21319   
```

```

21320   
```

```

21321   
```

```

21322   
```

```

21323   
```

```

21324   
```

```

21325 \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.

```

21326 \cs_new_protected:Npn \__coffin_display_attach:Nnnnn #1#2#3#4#5

```

```

21327  {
21328    \__coffin_calculate_intersection:Nnn #1 {#2} {#3}
21329    \dim_set:Nn \l__coffin_x_prime_dim { \l__coffin_x_dim }
21330    \dim_set:Nn \l__coffin_y_prime_dim { \l__coffin_y_dim }
21331    \dim_set:Nn \l__coffin_offset_x_dim
21332      { \l__coffin_display_x_dim - \l__coffin_x_prime_dim + #4 }
21333    \dim_set:Nn \l__coffin_offset_y_dim
21334      { \l__coffin_display_y_dim - \l__coffin_y_prime_dim + #5 }
21335    \hbox_set:Nn \l__coffin_aligned_coffin
21336      {
21337        \box_use:N \l__coffin_display_coffin
21338        \tex_kern:D -\box_wd:N \l__coffin_display_coffin
21339        \tex_kern:D \l__coffin_offset_x_dim
21340        \box_move_up:nn { \l__coffin_offset_y_dim } { \box_use:N #1 }
21341      }
21342    \box_set_ht:Nn \l__coffin_aligned_coffin
21343      { \box_ht:N \l__coffin_display_coffin }
21344    \box_set_dp:Nn \l__coffin_aligned_coffin
21345      { \box_dp:N \l__coffin_display_coffin }
21346    \box_set_wd:Nn \l__coffin_aligned_coffin
21347      { \box_wd:N \l__coffin_display_coffin }
21348    \box_set_eq:NN \l__coffin_display_coffin \l__coffin_aligned_coffin
21349  }

```

(End definition for `\coffin_display_handles:Nn` and others. These functions are documented on page 220.)

\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.

```

21350 \cs_new_protected:Npn \coffin_show_structure:N #1
21351  {
21352    \__coffin_if_exist:NT #1
21353    {
21354      \__msg_show_pre:nnxxxx { LaTeX / kernel } { show-coffin }
21355      { \token_to_str:N #1 }
21356      { \dim_eval:n { \coffin_ht:N #1 } }
21357      { \dim_eval:n { \coffin_dp:N #1 } }
21358      { \dim_eval:n { \coffin_wd:N #1 } }
21359      \__msg_show_wrap:n
21360      {
21361        \prop_map_function:cN
21362          { \l__coffin_poles_ \__int_value:w #1 _prop }
21363          \__msg_show_item_unbraced:nn
21364      }
21365    }
21366  }
21367 \cs_generate_variant:Nn \coffin_show_structure:N { c }

```

(End definition for `\coffin_show_structure:N`. This function is documented on page 221.)

\coffin_log_structure:N Redirect output of `\coffin_show_structure:N` to the log.

```

\coffin_log_structure:c
21368 \cs_new_protected:Npn \coffin_log_structure:N
21369  { \__msg_log_next: \coffin_show_structure:N }
21370 \cs_generate_variant:Nn \coffin_log_structure:N { c }

```

(End definition for `\coffin_log_structure:N`. This function is documented on page 221.)

38.8 Messages

```
21371 \__msg_kernel_new:nnn { kernel } { no-pole-intersection }
21372   { No-intersection-between-coffin-poles. }
21373   {
21374     \c__msg_error_text_tl
21375     LaTeX-was-asked-to-find-the-intersection-between-two-poles,-
21376     but-they-do-not-have-a-unique-meeting-point:-
21377     the-value-(0-pt,~0-pt)-will-be-used.
21378   }
21379 \__msg_kernel_new:nnn { kernel } { unknown-coffin }
21380   { Unknown-coffin-'#1'. }
21381   { The-coffin-'#1'-was-never-defined. }
21382 \__msg_kernel_new:nnn { kernel } { unknown-coffin-pole }
21383   { Pole-'#1'-unknown-for-coffin-'#2'. }
21384   {
21385     \c__msg_error_text_tl
21386     LaTeX-was-asked-to-find-a-typesetting-pole-for-a-coffin,-
21387     but-either-the-coffin-does-not-exist-or-the-pole-name-is-wrong.
21388   }
21389 \__msg_kernel_new:nnn { kernel } { show-coffin }
21390   {
21391     Size-of-coffin-'#1' : \\
21392     > ~ ht==#2 \\
21393     > ~ dp==#3 \\
21394     > ~ wd==#4 \\
21395     Poles-of-coffin-'#1' :
21396   }
21397 ⟨/initex | package⟩
```

39 I3color Implementation

```
21398 ⟨*initex | package⟩
```

\color_group_begin: Grouping for color is almost the same as using the basic `\group_begin:` and `\group_end:` functions. However, in vertical mode the end-of-group needs a `\par`, which in horizontal mode does nothing.

```
21399 \cs_new_eq:NN \color_group_begin: \group_begin:
21400 \cs_new_protected:Npn \color_group_end:
21401   {
21402     \par
21403     \group_end:
21404   }
```

(End definition for `\color_group_begin:` and `\color_group_end:`. These functions are documented on page 222.)

\color_ensure_current: A driver-independent wrapper for setting the foreground color to the current color “now”.

```
21405 \cs_new_protected:Npn \color_ensure_current:
21406   { \__driver_color_ensure_current: }
```

(End definition for `\color_ensure_current:`. This function is documented on page 222.)

```
21407 ⟨/initex | package⟩
```

40 I3sys implementation

21408 `(*initex | package)`

40.1 The name of the job

`\c_sys_jobname_str`

Inherited from the L^AT_EX3 name for the primitive: this needs to actually contain the text of the job name rather than the name of the primitive, of course.

```
21409 <*initex>
21410 \tex_everyjob:D \exp_after:wN
21411 {
21412   \tex_the:D \tex_everyjob:D
21413   \str_const:Nx \c_sys_jobname_str { \tex_jobname:D }
21414 }
21415 </initex>
21416 /*package*/
21417 \str_const:Nx \c_sys_jobname_str { \tex_jobname:D }
21418 </package>
```

(End definition for `\c_sys_jobname_str`. This variable is documented on page 223.)

40.2 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

```
21419 \int_const:Nn \c_sys_minute_int
21420   { \int_mod:nn { \tex_time:D } { 60 } }
21421 \int_const:Nn \c_sys_hour_int
21422   { \int_div_truncate:nn { \tex_time:D } { 60 } }
21423 \int_const:Nn \c_sys_day_int { \tex_day:D }
21424 \int_const:Nn \c_sys_month_int { \tex_month:D }
21425 \int_const:Nn \c_sys_year_int { \tex_year:D }
```

(End definition for `\c_sys_minute_int` and others. These variables are documented on page 223.)

40.3 Detecting the engine

`\sys_if_engine_luatex_p:`
`\sys_if_engine_luatex:TF`
`\sys_if_engine_pdftex_p:`
`\sys_if_engine_pdftex:TF`

Set up the engine tests on the basis exactly one test should be true. Mainly a case of looking for the appropriate marker primitive. For upT_EX, there is a complexity in that setting `-kanji-internal=sjis` or `-kanji-internal=euc` effective makes it more like pT_EX. In those cases we therefore report pT_EX rather than upT_EX.

```
21426 \clist_map_inline:nn { lua , pdf , p , up , xe }
21427 {
21428   \cs_new_eq:cN { sys_if_engine_ #1 tex:T } \use_none:n
21429   \cs_new_eq:cN { sys_if_engine_ #1 tex:F } \use:n
21430   \cs_new_eq:cN { sys_if_engine_ #1 tex:TF } \use_i:nn
21431   \cs_new_eq:cN { sys_if_engine_ #1 tex_p: } \c_false_bool
21432 }
21433 \cs_if_exist:NT \luatex_luatexversion:D
21434 {
21435   \cs_gset_eq:NN \sys_if_engine_luatex:T \use:n
21436   \cs_gset_eq:NN \sys_if_engine_luatex:F \use_none:n
21437   \cs_gset_eq:NN \sys_if_engine_luatex:TF \use_i:nn
21438   \cs_gset_eq:NN \sys_if_engine_luatex_p: \c_true_bool
21439   \str_const:Nn \c_sys_engine_str { luatex }
```

```

21440     }
21441 \cs_if_exist:NT \pdftex_pdftexversion:D
21442 {
21443     \cs_gset_eq:NN \sys_if_engine_pdftex:T \use:n
21444     \cs_gset_eq:NN \sys_if_engine_pdftex:F \use_none:n
21445     \cs_gset_eq:NN \sys_if_engine_pdftex:TF \use_i:nn
21446     \cs_gset_eq:NN \sys_if_engine_pdftex_p: \c_true_bool
21447     \str_const:Nn \c_sys_engine_str { pdftex }
21448 }
21449 \cs_if_exist:NT \ptex_kanjiskip:D
21450 {
21451     \bool_lazy_and:nTF
21452     { \cs_if_exist_p:N \uptex_disablecjktoken:D }
21453     { \int_compare_p:nNn { \ptex_jis:D "2121 } = { "3000 } }
21454     {
21455         \cs_gset_eq:NN \sys_if_engine_uptex:T \use:n
21456         \cs_gset_eq:NN \sys_if_engine_uptex:F \use_none:n
21457         \cs_gset_eq:NN \sys_if_engine_uptex:TF \use_i:nn
21458         \cs_gset_eq:NN \sys_if_engine_uptex_p: \c_true_bool
21459         \str_const:Nn \c_sys_engine_str { uptex }
21460     }
21461 }
21462     \cs_gset_eq:NN \sys_if_engine_ptex:T \use:n
21463     \cs_gset_eq:NN \sys_if_engine_ptex:F \use_none:n
21464     \cs_gset_eq:NN \sys_if_engine_ptex:TF \use_i:nn
21465     \cs_gset_eq:NN \sys_if_engine_ptex_p: \c_true_bool
21466     \str_const:Nn \c_sys_engine_str { ptex }
21467 }
21468 }
21469 \cs_if_exist:NT \xetex_XeTeXversion:D
21470 {
21471     \cs_gset_eq:NN \sys_if_engine_xetex:T \use:n
21472     \cs_gset_eq:NN \sys_if_engine_xetex:F \use_none:n
21473     \cs_gset_eq:NN \sys_if_engine_xetex:TF \use_i:nn
21474     \cs_gset_eq:NN \sys_if_engine_xetex_p: \c_true_bool
21475     \str_const:Nn \c_sys_engine_str { xetex }
21476 }

```

(End definition for `\sys_if_engine_luatex:TF` and others. These functions are documented on page [223](#).)

40.4 Detecting the output

`\sys_if_output_dvi_p:` This is a simple enough concept: the two views here are complementary.

```

21477 \int_compare:nNnTF
21478 { \cs_if_exist_use:N \pdftex_pdfoutput:D { 0 } } > { 0 }
21479 {
21480     \cs_new_eq:NN \sys_if_output_dvi:T \use_none:n
21481     \cs_new_eq:NN \sys_if_output_dvi:F \use:n
21482     \cs_new_eq:NN \sys_if_output_dvi:TF \use_i:nn
21483     \cs_new_eq:NN \sys_if_output_dvi_p: \c_false_bool
21484     \cs_new_eq:NN \sys_if_output_pdf:T \use:n
21485     \cs_new_eq:NN \sys_if_output_pdf:F \use_none:n
21486     \cs_new_eq:NN \sys_if_output_pdf:TF \use_i:nn

```

```

21487   \cs_new_eq:NN \sys_if_output_pdf_p: \c_true_bool
21488   \str_const:Nn \c_sys_output_str { pdf }
21489 }
21490 {
21491   \cs_new_eq:NN \sys_if_output_dvi:T \use:n
21492   \cs_new_eq:NN \sys_if_output_dvi:F \use:none:n
21493   \cs_new_eq:NN \sys_if_output_dvi:TF \use_i:nn
21494   \cs_new_eq:NN \sys_if_output_dvi_p: \c_true_bool
21495   \cs_new_eq:NN \sys_if_output_pdf:T \use:none:n
21496   \cs_new_eq:NN \sys_if_output_pdf:F \use:n
21497   \cs_new_eq:NN \sys_if_output_pdf:TF \use_ii:nn
21498   \cs_new_eq:NN \sys_if_output_pdf_p: \c_false_bool
21499   \str_const:Nn \c_sys_output_str { dvi }
21500 }

```

(End definition for `\sys_if_output_dvi:TF`, `\sys_if_output_pdf:TF`, and `\c_sys_output_str`. These functions are documented on page 224.)

```
21501 </initex | package>
```

41 I3deprecation implementation

```

21502 <*initex | package>
21503 <@=deprecation>
```

`__deprecation_error:Nnn` The `\outer` definition here ensures the command will not appear in an argument. Use this auxiliary on all commands that have been removed since 2015.

```

21504 \cs_new_protected:Npn \__deprecation_error:Nnn #1#2#3
21505 {
21506   \etex_protected:D \tex_outer:D \tex_eodef:D #1
21507   {
21508     \exp_not:N \__msg_kernel_expandable_error:nnnnn
21509     { kernel } { deprecated-command }
21510     { \tl_to_str:n {#3} } { \token_to_str:N #1 } { \tl_to_str:n {#2} }
21511     \exp_not:N \__msg_kernel_error:nnxxx
21512     { kernel } { deprecated-command }
21513     { \tl_to_str:n {#3} } { \token_to_str:N #1 } { \tl_to_str:n {#2} }
21514   }
21515 }
21516 \__deprecation_error:Nnn \c_job_name_tl { \c_sys_jobname_str } { 2017-01-01 }
21517 \__deprecation_error:Nnn \dim_case:nnn { \dim_case:nnF } { 2015-07-14 }
21518 \__deprecation_error:Nnn \int_case:nnn { \int_case:nnF } { 2015-07-14 }
21519 \__deprecation_error:Nnn \int_from_binary:n { \int_from_bin:n } { 2016-01-05 }
21520 \__deprecation_error:Nnn \int_from_hexadecimal:n { \int_from_hex:n } { 2016-01-05 }
21521 \__deprecation_error:Nnn \int_from_octal:n { \int_from_oct:n } { 2016-01-05 }
21522 \__deprecation_error:Nnn \int_to_binary:n { \int_to_bin:n } { 2016-01-05 }
21523 \__deprecation_error:Nnn \int_to_hexadecimal:n { \int_to_hex:n } { 2016-01-05 }
21524 \__deprecation_error:Nnn \int_to_octal:n { \int_to_oct:n } { 2016-01-05 }
21525 \__deprecation_error:Nnn \luatex_if_engine_p: { \sys_if_engine_luatex_p: } { 2017-01-01 }
21526 \__deprecation_error:Nnn \luatex_if_engine:F { \sys_if_engine_luatex:F } { 2017-01-01 }
21527 \__deprecation_error:Nnn \luatex_if_engine:T { \sys_if_engine_luatex:T } { 2017-01-01 }
21528 \__deprecation_error:Nnn \luatex_if_engine:TF { \sys_if_engine_luatex:TF } { 2017-01-01 }
21529 \__deprecation_error:Nnn \pdftex_if_engine_p: { \sys_if_engine_pdftex_p: } { 2017-01-01 }
21530 \__deprecation_error:Nnn \pdftex_if_engine:F { \sys_if_engine_pdftex:F } { 2017-01-01 }
```

```

21531 \__deprecation_error:Nnn \pdfTeX_if_engine:T { \sys_if_engine_pdfTeX:T } { 2017-01-01 }
21532 \__deprecation_error:Nnn \pdfTeX_if_engine:TF { \sys_if_engine_pdfTeX:TF } { 2017-01-01 }
21533 \__deprecation_error:Nnn \prop_get:cn { \prop_item:cn } { 2016-01-05 }
21534 \__deprecation_error:Nnn \prop_get:Nn { \prop_item:Nn } { 2016-01-05 }
21535 \__deprecation_error:Nnn \quark_if_recursion_tail_break:N { } { 2015-07-14 }
21536 \__deprecation_error:Nnn \quark_if_recursion_tail_break:n { } { 2015-07-14 }
21537 \__deprecation_error:Nnn \scan_align_safe_stop: { protected-commands } { 2017-01-01 }
21538 \__deprecation_error:Nnn \str_case:nnn { \str_case:nnF } { 2015-07-14 }
21539 \__deprecation_error:Nnn \str_case:onn { \str_case:onF } { 2015-07-14 }
21540 \__deprecation_error:Nnn \str_case_x:nnn { \str_case_x:nnF } { 2015-07-14 }
21541 \__deprecation_error:Nnn \tl_case:cnn { \tl_case:cnF } { 2015-07-14 }
21542 \__deprecation_error:Nnn \tl_case:Nnn { \tl_case:NnF } { 2015-07-14 }
21543 \__deprecation_error:Nnn \xetex_if_engine_p: { \sys_if_engine_xetex_p: } { 2017-01-01 }
21544 \__deprecation_error:Nnn \xetex_if_engine:F { \sys_if_engine_xetex:F } { 2017-01-01 }
21545 \__deprecation_error:Nnn \xetex_if_engine:T { \sys_if_engine_xetex:T } { 2017-01-01 }
21546 \__deprecation_error:Nnn \xetex_if_engine:TF { \sys_if_engine_xetex:TF } { 2017-01-01 }

```

(End definition for `__deprecation_error:Nnn`.)

`\deprecation_error:` Some commands were more recently deprecated and not yet removed; only make these into errors if the user requests it. This allows testing code even if it relies on other packages: load all other packages, call `\deprecation_error:`, and load the code that one is interested in testing.

One of the deprecated syntaxes is within floating point expressions; to keep related code in the same place we define in `\l3fp-parse` a version of `__fp_parse_round:Nw` producing an error.

```

21547 \cs_new_protected:Npn \deprecation_error:
21548   {
21549     \__deprecation_error:Nnn \tl_to_lowercase:n { } { 2017-12-31 }
21550     \__deprecation_error:Nnn \tl_to_uppercase:n { } { 2017-12-31 }
21551     \__deprecation_error:Nnn \ior_get_str>NN { \ior_str_get>NN } { 2017-12-31 }
21552     \__deprecation_error:Nnn \box_resize:Nnn { \box_resize_to_wd_and_ht_plus_dp:Nnn } { 201
21553     \__deprecation_error:Nnn \box_resize:cnn { \box_resize_to_wd_and_ht_plus_dp:cnn } { 201
21554     \__deprecation_error:Nnn \c_minus_one { - 1 } { 2018-12-31 }
21555     \__deprecation_error:Nnn \sort_ordered: { \sort_return_same: } { 2018-12-31 }
21556     \__deprecation_error:Nnn \sort_reversed: { \sort_return_swapped: } { 2018-12-31 }
21557     \cs_set_eq:NN \__fp_parse_round:Nw \__fp_parse_round_deprecation_error:Nw
21558     \cs_set_eq:NN \deprecation_error: \scan_stop:
21559   }

```

(End definition for `\deprecation_error:`. This function is documented on page ??.)

21560 `</initex | package>`

42 `\l3candidates` Implementation

21561 `<*initex | package>`

42.1 Additions to `\l3box`

21562 `<@=box>`

42.2 Viewing part of a box

`\box_clip:N` A wrapper around the driver-dependent code.
`\box_clip:c`

```

21563 \cs_new_protected:Npn \box_clip:N #1
21564   { \hbox_set:Nn #1 { \__driver_box_use_clip:N #1 } }
21565 \cs_generate_variant:Nn \box_clip:N { c }

```

(End definition for `\box_clip:N`. This function is documented on page 226.)

`\box_trim:Nnnnn` Trimming from the left- and right-hand edges of the box is easy: kern the appropriate parts off each side.

```

21566 \cs_new_protected:Npn \box_trim:Nnnnn #1#2#3#4#5
21567   {
21568     \hbox_set:Nn \l__box_internal_box
21569     {
21570       \tex_kern:D -\__dim_eval:w #2 \__dim_eval_end:
21571       \box_use:N #1
21572       \tex_kern:D -\__dim_eval:w #4 \__dim_eval_end:
21573     }

```

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`.

```

21574 \dim_compare:nNnTF { \box_dp:N #1 } > {#3}
21575   {
21576     \hbox_set:Nn \l__box_internal_box
21577     {
21578       \box_move_down:nn \c_zero_dim
21579       { \box_use:N \l__box_internal_box }
21580     }
21581     \box_set_dp:Nn \l__box_internal_box { \box_dp:N #1 - (#3) }
21582   }
21583   {
21584     \hbox_set:Nn \l__box_internal_box
21585     {
21586       \box_move_down:nn { #3 - \box_dp:N #1 }
21587       { \box_use:N \l__box_internal_box }
21588     }
21589     \box_set_dp:Nn \l__box_internal_box \c_zero_dim
21590   }

```

Same thing, this time from the top of the box.

```

21591 \dim_compare:nNnTF { \box_ht:N \l__box_internal_box } > {#5}
21592   {
21593     \hbox_set:Nn \l__box_internal_box
21594     {
21595       \box_move_up:nn \c_zero_dim
21596       { \box_use:N \l__box_internal_box }
21597     }
21598     \box_set_ht:Nn \l__box_internal_box
21599     { \box_ht:N \l__box_internal_box - (#5) }
21600   }
21601   {
21602     \hbox_set:Nn \l__box_internal_box
21603     {

```

```

21604         \box_move_up:nn { #5 - \box_ht:N \l_box_internal_box }
21605             { \box_use:N \l_box_internal_box }
21606     }
21607     \box_set_ht:Nn \l_box_internal_box \c_zero_dim
21608   }
21609   \box_set_eq:NN #1 \l_box_internal_box
21610 }
21611 \cs_generate_variant:Nn \box_trim:Nnnnn { c }

```

(End definition for `\box_trim:Nnnnn`. This function is documented on page 227.)

`\box_viewport:Nnnnn` 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.

```

21612 \cs_new_protected:Npn \box_viewport:Nnnnn #1#2#3#4#5
21613 {
21614     \hbox_set:Nn \l_box_internal_box
21615     {
21616         \tex_kern:D -\dim_eval:w #2 \dim_eval_end:
21617         \box_use:N #1
21618         \tex_kern:D \dim_eval:w #4 - \box_wd:N #1 \dim_eval_end:
21619     }
21620     \dim_compare:nNnTF {#3} < \c_zero_dim
21621     {
21622         \hbox_set:Nn \l_box_internal_box
21623         {
21624             \box_move_down:nn \c_zero_dim
21625                 { \box_use:N \l_box_internal_box }
21626         }
21627         \box_set_dp:Nn \l_box_internal_box { -\dim_eval:n {#3} }
21628     }
21629     {
21630         \hbox_set:Nn \l_box_internal_box
21631             { \box_move_down:nn {#3} { \box_use:N \l_box_internal_box } }
21632         \box_set_dp:Nn \l_box_internal_box \c_zero_dim
21633     }
21634     \dim_compare:nNnTF {#5} > \c_zero_dim
21635     {
21636         \hbox_set:Nn \l_box_internal_box
21637         {
21638             \box_move_up:nn \c_zero_dim
21639                 { \box_use:N \l_box_internal_box }
21640         }
21641         \box_set_ht:Nn \l_box_internal_box
21642         {
21643             #5
21644             \dim_compare:nNnT {#3} > \c_zero_dim
21645                 { - (#3) }
21646             }
21647         }
21648     {
21649         \hbox_set:Nn \l_box_internal_box
21650         {
21651             \box_move_up:nn { -\dim_eval:n {#5} }
21652                 { \box_use:N \l_box_internal_box }

```

```

21653         }
21654         \box_set_ht:Nn \l__box_internal_box \c_zero_dim
21655     }
21656     \box_set_eq:NN #1 \l__box_internal_box
21657 }
21658 \cs_generate_variant:Nn \box_viewport:Nnnnn { c }

```

(End definition for `\box_viewport:Nnnnn`. This function is documented on page 227.)

42.3 Additions to l3clist

`\clist_rand_item:n`
`\clist_rand_item:N`
`\clist_rand_item:c`
`_clist_rand_item:nn`

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.

```

21660 \cs_new:Npn \clist_rand_item:n #
21661   { \exp_args:Nf \_clist_rand_item:nn { \clist_count:n {#1} } {#1} }
21662 \cs_new:Npn \_clist_rand_item:nn #1#2
21663   {
21664     \int_compare:nNnF {#1} = 0
21665       { \clist_item:nn {#2} { \int_rand:nn { 1 } {#1} } }
21666   }
21667 \cs_new:Npn \clist_rand_item:N #
21668   {
21669     \clist_if_empty:NF #1
21670       { \clist_item:Nn #1 { \int_rand:nn { 1 } { \clist_count:N #1 } } }
21671   }
21672 \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 227.)

42.4 Additions to l3coffins

`\coffin_sin_fp`
`\coffin_cos_fp`

Used for rotations to get the sine and cosine values.

```

21674 \fp_new:N \l__coffin_sin_fp
21675 \fp_new:N \l__coffin_cos_fp

```

(End definition for `\l__coffin_sin_fp` and `\l__coffin_cos_fp`.)

`\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.

```
21676 \prop_new:N \l__coffin_bounding_prop
```

(End definition for `\l__coffin_bounding_prop`.)

`\coffin_bounding_shift_dim` The shift of the bounding box of a coffin from the real content.

```
21677 \dim_new:N \l__coffin_bounding_shift_dim
```

(End definition for `\l_coffin_bounding_shift_dim`.)

```
\l_coffin_left_corner_dim These are used to hold maxima for the various corner values: these thus define the
\l_coffin_right_corner_dim minimum size of the bounding box after rotation.
\l_coffin_bottom_corner_dim
\l_coffin_top_corner_dim
21678 \dim_new:N \l_coffin_left_corner_dim
21679 \dim_new:N \l_coffin_right_corner_dim
21680 \dim_new:N \l_coffin_bottom_corner_dim
21681 \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.

```
21682 \cs_new_protected:Npn \coffin_rotate:Nn #1#2
21683 {
21684     \fp_set:Nn \l_coffin_sin_fp { sind ( #2 ) }
21685     \fp_set:Nn \l_coffin_cos_fp { cosd ( #2 ) }
```

The corners and poles of the coffin can now be rotated around the origin. This is best achieved using mapping functions.

```
21686 \prop_map_inline:cn { l_coffin_corners_ \__int_value:w #1 _prop }
21687     { \__coffin_rotate_corner:Nnnn #1 {##1} ##2 }
21688 \prop_map_inline:cn { l_coffin_poles_ \__int_value:w #1 _prop }
21689     { \__coffin_rotate_pole:Nnnnn #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.

```
21690 \__coffin_set_bounding:N #1
21691 \prop_map_inline:Nn \l_coffin_bounding_prop
21692     { \__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.

```
21693 \__coffin_find_corner_maxima:N #1
21694 \__coffin_find_bounding_shift:
21695 \box_rotate:Nn #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.

```
21696 \hbox_set:Nn \l_coffin_internal_box
21697 {
21698     \tex_kern:D
21699     \__dim_eval:w
21700         \l_coffin_bounding_shift_dim - \l_coffin_left_corner_dim
21701     \__dim_eval_end:
21702     \box_move_down:nn { \l_coffin_bottom_corner_dim }
21703     { \box_use:N #1 }
21704 }
```

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.

```

21705      \box_set_ht:Nn \l__coffin_internal_box
21706      { \l__coffin_top_corner_dim - \l__coffin_bottom_corner_dim }
21707      \box_set_dp:Nn \l__coffin_internal_box { 0 pt }
21708      \box_set_wd:Nn \l__coffin_internal_box
21709      { \l__coffin_right_corner_dim - \l__coffin_left_corner_dim }
21710      \hbox_set:Nn #1 { \box_use: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.

```

21711      \prop_map_inline:cn { l__coffin_corners_ \__int_value:w #1 _prop }
21712      { \__coffin_shift_corner:Nnnn #1 {##1} ##2 }
21713      \prop_map_inline:cn { l__coffin_poles_ \__int_value:w #1 _prop }
21714      { \__coffin_shift_pole:Nnnnn #1 {##1} ##2 }
21715  }
21716 \cs_generate_variant:Nn \coffin_rotate:Nn { c }
```

(End definition for `\coffin_rotate:Nn`. This function is documented on page 227.)

`__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.

```

21717 \cs_new_protected:Npn \__coffin_set_bounding:N #1
21718 {
21719     \prop_put:Nnx \l__coffin_bounding_prop { tl }
21720     { { 0 pt } { \dim_eval:n { \box_ht:N #1 } } }
21721     \prop_put:Nnx \l__coffin_bounding_prop { tr }
21722     { { \dim_eval:n { \box_wd:N #1 } } { \dim_eval:n { \box_ht:N #1 } } }
21723     \dim_set:Nn \l__coffin_internal_dim { -\box_dp:N #1 }
21724     \prop_put:Nnx \l__coffin_bounding_prop { bl }
21725     { { 0 pt } { \dim_use:N \l__coffin_internal_dim } }
21726     \prop_put:Nnx \l__coffin_bounding_prop { br }
21727     { { \dim_eval:n { \box_wd:N #1 } } { \dim_use:N \l__coffin_internal_dim } }
21728 }
```

(End definition for `__coffin_set_bounding:N`.)

`__coffin_rotate_bounding:nnn`
`__coffin_rotate_corner:Nnnn` 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.

```

21729 \cs_new_protected:Npn \__coffin_rotate_bounding:nnn #1#2#3
21730 {
21731     \__coffin_rotate_vector:nnNN {#2} {#3} \l__coffin_x_dim \l__coffin_y_dim
21732     \prop_put:Nnx \l__coffin_bounding_prop {#1}
21733     { { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim } }
21734 }
21735 \cs_new_protected:Npn \__coffin_rotate_corner:Nnnn #1#2#3#4
21736 {
21737     \__coffin_rotate_vector:nnNN {#3} {#4} \l__coffin_x_dim \l__coffin_y_dim
21738     \prop_put:cnx { l__coffin_corners_ \__int_value:w #1 _prop } {#2}
21739     { { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim } }
21740 }
```

(End definition for `__coffin_rotate_bounding:n nn` and `__coffin_rotate_corner:N 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.

```
21741 \cs_new_protected:Npn \__coffin_rotate_pole:Nnnnnn #1#2#3#4#5#6
21742 {
21743     \__coffin_rotate_vector:nnNN {#3} {#4} \l__coffin_x_dim \l__coffin_y_dim
21744     \__coffin_rotate_vector:nnNN {#5} {#6}
21745     \l__coffin_x_prime_dim \l__coffin_y_prime_dim
21746     \__coffin_set_pole:Nnx #1 {#2}
21747     {
21748         { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim }
21749         { \dim_use:N \l__coffin_x_prime_dim }
21750         { \dim_use:N \l__coffin_y_prime_dim }
21751     }
21752 }
```

(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.

```
21753 \cs_new_protected:Npn \__coffin_rotate_vector:nnNN #1#2#3#4
21754 {
21755     \dim_set:Nn #3
21756     {
21757         \fp_to_dim:n
21758         {
21759             \dim_to_fp:n {#1} * \l__coffin_cos_fp
21760             - \dim_to_fp:n {#2} * \l__coffin_sin_fp
21761         }
21762     }
21763     \dim_set:Nn #4
21764     {
21765         \fp_to_dim:n
21766         {
21767             \dim_to_fp:n {#1} * \l__coffin_sin_fp
21768             + \dim_to_fp:n {#2} * \l__coffin_cos_fp
21769         }
21770     }
21771 }
```

(End definition for `__coffin_rotate_vector:nnNN`.)

`__coffin_find_corner_maxima:N` 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.

```
21772 \cs_new_protected:Npn \__coffin_find_corner_maxima:N #1
21773 {
21774     \dim_set:Nn \l__coffin_top_corner_dim { -\c_max_dim }
21775     \dim_set:Nn \l__coffin_right_corner_dim { -\c_max_dim }
21776     \dim_set:Nn \l__coffin_bottom_corner_dim { \c_max_dim }
```

```

21777     \dim_set:Nn \l__coffin_left_corner_dim { \c_max_dim }
21778     \prop_map_inline:cn { l__coffin_corners_ \__int_value:w #1 _prop }
21779         { \__coffin_find_corner_maxima_aux:nn ##2 }
21780     }
21781 \cs_new_protected:Npn \__coffin_find_corner_maxima_aux:nn #1#2
21782 {
21783     \dim_set:Nn \l__coffin_left_corner_dim
21784         { \dim_min:nn { \l__coffin_left_corner_dim } {#1} }
21785     \dim_set:Nn \l__coffin_right_corner_dim
21786         { \dim_max:nn { \l__coffin_right_corner_dim } {#1} }
21787     \dim_set:Nn \l__coffin_bottom_corner_dim
21788         { \dim_min:nn { \l__coffin_bottom_corner_dim } {#2} }
21789     \dim_set:Nn \l__coffin_top_corner_dim
21790         { \dim_max:nn { \l__coffin_top_corner_dim } {#2} }
21791 }

```

(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.

```

21792 \cs_new_protected:Npn \__coffin_find_bounding_shift:
21793 {
21794     \dim_set:Nn \l__coffin_bounding_shift_dim { \c_max_dim }
21795     \prop_map_inline:Nn \l__coffin_bounding_prop
21796         { \__coffin_find_bounding_shift_aux:nn ##2 }
21797 }
21798 \cs_new_protected:Npn \__coffin_find_bounding_shift_aux:nn #1#2
21799 {
21800     \dim_set:Nn \l__coffin_bounding_shift_dim
21801         { \dim_min:nn { \l__coffin_bounding_shift_dim } {#1} }
21802 }

```

(End definition for `__coffin_find_bounding_shift:` and `__coffin_find_bounding_shift_aux:nn`.)

`__coffin_shift_corner:Nnnn`
`__coffin_shift_pole:Nnnnnn`
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.

```

21803 \cs_new_protected:Npn \__coffin_shift_corner:Nnnn #1#2#3#4
21804 {
21805     \prop_put:cnx { l__coffin_corners_ \__int_value:w #1 _prop } {#2}
21806     {
21807         { \dim_eval:n { #3 - \l__coffin_left_corner_dim } }
21808         { \dim_eval:n { #4 - \l__coffin_bottom_corner_dim } }
21809     }
21810 }
21811 \cs_new_protected:Npn \__coffin_shift_pole:Nnnnnn #1#2#3#4#5#6
21812 {
21813     \prop_put:cnx { l__coffin_poles_ \__int_value:w #1 _prop } {#2}
21814     {
21815         { \dim_eval:n { #3 - \l__coffin_left_corner_dim } }
21816         { \dim_eval:n { #4 - \l__coffin_bottom_corner_dim } }
21817         {#5} {#6}
21818     }
21819 }

```

(End definition for `_coffin_shift_corner:Nnnn` and `_coffin_shift_pole:Nnnnnn`.)

42.6 Resizing coffins

`\l_coffin_scale_x_fp` Storage for the scaling factors in x and y , respectively.

```
21820 \fp_new:N \l_coffin_scale_x_fp  
21821 \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.

```
21822 \dim_new:N \l_coffin_scaled_total_height_dim  
21823 \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`
`\coffin_resize:cnn`

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.

```
21824 \cs_new_protected:Npn \coffin_resize:Nnn #1#2#3  
21825 {  
21826   \fp_set:Nn \l_coffin_scale_x_fp  
21827   { \dim_to_fp:n {#2} / \dim_to_fp:n { \coffin_wd:N #1 } }  
21828   \fp_set:Nn \l_coffin_scale_y_fp  
21829   {  
21830     \dim_to_fp:n {#3}  
21831     / \dim_to_fp:n { \coffin_ht:N #1 + \coffin_dp:N #1 }  
21832   }  
21833   \box_resize_to_wd_and_ht_plus_dp:Nnn #1 {#2} {#3}  
21834   \__coffin_resize_common:Nnn #1 {#2} {#3}  
21835 }  
21836 \cs_generate_variant:Nn \coffin_resize:Nnn { c }
```

(End definition for `\coffin_resize:Nnn`. This function is documented on page 227.)

`__coffin_resize_common:Nnn`

The poles and corners of the coffin are scaled to the appropriate places before actually resizing the underlying box.

```
21837 \cs_new_protected:Npn \__coffin_resize_common:Nnn #1#2#3  
21838 {  
21839   \prop_map_inline:cn { \__coffin_corners_ \__int_value:w #1 _prop }  
21840   { \__coffin_scale_corner:Nnnn #1 {##1} ##2 }  
21841   \prop_map_inline:cn { \__coffin_poles_ \__int_value:w #1 _prop }  
21842   { \__coffin_scale_pole:Nnnnnn #1 {##1} ##2 }  
21843 }
```

Negative x -scaling values will place the poles in the wrong location: this is corrected here.

```
21843 \fp_compare:nNnT \l_coffin_scale_x_fp < \c_zero_fp  
21844 {  
21845   \prop_map_inline:cn { \__coffin_corners_ \__int_value:w #1 _prop }  
21846   { \__coffin_x_shift_corner:Nnnn #1 {##1} ##2 }  
21847   \prop_map_inline:cn { \__coffin_poles_ \__int_value:w #1 _prop }  
21848   { \__coffin_x_shift_pole:Nnnnnn #1 {##1} ##2 }  
21849 }  
21850 }
```

(End definition for `__coffin_resize_common:Nnn`.)

\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 scaling is done the TeX way as this works properly with floating point values without needing to use the `fp` module.

```

21851 \cs_new_protected:Npn \coffin_scale:Nnn #1#2#3
21852 {
21853     \fp_set:Nn \l_coffin_scale_x_fp {#2}
21854     \fp_set:Nn \l_coffin_scale_y_fp {#3}
21855     \box_scale:Nnn #1 { \l_coffin_scale_x_fp } { \l_coffin_scale_y_fp }
21856     \dim_set:Nn \l_coffin_internal_dim
21857         { \coffin_ht:N #1 + \coffin_dp:N #1 }
21858     \dim_set:Nn \l_coffin_scaled_total_height_dim
21859         { \fp_abs:n { \l_coffin_scale_y_fp } \l_coffin_internal_dim }
21860     \dim_set:Nn \l_coffin_scaled_width_dim
21861         { -\fp_abs:n { \l_coffin_scale_x_fp } \coffin_wd:N #1 }
21862     \__coffin_resize_common:Nnn #1
21863         { \l_coffin_scaled_width_dim } { \l_coffin_scaled_total_height_dim }
21864     }
21865 \cs_generate_variant:Nn \coffin_scale:Nnn { c }
```

(End definition for `\coffin_scale:Nnn`. This function is documented on page 227.)

__coffin_scale_vector:nnNN This functions 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.

```

21866 \cs_new_protected:Npn \__coffin_scale_vector:nnNN #1#2#3#4
21867 {
21868     \dim_set:Nn #3
21869         { \fp_to_dim:n { \dim_to_fp:n {#1} * \l_coffin_scale_x_fp } }
21870     \dim_set:Nn #4
21871         { \fp_to_dim:n { \dim_to_fp:n {#2} * \l_coffin_scale_y_fp } }
21872 }
```

(End definition for `__coffin_scale_vector:nnNN`.)

__coffin_scale_corner:Nnnn Scaling both corners and poles is a simple calculation using the preceding vector scaling.
__coffin_scale_pole:Nnnnnn

```

21873 \cs_new_protected:Npn \__coffin_scale_corner:Nnnn #1#2#3#4
21874 {
21875     \__coffin_scale_vector:nnNN {#3} {#4} \l_coffin_x_dim \l_coffin_y_dim
21876     \prop_put:cnx { l_coffin_corners_ \__int_value:w #1 _prop } {#2}
21877         { { \dim_use:N \l_coffin_x_dim } { \dim_use:N \l_coffin_y_dim } }
21878 }
21879 \cs_new_protected:Npn \__coffin_scale_pole:Nnnnnn #1#2#3#4#5#6
21880 {
21881     \__coffin_scale_vector:nnNN {#3} {#4} \l_coffin_x_dim \l_coffin_y_dim
21882     \__coffin_set_pole:Nnx #1 {#2}
21883     {
21884         { \dim_use:N \l_coffin_x_dim } { \dim_use:N \l_coffin_y_dim }
21885         {#5} {#6}
21886     }
21887 }
```

(End definition for `_coffin_x_shift_corner:Nnnn` and `_coffin_x_shift_pole:Nnnnnn`.)

`_coffin_x_shift_corner:Nnnn` These functions correct for the *x* displacement that takes place with a negative horizontal scaling.
`_coffin_x_shift_pole:Nnnnnn`

```
21888 \cs_new_protected:Npn \_coffin_x_shift_corner:Nnnn #1#2#3#4
21889  {
21890    \prop_put:cnx { l__coffin_corners_ \__int_value:w #1 _prop } {#2}
21891    {
21892      { \dim_eval:n { #3 + \box_wd:N #1 } } {#4}
21893    }
21894  }
21895 \cs_new_protected:Npn \_coffin_x_shift_pole:Nnnnnn #1#2#3#4#5#6
21896  {
21897    \prop_put:cnx { l__coffin_poles_ \__int_value:w #1 _prop } {#2}
21898    {
21899      { \dim_eval:n { #3 + \box_wd:N #1 } } {#4}
21900      {#5} {#6}
21901    }
21902  }
```

(End definition for `_coffin_x_shift_corner:Nnnn` and `_coffin_x_shift_pole:Nnnnnn`.)

42.7 Additions to l3file

21903 ⟨@C=file⟩

`\file_if_exist_input:nTF` Input of a file with a test for existence cannot be done the usual way as the tokens to insert are in an odd place.

```
21904 \cs_new_protected:Npn \file_if_exist_input:n #1
21905  {
21906    \file_if_exist:nT {#1}
21907    { \_file_input:V \l_file_internal_name_tl }
21908  }
21909 \cs_new_protected:Npn \file_if_exist_input:nT #1#2
21910  {
21911    \file_if_exist:nT {#1}
21912    {
21913      #2
21914      \_file_input:V \l_file_internal_name_tl
21915    }
21916  }
21917 \cs_new_protected:Npn \file_if_exist_input:nF #1
21918  {
21919    \file_if_exist:nTF {#1}
21920    { \_file_input:V \l_file_internal_name_tl }
21921  }
21922 \cs_new_protected:Npn \file_if_exist_input:nTF #1#2
21923  {
21924    \file_if_exist:nTF {#1}
21925    {
21926      #2
21927      \_file_input:V \l_file_internal_name_tl
21928    }
21929  }
```

(End definition for \file_if_exist_input:nTF. This function is documented on page 228.)

\ior_log_streams: Redirect output of \ior_list_streams: to the log.

```
21930 \cs_new_protected:Npn \ior_log_streams:
21931   { \__msg_log_next: \ior_list_streams: }
```

(End definition for \ior_log_streams:. This function is documented on page 228.)

\iow_log_streams: Redirect output of \iow_list_streams: to the log.

```
21932 \cs_new_protected:Npn \iow_log_streams:
21933   { \__msg_log_next: \iow_list_streams: }
```

(End definition for \iow_log_streams:. This function is documented on page 228.)

42.8 Additions to l3int

```
21934 ⟨@C=int⟩
```

\int_rand:nn
 __int_rand:ww
 __int_rand_narrow:n
 __int_rand_narrow:nnn
 __int_rand_narrow:nnnn
Evaluate the argument and filter out the case where the lower bound #1 is more than the upper bound #2. Then determine whether the range is narrower than \c_fp_rand_size_int; #2-#1 may overflow for very large positive #2 and negative #1. If the range is wide, use slower code from l3fp. If the range is narrow, call __int_rand_narrow:nn {\⟨choices⟩} {#1} where ⟨choices⟩ is the number of possible outcomes. Then __int_rand_narrow:nnnn receives a random number reduced modulo ⟨choices⟩, the random number itself, ⟨choices⟩ and #1. To avoid bias, throw away the random number if it lies in the last, incomplete, interval of size ⟨choices⟩ in [0, \c_fp_rand_size_int - 1], and try again.

```
21935 \cs_if_exist:NTF \pdftex_uniformdeviate:D
21936   {
21937     \cs_new:Npn \int_rand:nn #1#2
21938       {
21939         \exp_after:wN \__int_rand:ww
21940         \__int_value:w \__int_eval:w #1 \exp_after:wN ;
21941         \__int_value:w \__int_eval:w #2 ;
21942       }
21943     \cs_new:Npn \__int_rand:ww #1; #2;
21944       {
21945         \int_compare:nNnTF {#1} > {#2}
21946           {
21947             \__msg_kernel_expandable_error:nnnn
21948               { kernel } { backward-range } {#1} {#2}
21949             \__int_rand:ww #2; #1;
21950           }
21951           {
21952             \int_compare:nNnTF {#1} > 0
21953               { \int_compare:nNnTF { #2 - #1 } < \c_fp_rand_size_int }
21954               { \int_compare:nNnTF {#2} < { #1 + \c_fp_rand_size_int } }
21955               {
21956                 \exp_args:Nf \__int_rand_narrow:nn
21957                   { \int_eval:n { #2 - #1 + 1 } } {#1}
21958               }
21959               { \fp_to_int:n { randint(#1,#2) } }
21960             }
21961           }
21962       }
```

```

21962   \cs_new:Npn \__int_rand_narrow:nn
21963   {
21964     \exp_args:No \__int_rand_narrow:nnn
21965       { \pdftex_uniformdeviate:D \c_fp_rand_size_int }
21966   }
21967   \cs_new:Npn \__int_rand_narrow:nnn #1#2
21968   {
21969     \exp_args:Nf \__int_rand_narrow:nnnn
21970       { \int_mod:nn {#1} {#2} } {#1} {#2}
21971   }
21972   \cs_new:Npn \__int_rand_narrow:nnnn #1#2#3#4
21973   {
21974     \int_compare:nNnTF {#2 - #1 + #3} > \c_fp_rand_size_int
21975       { \__int_rand_narrow:nn {#3} {#4} }
21976       { \int_eval:n {#4 + #1} }
21977   }
21978 }
21979 {
21980   \cs_new:Npn \int_rand:nn #1#2
21981   {
21982     \__msg_kernel_expandable_error:nn { kernel } { fp-no-random }
21983     \int_eval:n {#1}
21984   }
21985 }

```

(End definition for `\int_rand:nn` and others. These functions are documented on page 228.)

The following must be added to `\l3msg`.

```

21986 \cs_if_exist:NT \pdftex_uniformdeviate:D
21987   {
21988     \__msg_kernel_new:nnn { kernel } { backward-range }
21989       { Bounds~ordered~backwards~in~\int_rand:nn {#1}~{#2}. }
21990   }

```

42.9 Additions to `\l3msg`

`\l3msg`

Pass to an auxiliary the message to display and the module name

```

21992 \cs_new:Npn \msg_expandable_error:nnnnnn #1#2#3#4#5#6
21993   {
21994     \exp_args:Nf \__msg_expandable_error_module:nn
21995       {
21996         \exp_args:Nf \tl_to_str:n
21997           { \use:c { \c_msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6} }
21998       }
21999       {#1}
22000   }
22001 \cs_new:Npn \msg_expandable_error:nnnnnn #1#2#3#4#5
22002   { \msg_expandable_error:nnnnnn {#1} {#2} {#3} {#4} {#5} { } }
22003 \cs_new:Npn \msg_expandable_error:nnnn #1#2#3#4
22004   { \msg_expandable_error:nnnnnn {#1} {#2} {#3} {#4} { } { } }
22005 \cs_new:Npn \msg_expandable_error:nnn #1#2#3
22006   { \msg_expandable_error:nnnnnn {#1} {#2} {#3} { } { } { } }
22007 \cs_new:Npn \msg_expandable_error:nn #1#2

```

```

22008 { \msg_expandable_error:nnnnnn {\#1} {\#2} { } { } { } { } { } }
22009 \cs_generate_variant:Nn \msg_expandable_error:nnnnnn { nnffff }
22010 \cs_generate_variant:Nn \msg_expandable_error:nnnnn { nnfff }
22011 \cs_generate_variant:Nn \msg_expandable_error:nmmn { nnff }
22012 \cs_generate_variant:Nn \msg_expandable_error:nmn { nnf }
22013 \cs_new:Npn \__msg_expandable_error_module:nn #1#2
22014 {
22015     \exp_after:wN \exp_after:wN
22016     \exp_after:wN \use_none_delimit_by_q_stop:w
22017     \use:n { \::error ! ~ #2 : ~ #1 } \q_stop
22018 }

```

(End definition for `\msg_expandable_error:nnnnnn` and others. These functions are documented on page 229.)

42.10 Additions to `I3prop`

22019 ⟨@@=prop⟩

`\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.

```

22020 \cs_new:Npn \prop_count:N #1
22021 {
22022     \int_eval:n
22023     {
22024         0
22025         \prop_map_function:NN #1 \__prop_count:nn
22026     }
22027 }
22028 \cs_new:Npn \__prop_count:nn #1#2 { + 1 }
22029 \cs_generate_variant:Nn \prop_count:N { c }

```

(End definition for `\prop_count:N` and `__prop_count:nn`. These functions are documented on page 229.)

`\prop_map_tokens:Nn`
`\prop_map_tokens:cN`
`__prop_map_tokens:nwwn` The mapping is very similar to `\prop_map_function:NN`. It grabs one key–value pair at a time, and stops when reaching the marker key `\q_recursion_tail`, which cannot appear in normal keys since those are strings. The odd construction `\use:n {#1}` allows `#1` to contain any token without interfering with `\prop_map_break:`. Argument `#2` of `__prop_map_tokens:nwwn` is `\s__prop` the first time, and is otherwise empty.

```

22030 \cs_new:Npn \prop_map_tokens:Nn #1#2
22031 {
22032     \exp_last_unbraced:Nno \__prop_map_tokens:nwwn {#2} #1
22033     \__prop_pair:wn \q_recursion_tail \s__prop { }
22034     \__prg_break_point:Nn \prop_map_break: { }
22035 }
22036 \cs_new:Npn \__prop_map_tokens:nwwn #1#2 \__prop_pair:wn #3 \s__prop #4
22037 {
22038     \if_meaning:w \q_recursion_tail #3
22039         \exp_after:wN \prop_map_break:
22040     \fi:
22041     \use:n {#1} {#3} {#4}
22042     \__prop_map_tokens:nwwn {#1}
22043 }

```

```
22044 \cs_generate_variant:Nn \prop_map_tokens:Nn { c }
```

(End definition for `\prop_map_tokens:Nn` and `_prop_map_tokens:nwwn`. These functions are documented on page 229.)

```
\prop_rand_key_value:N  
\prop_rand_key_value:c  
  \_prop_rand:NN  
  \_prop_rand_item:Nw
```

Contrarily to `clist`, `seq` and `t1`, 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. At the end, leave either the key #3 or the value #4 in the input stream.

```
22045 \cs_new:Npn \prop_rand_key_value:N { \_prop_rand:NN \_prop_rand:nNn }  
22046 \cs_new:Npn \_prop_rand:nNn #1#2#3 { \exp_not:n { #1} {#3} }  
22047 \cs_new:Npn \_prop_rand:NN #1#  
22048 {  
22049   \prop_if_empty:NTF #2 { }  
22050   {  
22051     \exp_after:wN \_prop_rand_item:Nw \exp_after:wN #1  
22052     \int_value:w \int_rand:nn { 1 } { \prop_count:N #2 } #2  
22053     \q_stop  
22054   }  
22055 }  
22056 \cs_new:Npn \_prop_rand_item:Nw #1#2 \s__prop \_prop_pair:wn #3 \s__prop #4  
22057 {  
22058   \int_compare:nNnF {#2} > 1  
22059   { \use_i_delimit_by_q_stop:nw { #1 {#3} \exp_not:n {#4} } }  
22060   \exp_after:wN \_prop_rand_item:Nw \exp_after:wN #1  
22061   \int_value:w \int_eval:n { #2 - 1 } \s__prop  
22062 }  
22063 \cs_generate_variant:Nn \prop_rand_key_value:N { c }
```

(End definition for `\prop_rand_key_value:N`, `_prop_rand:NN`, and `_prop_rand_item:Nw`. These functions are documented on page 229.)

42.11 Additions to `I3seq`

```
22064 ⟨@@=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:Nnnwwn
```

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 the sequences will both be `\s__seq _seq_item:n`. The function to be mapped will then be applied to the two entries. When the code hits the end of one of the sequences, the break material will stop the entire loop and tidy up. This avoids needing to find the count of the two sequences, or worrying about which is longer.

```
22065 \cs_new:Npn \seq_mapthread_function:NNN #1#2#3  
22066 { \exp_after:wN \_seq_mapthread_function:wNN #2 \q_stop #1 #3 }  
22067 \cs_new:Npn \_seq_mapthread_function:wNN \s__seq #1 \q_stop #2#3  
22068 {  
22069   \exp_after:wN \_seq_mapthread_function:wNw #2 \q_stop #3  
22070   #1 { ? \_prg_break: } { }  
22071   \_prg_break_point:  
22072 }  
22073 \cs_new:Npn \_seq_mapthread_function:wNw \s__seq #1 \q_stop #2  
22074 {  
22075   \_seq_mapthread_function:Nnnwwn #2
```

```

22076      #1 { ? \_prg_break: } { }
22077      \q_stop
22078  }
22079 \cs_new:Npn \__seq_mapthread_function:Nnnwnn #1#2#3#4 \q_stop #5#6
22080  {
22081      \use_none:n #2
22082      \use_none:n #5
22083      #1 {#3} {#6}
22084      \__seq_mapthread_function:Nnnwnn #1 #4 \q_stop
22085  }
22086 \cs_generate_variant:Nn \seq_mapthread_function:NNN { Nc }
22087 \cs_generate_variant:Nn \seq_mapthread_function:NNN { c , cc }

(End definition for \seq_mapthread_function:NNN and others. These functions are documented on page 230.)

```

\seq_set_filter:NNn
\seq_gset_filter:NNn
__seq_set_filter:NNNn

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.

```

22088 \cs_new_protected:Npn \seq_set_filter:NNn
22089  { \__seq_set_filter:NNNn \tl_set:Nx }
22090 \cs_new_protected:Npn \seq_gset_filter:NNn
22091  { \__seq_set_filter:NNNn \tl_gset:Nx }
22092 \cs_new_protected:Npn \__seq_set_filter:NNNn #1#2#3#4
22093  {
22094      \__seq_push_item_def:n { \bool_if:nT {#4} { \__seq_wrap_item:n {##1} } }
22095      #1 #2 { #3 }
22096      \__seq_pop_item_def:
22097  }

```

(End definition for \seq_set_filter:NNn, \seq_gset_filter:NNn, and __seq_set_filter:NNNn. These functions are documented on page 230.)

\seq_set_map:NNn
\seq_gset_map:NNn
__seq_set_map:NNNn

Very similar to **\seq_set_filter:NNn**. We could actually merge the two within a single function, but it would have weird semantics.

```

22098 \cs_new_protected:Npn \seq_set_map:NNn
22099  { \__seq_set_map:NNNn \tl_set:Nx }
22100 \cs_new_protected:Npn \seq_gset_map:NNn
22101  { \__seq_set_map:NNNn \tl_gset:Nx }
22102 \cs_new_protected:Npn \__seq_set_map:NNNn #1#2#3#4
22103  {
22104      \__seq_push_item_def:n { \exp_not:N \__seq_item:n {#4} }
22105      #1 #2 { #3 }
22106      \__seq_pop_item_def:
22107  }

```

(End definition for \seq_set_map:NNn, \seq_gset_map:NNn, and __seq_set_map:NNNn. These functions are documented on page 230.)

\seq_rand_item:N Importantly, **\seq_item:Nn** only evaluates its argument once.

\seq_rand_item:c

```

22108 \cs_new:Npn \seq_rand_item:N #1
22109  {
22110      \seq_if_empty:NF #1
22111      { \seq_item:Nn #1 { \int_rand:nn { 1 } { \seq_count:N #1 } } }

```

```

22112     }
22113 \cs_generate_variant:Nn \seq_rand_item:N { c }

```

(End definition for `\seq_rand_item:N`. This function is documented on page 230.)

42.12 Additions to l3skip

```

22114 <@C=skip>

```

`\skip_split_finite_else_action:nnNN` This macro is useful when performing error checking in certain circumstances. If the `<skip>` register holds finite glue it sets #3 and #4 to the stretch and shrink component, resp. If it holds infinite glue set #3 and #4 to zero and issue the special action #2 which is probably an error message. Assignments are local.

```

22115 \cs_new:Npn \skip_split_finite_else_action:nnNN #1#2#3#4
22116 {
22117     \skip_if_finite:nTF {#1}
22118     {
22119         #3 = \etex_gluestretch:D #1 \scan_stop:
22120         #4 = \etex_glueshrink:D #1 \scan_stop:
22121     }
22122     {
22123         #3 = \c_zero_skip
22124         #4 = \c_zero_skip
22125         #2
22126     }
22127 }

```

(End definition for `\skip_split_finite_else_action:nnNN`. This function is documented on page 230.)

42.13 Additions to l3sys

```

22128 <@C=sys>

```

`\sys_if_rand_exist_p:` Currently, randomness exists under pdfTeX and LuaTeX.

`\sys_if_rand_exist:TF`

```

22129 \cs_if_exist:NTF \pdftex_uniformdeviate:D
22130 {
22131     \prg_new_conditional:Npnn \sys_if_rand_exist: { p , T , F , TF }
22132     { \prg_return_true: }
22133 }
22134 {
22135     \prg_new_conditional:Npnn \sys_if_rand_exist: { p , T , F , TF }
22136     { \prg_return_false: }
22137 }

```

(End definition for `\sys_if_rand_exist:TF`. This function is documented on page 231.)

`\sys_rand_seed:` Unpack the primitive.

```

22138 \cs_new:Npn \sys_rand_seed: { \tex_the:D \pdftex_randomseed:D }

```

(End definition for `\sys_rand_seed:`. This function is documented on page 231.)

`\sys_gset_rand_seed:n` The primitive always assigns the seed globally.

```

22139 \cs_new_protected:Npn \sys_gset_rand_seed:n #1
22140     { \pdftex_setrandomseed:D \__int_eval:w #1 \__int_eval_end: }

```

(End definition for `\sys_gset_rand_seed:n`. This function is documented on page 231.)

`\c_sys_shell_escape_int` Expose the engine's shell escape status to the user.

```
22141 \int_const:Nn \c_sys_shell_escape_int
22142 {
22143     \sys_if_engine_luatex:TF
22144     {
22145         \luatex_directlua:D
22146         {
22147             tex.sprint((status.shell_escape~or~os.execute()) .. " ")
22148         }
22149     }
22150     {
22151         \pdftex_shellescape:D
22152     }
22153 }
```

(End definition for `\c_sys_shell_escape_int`. This variable is documented on page 231.)

`\sys_if_shell_p:` Performs a check for whether shell escape is enabled. This will return true if either of `\sys_if_shell:TF` restricted or unrestricted shell escape is enabled.

```
22154 \prg_new_conditional:Nnn \sys_if_shell: { p , T , F , TF }
22155 {
22156     \if_int_compare:w \c_sys_shell_escape_int = 0 ~
22157         \prg_return_false:
22158     \else:
22159         \prg_return_true:
22160     \fi:
22161 }
```

(End definition for `\sys_if_shell:TF`. This function is documented on page 231.)

`\sys_if_shell_unrestricted_p:` Performs a check for whether *unrestricted* shell escape is enabled.

```
22162 \prg_new_conditional:Nnn \sys_if_shell_unrestricted: { p , T , F , TF }
22163 {
22164     \if_int_compare:w \c_sys_shell_escape_int = 1 ~
22165         \prg_return_true:
22166     \else:
22167         \prg_return_false:
22168     \fi:
22169 }
```

(End definition for `\sys_if_shell_unrestricted:TF`. This function is documented on page 231.)

`\sys_if_shell_unrestricted_p:` Performs a check for whether *restricted* shell escape is enabled. This will return 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:..`.

```
22170 \prg_new_conditional:Nnn \sys_if_shell_restricted: { p , T , F , TF }
22171 {
22172     \if_int_compare:w \c_sys_shell_escape_int = 2 ~
22173         \prg_return_true:
22174     \else:
22175         \prg_return_false:
22176     \fi:
22177 }
```

(End definition for `\sys_if_shell_unrestricted:TF`. This function is documented on page 231.)

`\c__sys_shell_stream_int` This is not needed for LuaTeX: shell escape there isn't done using a TeX interface

```
22178 \sys_if_engine_luatex:F
22179 { \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.

```
22180 \sys_if_engine_luatex:TF
22181 {
22182     \cs_new_protected:Npn \sys_shell_now:n #1
22183     {
22184         \luatex_directlua:D
22185         {
22186             os.execute(
22187                 \luatex_luaescapestring:D { \etex_detokenize:D {#1} }
22188                 ")
22189         }
22190     }
22191 }
22192 {
22193     \cs_new_protected:Npn \sys_shell_now:n #1
22194     {
22195         \iow_now:Nn \c__sys_shell_stream_int { #1 }
22196     }
22197 }
22198 \cs_generate_variant:Nn \sys_shell_now:n { x }
```

(End definition for `\sys_shell_now:n`. This function is documented on page 232.)

`\sys_shell_shipout:n` Execute commands through shell escape at shipout.

```
22199 \sys_if_engine_luatex:TF
22200 {
22201     \cs_new_protected:Npn \sys_shell_shipout:n #1
22202     {
22203         \luatex_latelua:D
22204         {
22205             os.execute(
22206                 \luatex_luaescapestring:D { \etex_detokenize:D {#1} }
22207                 ")
22208         }
22209     }
22210 }
22211 {
22212     \cs_new_protected:Npn \sys_shell_shipout:n #1
22213     {
22214         \iow_shipout:Nn \c__sys_shell_stream_int { #1 }
22215     }
22216 }
22217 \cs_generate_variant:Nn \sys_shell_shipout:n { x }
```

(End definition for `\sys_shell_shipout:n`. This function is documented on page 232.)

42.14 Additions to `\tl`

22218 `\tl_if_single_token_p:n`

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.

```
22219 \prg_new_conditional:Npnn \tl_if_single_token:n #1 { p , T , F , TF }
22220  {
22221    \tl_if_head_is_N_type:nTF {#1}
22222      { \__tl_if_empty_return:o { \use_none:n #1 } }
22223      {
22224        \tl_if_empty:nTF {#1}
22225          { \prg_return_false: }
22226          { \__tl_if_empty_return:o { \exp:w \exp_end_continue_f:w #1 } }
22227      }
22228  }
```

(End definition for `\tl_if_single_token:nTF`. This function is documented on page 232.)

`\tl_reverse_tokens:n` The same as `\tl_reverse:n` but with recursion within brace groups.

```
22229 \cs_new:Npn \tl_reverse_tokens:n #1
22230  {
22231    \etex_unexpanded:D \exp_after:wN
22232      {
22233        \exp:w
22234        \__tl_act:NNNnn
22235        \__tl_reverse_normal:nN
22236        \__tl_reverse_group:nn
22237        \__tl_reverse_space:n
22238        { }
22239        {#1}
22240      }
22241  }
22242 \cs_new:Npn \__tl_reverse_group:nn #1
22243  {
22244    \__tl_act_group_recurse:Nnn
22245    \__tl_act_reverse_output:n
22246    { \tl_reverse_tokens:n }
22247 }
```

In many applications of `__tl_act:NNNnn`, we need to recursively apply some transformation within brace groups, then output. In this code, #1 is the output function, #2 is the transformation, which should expand in two steps, and #3 is the group.

```
22248 \cs_new:Npn \__tl_act_group_recurse:Nnn #1#2#3
22249  {
22250    \exp_args:Nf #1
22251    { \exp_after:wN \exp_after:wN \exp_after:wN { #2 {#3} } }
22252 }
```

(End definition for `\tl_reverse_tokens:n`, `__tl_reverse_group:nn`, and `__tl_act_group_recurse:Nnn`. These functions are documented on page 232.)

```
\tl_count_tokens:n
```

```
\_tl_act_count_normal:nN
\_tl_act_count_group:nn
\_tl_act_count_space:n

22253 \cs_new:Npn \tl_count_tokens:n #1
22254   {
22255     \int_eval:n
22256     {
22257       \_tl_act:NNNnn
22258       \_tl_act_count_normal:nN
22259       \_tl_act_count_group:nn
22260       \_tl_act_count_space:n
22261       {
22262         \#1
22263       }
22264     }
22265   \cs_new:Npn \_tl_act_count_normal:nN #1 { 1 + }
22266   \cs_new:Npn \_tl_act_count_space:n #1 { 1 + }
22267   \cs_new:Npn \_tl_act_count_group:nn #1 #2
22268     { 2 + \tl_count_tokens:n {#2} + }
```

(End definition for `\tl_count_tokens:n` and others. These functions are documented on page 232.)

```
\tl_set_from_file:Nnn
```

```
\tl_set_from_file:cnn
\tl_gset_from_file:Nnn
\tl_gset_from_file:cnn

\_tl_set_from_file>NNnn
  \_tl_from_file_do:w
```

The approach here is similar to that for doing a rescan, and so the same internals can be reused. Thus the plan is to insert a pair of tokens of the same charcode but different catcodes after the file has been read. This plus `\exp_not:N` allows the primitive to be used to carry out a set operation.

```
22269 \cs_new_protected:Npn \tl_set_from_file:Nnn
22270   { \_tl_set_from_file>NNnn \tl_set:Nn }
22271 \cs_new_protected:Npn \tl_gset_from_file:Nnn
22272   { \_tl_set_from_file>NNnn \tl_gset:Nn }
22273 \cs_generate_variant:Nn \tl_set_from_file:Nnn { c }
22274 \cs_generate_variant:Nn \tl_gset_from_file:Nnn { c }
22275 \cs_new_protected:Npn \_tl_set_from_file>NNnn #1#2#3#4
22276   {
22277     \_file_if_exist:nT {#4}
22278     {
22279       \group_begin:
22280         \exp_args:No \etex_everyeof:D
22281           { \c_tl_rescan_marker_tl \exp_not:N }
22282         #3 \scan_stop:
22283         \exp_after:wN \_tl_from_file_do:w
22284         \exp_after:wN \prg_do_nothing:
22285           \tex_input:D \l_file_internal_name_tl \scan_stop:
22286         \exp_args:NNNo \group_end:
22287         #1 #2 \l_tl_internal_a_tl
22288     }
22289   }
22290 \exp_args:Nno \use:nn
22291   { \cs_new_protected:Npn \_tl_from_file_do:w #1 }
22292   { \c_tl_rescan_marker_tl }
22293   { \tl_set:No \l_tl_internal_a_tl {#1} }
```

(End definition for `\tl_set_from_file:Nnn` and others. These functions are documented on page 235.)

```
\tl_set_from_file_x:Nnn
\tl_set_from_file_x:cnn
\color{red}\tl_gset_from_file_x:Nnn
\color{black}\tl_gset_from_file_x:cnn
\color{black}\_tl_set_from_file_x:NNnn
```

When reading a file and allowing expansion of the content, the set up only needs to prevent TeX complaining about the end of the file. That is done simply, with a group then used to trap the definition needed. Once the business is done using some scratch space, the tokens can be transferred to the real target.

```
22294 \cs_new_protected:Npn \tl_set_from_file_x:Nnn
22295   { \_tl_set_from_file_x:NNnn \tl_set:Nn }
22296 \cs_new_protected:Npn \tl_gset_from_file_x:Nnn
22297   { \_tl_set_from_file_x:NNnn \tl_gset:Nn }
22298 \cs_generate_variant:Nn \tl_set_from_file_x:Nnn { c }
22299 \cs_generate_variant:Nn \tl_gset_from_file_x:Nnn { c }
22300 \cs_new_protected:Npn \_tl_set_from_file_x:NNnn #1#2#3#4
22301   {
22302     \_file_if_exist:nT {#4}
22303   {
22304     \group_begin:
22305       \etex_everyeof:D { \exp_not:N }
22306       #3 \scan_stop:
22307       \tl_set:Nx \l__tl_internal_a_tl
22308         { \tex_input:D \l__file_internal_name_tl \c_space_token }
22309       \exp_args:NNNo \group_end:
22310       #1 #2 \l__tl_internal_a_tl
22311   }
22312 }
```

(End definition for `\tl_set_from_file_x:Nnn`, `\tl_gset_from_file_x:Nnn`, and `_tl_set_from_file_x:NNnn`. These functions are documented on page 236.)

42.14.1 Unicode case changing

The mechanisms needed for case changing are somewhat involved, particularly to allow for all of the special cases. These functions also require the appropriate data extracted from the Unicode documentation (either manually or automatically).

```
\tl_if_head_eq_catcode:oNTF Extra variants.
```

```
22313 \cs_generate_variant:Nn \tl_if_head_eq_catcode:nNTF { o }
```

(End definition for `\tl_if_head_eq_catcode:oNTF`. This function is documented on page 44.)

```
\color{red}\tl_lower_case:n
\color{red}\tl_upper_case:n
\color{red}\tl_mixed_case:n
\color{red}\tl_lower_case:nn
\color{red}\tl_upper_case:nn
\color{red}\tl_mixed_case:nn
```

The user level functions here are all wrappers around the internal functions for case changing. Note that `\tl_mixed_case:nn` could be done without an internal, but this way the logic is slightly clearer as everything essentially follows the same path.

```
22314 \cs_new:Npn \tl_lower_case:n { \_tl_change_case:nnn { lower } { } }
22315 \cs_new:Npn \tl_upper_case:n { \_tl_change_case:nnn { upper } { } }
22316 \cs_new:Npn \tl_mixed_case:n { \_tl_mixed_case:nn { } }
22317 \cs_new:Npn \tl_lower_case:nn { \_tl_change_case:nnn { lower } }
22318 \cs_new:Npn \tl_upper_case:nn { \_tl_change_case:nnn { upper } }
22319 \cs_new:Npn \tl_mixed_case:nn { \_tl_mixed_case:nn }
```

(End definition for `\tl_lower_case:n` and others. These functions are documented on page 232.)

```
\_tl_change_case:nnn
\_tl_change_case_aux:nnn
\_tl_change_case_loop:wnn
\_tl_change_case_output:nwn
\_tl_change_case_output:Vwn
\_tl_change_case_output:own
\_tl_change_case_output:vwn
\_tl_change_case_output:fwn
  \_tl_change_case_end:wn
\_tl_change_case_group:nwnn
\_tl_change_case_space:wnn
  \_tl_change_case_N_type:NNnnn
  \_tl_change_case_N_type:NNNnnnn
  \_tl_change_case_math:NNNnnnn
```

space. These different cases all work by re-reading #1 in the appropriate way, hence the repetition of #1 \q_recursion_stop.

```

22320 \cs_new:Npn \__tl_change_case:nnn #1#2#3
22321 {
22322     \etex_unexpanded:D \exp_after:wN
22323     {
22324         \exp:w
22325         \__tl_change_case_aux:nnn {#1} {#2} {#3}
22326     }
22327 }
22328 \cs_new:Npn \__tl_change_case_aux:nnn #1#2#3
22329 {
22330     \group_align_safe_begin:
22331     \__tl_change_case_loop:wnn
22332     #3 \q_recursion_tail \q_recursion_stop {#1} {#2}
22333     \__tl_change_case_result:n { }
22334 }
22335 \cs_new:Npn \__tl_change_case_loop:wnn #1 \q_recursion_stop
22336 {
22337     \tl_if_head_is_N_type:nTF {#1}
22338     { \__tl_change_case_N_type:Nwnn }
22339     {
22340         \tl_if_head_is_group:nTF {#1}
22341         { \__tl_change_case_group:nwnn }
22342         { \__tl_change_case_space:wnn }
22343     }
22344     #1 \q_recursion_stop
22345 }
```

Earlier versions of the code where only x-type expandable rather than f-type: this causes issues with nesting and so the slight performance hit is taken for a better outcome in usability terms. Setting up for f-type expandability has two requirements: a marker token after the main loop (see above) and a mechanism to “load” and finalise the result. That is handled in the code below, which includes the necessary material to end the \exp:w expansion.

```

22346 \cs_new:Npn \__tl_change_case_output:nwn #1#2 \__tl_change_case_result:n #3
22347 { #2 \__tl_change_case_result:n { #3 #1 } }
22348 \cs_generate_variant:Nn \__tl_change_case_output:nwn { V , o , v , f }
22349 \cs_new:Npn \__tl_change_case_end:wn #1 \__tl_change_case_result:n #2
22350 {
22351     \group_align_safe_end:
22352     \exp_end:
22353     #2
22354 }
```

Handling for the cases where the current argument is a brace group or a space is relatively easy. For the brace case, the routine works recursively, using the expandability of the mechanism to ensure that the result is finalised before storage. For the space case it is simply a question of removing the space in the input and storing it in the output. In both cases, and indeed for the N-type grabber, after removing the current item from the input __tl_change_case_loop:wnn is inserted in front of the remaining tokens.

```

22355 \cs_new:Npn \__tl_change_case_group:nwnn #1#2 \q_recursion_stop #3#4
22356 {
22357     \__tl_change_case_output:own
```

```

22358 {
22359   \exp_after:wN
22360   {
22361     \exp:w
22362     \__tl_change_case_aux:nnn {#3} {#4} {#1}
22363   }
22364 }
22365 \__tl_change_case_loop:wnn #2 \q_recursion_stop {#3} {#4}
22366 }
22367 \exp_last_unbraced:NNo \cs_new:Npn \__tl_change_case_space:wnn \c_space_tl
22368 {
22369   \__tl_change_case_output:nwn { ~ }
22370   \__tl_change_case_loop:wnn
22371 }

```

For N-type arguments there are several stages to the approach. First, a simply check for the end-of-input marker, which if found triggers the final clean up and output step. Assuming that is not the case, the first check is for math-mode escaping: this test can encompass control sequences or other N-type tokens so is handled up front.

```

22372 \cs_new:Npn \__tl_change_case_N_type:Nwnn #1#2 \q_recursion_stop
22373 {
22374   \quark_if_recursion_tail_stop_do:Nn #1
22375   { \__tl_change_case_end:wn }
22376   \exp_after:wN \__tl_change_case_N_type:NNNnnn
22377   \exp_after:wN #1 \l_tl_case_change_math_tl
22378   \q_recursion_tail ? \q_recursion_stop {#2}
22379 }

```

Looking for math mode escape first requires a loop over the possible token pairs to see if the current input (#1) matches an open-math case (#2). If it does then this test loop is ended and a new input-gathering one is begun. The latter simply transfers material from the input to the output without any expansion, testing each N-type token to see if it matches the close-math case required. If that is the situation then the “math loop” stops and resumes the main loop: as that might be either the standard case-changing one or the mixed-case alternative, it is not hard-coded into the math loop but is rather passed as argument #3 to __tl_change_case_math:NNNnnn. If no close-math token is found then the final clean-up will be forced (*i.e.* there is no assumption of “well-behaved” code in terms of math mode).

```

22380 \cs_new:Npn \__tl_change_case_N_type:NNNnnn #1#2#3
22381 {
22382   \quark_if_recursion_tail_stop_do:Nn #2
22383   { \__tl_change_case_N_type:Nnnn #1 }
22384   \token_if_eq_meaning:NNTF #1 #2
22385   {
22386     \use_i_delimit_by_q_recursion_stop:nw
22387     {
22388       \__tl_change_case_math:NNNnnn
22389       #1 #3 \__tl_change_case_loop:wnn
22390     }
22391   }
22392   { \__tl_change_case_N_type:NNNnnn #1 }
22393 }
22394 \cs_new:Npn \__tl_change_case_math:NNNnnn #1#2#3#4
22395 {

```

```

22396      \__tl_change_case_output:nwn {#1}
22397      \__tl_change_case_math_loop:wNNnn #4 \q_recursion_stop #2 #3
22398  }
22399 \cs_new:Npn \__tl_change_case_math_loop:wNNnn #1 \q_recursion_stop
22400  {
22401      \tl_if_head_is_N_type:nTF {#1}
22402      { \__tl_change_case_math:NwNNnn }
22403      {
22404          \tl_if_head_is_group:nTF {#1}
22405          { \__tl_change_case_math_group:nwNNnn }
22406          { \__tl_change_case_math_space:wNNnn }
22407      }
22408      #1 \q_recursion_stop
22409  }
22410 \cs_new:Npn \__tl_change_case_math:NwNNnn #1#2 \q_recursion_stop #3#4
22411  {
22412      \token_if_eq_meaning:NNTF \q_recursion_tail #1
22413      { \__tl_change_case_end:wn }
22414      {
22415          \__tl_change_case_output:nwn {#1}
22416          \token_if_eq_meaning:NNTF #1 #3
22417          { #4 #2 \q_recursion_stop }
22418          { \__tl_change_case_math_loop:wNNnn #2 \q_recursion_stop #3#4 }
22419      }
22420  }
22421 \cs_new:Npn \__tl_change_case_math_group:nwNNnn #1#2 \q_recursion_stop
22422  {
22423      \__tl_change_case_output:nwn { {#1} }
22424      \__tl_change_case_math_loop:wNNnn #2 \q_recursion_stop
22425  }
22426 \exp_last_unbraced:NNo
22427     \cs_new:Npn \__tl_change_case_math_space:wNNnn \c_space_tl
22428  {
22429      \__tl_change_case_output:nwn { ~ }
22430      \__tl_change_case_math_loop:wNNnn
22431  }

```

Once potential math-mode cases are filtered out the next stage is to test if the token grabbed is a control sequence: they cannot be used in the lookup table and also may require expansion. At this stage the loop code starting `__tl_change_case_loop:wnn` is inserted: all subsequent steps in the code which need a look-ahead are coded to rely on this and thus have w-type arguments if they may do a look-ahead.

```

22432 \cs_new:Npn \__tl_change_case_N_type:Nnnn #1#2#3#4
22433  {
22434      \token_if_cs:NTF #1
22435      { \__tl_change_case_cs_letterlike:Nnn #1 {#3} { } }
22436      { \__tl_change_case_char:Nnn #1 {#3} {#4} }
22437      \__tl_change_case_loop:wnn #2 \q_recursion_stop {#3} {#4}
22438  }

```

For character tokens there are some special cases to deal with then the majority of changes are covered by using the TeX data as a lookup along with expandable character generation. This avoids needing a very large number of macros or (as seen in earlier versions) a somewhat tricky split of the characters into various blocks. Notice that the

special case code may do a look-ahead so requires a final w-type argument whereas the core lookup table does not and also guarantees an output so f-type expansion may be used to obtain the case-changed result.

```

22439 \cs_new:Npn \__tl_change_case_char:Nnn #1#2#3
22440   {
22441     \cs_if_exist_use:cF { __tl_change_case_ #2 _ #3 :Nnw }
22442     { \use_ii:nn }
22443     #1
22444     {
22445       \use:c { __tl_change_case_ #2 _ sigma:Nnw } #1
22446       { \__tl_change_case_char:nN {#2} #1 }
22447     }
22448   }

```

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.)

```

22449 \cs_if_exist:NTF \utex_char:D
22450   {
22451     \cs_new:Npn \__tl_change_case_char:nN #1#2
22452     { \__tl_change_case_char_auxi:nN {#1} #2 }
22453   }
22454   {
22455     \cs_new:Npn \__tl_change_case_char:nN #1#2
22456     {
22457       \int_compare:nNnTF { '#2 } > { "80 "
22458         {
22459           \int_compare:nNnTF { '#2 } < { "E0 "
22460             { \__tl_change_case_char_UTFviii:nNNN {#1} #2 }
22461             {
22462               \int_compare:nNnTF { '#2 } < { "F0 "
22463                 { \__tl_change_case_char_UTFviii:nNNNN {#1} #2 }
22464                 { \__tl_change_case_char_UTFviii:nNNNNN {#1} #2 }
22465               }
22466             }
22467             { \__tl_change_case_char_auxi:nN {#1} #2 }
22468           }
22469         }
22470     \cs_new:Npn \__tl_change_case_char_auxi:nN #1#2
22471     {
22472       \__tl_change_case_output:fwn
22473       {
22474         \cs_if_exist_use:cF { c__unicode_ #1 _ \token_to_str:N #2 _tl }
22475           { \__tl_change_case_char_auxii:nN {#1} #2 }
22476       }
22477     }
22478 \cs_if_exist:NTF \utex_char:D
22479   {
22480     \cs_new:Npn \__tl_change_case_char_auxii:nN #1#2
22481     {
22482       \int_compare:nNnTF { \use:c { __tl_lookup_ #1 :N } #2 } = { 0 }
```

```

22483 { \exp_stop_f: #2 }
22484 {
22485   \char_generate:nn
22486     { \use:c { __tl_lookup_ #1 :N } #2 }
22487     { \char_value_catcode:n { \use:c { __tl_lookup_ #1 :N } #2 } }
22488   }
22489 }
22490 \cs_new_protected:Npn \__tl_lookup_lower:N #1 { \tex_lccode:D '#1 }
22491 \cs_new_protected:Npn \__tl_lookup_upper:N #1 { \tex_uccode:D '#1 }
22492 \cs_new_eq:NN \__tl_lookup_title:N \__tl_lookup_upper:N
22493 }
22494 {
22495   \cs_new:Npn \__tl_change_case_char_auxii:nN #1#2 { \exp_stop_f: #2 }
22496   \cs_new:Npn \__tl_change_case_char_UTFviii:nNNN #1#2#3#4
22497     { \__tl_change_case_char_UTFviii:nnN {#1} {#2#4} #3 }
22498   \cs_new:Npn \__tl_change_case_char_UTFviii:nNNNN #1#2#3#4#5
22499     { \__tl_change_case_char_UTFviii:nnN {#1} {#2#4#5} #3 }
22500   \cs_new:Npn \__tl_change_case_char_UTFviii:nNNNNN #1#2#3#4#5#6
22501     { \__tl_change_case_char_UTFviii:nnN {#1} {#2#4#5#6} #3 }
22502   \cs_new:Npn \__tl_change_case_char_UTFviii:nnN #1#2#3
22503     {
22504       \cs_if_exist:cTF { c__unicode_ #1 _ \tl_to_str:n {#2} _tl }
22505         {
22506           \__tl_change_case_output:vwn
22507             { c__unicode_ #1 _ \tl_to_str:n {#2} _tl }
22508         }
22509         { \__tl_change_case_output:nwn {#2} }
22510       #3
22511     }
22512   }

```

Before dealing with general control sequences there are the special ones to deal with. Letter-like control sequences are a simple look-up, while for accents the loop is much as done elsewhere. Notice that we have a no-op test to make sure there is no unexpected expansion of letter-like input. The third argument here is needed for mixed casing, where it if there is a hit there has to be a change-of-path.

```

22513 \cs_new:Npn \__tl_change_case_cs_letterlike:Nnn #1#2#3
22514   {
22515     \cs_if_exist:cTF { c__tl_change_case_ #2 _ \token_to_str:N #1 _tl }
22516     {
22517       \__tl_change_case_output:vwn
22518         { c__tl_change_case_ #2 _ \token_to_str:N #1 _tl }
22519       #3
22520     }
22521   {
22522     \cs_if_exist:cTF
22523       {
22524         c__tl_change_case_
22525         \str_if_eq:nnTF {#2} { lower } { upper } { lower }
22526           _ \token_to_str:N #1 _tl
22527       }
22528     {
22529       \__tl_change_case_output:nwn {#1}
22530       #3

```

```

22531     }
22532     {
22533         \exp_after:wN \__tl_change_case_cs_accents:NN
22534             \exp_after:wN #1 \l_tl_case_change_accents_tl
22535                 \q_recursion_tail \q_recursion_stop
22536             }
22537         }
22538     }
22539 \cs_new:Npn \__tl_change_case_cs_accents:NN #1#2
22540     {
22541         \quark_if_recursion_tail_stop_do:Nn #2
22542             { \__tl_change_case_cs:N #1 }
22543         \str_if_eq:nnTF {#1} {#2}
22544             {
22545                 \use_i_delimit_by_q_recursion_stop:nw
22546                     { \__tl_change_case_output:nwn {#1} }
22547             }
22548             { \__tl_change_case_cs_accents:NN #1 }
22549     }

```

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: that comes *after* the loop function which is therefore rearranged. In a L^AT_EX 2 _{ε} context, \protect needs to be treated specially, to prevent expansion of the next token but output it without braces.

```

22550 \cs_new:Npn \__tl_change_case_cs:N #1
22551     {
22552     (*package)
22553         \str_if_eq:nnTF {#1} { \protect } { \__tl_change_case_protect:wNN }
22554     
```

 $\langle/$ package>
 \rangle
 $\exp_after:wN __tl_change_case_cs:NN$
 $\exp_after:wN #1 \l_tl_case_change_exclude_tl$
 $\q_recursion_tail \q_recursion_stop$
 $\}$
 $__tl_change_case_cs_accents:NN #1#2$
 $\{$
 $\quark_if_recursion_tail_stop_do:Nn #2$
 $\{$
 $__tl_change_case_cs_expand:Nnw #1$
 $\{ __tl_change_case_output:nwn {#1} \}$
 $\}$
 $\str_if_eq:nnTF {#1} {#2}$
 $\{$
 $\use_i_delimit_by_q_recursion_stop:nw$
 $\{ __tl_change_case_cs:NNn #1 \}$
 $\}$
 $\{ __tl_change_case_cs:NN #1 \}$
 $\}$
 $__tl_change_case_output:nwn { #1 {#3} }$
 $#2$
 $\}$
 $\langle/$ package>

```

22579 \cs_new:Npn \__tl_change_case_protect:wNN #1 \q_recursion_stop #2 #3
22580   { \__tl_change_case_output:nwn { \protect #3 } #2 }
22581 
```

When a control sequence is not on the exclude list the other test if to see if it is expandable. Once again, if there is a hit then the loop function is grabbed as part of the clean-up and reinserted before the now expanded material. The test for expandability has to check for end-of-recursion as it is needed by the look-ahead code which might hit the end of the input. The test is done in two parts as \bool_if:nTF will choke if #1 is (!

```

22582 \cs_new:Npn \__tl_change_case_if_expandable:NTF #1
22583   {
22584     \token_if_expandable:NTF #1
22585     {
22586       \bool_lazy_any:nTF
22587       {
22588         { \token_if_eq_meaning_p:NN \q_recursion_tail #1 }
22589         { \token_if_protected_macro_p:N      #1 }
22590         { \token_if_protected_long_macro_p:N #1 }
22591       }
22592       { \use_i:nn }
22593       { \use_i:nn }
22594     }
22595     { \use_ii:nn }
22596   }
22597 \cs_new:Npn \__tl_change_case_cs_expand:Nnw #1#2
22598   {
22599     \__tl_change_case_if_expandable:NTF #1
22600     { \__tl_change_case_cs_expand>NN #1 }
22601     { #2 }
22602   }
22603 \cs_new:Npn \__tl_change_case_cs_expand>NN #1#2
22604   { \exp_after:wN #2 #1 }

```

(End definition for __tl_change_case:nnn and others.)

__tl_change_case_lower_sigma:Nnw
__tl_change_case_lower_sigma:w
__tl_change_case_lower_sigma:Nw
__tl_change_case_upper_sigma:Nnw

```

22605 \cs_new:Npn \__tl_change_case_lower_sigma:Nnw #1#2#3#4 \q_recursion_stop
22606   {
22607     \int_compare:nNnTF { '#1 } = { "03A3" }
22608     {
22609       \__tl_change_case_output:fwn
22610       { \__tl_change_case_lower_sigma:w #4 \q_recursion_stop }
22611     }
22612     {#2}
22613     #3 #4 \q_recursion_stop
22614   }
22615 \cs_new:Npn \__tl_change_case_lower_sigma:w #1 \q_recursion_stop
22616   {
22617     \tl_if_head_is_N_type:nTF {#1}
22618     { \__tl_change_case_lower_sigma:Nw #1 \q_recursion_stop }
22619     { \c_unicode_final_sigma_tl }
22620   }
22621 \cs_new:Npn \__tl_change_case_lower_sigma:Nw #1#2 \q_recursion_stop

```

```

22622 {
22623   \_tl_change_case_if_expandable:NTF #1
22624   {
22625     \exp_after:wN \_tl_change_case_lower_sigma:w #1
22626     #2 \q_recursion_stop
22627   }
22628   {
22629     \token_if_letter:NTF #1
22630     { \c_unicode_std_sigma_t1 }
22631     { \c_unicode_final_sigma_t1 }
22632   }
22633 }
```

Simply skip to the final step for upper casing.

```

22634 \cs_new_eq:NN \_tl_change_case_upper_sigma:Nnw \use_i:nn
(End definition for \_tl_change_case_lower_sigma:Nnw and others.)
```

_tl_change_case_lower_tr:Nnw
_tl_change_case_lower_tr_auxi:Nw
_tl_change_case_lower_tr_auxii:Nw
_tl_change_case_upper_tr:Nnw
_tl_change_case_lower_az:Nnw
_tl_change_case_upper_az:Nnw

```

22635 \cs_if_exist:NTF \utex_char:D
22636   {
22637     \cs_new:Npn \_tl_change_case_lower_tr:Nnw #1#2
22638     {
22639       \int_compare:nNnTF { '#1 } = { "0049 }
22640       { \_tl_change_case_lower_tr_auxi:Nw }
22641       {
22642         \int_compare:nNnTF { '#1 } = { "0130 }
22643         { \_tl_change_case_output:nwn { i } }
22644         {#2}
22645       }
22646     }
```

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, which is done by the \use_i:nn (it grabs _tl_change_case_loop:wn and the dot-above char and discards the latter).

```

22647   \cs_new:Npn \_tl_change_case_lower_tr_auxi:Nw #1#2 \q_recursion_stop
22648   {
22649     \tl_if_head_is_N_type:nTF {#2}
22650     { \_tl_change_case_lower_tr_auxii:Nw #2 \q_recursion_stop }
22651     { \_tl_change_case_output:Vwn \c_unicode_dotless_i_t1 }
22652     #1 #2 \q_recursion_stop
22653   }
22654   \cs_new:Npn \_tl_change_case_lower_tr_auxii:Nw #1#2 \q_recursion_stop
22655   {
22656     \_tl_change_case_if_expandable:NTF #1
22657     {
22658       \exp_after:wN \_tl_change_case_lower_tr_auxi:Nw #1
22659       #2 \q_recursion_stop
22660     }
22661   {
22662     \bool_lazy_or:nnTF
```

```

22663      { \token_if_cs_p:N #1 }
22664      { ! \int_compare_p:nNn { '#1 } = { "0307" } }
22665      { \__tl_change_case_output:Vwn \c_unicode_dotless_i_tl }
22666      {
22667          \__tl_change_case_output:nwn { i }
22668          \use_i:nn
22669      }
22670  }
22671 }
22672 }
```

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.

```

22673  {
22674      \cs_new:Npn \__tl_change_case_lower_tr:Nnw #1#2
22675      {
22676          \int_compare:nNnTF { '#1 } = { "0049" }
22677          { \__tl_change_case_output:Vwn \c_unicode_dotless_i_tl }
22678          {
22679              \int_compare:nNnTF { '#1 } = { 196 }
22680              { \__tl_change_case_lower_tr_auxi:Nw #1 {#2} }
22681              {#2}
22682          }
22683      }
22684      \cs_new:Npn \__tl_change_case_lower_tr_auxi:Nw #1#2#3#4
22685      {
22686          \int_compare:nNnTF { '#4 } = { 176 }
22687          {
22688              \__tl_change_case_output:nwn { i }
22689              #3
22690          }
22691          {
22692              #2
22693              #3 #4
22694          }
22695      }
22696  }
```

Upper casing is easier: just one exception with no context.

```

22697 \cs_new:Npn \__tl_change_case_upper_tr:Nnw #1#2
22698  {
22699      \int_compare:nNnTF { '#1 } = { "0069" }
22700      { \__tl_change_case_output:Vwn \c_unicode_dotted_I_tl }
22701      {#2}
22702  }
```

Straight copies.

```

22703 \cs_new_eq:NN \__tl_change_case_lower_az:Nnw \__tl_change_case_lower_tr:Nnw
22704 \cs_new_eq:NN \__tl_change_case_upper_az:Nnw \__tl_change_case_upper_tr:Nnw
```

(End definition for __tl_change_case_lower_tr:Nnw and others.)

__tl_change_case_lower_lt:Nnw
__tl_change_case_lower_lt:nNnw
__tl_change_case_lower_lt:nnw
__tl_change_case_lower_lt:Nw
__tl_change_case_lower_lt:NNw
__tl_change_case_upper_lt:Nnw
__tl_change_case_upper_lt:nnw
__tl_change_case_upper_lt:Nw
__tl_change_case_upper_lt:NNw

For Lithuanian, the issue to be dealt with is dots over lower case letters: these should be present if there is another accent. That means that there is some work to do when

lower casing I and J. The first step is a simple match attempt: \c__tl_accents_lt_tl contains accented upper case letters which should gain a dot-above char in their lower case form. This is done using f-type expansion so only one pass is needed to find if it works or not. If there was no hit, the second stage is to check for I, J and I-ogonek, and if the current char is a match to look for a following accent.

```

22705 \cs_new:Npn \__tl_change_case_lower_lt:Nnw #1
22706 {
22707   \exp_args:Nf \__tl_change_case_lower_lt:nNnw
22708   { \str_case:nVF #1 \c__unicode_accents_lt_tl \exp_stop_f: }
22709   #1
22710 }
22711 \cs_new:Npn \__tl_change_case_lower_lt:nNnw #1#2
22712 {
22713   \tl_if_blank:nTF {#1}
22714   {
22715     \exp_args:Nf \__tl_change_case_lower_lt:nnw
22716     {
22717       \int_case:nnF {'#2}
22718       {
22719         { "0049 } i
22720         { "004A } j
22721         { "012E } \c__unicode_i_ogonek_tl
22722       }
22723       \exp_stop_f:
22724     }
22725   }
22726   {
22727     \__tl_change_case_output:nwn {#1}
22728     \use_none:n
22729   }
22730 }
22731 \cs_new:Npn \__tl_change_case_lower_lt:nnw #1#2
22732 {
22733   \tl_if_blank:nTF {#1}
22734   {#2}
22735   {
22736     \__tl_change_case_output:nwn {#1}
22737     \__tl_change_case_lower_lt:Nw
22738   }
22739 }
```

Grab the next char and see if it is one of the accents used in Lithuanian: if it is, add the dot-above char into the output.

```

22740 \cs_new:Npn \__tl_change_case_lower_lt:Nw #1#2 \q_recursion_stop
22741 {
22742   \tl_if_head_is_N_type:nT {#2}
22743   { \__tl_change_case_lower_lt>NNw }
22744   #1 #2 \q_recursion_stop
22745 }
22746 \cs_new:Npn \__tl_change_case_lower_lt>NNw #1#2#3 \q_recursion_stop
22747 {
22748   \__tl_change_case_if_expandable:NTF #2
22749   {
22750     \exp_after:wN \__tl_change_case_lower_lt:Nw \exp_after:wN #1 #2
```

```

22751      #3 \q_recursion_stop
22752  }
22753  {
22754      \bool_lazy_and:nnt
22755      { ! \token_if_cs_p:N #2 }
22756      {
22757          \bool_lazy_any_p:n
22758          {
22759              \int_compare_p:nNn { '#2 } = { "0300" }
22760              \int_compare_p:nNn { '#2 } = { "0301" }
22761              \int_compare_p:nNn { '#2 } = { "0303" }
22762          }
22763      }
22764      { \__tl_change_case_output:Vwn \c_unicode_dot_above_tl }
22765      #1 #2#3 \q_recursion_stop
22766  }
22767  }

```

For upper casing, the test required is for a dot-above char after an I, J or I-ogonek. First a test for the appropriate letter, and if found a look-ahead and potentially one token dropped.

```

22768 \cs_new:Npn \__tl_change_case_upper_lt:Nnw #1
22769  {
22770      \exp_args:Nf \__tl_change_case_upper_lt:nnw
22771      {
22772          \int_case:nnF {'#1}
22773          {
22774              { "0069" } I
22775              { "006A" } J
22776              { "012F" } \c_unicode_I_ogonek_tl
22777          }
22778          \exp_stop_f:
22779      }
2280  }
2281 \cs_new:Npn \__tl_change_case_upper_lt:nnw #1#2
2282  {
2283      \tl_if_blank:nTF {#1}
2284      {#2}
2285      {
2286          \__tl_change_case_output:nnn {#1}
2287          \__tl_change_case_upper_lt:Nw
2288      }
2289  }
2290 \cs_new:Npn \__tl_change_case_upper_lt:Nw #1#2 \q_recursion_stop
2291  {
2292      \tl_if_head_is_N_type:nT {#2}
2293      { \__tl_change_case_upper_lt>NNw }
2294      #1 #2 \q_recursion_stop
2295  }
2296 \cs_new:Npn \__tl_change_case_upper_lt>NNw #1#2#3 \q_recursion_stop
2297  {
2298      \__tl_change_case_if_expandable:NTF #2
2299      {
2300          \exp_after:wN \__tl_change_case_upper_lt:Nw \exp_after:wN #1 #2

```

```

22801      #3 \q_recursion_stop
22802 }
22803 {
22804     \bool_lazy_and:nTF
22805     { ! \token_if_cs_p:N #2 }
22806     { \int_compare_p:nNn { '#2 } = { "0307" } }
22807     { #1 }
22808     { #1 #2 }
22809     #3 \q_recursion_stop
22810   }
22811 }
```

(End definition for `_tl_change_case_lower_lt:Nnw` and others.)

`_tl_change_case_upper_de-alt:Nnw` A simple alternative version for German.

```

22812 \cs_new:cpn { __tl_change_case_upper_de-alt:Nnw } #1#2
22813 {
22814   \int_compare:nNnTF { '#1 } = { 223 }
22815   { \_tl_change_case_output:Vwn \c__unicode_upper_Eszett_tl }
22816   { #2 }
22817 }
```

(End definition for `_tl_change_case_upper_de-alt:Nnw`.)

`_unicode_codepoint_to_UTFviii:n` This code will convert a codepoint into the correct UTF-8 representation. As there are a variable number of octets, the result starts with the numeral 1–4 to indicate the nature of the returned value. Note that this code will cover the full range even though at this stage it is not required here. Also note that longer-term this is likely to need a public interface and/or moving to `I3str` (see experimental string conversions). In terms of the algorithm itself, see <https://en.wikipedia.org/wiki/UTF-8> for the octet pattern.

```

22818 \cs_new:Npn \_unicode_codepoint_to_UTFviii:n #1
22819 {
22820   \exp_args:Nf \_unicode_codepoint_to_UTFviii_auxi:n
22821   { \int_eval:n {#1} }
22822 }
22823 \cs_new:Npn \_unicode_codepoint_to_UTFviii_auxi:n #1
22824 {
22825   \if_int_compare:w #1 > "80 ~
22826   \if_int_compare:w #1 < "800 ~
22827     2
22828     \_unicode_codepoint_to_UTFviii_auxi:Nnn C {#1} { 64 }
22829     \_unicode_codepoint_to_UTFviii_auxi:n {#1}
22830   \else:
22831     \if_int_compare:w #1 < "10000 ~
22832       3
22833       \_unicode_codepoint_to_UTFviii_auxi:Nnn E {#1} { 64 * 64 }
22834       \_unicode_codepoint_to_UTFviii_auxi:n {#1}
22835       \_unicode_codepoint_to_UTFviii_auxi:n
22836       { \int_div_truncate:nn {#1} { 64 } }
22837   \else:
22838     4
22839     \_unicode_codepoint_to_UTFviii_auxi:Nnn F
22840     {#1} { 64 * 64 * 64 }
22841     \_unicode_codepoint_to_UTFviii_auxi:n
```

```

22842     { \int_div_truncate:nn {#1} { 64 * 64 } }
22843     \__unicode_codepoint_to_UTFviii_auxiii:n
22844     { \int_div_truncate:nn {#1} { 64 } }
22845     \__unicode_codepoint_to_UTFviii_auxiii:n {#1}
22846
22847     \fi:
22848     \fi:
22849   \else:
22850     1 {#1}
22851   \fi:
22852 }
22853 \cs_new:Npn \__unicode_codepoint_to_UTFviii_auxii:Nnn #1#2#3
22854   { { \int_eval:n { "#10 + \int_div_truncate:nn {#2} {#3} } } }
22855 \cs_new:Npn \__unicode_codepoint_to_UTFviii_auxiii:n #
22856   { { \int_eval:n { \int_mod:nn {#1} { 64 } + 128 } } }

(End definition for \__unicode_codepoint_to_UTFviii:n and others.)

```

\c__unicode_std_sigma_tl The above needs various special token lists containg pre-formed characters. This set are only available in Unicode engines, with no-op definitions for 8-bit use.

```

22857 \cs_if_exist:NTF \utex_char:D
22858   {
22859     \tl_const:Nx \c__unicode_std_sigma_tl { \utex_char:D "03C3 ~ }
22860     \tl_const:Nx \c__unicode_final_sigma_tl { \utex_char:D "03C2 ~ }
22861     \tl_const:Nx \c__unicode_accents_lt_tl
22862     {
22863       \utex_char:D "00CC ~
22864         { \utex_char:D "0069 ~ \utex_char:D "0307 ~ \utex_char:D "0300 ~ }
22865       \utex_char:D "00CD ~
22866         { \utex_char:D "0069 ~ \utex_char:D "0307 ~ \utex_char:D "0301 ~ }
22867       \utex_char:D "0128 ~
22868         { \utex_char:D "0069 ~ \utex_char:D "0307 ~ \utex_char:D "0303 ~ }
22869     }
22870     \tl_const:Nx \c__unicode_dot_above_tl { \utex_char:D "0307 ~ }
22871     \tl_const:Nx \c__unicode_upper_Eszett_tl { \utex_char:D "1E9E ~ }
22872   }
22873   {
22874     \tl_const:Nn \c__unicode_std_sigma_tl { }
22875     \tl_const:Nn \c__unicode_final_sigma_tl { }
22876     \tl_const:Nn \c__unicode_accents_lt_tl { }
22877     \tl_const:Nn \c__unicode_dot_above_tl { }
22878     \tl_const:Nn \c__unicode_upper_Eszett_tl { }
22879   }

```

(End definition for \c__unicode_std_sigma_tl and others.)

\c__unicode_dotless_i_tl For cases where there is an 8-bit option in the T1 font set up, a variant is provided in both cases.

```

22880 \group_begin:
22881   \cs_if_exist:NTF \utex_char:D
22882   {
22883     \cs_set_protected:Npn \__tl_tmp:w #1#2
22884       { \tl_const:Nx #1 { \utex_char:D "#2 ~ } }
22885   }

```

```

22886   {
22887     \cs_set_protected:Npn \__tl_tmp:w #1#2
22888     {
22889       \group_begin:
2290         \cs_set_protected:Npn \__tl_tmp:w ##1##2##3
2291         {
2292           \tl_const:Nx #1
2293           {
2294             \exp_after:wN \exp_after:wN \exp_after:wN
2295               \exp_not:N \__char_generate:nn {##2} { 13 }
2296             \exp_after:wN \exp_after:wN \exp_after:wN
2297               \exp_not:N \__char_generate:nn {##3} { 13 }
2298           }
2299         }
2300       \tl_set:Nx \l__tl_internal_a_tl
2301         { \__unicode_codepoint_to_UTFviii:n {"#2} }
2302       \exp_after:wN \__tl_tmp:w \l__tl_internal_a_tl
2303     \group_end:
2304   }
2305 }
2306 \__tl_tmp:w \c__unicode_dotless_i_tl { 0131 }
2307 \__tl_tmp:w \c__unicode_dotted_I_tl { 0130 }
2308 \__tl_tmp:w \c__unicode_i_ogonek_tl { 012F }
2309 \__tl_tmp:w \c__unicode_I_ogonek_tl { 012E }
2310 \group_end:

```

(End definition for `\c__unicode_dotless_i_tl` and others.)

For 8-bit engines we now need to define the case-change data for the multi-octet mappings. 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.

```

2311 \group_begin:
2312   \bool_lazy_or:nnT
2313   { \sys_if_engine_pdftex_p: }
2314   { \sys_if_engine_uptex_p: }
2315   {
2316     \cs_set_protected:Npn \__tl_loop:nn #1#2
2317     {
2318       \quark_if_recursion_tail_stop:n {#1}
2319       \tl_set:Nx \l__tl_internal_a_tl
2320       {
2321         \__unicode_codepoint_to_UTFviii:n {"#1}
2322         \__unicode_codepoint_to_UTFviii:n {"#2}
2323       }
2324       \exp_after:wN \__tl_tmp:w \l__tl_internal_a_tl
2325       \__tl_loop:nn
2326     }
2327     \cs_set_protected:Npn \__tl_tmp:w #1#2#3#4#5#6
2328     {
2329       \tl_const:cx
2330       {
2331         \c__unicode_lower_
2332         \char_generate:nn {#2} { 12 }
2333         \char_generate:nn {#3} { 12 }
2334       }
2335     }
2336   }
2337 
```

```

22934     _tl
22935 }
22936 {
22937     \exp_after:wN \exp_after:wN \exp_after:wN
22938         \exp_not:N \__char_generate:nn {#5} { 13 }
22939     \exp_after:wN \exp_after:wN \exp_after:wN
22940         \exp_not:N \__char_generate:nn {#6} { 13 }
22941 }
22942 \tl_const:cx
22943 {
22944     c__unicode_upper_
22945     \char_generate:nn {#5} { 12 }
22946     \char_generate:nn {#6} { 12 }
22947     _tl
22948 }
22949 {
22950     \exp_after:wN \exp_after:wN \exp_after:wN
22951         \exp_not:N \__char_generate:nn {#2} { 13 }
22952     \exp_after:wN \exp_after:wN \exp_after:wN
22953         \exp_not:N \__char_generate:nn {#3} { 13 }
22954 }
22955 }
22956 \__tl_loop:nn
22957 { 00C0 } { 00E0 }
22958 { 00C2 } { 00E2 }
22959 { 00C3 } { 00E3 }
22960 { 00C4 } { 00E4 }
22961 { 00C5 } { 00E5 }
22962 { 00C6 } { 00E6 }
22963 { 00C7 } { 00E7 }
22964 { 00C8 } { 00E8 }
22965 { 00C9 } { 00E9 }
22966 { 00CA } { 00EA }
22967 { 00CB } { 00EB }
22968 { 00CC } { 00EC }
22969 { 00CD } { 00ED }
22970 { 00CE } { 00EE }
22971 { 00CF } { 00EF }
22972 { 00D0 } { 00F0 }
22973 { 00D1 } { 00F1 }
22974 { 00D2 } { 00F2 }
22975 { 00D3 } { 00F3 }
22976 { 00D4 } { 00F4 }
22977 { 00D5 } { 00F5 }
22978 { 00D6 } { 00F6 }
22979 { 00D8 } { 00F8 }
22980 { 00D9 } { 00F9 }
22981 { 00DA } { 00FA }
22982 { 00DB } { 00FB }
22983 { 00DC } { 00FC }
22984 { 00DD } { 00FD }
22985 { 00DE } { 00FE }
22986 { 0100 } { 0101 }
22987 { 0102 } { 0103 }

```

22988 { 0104 } { 0105 }
22989 { 0106 } { 0107 }
22990 { 0108 } { 0109 }
22991 { 010A } { 010B }
22992 { 010C } { 010D }
22993 { 010E } { 010F }
22994 { 0110 } { 0111 }
22995 { 0112 } { 0113 }
22996 { 0114 } { 0115 }
22997 { 0116 } { 0117 }
22998 { 0118 } { 0119 }
22999 { 011A } { 011B }
23000 { 011C } { 011D }
23001 { 011E } { 011F }
23002 { 0120 } { 0121 }
23003 { 0122 } { 0123 }
23004 { 0124 } { 0125 }
23005 { 0128 } { 0129 }
23006 { 012A } { 012B }
23007 { 012C } { 012D }
23008 { 012E } { 012F }
23009 { 0132 } { 0133 }
23010 { 0134 } { 0135 }
23011 { 0136 } { 0137 }
23012 { 0139 } { 013A }
23013 { 013B } { 013C }
23014 { 013E } { 013F }
23015 { 0141 } { 0142 }
23016 { 0143 } { 0144 }
23017 { 0145 } { 0146 }
23018 { 0147 } { 0148 }
23019 { 014A } { 014B }
23020 { 014C } { 014D }
23021 { 014E } { 014F }
23022 { 0150 } { 0151 }
23023 { 0152 } { 0153 }
23024 { 0154 } { 0155 }
23025 { 0156 } { 0157 }
23026 { 0158 } { 0159 }
23027 { 015A } { 015B }
23028 { 015C } { 015D }
23029 { 015E } { 015F }
23030 { 0160 } { 0161 }
23031 { 0162 } { 0163 }
23032 { 0164 } { 0165 }
23033 { 0168 } { 0169 }
23034 { 016A } { 016B }
23035 { 016C } { 016D }
23036 { 016E } { 016F }
23037 { 0170 } { 0171 }
23038 { 0172 } { 0173 }
23039 { 0174 } { 0175 }
23040 { 0176 } { 0177 }
23041 { 0178 } { 00FF }

```

23042      { 0179 } { 017A }
23043      { 017B } { 017C }
23044      { 017D } { 017E }
23045      { 01CD } { 01CE }
23046      { 01CF } { 01D0 }
23047      { 01D1 } { 01D2 }
23048      { 01D3 } { 01D4 }
23049      { 01E2 } { 01E3 }
23050      { 01E6 } { 01E7 }
23051      { 01E8 } { 01E9 }
23052      { 01EA } { 01EB }
23053      { 01F4 } { 01F5 }
23054      { 0218 } { 0219 }
23055      { 021A } { 021B }
23056      \q_recursion_tail ?
23057      \q_recursion_stop
23058      \cs_set_protected:Npn \__tl_tmp:w #1#2#3
23059      {
23060          \group_begin:
23061          \cs_set_protected:Npn \__tl_tmp:##1##2##3
23062          {
23063              \tl_const:cx
23064              {
23065                  c__unicode_ #3 _
23066                  \char_generate:nn {##2} { 12 }
23067                  \char_generate:nn {##3} { 12 }
23068                  _tl
23069              }
23070              {##2}
23071          }
23072          \tl_set:Nx \l__tl_internal_a_tl
23073          { \__unicode_codepoint_to_UTFviii:n { "#1" } }
23074          \exp_after:wN \__tl_tmp:w \l__tl_internal_a_tl
23075          \group_end:
23076      }
23077      \__tl_tmp:w { 00DF } { SS } { upper }
23078      \__tl_tmp:w { 00DF } { Ss } { title }
23079      \__tl_tmp:w { 0131 } { I } { upper }
23080  }
23081  \group_end:

```

The (fixed) look-up mappings for letter-like control sequences.

```

23082  \group_begin:
23083  \cs_set_protected:Npn \__tl_change_case_setup:NN #1#2
23084  {
23085      \quark_if_recursion_tail_stop:N #1
23086      \tl_const:cn { c__tl_change_case_lower_ \token_to_str:N #1 _tl } { #2 }
23087      \tl_const:cn { c__tl_change_case_upper_ \token_to_str:N #2 _tl } { #1 }
23088      \__tl_change_case_setup:NN
23089  }
23090  \__tl_change_case_setup:NN
23091  \AA \aa
23092  \AE \ae
23093  \DH \dh
23094  \DJ \dj

```

```

23095 \IJ \ij
23096 \L \l
23097 \NG \ng
23098 \O \o
23099 \OE \oe
23100 \SS \ss
23101 \TH \th
23102 \q_recursion_tail ?
23103 \q_recursion_stop
23104 \tl_const:cn { c__tl_change_case_upper_ \token_to_str:N \i _tl } { I }
23105 \tl_const:cn { c__tl_change_case_upper_ \token_to_str:N \j _tl } { J }
23106 \group_end:

```

\l_t1_case_change_accents_t1

A list of accents to leave alone.

```

23107 \tl_new:N \l_t1_case_change_accents_t1
23108 \tl_set:Nn \l_t1_case_change_accents_t1
23109 { \" \' \\" \^ \` \~ \c \H \k \r \t \u \v }

```

(End definition for \l_t1_case_change_accents_t1. This variable is documented on page 234.)

```

\_\_tl_mixed_case:nn
\_\_tl_mixed_case_aux:nn
\_\_tl_mixed_case_loop:wn
\_\_tl_mixed_case_group:nwn
\_\_tl_mixed_case_space:wn
\_\_tl_mixed_case_N_type:Nwn
  \_\_tl_mixed_case_N_type:NNNnn
\_\_tl_mixed_case_N_type:Nnn
  \_\_tl_mixed_case_letterlike:Nw
\_\_tl_mixed_case_char:N
\_\_tl_mixed_case_skip:N
\_\_tl_mixed_case_skip:NN
  \_\_tl_mixed_case_skip_tidy:Nwn
\_\_tl_mixed_case_char:nN

```

```

23110 \cs_new:Npn \_\_tl_mixed_case:nn #1#2
23111 {
23112   \etex_unexpanded:D \exp_after:wn
23113   {
23114     \exp:w
23115     \_\_tl_mixed_case_aux:nn {#1} {#2}
23116   }
23117 }
23118 \cs_new:Npn \_\_tl_mixed_case_aux:nn #1#2
23119 {
23120   \group_align_safe_begin:
23121   \_\_tl_mixed_case_loop:wn
23122   #2 \q_recursion_tail \q_recursion_stop {#1}
23123   \_\_tl_change_case_result:n { }
23124 }
23125 \cs_new:Npn \_\_tl_mixed_case_loop:wn #1 \q_recursion_stop
23126 {
23127   \tl_if_head_is_N_type:nTF {#1}
23128   { \_\_tl_mixed_case_N_type:Nwn }
23129   {

```

Mixed (title) casing requires some custom handling of the case changing of the first letter in the input followed by a switch to the normal lower casing routine. That could be covered by passing a set of functions to generic routines, but at the cost of making the process rather opaque. Instead, the approach taken here is to use a dedicated set of functions which keep the different loop requirements clearly separate.

The main loop looks for the first “real” char in the input (skipping any pre-letter chars). Once one is found, it is case changed to upper case but first checking that there is not an entry in the exceptions list. Note that simply grabbing the first token in the input is no good here: it can’t handle pre-letter tokens or any special treatment of the first letter found (*e.g.* words starting with *i* in Turkish). Spaces at the start of the input are passed through without counting as being the “start” of the first word, while a brace group is assumed to be contain the first char with everything after the brace therefore lower cased.

```

23130     \tl_if_head_is_group:nTF {#1}
23131         { \__tl_mixed_case_group:nwn }
23132         { \__tl_mixed_case_space:wn }
23133     }
23134     #1 \q_recursion_stop
23135   }
23136 \cs_new:Npn \__tl_mixed_case_group:nwn #1#2 \q_recursion_stop #3
23137   {
23138     \__tl_change_case_output:own
23139   {
23140     \exp_after:wn
23141     {
23142       \exp:w
23143       \__tl_mixed_case_aux:nn {#3} {#1}
23144     }
23145   }
23146   \__tl_change_case_loop:wnn #2 \q_recursion_stop { lower } {#3}
23147 }
23148 \exp_last_unbraced:NNo \cs_new:Npn \__tl_mixed_case_space:wn \c_space_tl
23149   {
23150     \__tl_change_case_output:nwn { ~ }
23151     \__tl_mixed_case_loop:wn
23152   }
23153 \cs_new:Npn \__tl_mixed_case_N_type:Nwn #1#2 \q_recursion_stop
23154   {
23155     \quark_if_recursion_tail_stop_do:Nn #1
23156     { \__tl_change_case_end:wn }
23157     \exp_after:wN \__tl_mixed_case_N_type:NNNnn
23158     \exp_after:wN #1 \l_tl_case_change_math_tl
23159     \q_recursion_tail ? \q_recursion_stop {#2}
23160   }
23161 \cs_new:Npn \__tl_mixed_case_N_type:NNNnn #1#2#3
23162   {
23163     \quark_if_recursion_tail_stop_do:Nn #2
23164     { \__tl_mixed_case_N_type:Nnn #1 }
23165     \token_if_eq_meaning:NNTF #1 #2
23166     {
23167       \use_i_delimit_by_q_recursion_stop:nw
23168       {
23169         \__tl_change_case_math:NNNnnn
23170         #1 #3 \__tl_mixed_case_loop:wn
23171       }
23172     }
23173   { \__tl_mixed_case_N_type:NNNnn #1 }
23174 }

```

The business end of the loop is here: there is first a need to deal with any control sequence cases before looking for characters to skip. If there is a hit for a letter-like control sequence, switch to lower casing.

```

23175 \cs_new:Npn \__tl_mixed_case_N_type:Nnn #1#2#3
23176   {
23177     \token_if_cs:NTF #1
23178     {
23179       \__tl_change_case_cs_letterlike:Nnn #1 { upper }

```

```

23180      { \__tl_mixed_case_letterlike:Nw }
23181      \__tl_mixed_case_loop:wn #2 \q_recursion_stop {#3}
23182    }
23183    {
23184      \__tl_mixed_case_char:Nn #1 {#3}
23185      \__tl_change_case_loop:wnn #2 \q_recursion_stop { lower } {#3}
23186    }
23187  }
23188 \cs_new:Npn \__tl_mixed_case_letterlike:Nw #1#2 \q_recursion_stop
23189   { \__tl_change_case_loop:wnn #2 \q_recursion_stop { lower } }

```

As detailed above, handling a mixed case char means first looking for exceptions then treating as an upper cased letter, but with a list of tokens to skip over too.

```

23190 \cs_new:Npn \__tl_mixed_case_char:Nn #1#2
23191   {
23192     \cs_if_exist_use:cF { \__tl_change_case_mixed_ #2 :Nnw }
23193     {
23194       \cs_if_exist_use:cF { \__tl_change_case_upper_ #2 :Nnw }
23195       { \use_i:nn }
23196     }
23197     #1
23198     { \__tl_mixed_case_skip:N #1 }
23199   }
23200 \cs_new:Npn \__tl_mixed_case_skip:N #1
23201   {
23202     \exp_after:wN \__tl_mixed_case_skip>NN
23203     \exp_after:wN #1 \l_tl_mixed_case_ignore_tl
23204     \q_recursion_tail \q_recursion_stop
23205   }
23206 \cs_new:Npn \__tl_mixed_case_skip>NN #1#2
23207   {
23208     \quark_if_recursion_tail_stop_do:nn {#2}
23209     { \__tl_mixed_case_char:N #1 }
23210     \int_compare:nNnT { '#1 } = { '#2 }
23211     {
23212       \use_i_delimit_by_q_recursion_stop:nw
23213       {
23214         \__tl_change_case_output:nwn {#1}
23215         \__tl_mixed_case_skip_tidy:Nwn
23216       }
23217     }
23218     \__tl_mixed_case_skip>NN #1
23219   }
23220 \cs_new:Npn \__tl_mixed_case_skip_tidy:Nwn #1#2 \q_recursion_stop #3
23221   {
23222     \__tl_mixed_case_loop:wn #2 \q_recursion_stop
23223   }
23224 \cs_new:Npn \__tl_mixed_case_char:N #1
23225   {
23226     \cs_if_exist:cTF { c__unicode_title_ #1 _tl }
23227     {
23228       \__tl_change_case_output:fwn
23229       { \tl_use:c { c__unicode_title_ #1 _tl } }
23230     }

```

```

23231     { \__tl_change_case_char:nN { upper } #1 }
23232 }
```

(End definition for `__tl_mixed_case:nn` and others.)

`__tl_change_case_mixed_nl:Nnw` `__tl_change_case_mixed_nl:Nw` `__tl_change_case_mixed_nl>NNw`

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.

```

23233 \cs_new:Npn \__tl_change_case_mixed_nl:Nnw #1
23234 {
23235     \bool_lazy_or:nnTF
23236     { \int_compare_p:nNn { '#1 } = { 'i } }
23237     { \int_compare_p:nNn { '#1 } = { 'I } }
23238     {
23239         \__tl_change_case_output:nwn { I }
23240         \__tl_change_case_mixed_nl:Nw
23241     }
23242 }
23243 \cs_new:Npn \__tl_change_case_mixed_nl:Nw #1#2 \q_recursion_stop
23244 {
23245     \tl_if_head_is_N_type:nT {#2}
23246     { \__tl_change_case_mixed_nl>NNw }
23247     #1 #2 \q_recursion_stop
23248 }
23249 \cs_new:Npn \__tl_change_case_mixed_nl>NNw #1#2#3 \q_recursion_stop
23250 {
23251     \__tl_change_case_if_expandable:NTF #2
23252     {
23253         \exp_after:wN \__tl_change_case_mixed_nl:Nw \exp_after:wN #1 #2
23254         #3 \q_recursion_stop
23255     }
23256 }
23257 \bool_lazy_and:nnTF
23258     { ! ( \token_if_cs_p:N #2 ) }
23259 {
23260     \bool_lazy_or_p:nn
23261     { \int_compare_p:nNn { '#2 } = { 'j } }
23262     { \int_compare_p:nNn { '#2 } = { 'J } }
23263 }
23264 {
23265     \__tl_change_case_output:nwn { J }
23266     #1
23267 }
23268     { #1 #2 }
23269     #3 \q_recursion_stop
23270 }
23271 }
```

(End definition for `__tl_change_case_mixed_nl:Nnw`, `__tl_change_case_mixed_nl:Nw`, and `__tl_change_case_mixed_nl>NNw`.)

`\l_tl_case_change_math_tl` The list of token pairs which are treated as math mode and so not case changed.

```

23272 \tl_new:N \l_tl_case_change_math_tl
23273 (*package)
23274 \tl_set:Nn \l_tl_case_change_math_tl
23275     { $ $ \(\) }
```

```
23276  </package>
```

(End definition for `\l_t1_case_change_math_t1`. This variable is documented on page 233.)

\l_t1_case_change_exclude_t1 The list of commands for which an argument is not case changed.

```
23277  \tl_new:N \l_t1_case_change_exclude_t1
23278  (*package)
23279  \tl_set:Nn \l_t1_case_change_exclude_t1
23280  { \cite \ensuremath \label \ref }
23281  </package>
```

(End definition for `\l_t1_case_change_exclude_t1`. This variable is documented on page 234.)

\l_t1_mixed_case_ignore_t1 Characters to skip over when finding the first letter in a word to be mixed cased.

```
23282  \tl_new:N \l_t1_mixed_case_ignore_t1
23283  \tl_set:Nx \l_t1_mixed_case_ignore_t1
23284  {
23285  ( % )
23286  [ % ]
23287  \cs_to_str:N \{ % \}
23288  '
23289  -
23290 }
```

(End definition for `\l_t1_mixed_case_ignore_t1`. This variable is documented on page 235.)

42.14.2 Other additions to l3tl

\tl_rand_item:n Importantly `\tl_item:nn` only evaluates its argument once.

```
23291  \cs_new:Npn \tl_rand_item:n #1
23292  {
23293  \tl_if_blank:nF {#1}
23294  { \tl_item:nn {#1} { \int_rand:nn { 1 } { \tl_count:n {#1} } } }
23295  }
23296  \cs_new:Npn \tl_rand_item:N { \exp_args:No \tl_rand_item:n }
23297  \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 236.)

\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:nnw` receives the number of items to skip at the beginning of the token list, and the index of the last item to keep. 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 mistake), then repeatedly call `_tl_range_collect:w` to store in its second argument the items to be kept. 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`.

```
23298  \cs_new:Npn \tl_range:Nnn { \exp_args:No \tl_range:nnn }
23299  \cs_generate_variant:Nn \tl_range:Nnn { c }
23300  \cs_new:Npn \tl_range:nnn #1#2#3
23301  {
```

```

23302     \tl_head:f
23303     {
23304         \exp_args:Nf \__tl_range:nnnw { \tl_count:n {#1} } {#2} {#3}
23305         #1
23306     }
23307 }
23308 \cs_new:Npn \__tl_range:nnnw #1#2#3
23309 {
23310     \exp_args:Nff \__tl_range:nnw
23311     {
23312         \exp_args:Nf \__tl_range_normalize:nn
23313         { \int_eval:n {#2 - 1} } {#1}
23314     }
23315     {
23316         \exp_args:Nf \__tl_range_normalize:nn
23317         { \int_eval:n {#3} } {#1}
23318     }
23319 }
23320 \cs_new:Npn \__tl_range:nnw #1#2
23321 {
23322     \if_int_compare:w #2 > #1 \exp_stop_f: \else:
23323         \exp_after:wN { \exp_after:wN }
23324     \fi:
23325     \exp_after:wN \__tl_range_collect:w
23326     \__int_value:w \__int_eval:w #2 - #1 \exp_after:wN ;
23327     \exp_after:wN { \exp_after:wN }
23328     \exp:w \__tl_range_skip:w #1 ; { }
23329 }
23330 \cs_new:Npn \__tl_range_skip:w #1 ; #2
23331 {
23332     \if_int_compare:w #1 > 0 \exp_stop_f:
23333         \exp_after:wN \__tl_range_skip:w
23334         \__int_value:w \__int_eval:w #1 - 1 \exp_after:wN ;
23335     \else:
23336         \exp_after:wN \exp_end:
23337     \fi:
23338 }
23339 \cs_new:Npn \__tl_range_collect:w #1 ; #2#3
23340 {
23341     \if_int_compare:w #1 > 1 \exp_stop_f:
23342         \exp_after:wN \__tl_range_collect:w
23343         \__int_value:w \__int_eval:w #1 - 1 \exp_after:wN ;
23344     \fi:
23345     { #2 {#3} }
23346 }

```

(End definition for `\tl_range:Nnn` and others. These functions are documented on page 236.)

`__tl_range_normalize:nn` This function converts an `<index>` argument into an explicit position in the token list (a result of 0 denoting “out of bounds”). Expects two explicit integer arguments: the `<index> #1` and the string count `#2`. If `#1` is negative, replace it by `#1 + #2 + 1`, then limit to the range $[0, #2]$.

```

23347 \cs_new:Npn \__tl_range_normalize:nn #1#2
23348 {

```

```

23349   \int_eval:n
23350   {
23351     \if_int_compare:w #1 < 0 \exp_stop_f:
23352       \if_int_compare:w #1 < -#2 \exp_stop_f:
23353         0
23354       \else:
23355         #1 + #2 + 1
23356     \fi:
23357   \else:
23358     \if_int_compare:w #1 < #2 \exp_stop_f:
23359       #1
23360     \else:
23361       #2
23362     \fi:
23363   \fi:
23364 }
23365 }
```

(End definition for `_tl_range_normalize:nn.`)

42.15 Additions to `I3tokens`

`\@=peek`

```

\peek_N_type:TF
\_peek_execute_branches_N_type:
\_peek_N_type:w
\_peek_N_type_aux:nnw
```

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:nTF` 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, `\use_none_delimit_by_q_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.

```

23367 \group_begin:
23368   \cs_set_protected:Npn \_peek_tmp:w #1 \q_stop
23369   {
23370     \cs_new_protected:Npn \_peek_execute_branches_N_type:
23371     {
23372       \if_int_odd:w
23373         \if_catcode:w \exp_not:N \l_peek_token { 0 \exp_stop_f: \fi:
23374           \if_catcode:w \exp_not:N \l_peek_token } 0 \exp_stop_f: \fi:
23375           \if_meaning:w \l_peek_token \c_space_token 0 \exp_stop_f: \fi:
23376             1 \exp_stop_f:
23377           \exp_after:wN \_peek_N_type:w
23378             \token_to_meaning:N \l_peek_token
23379             \q_mark \_peek_N_type_aux:nnw
23380             #1 \q_mark \use_none_delimit_by_q_stop:w
```

```

23381           \q_stop
23382           \exp_after:wN \__peek_true:w
23383     \else:
23384       \exp_after:wN \__peek_false:w
23385     \fi:
23386   }
23387   \cs_new_protected:Npn \__peek_N_type:w ##1 #1 ##2 \q_mark ##3
23388   { ##3 {##1} {##2} }
23389 }
23390 \exp_after:wN \__peek_tmp:w \tl_to_str:n { outer } \q_stop
23391 \group_end:
23392 \cs_new_protected:Npn \__peek_N_type_aux:nnw #1 #2 #3 \fi:
23393 {
23394   \fi:
23395   \tl_if_in:ntF {#1} { \tl_to_str:n {ma} }
23396   { \__peek_true:w }
23397   { \tl_if_empty:nTF {#2} { \__peek_true:w } { \__peek_false:w } }
23398 }
23399 \cs_new_protected:Npn \peek_N_type:TF
23400 { \__peek_token_generic:NNTF \__peek_execute_branches_N_type: \scan_stop: }
23401 \cs_new_protected:Npn \peek_N_type:T
23402 { \__peek_token_generic:NNT \__peek_execute_branches_N_type: \scan_stop: }
23403 \cs_new_protected:Npn \peek_N_type:F
23404 { \__peek_token_generic:NNF \__peek_execute_branches_N_type: \scan_stop: }

(End definition for \peek_N_type:TF and others. These functions are documented on page 237.)
```

23405 ⟨/initex | package⟩

43 I3luateX implementation

23406 ⟨*initex | package⟩

43.1 Breaking out to Lua

23407 ⟨*tex⟩

\lua_now_x:n Wrappers around the primitives. As with engines other than LuaTeX these have to be macros, we give them the same status in all cases. When LuaTeX is not in use, simply give an error message/

23408 \cs_new:Npn \lua_now_x:n #1 { \luatex_directlua:D {#1} }

23409 \cs_new:Npn \lua_now:n #1 { \lua_now_x:n { \exp_not:n {#1} } }

23410 \cs_new_protected:Npn \lua_shipout_x:n #1 { \luatex_latelua:D {#1} }

23411 \cs_new_protected:Npn \lua_shipout:n #1

23412 { \lua_shipout_x:n { \exp_not:n {#1} } }

23413 \cs_new:Npn \lua_escape_x:n #1 { \luatex_luaescapestring:D {#1} }

23414 \cs_new:Npn \lua_escape:n #1 { \lua_escape_x:n { \exp_not:n {#1} } }

23415 \sys_if_engine_luatex:F

23416 {

23417 \clist_map_inline:nn

23418 { \lua_now_x:n , \lua_now:n , \lua_escape_x:n , \lua_escape:n }

23419 {

23420 \cs_set:Npn #1 ##1

23421 {

23422 __msg_kernel_expandable_error:nnn

```

23423     { kernel } { luatex-required } { #1 }
23424   }
23425   }
23426 \clist_map_inline:nn
23427   { \lua_shipout_x:n , \lua_shipout:n }
23428   {
23429     \cs_set_protected:Npn #1 ##1
23430     {
23431       \__msg_kernel_error:nnn
23432         { kernel } { luatex-required } { #1 }
23433     }
23434   }
23435 }
```

(End definition for `\lua_now_x:n` and others. These functions are documented on page 238.)

43.2 Messages

```

23436 \__msg_kernel_new:nnnn { kernel } { luatex-required }
23437   { LuaTeX-engine-not-in-use!~Ignoring~#1. }
23438   {
23439     The~feature~you~are~using~is~only~available~
23440     with~the~LuaTeX-engine.~LaTeX3~ignored~'#1'.
23441   }
23442 </tex>
```

43.3 Lua functions for internal use

```
23443 <*lua>
```

`13kernel` Create a table for the kernel's own use.

```
23444 13kernel = 13kernel or {}
```

(End definition for `13kernel`.)

Various local copies of standard functions: naming convention is to retain the full text but replace all `.` by `_`.

```

23445 local tex_setcatcode    = tex.setcatcode
23446 local tex_sprint        = tex.sprint
23447 local tex_write         = tex.write
23448 local unicode_utf8_char = unicode.utf8.char
```

`13kernelstrcmp` String comparison which gives the same results as pdfTeX's `\pdfstrcmp`, although the ordering should likely not be relied upon!

```

23449 local function strcmp (A, B)
23450   if A == B then
23451     tex_write("0")
23452   elseif A < B then
23453     tex_write("-1")
23454   else
23455     tex_write("1")
23456   end
23457 end
23458 13kernelstrcmp = strcmp
```

(End definition for `13kernelstrcmp`.)

13kernel.charcat

Creating arbitrary chars needs a category code table. As set up here, one may have been assigned earlier (see l3bootstrap) or a hard-coded one is used. The latter is intended for format mode and should be adjusted to match an eventual allocator.

```
23459 local charcat_table = l3kernel.charcat_table or 1
23460 local function charcat (charcode, catcode)
23461   tex_setcatcode(charcat_table, charcode, catcode)
23462   tex_sprint(charcat_table, unicode_utf8_char(charcode))
23463 end
23464 l3kernel.charcat = charcat
```

(End definition for l3kernel.charcat.)

43.4 Generic Lua and font support

```
23465 ⟨*initex⟩
```

A small amount of generic code is used by almost all LuaTeX material so needs to be loaded by the format.

```
23466 attribute_count_name = "g__alloc_attribute_int"
23467 bytecode_count_name = "g__alloc_bytecode_int"
23468 chunkname_count_name = "g__alloc_chunkname_int"
23469 whatsit_count_name = "g__alloc_whatsit_int"
23470 require("ltluatex")
```

With the above available the font loader code used by plain T_EX and L^AT_EX 2_ε when used with LuaTeX can be loaded here. This is thus being treated more-or-less as part of the engine itself.

```
23471 require("luaotfload-main")
23472 local _void = luaotfload.main()
23473 ⟨/initex⟩
23474 ⟨/lua⟩
23475 ⟨/initex | package⟩
```

44 I3drivers Implementation

```
23476 ⟨*initex | package⟩
23477 ⟨@C=driver⟩
```

Whilst there is a reasonable amount of code overlap between drivers, it is much clearer to have the blocks more-or-less separated than run in together and DocStripped out in parts. As such, most of the following is set up on a per-driver basis, though there is some common code (again given in blocks not interspersed with other material).

All the file identifiers are up-front so that they come out in the right place in the files.

```
23478 ⟨*package⟩
23479 \ProvidesExplFile
23480 ⟨*dvipdfmx⟩
23481   {l3dvidpdfmx.def}{2017/03/18}{}{L3 Experimental driver: dvipdfmx}
23482   ⟨/dvipdfmx⟩
23483 ⟨*dvips⟩
23484   {l3dvips.def}{2017/03/18}{}{L3 Experimental driver: dvips}
23485   ⟨/dvips⟩
23486   ⟨/dvips⟩
```

```

23488  {*dvisvgm}
23489   {l3dvisvgm.def}{2017/03/18}{}
23490   {L3 Experimental driver: dvisvgm}
23491 
```

```

23492  {*pdfmode}
23493   {l3pdfmode.def}{2017/03/18}{}
23494   {L3 Experimental driver: PDF mode}
23495 
```

```

23496  {*xdvipdfmx}
23497   {l3xdvipdfmx.def}{2017/03/18}{}
23498   {L3 Experimental driver: xdvipdfmx}
23499 
```

```

23500 
```

44.1 pdfmode driver

```

23501 
```

The direct PDF driver covers both pdf_TE_X and Lu_AT_E_X. The latter renames/restructures the driver primitives but this can be handled at one level of abstraction. As such, we avoid using two separate drivers for this material at the cost of some **x**-type definitions to get everything expanded up-front.

44.1.1 Basics

__driver_literal:n: This is equivalent to \special{pdf:} but the engine can track it. Without the **direct** keyword everything is kept in sync: the transformation matrix is set to the current point automatically. Note that this is still inside the text (BT ... ET block).

```

23502 \cs_new_protected:Npx \_\_driver_literal:n #1
23503 {
23504   \cs_if_exist:NTF \luatex_pdfextension:D
23505     { \luatex_pdfextension:D literal }
23506     { \pdftex_pafliteral:D }
23507     {#1}
23508 }
```

(End definition for __driver_literal:n.)

__driver_scope_begin: Higher-level interfaces for saving and restoring the graphic state.

```

\_\_driver_scope_end:
23509 \cs_new_protected:Npx \_\_driver_scope_begin:
23510 {
23511   \cs_if_exist:NTF \luatex_pdfextension:D
23512     { \luatex_pdfextension:D save \scan_stop: }
23513     { \pdftex_pdfsave:D }
23514 }
23515 \cs_new_protected:Npx \_\_driver_scope_end:
23516 {
23517   \cs_if_exist:NTF \luatex_pdfextension:D
23518     { \luatex_pdfextension:D restore \scan_stop: }
23519     { \pdftex_pdfrestore:D }
23520 }
```

(End definition for __driver_scope_begin: and __driver_scope_end:.)

__driver_matrix:n: Here the appropriate function is set up to insert an affine matrix into the PDF. With pdf_TE_X and Lu_AT_E_X in direct PDF output mode there is a primitive for this, which only needs the rotation/scaling/skew part.

```

23521 \cs_new_protected:Npx \__driver_matrix:n #1
23522 {
23523   \cs_if_exist:NTF \luatex_pdfextension:D
23524     { \luatex_pdfextension:D setmatrix }
23525     { \pdftex_pdfsetmatrix:D }
23526     {#1}
23527 }

```

(End definition for `__driver_matrix:n`.)

44.1.2 Box operations

`__driver_box_use_clip:N`

The general method is to save the current location, define a clipping path equivalent to the bounding box, then insert the content at the current position and in a zero width box. The “real” width is then made up using a horizontal skip before tidying up. There are other approaches that can be taken (for example using XForm objects), but the logic here shares as much code as possible and uses the same conversions (and so same rounding errors) in all cases.

```

23528 \cs_new_protected:Npn \__driver_box_use_clip:N #1
23529 {
23530   \__driver_scope_begin:
23531   \__driver_literal:n
23532   {
23533     O~
23534     \dim_to_decimal_in_bp:n { -\box_dp:N #1 } ~
23535     \dim_to_decimal_in_bp:n { \box_wd:N #1 } ~
23536     \dim_to_decimal_in_bp:n { \box_ht:N #1 + \box_dp:N #1 } ~
23537     re~W~n
23538   }
23539   \hbox_overlap_right:n { \box_use:N #1 }
23540   \__driver_scope_end:
23541   \skip_horizontal:n { \box_wd:N #1 }
23542 }

```

(End definition for `__driver_box_use_clip:N`.)

`__driver_box_use_rotate:Nn`

`\l__driver_cos_fp`
`\l__driver_sin_fp`

Rotations are set using an affine transformation matrix which therefore requires sine/cosine values not the angle itself. We store the rounded values to avoid rounding twice. There are also a couple of comparisons to ensure that -0 is not written to the output, as this avoids any issues with problematic display programs. Note that numbers are compared to 0 after rounding.

```

23543 \cs_new_protected:Npn \__driver_box_use_rotate:Nn #1#2
23544 {
23545   \__driver_scope_begin:
23546   \box_set_wd:Nn #1 \c_zero_dim
23547   \fp_set:Nn \l__driver_cos_fp { round ( cosd ( #2 ) , 5 ) }
23548   \fp_compare:nNnTF \l__driver_cos_fp = \c_zero_fp
23549     { \fp_zero:N \l__driver_cos_fp }
23550   \fp_set:Nn \l__driver_sin_fp { round ( sind ( #2 ) , 5 ) }
23551   \__driver_matrix:n
23552   {
23553     \fp_use:N \l__driver_cos_fp \c_space_tl
23554     \fp_compare:nNnTF \l__driver_sin_fp = \c_zero_fp
23555     { 0~0 }

```

```

23556      {
23557          \fp_use:N \l__driver_sin_fp
23558          \c_space_tl
23559          \fp_eval:n { -\l__driver_sin_fp }
23560      }
23561      \c_space_tl
23562      \fp_use:N \l__driver_cos_fp
23563  }
23564      \box_use:N #1
23565      \__driver_scope_end:
23566  }
23567 \fp_new:N \l__driver_cos_fp
23568 \fp_new:N \l__driver_sin_fp

```

(End definition for `__driver_box_use_rotate:Nn`, `\l__driver_cos_fp`, and `\l__driver_sin_fp`.)

`__driver_box_use_scale:Nnn` The same idea as for rotation but without the complexity of signs and cosines.

```

23569 \cs_new_protected:Npn \__driver_box_use_scale:Nnn #1#2#3
23570  {
23571      \__driver_scope_begin:
23572      \__driver_matrix:n
23573      {
23574          \fp_eval:n { round ( #2 , 5 ) } ~
23575          0~0~
23576          \fp_eval:n { round ( #3 , 5 ) }
23577      }
23578      \hbox_overlap_right:n { \box_use:N #1 }
23579      \__driver_scope_end:
23580  }

```

(End definition for `__driver_box_use_scale:Nnn`.)

44.1.3 Color

`\l__driver_color_current_tl` The current color in driver-dependent format: pick up the package-mode data if available.

```

23581 \tl_new:N \l__driver_color_current_tl
23582 \tl_set:Nn \l__driver_color_current_tl { 0g~0~0-G }
23583 (*package)
23584 \AtBeginDocument
23585  {
23586      \@ifpackageloaded { color }
23587          { \tl_set:Nn \l__driver_color_current_tl { \current@color } }
23588          { }
23589  }
23590 
```

(End definition for `\l__driver_color_current_tl`.)

`\l__driver_color_stack_int` pdfTeX and LuaTeX have multiple stacks available, and to track which one is in use a variable is required.

```
23591 \int_new:N \l__driver_color_stack_int
```

(End definition for `\l__driver_color_stack_int`.)

`_driver_color_ensure_current:` There is a dedicated primitive/primitive interface for setting colors. As with scoping, this approach is not suitable for cached operations.

```

23592 \cs_new_protected:Npx \_driver_color_ensure_current:
23593 {
23594     \cs_if_exist:NTF \luatex_pdfextension:D
23595         { \luatex_pdfextension:D colorstack }
23596         { \pdftex_pdfcolorstack:D }
23597         \exp_not:N \l__driver_color_stack_int push
23598             { \exp_not:N \l__driver_color_current_tl }
23599     \group_insert_after:N \exp_not:N \_driver_color_reset:
23600 }
23601 \cs_new_protected:Npx \_driver_color_reset:
23602 {
23603     \cs_if_exist:NTF \luatex_pdfextension:D
23604         { \luatex_pdfextension:D colorstack }
23605         { \pdftex_pdfcolorstack:D }
23606         \exp_not:N \l__driver_color_stack_int pop \scan_stop:
23607 }
```

(End definition for `_driver_color_ensure_current:` and `_driver_color_reset:`)

44.2 Images

`\l__driver_image_attr_tl`

In PDF mode, additional attributes of an image (such as page number) are needed both to obtain the bounding box and when inserting the image: this occurs as the image dictionary approach means they are read as part of the bounding box operation. As such, it is easier to track additional attributes using a dedicated `t1` rather than build up the same data twice.

```
23608 \tl_new:N \l__driver_image_attr_tl
```

(End definition for `\l__driver_image_attr_tl`.)

`_driver_image_getbb_jpg:n`
`_driver_image_getbb_pdf:n`
`_driver_image_getbb_png:n`
`_driver_image_getbb_auxi:n`
`_driver_image_getbb_auxii:n`

Getting the bounding box here requires us to box up the image and measure it. To deal with the difference in feature support in bitmap and vector images but keeping the common parts, there is a little work to do in terms of auxiliaries. The key here is to notice that we need two forms of the attributes: a “short” set to allow us to track for caching, and the full form to pass to the primitive. Note that in `pdftex.def` the short reference is stored to be used in the inclusion stage: may be required when there are more aspects to track.

```

23609 \cs_new_protected:Npn \_driver_image_getbb_jpg:n #1
23610 {
23611     \int_zero:N \l__image_page_int
23612     \tl_set:Nx \l__driver_image_attr_tl
23613     {
23614         \bool_if:NT \l__image_interpolate_bool
23615             { :I }
23616     }
23617     \_driver_image_getbb_auxi:n {#1}
23618 }
23619 \cs_new_eq:NN \_driver_image_getbb_png:n \_driver_image_getbb_jpg:n
23620 \cs_new_protected:Npn \_driver_image_getbb_pdf:n #1
23621 {
23622     \bool_set_false:N \l__image_interpolate_bool
```

```

23623     \tl_set:Nx \l__driver_image_attr_tl
23624     {
23625         \int_compare:nNnT \l__image_page_int > 0
23626             { :P \int_use:N \l__image_page_int }
23627         }
23628     \__driver_image_getbb_auxi:n {#1}
23629 }
23630 \cs_new_protected:Npn \__driver_image_getbb_auxi:n #1
23631 {
23632     \dim_if_exist:cTF { c__image_ #1 \l__driver_image_attr_tl _ht_dim }
23633     {
23634         \dim_set_eq:Nc \l__image_ht_dim
23635             { c__image_ #1 \l__driver_image_attr_tl _ht_dim }
23636         \dim_set_eq:Nc \l__image_wd_dim
23637             { c__image_ #1 \l__driver_image_attr_tl _wd_dim }
23638     }
23639     { \__driver_image_getbb_auxii:n {#1} }
23640 }
23641 % \begin{macrocode}
23642 % Measuring the image is done by boxing up: for PDF images we could
23643 % use \pdftex_pfdximagebbox:D, but if doesn't work for other types.
23644 % \begin{macrocode}
23645 \cs_new_protected:Npn \__driver_image_getbb_auxii:n #1
23646 {
23647     \tex_immediate:D \pdftex_pfdximage:D
23648         \bool_if:NT \l__image_interpolate_bool
23649             { attr ~ { /Interpolate~true } }
23650         \int_compare:nNnT \l__image_page_int > 0
23651             { page ~ \int_use:N \l__image_page_int }
23652             {#1}
23653     \hbox_set:Nn \l__image_tmp_box
23654         { \pdftex_pfdrefximage:D \pdftex_pfdlastximage:D }
23655     \dim_set:Nn \l__image_ht_dim { \box_ht:N \l__image_tmp_box }
23656     \dim_set:Nn \l__image_wd_dim { \box_wd:N \l__image_tmp_box }
23657     \int_const:cn { c__image_ #1 \l__driver_image_attr_tl _int }
23658         { \tex_the:D \pdftex_pfdlastximage:D }
23659     \dim_const:cn { c__image_ #1 \l__driver_image_attr_tl _ht_dim }
23660         { \l__image_ht_dim }
23661     \dim_const:cn { c__image_ #1 \l__driver_image_attr_tl _wd_dim }
23662         { \l__image_wd_dim }
23663 }

```

(End definition for `__driver_image_getbb_jpg:n` and others.)

`_driver_image_include_jpg:n` Images are already loaded for the measurement part of the code, so inclusion is straightforward, with only any attributes to worry about. The latter carry through from determination of the bounding box.

```

23664 \cs_new_protected:Npn \__driver_image_include_jpg:n #1
23665 {
23666     \pdftex_pfdrefximage:D
23667         \int_use:c { c__image_ #1 \l__driver_image_attr_tl _int }
23668 }
23669 \cs_new_eq:NN \__driver_image_include_pdf:n \__driver_image_include_jpg:n
23670 \cs_new_eq:NN \__driver_image_include_png:n \__driver_image_include_jpg:n

```

(End definition for `_driver_image_include_jpg:n`, `_driver_image_include_pdf:n`, and `_driver_image_include_png:n`.)

23671 `</pdfmode>`

44.3 dvipdfmx driver

23672 `(*dvipdfmx | xdvipdfmx)`

The `dvipdfmx` shares code with the PDF mode one (using the common section to this file) but also with `xdvipdfmx`. The latter is close to identical to `dvipdfmx` and so all of the code here is extracted for both drivers, with some clean up for `xdvipdfmx` as required.

44.3.1 Basics

`_driver_literal:n` Equivalent to `pdf:content` but favored as the link to the pdfTeX primitive approach is clearer. Some higher-level operations use `\tex_special:D` directly: see the later comments on where this is useful.

23673 `\cs_new_protected:Npn _driver_literal:n #1`
23674 `{ \tex_special:D { pdf:literal~ #1 } }`

(End definition for `_driver_literal:n`.)

`_driver_scope_begin:` Scoping is done using the driver-specific specials.

`_driver_scope_end:`

23675 `\cs_new_protected:Npn _driver_scope_begin:`
23676 `{ \tex_special:D { x:gsave } }`
23677 `\cs_new_protected:Npn _driver_scope_end:`
23678 `{ \tex_special:D { x:restore } }`

(End definition for `_driver_scope_begin:` and `_driver_scope_end:)`)

44.3.2 Box operations

`_driver_box_use_clip:N` The code here is identical to that for `pdfmode`: unlike rotation and scaling, there is no higher-level support in the driver for clipping.

23679 `\cs_new_protected:Npn _driver_box_use_clip:N #1`
23680 `{`
23681 `_driver_scope_begin:`
23682 `_driver_literal:n`
23683 `{`
23684 `O~`
23685 `\dim_to_decimal_in_bp:n { -\box_dp:N #1 } ~`
23686 `\dim_to_decimal_in_bp:n { \box_wd:N #1 } ~`
23687 `\dim_to_decimal_in_bp:n { \box_ht:N #1 + \box_dp:N #1 } ~`
23688 `re~W~n`
23689 `}`
23690 `\hbox_overlap_right:n { \box_use:N #1 }`
23691 `_driver_scope_end:`
23692 `\skip_horizontal:n { \box_wd:N #1 }`
23693 `}`

(End definition for `_driver_box_use_clip:N`.)

__driver_box_use_rotate:Nn

Rotating in (x)dvipdmg can be implemented using either PDF or driver-specific code. The former approach however is not “aware” of the content of boxes: this means that any links embedded will not be adjusted by the rotation. As such, the driver-native approach is preferred: the code therefore is similar (though not identical) to the dvips version (notice the rotation angle here is positive). As for dvips, zero rotation is written as 0 not -0.

```
23694 \cs_new_protected:Npn \_\_driver_box_use_rotate:Nn #1#2
23695 {
23696     \_\_driver_scope_begin:
23697     \tex_special:D
23698     {
23699         x:rotate-
23700         \fp_compare:nNnTF {#2} = \c_zero_fp
23701             { 0 }
23702             { \fp_eval:n { round ( #2 , 5 ) } }
23703     }
23704     \box_use:N #1
23705     \_\_driver_scope_end:
23706 }
```

(End definition for __driver_box_use_rotate:Nn.)

__driver_box_use_scale:Nnn

Much the same idea for scaling: use the higher-level driver operation to allow for box content.

```
23707 \cs_new_protected:Npn \_\_driver_box_use_scale:Nnn #1#2#3
23708 {
23709     \_\_driver_scope_begin:
23710     \tex_special:D
23711     {
23712         x:scale-
23713         \fp_eval:n { round ( #2 , 5 ) } ~
23714         \fp_eval:n { round ( #3 , 5 ) }
23715     }
23716     \hbox_overlap_right:n { \box_use:N #1 }
23717     \_\_driver_scope_end:
23718 }
```

(End definition for __driver_box_use_scale:Nnn.)

44.3.3 Color

\l__driver_color_current_tl

The current color in driver-dependent format.

```
23719 \tl_new:N \l\_\_driver_color_current_tl
23720 \tl_set:Nn \l\_\_driver_color_current_tl { [ 0 ] }
23721 (*package)
23722 \AtBeginDocument
23723 {
23724     \Oifpackageloaded { color }
23725         { \tl_set:Nn \l\_\_driver_color_current_tl { \current@color } }
23726         { }
23727 }
23728 
```

(End definition for \l__driver_color_current_tl.)

```

\__driver_color_ensure_current: Directly set the color using the specials with optimisation support.
\__driver_color_reset:
23729 \cs_new_protected:Npn \__driver_color_ensure_current:
23730 {
23731     \tex_special:D { pdf:bcolor~\l__driver_color_current_tl }
23732     \group_insert_after:N \__driver_color_reset:
23733 }
23734 \cs_new_protected:Npn \__driver_color_reset:
23735     { \tex_special:D { pdf:ecolor } }

(End definition for \__driver_color_ensure_current: and \__driver_color_reset:.)
```

44.4 Images

```

\__driver_image_getbb_eps:n Simply use the generic functions: only for dvipdfmx in the extraction cases.
\__driver_image_getbb_jpg:n
\__driver_image_getbb_pdf:n
\__driver_image_getbb_png:n
23736 \cs_new_eq:NN \__driver_image_getbb_eps:n \__image_read_bb:n
23737 (*dvipdfmx)
23738 \cs_new_protected:Npn \__driver_image_getbb_jpg:n #1
23739 {
23740     \int_zero:N \l__image_page_int
23741     \__image_extract_bb:n {\#1}
23742 }
23743 \cs_new_eq:NN \__driver_image_getbb_png:n \__driver_image_getbb_jpg:n
23744 \cs_new_protected:Npn \__driver_image_getbb_pdf:n #1
23745 {
23746     \bool_set_false:N \l__image_interpolate_tl
23747     \__image_extract_bb:n {\#1}
23748 }
23749 
```

(End definition for __driver_image_getbb_eps:n and others.)

\g__driver_image_int Used to track the object number associated with each image.

```
23750 \int_new:N \g__driver_image_int
```

(End definition for \g__driver_image_int.)

```

\__driver_image_include_eps:n The special syntax depends on the file type.
\__driver_image_include_jpg:n
\__driver_image_include_pdf:n
\__driver_image_include_png:n
\__driver_image_include_auxi:nn
\__driver_image_include_auxi:nnn
\__driver_image_include_auxii:xnn
\__driver_image_include_auxiii:nn
23751 \cs_new_protected:Npn \__driver_image_include_eps:n #1
23752 {
23753     \tex_special:D { PSfile = #1 }
23754 }
23755 \cs_new_protected:Npn \__driver_image_include_jpg:n #1
23756     { \__driver_image_include_auxi:nn {\#1} { image } }
23757 \cs_new_eq:NN \__driver_image_include_png:n \__driver_image_include_png:n
23758 \cs_new_protected:Npn \__driver_image_include_pdf:n #1
23759     { \__driver_image_include_auxi:nn {\#1} { epdf } }
```

Image inclusion is set up to use the fact that each image is stored in the PDF as an XObject. This means that we can include repeated images only once and refer to them. To allow that, track the nature of each image: much the same as for the direct PDF mode case.

```
23760 \cs_new_protected:Npn \__driver_image_include_auxi:nn #1#2
23761 {
23762     \__driver_image_include_auxii:xnn
```

```

23763   {
23764     \int_compare:nNnT \l__image_page_int > 0
23765       { :P \int_use:N \l__image_page_int }
23766       \bool_if:NT \l__image_interpolate_bool
23767         { :I }
23768     }
23769     {\#1} {\#2}
23770   }
23771 \cs_new_protected:Npn \__driver_image_include_auxii:nnn #1#2#3
23772   {
23773     \int_if_exist:cTF { c__image_ #2#1 _int }
23774     {
23775       \tex_special:D
23776         { pdf:usexobj~@image \int_use:c { c__image_ #2#1 _int } }
23777     }
23778     { \__driver_image_include_auxiii:nn {\#2} {\#1} {\#3} }
23779   }
23780 \cs_generate_variant:Nn \__driver_image_include_auxii:nnn { x }
23781 \cs_new_protected:Npn \__driver_image_include_auxiii:nnn #1#2#3
23782   {
23783     \int_gincr:N \g__driver_image_int
23784     \int_const:cn { c__image_ #1#2 _int } { \g__driver_image_int }
23785     \tex_special:D
23786     {
23787       pdf:#3-
23788       @image \int_use:c { c__image_ #1#2 _int }
23789       \int_compare:nNnT \l__image_page_int > 0
23790         { page ~ \int_use:N \l__image_page_int \c_space_tl }
23791         (#1)
23792         \bool_if:NT \l__image_interpolate_bool
23793           { <</Interpolate+true>> }
23794     }
23795   }
23796   }

(End definition for \__driver_image_include_eps:n and others.)
23796 </dvipdfmx | xdvipdfmx>

```

44.5 xdvipdfmx driver

```
23797 <*xdvipdfmx>
```

44.5.1 Color

`_driver_color_ensure_current:` Older L^AT_EX 2 _{ε} drivers uses dvips-like specials so there has to be a change of set up if color is loaded and if the current color doesn't match the pattern expected for dvipdfmx.

```

23798 <*package>
23799 \AtBeginDocument
23800   {
23801     \@ifpackageloaded { color }
23802     {
23803       \cs_set_protected:Npn \__driver_tmp:w #1 [ #2 ] #3 \q_stop
23804         {
23805           \tl_if_empty:nT {\#2}
23806             {

```

```

23807   \cs_set_protected:Npn \__driver_color_ensure_current:
23808   {
23809     \tex_special:D { color~push~\l__driver_color_current_tl }
23810     \group_insert_after:N \__driver_color_reset:
23811   }
23812   \cs_set_protected:Npn \__driver_color_reset:
23813   {
23814     \tex_special:D { color~pop } }
23815   }
23816   \exp_after:wN \__driver_tmp:w \current@color [ ] \q_stop
23817 }
23818 {
23819 }
23820 
```

(End definition for `__driver_color_ensure_current:` and `__driver_color_reset..`)

44.6 Images

For `xdvipdfmx`, there are two primitives that allow us to obtain the bounding box without needing `extractbb`.

```

23821 \cs_new_protected:Npn \__driver_image_getbb_jpg:n #1
23822 {
23823   \int_zero:N \l__image_page_int
23824   \__driver_image_getbb_auxi:nN {#1} \xetex_picfile:D
23825 }
23826 \cs_new_eq:NN \__driver_image_getbb_png:n \__driver_image_getbb_jpg:n
23827 \cs_new_protected:Npn \__driver_image_getbb_pdf:n #1
23828 {
23829   \__driver_image_getbb_auxi:nN {#1} \xetex_pdffile:D
23830 }
23831 \int_compare:nNnTF \l__image_page_int > 0
23832 {
23833   \__driver_image_getbb_auxii:VnN \l__image_page_int {#1} #2
23834 }
23835 \cs_new_protected:Npn \__driver_image_getbb_auxii:nnN #1#2#3
23836 {
23837   \__driver_image_getbb_auxiii:nNnn {#2} #3 { :P #1 } { page #1 }
23838 \cs_generate_variant:Nn \__driver_image_getbb_auxii:nnN { V }
23839 \cs_new_protected:Npn \__driver_image_getbb_auxiii:nNnn #1#2#3#4
23840 {
23841   \dim_if_exist:cTF { c__image_ #1#3 _ht_dim }
23842 {
23843   \dim_set_eq:Nc \l__image_ht_dim { c__image_ #1#3 _ht_dim }
23844   \dim_set_eq:Nc \l__image_wd_dim { c__image_ #1#3 _wd_dim }
23845 }
23846 }
23847 \cs_new_protected:Npn \__driver_image_getbb_auxvi:nNnn #1#2#3#4
23848 {
23849   \hbox_set:Nn \l__image_tmp_box { #2 #1 ~ #4 }
23850   \dim_set:Nn \l__image_ht_dim { \box_ht:N \l__image_tmp_box }
23851   \dim_set:Nn \l__image_wd_dim { \box_wd:N \l__image_tmp_box }
23852   \dim_const:cn { c__image_ #1#3 _ht_dim }
23853   { \l__image_ht_dim }

```

```

23854     \dim_const:cn { c_image_ #1#3 _wd_dim }
23855     { \l_image_wd_dim }
23856 }

```

(End definition for `_driver_image_getbb_jpg:n` and others.)

```

23857 
```

44.7 Drawing commands: pdfmode and (x)dvipdfmx

Both `pdfmode` and `(x)dvipdfmx` directly produce PDF output and understand a shared set of specials for drawing commands.

```

23858 
```

44.8 Drawing

`_driver_draw_literal:n`

```

\_driver_draw_literal:x
23859 \cs_new_eq:NN \_driver_draw_literal:n \_driver_literal:n
23860 \cs_generate_variant:Nn \_driver_draw_literal:n { x }

```

(End definition for `_driver_draw_literal:n`.)

`_driver_draw_begin:`

`_driver_draw_end:`

No special requirements here, so simply set up a drawing scope.

```

23861 \cs_new_protected:Npn \_driver_draw_begin:
23862   { \_driver_scope_begin: }
23863 \cs_new_protected:Npn \_driver_draw_end:
23864   { \_driver_scope_end: }

```

(End definition for `_driver_draw_begin:` and `_driver_draw_end:..`)

`_driver_draw_scope_begin:`

In contrast to a general scope, a drawing scope is always done using the PDF operators so is the same for all relevant drivers.

```

23865 \cs_new_protected:Npn \_driver_draw_scope_begin:
23866   { \_driver_literal:n { q } }
23867 \cs_new_protected:Npn \_driver_draw_scope_end:
23868   { \_driver_literal:n { Q } }

```

(End definition for `_driver_draw_scope_begin:` and `_driver_draw_scope_end:..`)

`_driver_draw_moveto:nn`

`_driver_draw_lineto:nn`

`_driver_draw_rectangle:nnnn`

`_driver_draw_curveto:nnnnnn`

Path creation operations all resolve directly to PDF primitive steps, with only the need to convert to `bp`. Notice that `x`-type expansion is included here to ensure that any variable values are forced to literals before any possible caching.

```

23869 \cs_new_protected:Npn \_driver_draw_moveto:nn #1#2
23870   {
23871     \_driver_literal:x
23872     { \dim_to_decimal_in_bp:n {#1} ~ \dim_to_decimal_in_bp:n {#2} ~ m }
23873   }
23874 \cs_new_protected:Npn \_driver_draw_lineto:nn #1#2
23875   {
23876     \_driver_literal:x
23877     { \dim_to_decimal_in_bp:n {#1} ~ \dim_to_decimal_in_bp:n {#2} ~ l }
23878   }
23879 \cs_new_protected:Npn \_driver_draw_curveto:nnnnnn #1#2#3#4#5#6
23880   {

```

```

23881     \__driver_draw_literal:x
23882     {
23883         \dim_to_decimal_in_bp:n {#1} ~ \dim_to_decimal_in_bp:n {#2} ~
23884         \dim_to_decimal_in_bp:n {#3} ~ \dim_to_decimal_in_bp:n {#4} ~
23885         \dim_to_decimal_in_bp:n {#5} ~ \dim_to_decimal_in_bp:n {#6} ~
23886         c
23887     }
23888 }
23889 \cs_new_protected:Npn \__driver_draw_rectangle:nnnn #1#2#3#4
23890 {
23891     \__driver_draw_literal:x
23892     {
23893         \dim_to_decimal_in_bp:n {#1} ~ \dim_to_decimal_in_bp:n {#2} ~
23894         \dim_to_decimal_in_bp:n {#3} ~ \dim_to_decimal_in_bp:n {#4} ~
23895         re
23896     }
23897 }

```

(End definition for `__driver_draw_moveto:nn` and others.)

`__driver_draw_evenodd_rule:` The even-odd rule here can be implemented as a simply switch.

```

23898 \cs_new_protected:Npn \__driver_draw_evenodd_rule:
23899 {
23900     \bool_gset_true:N \g__driver_draw_eor_bool
23901 \cs_new_protected:Npn \__driver_draw_nonzero_rule:
23902 {
23903     \bool_gset_false:N \g__driver_draw_eor_bool
23904 \bool_new:N \g__driver_draw_eor_bool

```

(End definition for `__driver_draw_evenodd_rule:`, `__driver_draw_nonzero_rule:`, and `\g__driver_draw_eor_bool`.)

`__driver_draw_closepath:` Converting paths to output is again a case of mapping directly to PDF operations.

```

23903 \cs_new_protected:Npn \__driver_draw_closepath:
23904 {
23905     \__driver_draw_literal:n { h } }
23906 \cs_new_protected:Npn \__driver_draw_stroke:
23907 {
23908     \__driver_draw_literal:n { S } }
23909 \cs_new_protected:Npn \__driver_draw_fill:
23910 {
23911     \__driver_draw_literal:x
23912     { f \bool_if:NT \g__driver_draw_eor_bool * }
23913 }
23914 \cs_new_protected:Npn \__driver_draw_closestroke:
23915 {
23916     \__driver_draw_literal:x
23917     { B \bool_if:NT \g__driver_draw_eor_bool * }
23918 }
23919 \cs_new_protected:Npn \__driver_draw_clip:
23920 {
23921     \__driver_draw_literal:x
23922     { W \bool_if:NT \g__driver_draw_eor_bool * }
23923 }
23924 \cs_new_protected:Npn \__driver_draw_discardpath:
23925 {
23926     \__driver_draw_literal:n { n } }

```

(End definition for `_driver_draw_closepath:` and others.)

`_driver_draw_dash:nn` Converting paths to output is again a case of mapping directly to PDF operations.

```
23926 \cs_new_protected:Npn \_driver_draw_dash:nn #1#2
23927 {
23928     \_driver_draw_literal:x
23929     {
23930         [ ~
23931             \clist_map_function:nN {#1} \_driver_draw_dash:n
23932         ] ~
23933             \dim_to_decimal_in_bp:n {#2} ~ d
23934     }
23935 }
23936 \cs_new:Npn \_driver_draw_dash:n #1
23937     { \dim_to_decimal_in_bp:n {#1} ~ }
23938 \cs_new_protected:Npn \_driver_draw_linewidth:n #1
23939 {
23940     \_driver_draw_literal:x
23941     { \dim_to_decimal_in_bp:n {#1} ~ w }
23942 }
23943 \cs_new_protected:Npn \_driver_draw_miterlimit:n #1
23944     { \_driver_draw_literal:x { \fp_eval:n {#1} ~ M } }
23945 \cs_new_protected:Npn \_driver_draw_cap_but:
23946     { \_driver_draw_literal:n { 0 ~ J } }
23947 \cs_new_protected:Npn \_driver_draw_cap_round:
23948     { \_driver_draw_literal:n { 1 ~ J } }
23949 \cs_new_protected:Npn \_driver_draw_cap_rectangle:
23950     { \_driver_draw_literal:n { 2 ~ J } }
23951 \cs_new_protected:Npn \_driver_draw_join_miter:
23952     { \_driver_draw_literal:n { 0 ~ j } }
23953 \cs_new_protected:Npn \_driver_draw_join_round:
23954     { \_driver_draw_literal:n { 1 ~ j } }
23955 \cs_new_protected:Npn \_driver_draw_join_bevel:
23956     { \_driver_draw_literal:n { 2 ~ j } }
```

(End definition for `_driver_draw_dash:nn` and others.)

Yet more fast conversion, all using the FPU to allow for expressions in numerical input.

```
23957 \cs_new_protected:Npn \_driver_draw_color_cmyk:nnnn
23958 {
23959     \use:x
23960     {
23961         \_driver_draw_color_cmyk_aux:nnnn
23962             { \fp_eval:n {#1} }
23963             { \fp_eval:n {#2} }
23964             { \fp_eval:n {#3} }
23965             { \fp_eval:n {#4} }
23966     }
23967 }
23968 \cs_new_protected:Npn \_driver_draw_color_cmyk_aux:nnnn #1#2#3#4
23969 {
23970     \_driver_draw_literal:n
23971     { #1 ~ #2 ~ #3 ~ #4 ~ k ~ #1 ~ #2 ~ #3 ~ #4 ~ K }
23972 }
```

```

23973 \cs_new_protected:Npn \__driver_draw_color_cmyk_fill:nnnn #1#2#3#4
23974 {
23975   \__driver_draw_literal:x
23976   {
23977     \fp_eval:n {#1} ~ \fp_eval:n {#2} ~
23978     \fp_eval:n {#3} ~ \fp_eval:n {#4} ~
23979     k
23980   }
23981 }
23982 \cs_new_protected:Npn \__driver_draw_color_cmyk_stroke:nnnn #1#2#3#4
23983 {
23984   \__driver_draw_literal:x
23985   {
23986     \fp_eval:n {#1} ~ \fp_eval:n {#2} ~
23987     \fp_eval:n {#3} ~ \fp_eval:n {#4} ~
23988     K
23989   }
23990 }
23991 \cs_new_protected:Npn \__driver_draw_color_gray:n #1
23992 {
23993   \use:x
23994   { \__driver_draw_color_gray_aux:n { \fp_eval:n {#1} } }
23995 }
23996 \cs_new_protected:Npn \__driver_draw_color_gray_aux:n #1
23997 {
23998   \__driver_draw_literal:n { #1 ~ g ~ #1 ~ G }
23999 }
24000 \cs_new_protected:Npn \__driver_draw_color_gray_fill:n #1
24001 { \__driver_draw_literal:x { \fp_eval:n {#1} ~ g } }
24002 \cs_new_protected:Npn \__driver_draw_color_gray_stroke:n #1
24003 { \__driver_draw_literal:x { \fp_eval:n {#1} ~ G } }
24004 \cs_new_protected:Npn \__driver_draw_color_rgb:nnn #1#2#3
24005 {
24006   \use:x
24007   {
24008     \__driver_draw_color_rgb_aux:nnn
24009     { \fp_eval:n {#1} }
24010     { \fp_eval:n {#2} }
24011     { \fp_eval:n {#3} }
24012   }
24013 }
24014 \cs_new_protected:Npn \__driver_draw_color_rgb_aux:nnn #1#2#3
24015 {
24016   \__driver_draw_literal:n
24017   { #1 ~ #2 ~ #3 ~ rg ~ #1 ~ #2 ~ #3 ~ RG }
24018 }
24019 \cs_new_protected:Npn \__driver_draw_color_rgb_fill:nnn #1#2#3
24020 {
24021   \__driver_draw_literal:x
24022   { \fp_eval:n {#1} ~ \fp_eval:n {#2} ~ \fp_eval:n {#3} ~ rg }
24023 }
24024 \cs_new_protected:Npn \__driver_draw_color_rgb_stroke:nnn #1#2#3
24025 {
24026   \__driver_draw_literal:x

```

```

24027      { \fp_eval:n {#1} ~ \fp_eval:n {#2} ~ \fp_eval:n {#3} ~ RG }
24028  }

```

(End definition for `_driver_draw_color_cmyk:nnnn` and others.)

`_driver_draw_transformcm:nnnnnn`

The first four arguments here are floats (the affine matrix), the last two are a displacement vector. Once again, force evaluation to allow for caching.

```

24029 \cs_new_protected:Npn \_driver_draw_transformcm:nnnnnn #1#2#3#4#5#6
24030  {
24031      \_driver_draw_literal:x
24032  {
24033      \fp_eval:n {#1} ~ \fp_eval:n {#2} ~
24034      \fp_eval:n {#3} ~ \fp_eval:n {#4} ~
24035      \dim_to_decimal_in_bp:n {#5} ~ \dim_to_decimal_in_bp:n {#6} ~
24036      cm
24037  }
24038 }

```

(End definition for `_driver_draw_transformcm:nnnnnn`.)

`_driver_draw_hbox:Nnnnnnn`

`\l_driver_tmp_box`

Inserting a TeX box transformed to the requested position and using the current matrix is done using a mixture of TeX and low-level manipulation. The offset can be handled by TeX, so only any rotation/skew/scaling component needs to be done using the matrix operation. As this operation can never be cached, the scope is set directly not using the `draw` version.

```

24039 \cs_new_protected:Npn \_driver_draw_hbox:Nnnnnnn #1#2#3#4#5#6#7
24040  {
24041      \hbox_set:Nn \l_driver_tmp_box
24042  {
24043      \tex kern:D \_dim_eval:w #6 \_dim_eval_end:
24044      \_driver_scope_begin:
24045      \_driver_draw_transformcm:nnnnnn {#2} {#3} {#4} {#5}
24046      { Opt } { Opt }
24047      \box_move_up:nn {#7} { \box_use:N #1 }
24048      \_driver_scope_end:
24049  }
24050      \box_set_wd:Nn \l_driver_tmp_box { Opt }
24051      \box_set_ht:Nn \l_driver_tmp_box { Opt }
24052      \box_set_dp:Nn \l_driver_tmp_box { Opt }
24053      \box_use:N \l_driver_tmp_box
24054  }
24055 \box_new:N \l_driver_tmp_box

```

(End definition for `_driver_draw_hbox:Nnnnnnn` and `\l_driver_tmp_box`.)

24056 </dvipdfmx | pdfmode | xdvipdfmx>

44.9 dvips driver

24057 `(*dvips)`

44.9.1 Basics

`_driver_literal:n`

In the case of `dvips` there is no build-in saving of the current position, and so some additional PostScript is required to set up the transformation matrix and also to restore

it afterwards. Notice the use of the stack to save the current position “up front” and to move back to it at the end of the process.

```

24058 \cs_new_protected:Npn \__driver_literal:n #1
24059   {
24060     \tex_special:D
24061     {
24062       ps:
24063         currentpoint~
24064         currentpoint-translate~
24065         #1 ~
24066         neg-exch-neg-exch-translate
24067     }
24068 }
```

(End definition for __driver_literal:n.)

__driver_scope_begin: Scope saving/restoring is done directly with no need to worry about the transformation matrix. General scoping is only for the graphics stack so the lower-cost `gsave/grestore` pair are used.

```

24069 \cs_new_protected:Npn \__driver_scope_begin:
24070   { \tex_special:D { ps:gsave } }
24071 \cs_new_protected:Npn \__driver_scope_end:
24072   { \tex_special:D { ps:grestore } }
```

(End definition for __driver_scope_begin: and __driver_scope_end:.)

44.10 Driver-specific auxiliaries

__driver_absolute_lengths:n The dvips driver scales all absolute dimensions based on the output resolution selected and any TeX magnification. Thus for any operation involving absolute lengths there is a correction to make. This is based on `normalscale` from `special.pro` but using the stack rather than a definition to save the current matrix.

```

24073 \cs_new:Npn \__driver_absolute_lengths:n #1
24074   {
24075     matrix~currentmatrix~
24076     Resolution-72-div-VResolution-72-div-scale~
24077     DVImag~dup~scale~
24078     #1 ~
24079     setmatrix
24080 }
```

(End definition for __driver_absolute_lengths:n.)

44.10.1 Box operations

__driver_box_use_clip:N Much the same idea as for the PDF mode version but with a slightly different syntax for creating the clip path. To avoid any scaling issues we need the absolute length auxiliary here.

```

24081 \cs_new_protected:Npn \__driver_box_use_clip:N #1
24082   {
24083     \__driver_scope_begin:
24084     \__driver_literal:n
24085   }
```

```

24086     \__driver_absolute_lengths:n
24087     {
24088         0 ~
24089         \dim_to_decimal_in_bp:n { \box_dp:N #1 } ~
24090         \dim_to_decimal_in_bp:n { \box_wd:N #1 } ~
24091         \dim_to_decimal_in_bp:n { -\box_ht:N #1 - \box_dp:N #1 } ~
24092         rectclip
24093     }
24094 }
24095 \hbox_overlap_right:n { \box_use:N #1 }
24096 \__driver_scope_end:
24097 \skip_horizontal:n { \box_wd:N #1 }
24098 }

```

(End definition for `__driver_box_use_clip:N.`)

__driver_box_use_rotate:Nn Rotating using dvips does not require that the box dimensions are altered and has a very convenient built-in operation. Zero rotation must be written as 0 not -0 so there is a quick test.

```

24099 \cs_new_protected:Npn \__driver_box_use_rotate:Nn #1#2
24100 {
24101     \__driver_scope_begin:
24102     \__driver_literal:n
24103     {
24104         \fp_compare:nNnTF {#2} = \c_zero_fp
24105         { 0 }
24106         { \fp_eval:n { round ( -#2 , 5 ) } } ~
24107         rotate
24108     }
24109     \box_use:N #1
24110     \__driver_scope_end:
24111 }
24112 % \end{macro}
24113 %
24114 % \begin{macro}{\__driver_box_use_scale:Nnn}
24115 %   The \texttt{\{dvips\}} driver once again has a dedicated operation we can
24116 %   use here.
24117 %   \begin{macrocode}
24118 \cs_new_protected:Npn \__driver_box_use_scale:Nnn #1#2#3
24119 {
24120     \__driver_scope_begin:
24121     \__driver_literal:n
24122     {
24123         \fp_eval:n { round ( #2 , 5 ) } ~
24124         \fp_eval:n { round ( #3 , 5 ) } ~
24125         scale
24126     }
24127     \hbox_overlap_right:n { \box_use:N #1 }
24128     \__driver_scope_end:
24129 }

```

(End definition for `__driver_box_use_rotate:Nn.`)

44.10.2 Color

\l__driver_color_current_tl The current color in driver-dependent format.

```
24130 \tl_new:N \l__driver_color_current_tl
24131 \tl_set:Nn \l__driver_color_current_tl { gray~0 }
24132 (*package)
24133 \AtBeginDocument
24134 {
24135     \@ifpackageloaded { color }
24136         { \tl_set:Nn \l__driver_color_current_tl { \current@color } }
24137         { }
24138     }
24139 }/package)
```

(End definition for \l__driver_color_current_tl.)

_driver_color_ensure_current: Directly set the color using the specials: no optimisation here.

```
24140 \cs_new_protected:Npn \_driver_color_ensure_current:
24141 {
24142     \tex_special:D { color-push~\l__driver_color_current_tl }
24143     \group_insert_after:N \_driver_color_reset:
24144 }
24145 \cs_new_protected:Npn \_driver_color_reset:
24146     { \tex_special:D { color-pop } }
```

(End definition for _driver_color_ensure_current: and _driver_color_reset:.)

44.11 Images

_driver_image_getbb_eps:n Simply use the generic function.

```
24147 \cs_new_eq:NN \_driver_image_getbb_eps:n \_image_read_bb:n
24148 (End definition for \_driver_image_getbb_eps:n.)
```

_driver_image_include_eps:n The special syntax is relatively clear here: remember we need PostScript sizes here.

```
24149 \cs_new_protected:Npn \_driver_image_include_eps:n #1
24150 {
24151     \tex_special:D { PSfile = #1 }
```

(End definition for _driver_image_include_eps:n.)

44.12 Drawing

_driver_draw_literal:n _driver_draw_literal:x Literals with no positioning (using ps: each one is positioned but cut off from everything else, so no good for the stepwise approach needed here).

```
24152 \cs_new_protected:Npn \_driver_draw_literal:n #1
24153     { \tex_special:D { ps:: - #1 } }
24154 \cs_generate_variant:Nn \_driver_draw_literal:n { x }
```

(End definition for _driver_draw_literal:n.)

`__driver_draw_begin:` The `ps::[begin]` special here deals with positioning but allows us to continue on to a matching `ps::[end]`: contrast with `ps:`, which positions but where we can't split material between separate calls. The `@beginspecial/@endspecial` pair are from `special.pro` and correct the scale and y -axis direction. The reference point at the start of the box is saved (as `13x/13y`) as it is needed when inserting various items.

```
24155 \cs_new_protected:Npn \__driver_draw_begin:
24156 {
24157   \tex_special:D { ps::[begin] }
24158   \tex_special:D { ps::~save }
24159   \tex_special:D { ps::~/13x~currentpoint~/13y~exch~def~def }
24160   \tex_special:D { ps::~@beginspecial }
24161 }
24162 \cs_new_protected:Npn \__driver_draw_end:
24163 {
24164   \tex_special:D { ps::~@endspecial }
24165   \tex_special:D { ps::~restore }
24166   \tex_special:D { ps::[end] }
24167 }
```

(End definition for `__driver_draw_begin:` and `__driver_draw_end:..`)

`__driver_draw_scope_begin:` Scope here may need to contain saved definitions, so the entire memory rather than just the graphic state has to be sent to the stack.

```
24168 \cs_new_protected:Npn \__driver_draw_scope_begin:
24169   { \__driver_draw_literal:n { save } }
24170 \cs_new_protected:Npn \__driver_draw_scope_end:
24171   { \__driver_draw_literal:n { restore } }
```

(End definition for `__driver_draw_scope_begin:` and `__driver_draw_scope_end:..`)

`__driver_draw_moveto:nn` `__driver_draw_lineto:nn` Path creation operations mainly resolve directly to PostScript primitive steps, with only the need to convert to bp. Notice that x-type expansion is included here to ensure that any variable values are forced to literals before any possible caching. There is no native rectangular path command (without also clipping, filling or stroking), so that task is done using a small amount of PostScript.

```
24172 \cs_new_protected:Npn \__driver_draw_moveto:nn #1#2
24173 {
24174   \__driver_draw_literal:x
24175   { \dim_to_decimal_in_bp:n {#1} ~ \dim_to_decimal_in_bp:n {#2} ~ moveto }
24176 }
24177 \cs_new_protected:Npn \__driver_draw_lineto:nn #1#2
24178 {
24179   \__driver_draw_literal:x
24180   { \dim_to_decimal_in_bp:n {#1} ~ \dim_to_decimal_in_bp:n {#2} ~ lineto }
24181 }
24182 \cs_new_protected:Npn \__driver_draw_rectangle:nnnn #1#2#3#4
24183 {
24184   \__driver_draw_literal:x
24185   {
24186     \dim_to_decimal_in_bp:n {#4} ~ \dim_to_decimal_in_bp:n {#3} ~
24187     \dim_to_decimal_in_bp:n {#1} ~ \dim_to_decimal_in_bp:n {#2} ~
24188     moveto~dup~0~rlineto~exch~0~exch~rlineto~neg~0~rlineto~clospath
24189 }
```

```

24190     }
24191 \cs_new_protected:Npn \__driver_draw_curveto:nnnnnn #1#2#3#4#5#6
24192 {
24193     \__driver_draw_literal:x
24194     {
24195         \dim_to_decimal_in_bp:n {#1} ~ \dim_to_decimal_in_bp:n {#2} ~
24196         \dim_to_decimal_in_bp:n {#3} ~ \dim_to_decimal_in_bp:n {#4} ~
24197         \dim_to_decimal_in_bp:n {#5} ~ \dim_to_decimal_in_bp:n {#6} ~
24198         curveto
24199     }
24200 }

```

(End definition for `__driver_draw_moveto:nn` and others.)

`__driver_draw_evenodd_rule:`
`__driver_draw_nonzero_rule:`

`\g__driver_draw_eor_bool`

The even-odd rule here can be implemented as a simply switch.

```

24201 \cs_new_protected:Npn \__driver_draw_evenodd_rule:
24202     { \bool_gset_true:N \g__driver_draw_eor_bool }
24203 \cs_new_protected:Npn \__driver_draw_nonzero_rule:
24204     { \bool_gset_false:N \g__driver_draw_eor_bool }
24205 \bool_new:N \g__driver_draw_eor_bool

```

(End definition for `__driver_draw_evenodd_rule:`, `__driver_draw_nonzero_rule:`, and `\g__driver_draw_eor_bool`.)

`__driver_draw_closepath:`
`__driver_draw_stroke:`
`__driver_draw_closestroke:`
`__driver_draw_fill:`
`__driver_draw_fillstroke:`
`__driver_draw_clip:`
`__driver_draw_discardpath:`
`\g__driver_draw_clip_bool`

Unlike PDF, PostScript doesn't track separate colors for strokes and other elements. It is also desirable to have the `clip` keyword after a stroke or fill. To achieve those outcomes, there is some work to do. For color, if a stroke or fill color is defined it is used for the relevant operation, with a graphic scope inserted as required. That does mean that once such a color is set all further uses inside the same scope have to use scoping: see also the color set up functions. For clipping, the required ordering is achieved using a T_EX switch. All of the operations end with a new path instruction as they do not terminate (again in contrast to PDF).

```

24206 \cs_new_protected:Npn \__driver_draw_closepath:
24207     { \__driver_draw_literal:n { closepath } }
24208 \cs_new_protected:Npn \__driver_draw_stroke:
24209     {
24210         \__driver_draw_literal:n { currentdict~/13sc-known~{gsave~13sc}~if }
24211         \__driver_draw_literal:n { stroke }
24212         \__driver_draw_literal:n { currentdict~/13sc-known~{grestore}~if }
24213         \bool_if:NT \g__driver_draw_clip_bool
24214         {
24215             \__driver_draw_literal:x
24216             {
24217                 \bool_if:NT \g__driver_draw_eor_bool { eo }
24218                 clip
24219             }
24220         }
24221         \__driver_draw_literal:n { newpath }
24222         \bool_gset_false:N \g__driver_draw_clip_bool
24223     }
24224 \cs_new_protected:Npn \__driver_draw_closestroke:
24225     {
24226         \__driver_draw_closepath:
24227         \__driver_draw_stroke:

```

```

24228     }
24229 \cs_new_protected:Npn \__driver_draw_fill:
2430 {
2431     \__driver_draw_literal:n { currentdict~/13fc~known~{gsave~13fc}~if }
2432     \__driver_draw_literal:x
2433     {
2434         \bool_if:NT \g__driver_draw_eor_bool { eo }
2435         fill
2436     }
2437     \__driver_draw_literal:n { currentdict~/13fc~known~{grestore}~if }
2438     \bool_if:NT \g__driver_draw_clip_bool
2439     {
2440         \__driver_draw_literal:x
2441         {
2442             \bool_if:NT \g__driver_draw_eor_bool { eo }
2443             clip
2444         }
2445     }
2446     \__driver_draw_literal:n { newpath }
2447     \bool_gset_false:N \g__driver_draw_clip_bool
2448 }
2449 \cs_new_protected:Npn \__driver_draw_fillstroke:
2450 {
2451     \__driver_draw_literal:n { currentdict~/13fc~known~{gsave~13fc}~if }
2452     \__driver_draw_literal:x
2453     {
2454         \bool_if:NT \g__driver_draw_eor_bool { eo }
2455         fill
2456     }
2457     \__driver_draw_literal:n { currentdict~/13fc~known~{grestore}~if }
2458     \__driver_draw_literal:n { currentdict~/13sc~known~{gsave~13sc}~if }
2459     \__driver_draw_literal:n { stroke }
2460     \__driver_draw_literal:n { currentdict~/13sc~known~{grestore}~if }
2461     \bool_if:NT \g__driver_draw_clip_bool
2462     {
2463         \__driver_draw_literal:x
2464         {
2465             \bool_if:NT \g__driver_draw_eor_bool { eo }
2466             clip
2467         }
2468     }
2469     \__driver_draw_literal:n { newpath }
2470     \bool_gset_false:N \g__driver_draw_clip_bool
2471 }
2472 \cs_new_protected:Npn \__driver_draw_clip:
2473 {
2474     \bool_gset_true:N \g__driver_draw_clip_bool
2475     \bool_new:N \g__driver_draw_clip_bool
2476 \cs_new_protected:Npn \__driver_draw_discardpath:
2477 {
2478     \bool_if:NT \g__driver_draw_clip_bool
2479     {
2480         \__driver_draw_literal:x
2481         {
2482             \bool_if:NT \g__driver_draw_eor_bool { eo }

```

```

24282         clip
24283     }
24284 }
24285 \__driver_draw_literal:n { newpath }
24286 \bool_gset_false:N \g__driver_draw_clip_bool
24287 }

```

(End definition for `__driver_draw_closepath`: and others.)

`__driver_draw_dash:nn`

`__driver_draw_dash:n`

`__driver_draw_linewidth:n`

`__driver_draw_miterlimit:n`

`__driver_draw_cap_but:`

`__driver_draw_cap_round:`

`__driver_draw_cap_rectangle:`

`__driver_draw_join_miter:`

`__driver_draw_join_round:`

`__driver_draw_join_bevel:`

Converting paths to output is again a case of mapping directly to PostScript operations.

```

24288 \cs_new_protected:Npn \__driver_draw_dash:nn #1#2
24289 {
24290     \__driver_draw_literal:x
24291     {
24292         [ ~
24293             \clist_map_function:nN {#1} \__driver_draw_dash:n
24294         ] ~
24295             \dim_to_decimal_in_bp:n {#2} ~ setdash
24296     }
24297 }
24298 \cs_new:Npn \__driver_draw_dash:n #1
24299     { \dim_to_decimal_in_bp:n {#1} ~ }
24300 \cs_new_protected:Npn \__driver_draw_linewidth:n #1
24301 {
24302     \__driver_draw_literal:x
24303     { \dim_to_decimal_in_bp:n {#1} ~ setlinewidth }
24304 }
24305 \cs_new_protected:Npn \__driver_draw_miterlimit:n #1
24306     { \__driver_draw_literal:x { \fp_eval:n {#1} ~ setmiterlimit } }
24307 \cs_new_protected:Npn \__driver_draw_cap_but:
24308     { \__driver_draw_literal:n { 0 ~ setlinecap } }
24309 \cs_new_protected:Npn \__driver_draw_cap_round:
24310     { \__driver_draw_literal:n { 1 ~ setlinecap } }
24311 \cs_new_protected:Npn \__driver_draw_cap_rectangle:
24312     { \__driver_draw_literal:n { 2 ~ setlinecap } }
24313 \cs_new_protected:Npn \__driver_draw_join_miter:
24314     { \__driver_draw_literal:n { 0 ~ setlinejoin } }
24315 \cs_new_protected:Npn \__driver_draw_join_round:
24316     { \__driver_draw_literal:n { 1 ~ setlinejoin } }
24317 \cs_new_protected:Npn \__driver_draw_join_bevel:
24318     { \__driver_draw_literal:n { 2 ~ setlinejoin } }

```

(End definition for `__driver_draw_dash:nn` and others.)

To allow color to be defined for strokes and fills separately and to respect scoping, the data needs to be stored at the PostScript level. We cannot undefine (local) fill/stroke colors once set up but we can set them blank to improve performance slightly.

```

24319 \cs_new_protected:Npn \__driver_draw_color_reset:
24320 {
24321     \__driver_draw_literal:n { currentdic~/l3fc~known~{ /l3fc~ { } ~def }~if }
24322     \__driver_draw_literal:n { currentdic~/l3sc~known~{ /l3sc~ { } ~def }~if }
24323 }
24324 \cs_new_protected:Npn \__driver_draw_color_cmyk:nnnn #1#2#3#4
24325 {

```

```

24326     \__driver_draw_literal:x
24327     {
24328         \fp_eval:n {#1} ~ \fp_eval:n {#2} ~
24329         \fp_eval:n {#3} ~ \fp_eval:n {#4} ~
24330         setcmykcolor ~
24331     }
24332     \__driver_draw_color_reset:
24333 }
24334 \cs_new_protected:Npn \__driver_draw_color_cmyk_fill:nnnn #1#2#3#4
24335 {
24336     \__driver_draw_literal:x
24337     {
24338         /13fc ~
24339     {
24340         \fp_eval:n {#1} ~ \fp_eval:n {#2} ~
24341         \fp_eval:n {#3} ~ \fp_eval:n {#4} ~
24342         setcmykcolor
24343     } ~
24344     def
24345 }
24346 }
24347 \cs_new_protected:Npn \__driver_draw_color_cmyk_stroke:nnnn #1#2#3#4
24348 {
24349     \__driver_draw_literal:x
24350     {
24351         /13sc ~
24352     {
24353         \fp_eval:n {#1} ~ \fp_eval:n {#2} ~
24354         \fp_eval:n {#3} ~ \fp_eval:n {#4} ~
24355         setcmykcolor
24356     } ~
24357     def
24358 }
24359 }
24360 \cs_new_protected:Npn \__driver_draw_color_gray:n #1
24361 {
24362     \__driver_draw_literal:x { fp_eval:n {#1} ~ setgray }
24363     \__driver_draw_color_reset:
24364 }
24365 \cs_new_protected:Npn \__driver_draw_color_gray_fill:n #1
24366 { \__driver_draw_literal:x { /13fc ~ { \fp_eval:n {#1} ~ setgray } ~ def } }
24367 \cs_new_protected:Npn \__driver_draw_color_gray_stroke:n #1
24368 { \__driver_draw_literal:x { /13sc ~ { \fp_eval:n {#1} ~ setgray } ~ def } }
24369 \cs_new_protected:Npn \__driver_draw_color_rgb:nnn #1#2#3
24370 {
24371     \__driver_draw_literal:x
24372     {
24373         \fp_eval:n {#1} ~ \fp_eval:n {#2} ~ \fp_eval:n {#3} ~
24374         setrgbcolor
24375     }
24376     \__driver_draw_color_reset:
24377 }
24378 \cs_new_protected:Npn \__driver_draw_color_rgb_fill:nnn #1#2#3
24379 {

```

```

24380     \__driver_draw_literal:x
24381     {
24382         /13fc ~
24383         {
24384             \fp_eval:n {#1} ~ \fp_eval:n {#2} ~ \fp_eval:n {#3} ~
24385             setrgbcolor
24386         } ~
24387         def
24388     }
24389 }
24390 \cs_new_protected:Npn \__driver_draw_color_rgb_stroke:nnn #1#2#3
24391 {
24392     \__driver_draw_literal:x
24393     {
24394         /13sc ~
24395         {
24396             \fp_eval:n {#1} ~ \fp_eval:n {#2} ~ \fp_eval:n {#3} ~
24397             setrgbcolor
24398         } ~
24399         def
24400     }
24401 }

```

(End definition for `__driver_draw_color_reset:` and others.)

`__driver_draw_transformcm:nnnnnn` The first four arguments here are floats (the affine matrix), the last two are a displacement vector. Once again, force evaluation to allow for caching.

```

24402 \cs_new_protected:Npn \__driver_draw_transformcm:nnnnnn #1#2#3#4#5#6
24403 {
24404     \__driver_draw_literal:x
24405     {
24406         [
24407             \fp_eval:n {#1} ~ \fp_eval:n {#2} ~
24408             \fp_eval:n {#3} ~ \fp_eval:n {#4} ~
24409             \dim_to_decimal_in_bp:n {#5} ~ \dim_to_decimal_in_bp:n {#6} ~
24410         ] ~
24411         concat
24412     }
24413 }

```

(End definition for `__driver_draw_transformcm:nnnnnn`.)

`__driver_draw_hbox:Nnnnnnn` Inside a picture `@beginspecial`/`@endspecial` are active, which is normally a good thing but means that the position and scaling will be off if the box is inserted directly. Instead, we need to reverse the effect of the (normally desirable) shift/scaling within the box. That requires knowing where the reference point for the drawing is: saved as $13x/13y$ at the start of the picture. Transformation here is relative to the drawing origin so has to be done purely in driver code not using TeX offsets.

```

24414 \cs_new_protected:Npn \__driver_draw_hbox:Nnnnnnn #1#2#3#4#5#6#7
24415 {
24416     \__driver_scope_begin:
24417     \tex_special:D { ps::[end] }
24418     \__driver_draw_transformcm:nnnnnn {#2} {#3} {#4} {#5} {#6} {#7}
24419     \tex_special:D { ps::~72~Resolution~div~72~VResolution~div~neg-scale }

```

```

24420      \tex_special:D { ps::-magscale-{1~DVImag~div~dup~scale}~if }
24421      \tex_special:D { ps::-13x~neg~13y~neg~translate }
24422      \group_begin:
24423          \box_set_wd:Nn #1 { Opt }
24424          \box_set_ht:Nn #1 { Opt }
24425          \box_set_dp:Nn #1 { Opt }
24426          \box_use:N #1
24427      \group_end:
24428      \tex_special:D { ps::[begin] }
24429      \__driver_scope_end:
24430  }

```

(End definition for `__driver_draw_hbox:Nnnnnnn.`)

24431 ⟨/dvips⟩

44.13 dvisvgm driver

24432 ⟨*dvisvgm⟩

44.13.1 Basics

`__driver_literal:n` Unlike the other drivers, the requirements for making SVG files mean that we can't conveniently transform all operations to the current point. That makes life a bit more tricky later as that needs to be accounted for. A new line is added after each call to help to keep the output readable for debugging.

```

24433 \cs_new_protected:Npn \__driver_literal:n #1
24434     { \tex_special:D { dvisvgm:raw~ #1 { ?nl } } }

```

(End definition for `__driver_literal:n.`)

`__driver_scope_begin:` A scope in SVG terms is slightly different to the other drivers as operations have to be “tied” to these not simply inside them.

```

24435 \cs_new_protected:Npn \__driver_scope_begin:
24436     { \__driver_literal:n { <g> } }
24437 \cs_new_protected:Npn \__driver_scope_end:
24438     { \__driver_literal:n { </g> } }

```

(End definition for `__driver_scope_begin:` and `__driver_scope_end:.`)

44.14 Driver-specific auxiliaries

`__driver_scope_begin:n` In SVG transformations, clips and so on are attached directly to scopes so we need a way or allowing for that. This is rather more useful than `__driver_scope_begin:` as a result. No assumptions are made about the nature of the scoped operation(s).

```

24439 \cs_new_protected:Npn \__driver_scope_begin:n #1
24440     { \__driver_literal:n { <g~ #1 > } }

```

(End definition for `__driver_scope_begin:n.`)

44.14.1 Box operations

`__driver_box_use_clip:N`
`\g__driver_clip_path_int`

Clipping in SVG is more involved than with other drivers. The first issue is that the clipping path must be defined separately from where it is used, so we need to track how many paths have applied. The naming here uses `13cp` as the namespace with a number following. Rather than use a rectangular operation, we define the path manually as this allows it to have a depth: easier than the alternative approach of shifting content up and down using scopes to allow for the depth of the TeX box and keep the reference point the same!

```

24441 \cs_new_protected:Npn \__driver_box_use_clip:N #1
24442   {
24443     \int_gincr:N \g__driver_clip_path_int
24444     \__driver_literal:n
24445       { < clipPath-id = " 13cp \int_use:N \g__driver_clip_path_int " > }
24446     \__driver_literal:n
24447       {
24448         <
24449           path ~ d =
24450             "
24451             M ~ O ~
24452               \dim_to_decimal:n { -\box_dp:N #1 } ~
24453               L ~ \dim_to_decimal:n { \box_wd:N #1 } ~
24454                 \dim_to_decimal:n { -\box_dp:N #1 } ~
24455               L ~ \dim_to_decimal:n { \box_wd:N #1 } ~
24456                 \dim_to_decimal:n { \box_ht:N #1 + \box_dp:N #1 } ~
24457               L ~ O ~
24458                 \dim_to_decimal:n { \box_ht:N #1 + \box_dp:N #1 } ~
24459               Z
24460             "
24461         />
24462       }
24463     \__driver_literal:n
24464       { < /clipPath > }

```

In general the SVG set up does not try to transform coordinates to the current point. For clipping we need to do that, so have a transformation here to get us to the right place, and a matching one just before the TeX box is inserted to get things back on track. The clip path needs to come between those two such that if lines up with the current point, as does the TeX box.

```

24465   \__driver_scope_begin:n
24466   {
24467     transform =
24468     "
24469       translate ( { ?x } , { ?y } ) ~
24470         scale ( 1 , -1 )
24471     "
24472   }
24473   \__driver_scope_begin:n
24474   {
24475     clip-path = "url ( \c_hash_str 13cp \int_use:N \g__driver_clip_path_int ) "
24476   }
24477   \__driver_scope_begin:n
24478   {
24479     transform =

```

```

24480      "
24481      scale ( -1 , 1 ) ~
24482      translate ( { ?x } , { ?y } ) ~
24483      scale ( -1 , -1 )
24484      "
24485      }
24486      \box_use:N #1
24487      \__driver_scope_end:
24488      \__driver_scope_end:
24489      \__driver_scope_end:
24490      % \skip_horizontal:n { \box_wd:N #1 }
24491      }
24492 \int_new:N \g__driver_clip_path_int

```

(End definition for `__driver_box_use_clip:N` and `\g__driver_clip_path_int`.)

`__driver_box_use_rotate:Nn`

Rotation has a dedicated operation which includes a centre-of-rotation optional pair. That can be picked up from the driver syntax, so there is no need to worry about the transformation matrix.

```

24493 \cs_new_protected:Npn \__driver_box_use_rotate:Nn #1#2
24494 {
24495     \__driver_scope_begin:n
24496     {
24497         transform =
24498         "
24499         rotate
24500         ( \fp_eval:n { round ( -#2 , 5 ) } , ~ { ?x } , ~ { ?y } )
24501         "
24502     }
24503     \box_use:N #1
24504     \__driver_scope_end:
24505 }

```

(End definition for `__driver_box_use_rotate:Nn`.)

`__driver_box_use_scale:Nnn`

In contrast to rotation, we have to account for the current position in this case. That is done using a couple of translations in addition to the scaling (which is therefore done backward with a flip).

```

24506 \cs_new_protected:Npn \__driver_box_use_scale:Nnn #1#2#3
24507 {
24508     \__driver_scope_begin:n
24509     {
24510         transform =
24511         "
24512         translate ( { ?x } , { ?y } ) ~
24513         scale
24514         (
24515             \fp_eval:n { round ( -#2 , 5 ) } ,
24516             \fp_eval:n { round ( -#3 , 5 ) }
24517         ) ~
24518         translate ( { ?x } , { ?y } ) ~
24519         scale ( -1 )
24520         "
24521 }

```

```

24522      \hbox_overlap_right:n { \box_use:N #1 }
24523      \_driver_scope_end:
24524  }

```

(End definition for `_driver_box_use_scale:Nnn`.)

44.14.2 Color

`\l_driver_color_current_tl` The current color in driver-dependent format: the same as for dvips.

```

24525 \tl_new:N \l_driver_color_current_tl
24526 \tl_set:Nn \l_driver_color_current_tl { gray~0 }
24527 (*package)
24528 \AtBeginDocument
24529 {
24530     \@ifpackageloading { color }
24531         { \tl_set:Nn \l_driver_color_current_tl { \current@color } }
24532         { }
24533     }
24534 
```

(End definition for `\l_driver_color_current_tl`.)

`_driver_color_ensure_current:` Directly set the color: same as dvips.

```

\l_driver_color_reset:
24535 \cs_new_protected:Npn \_driver_color_ensure_current:
24536 {
24537     \tex_special:D { color-push~\l_driver_color_current_tl }
24538     \group_insert_after:N \_driver_color_reset:
24539 }
24540 \cs_new_protected:Npn \_driver_color_reset:
24541     { \tex_special:D { color-pop } }

```

(End definition for `_driver_color_ensure_current:` and `_driver_color_reset:..`)

44.15 Drawing

`_driver_draw_literal:n` The same as the more general literal call.

```

24542 \cs_new_eq:NN \_driver_draw_literal:n \_driver_literal:n
24543 \cs_generate_variant:Nn \_driver_draw_literal:n { x }

```

(End definition for `_driver_draw_literal:n`.)

`_driver_draw_begin:` `_driver_draw_end:` A drawing needs to be set up such that the co-ordinate system is translated. That is done inside a scope, which as described below

```

24544 \cs_new_protected:Npn \_driver_draw_begin:
24545 {
24546     \_driver_scope_begin:
24547     \_driver_scope:n { transform="translate({?x},{?y})~scale(1,-1)" }
24548 }
24549 \cs_new_protected:Npn \_driver_draw_end:
24550     { \_driver_scope_end: }

```

(End definition for `_driver_draw_begin:` and `_driver_draw_end:..`)

```
\__driver_draw_scope_begin:  
\__driver_draw_scope_end:  
  \__driver_draw_scope:n  
  \__driver_draw_scope:x  
\g__driver_draw_scope_int  
\l__driver_draw_scope_int
```

Several settings that with other drivers are “stand alone” have to be given as part of a scope in SVG. As a result, there is a need to provide a mechanism to automatically close these extra scopes. That is done using a dedicated function and a pair of tracking variables. Within each graphics scope we use a global variable to do the work, with a group used to save the value between scopes. The result is that no direct action is needed when creating a scope.

```
24551 \cs_new_protected:Npn \__driver_draw_scope_begin:  
24552 {  
24553   \int_set_eq:NN  
24554   \l__driver_draw_scope_int  
24555   \g__driver_draw_scope_int  
24556   \group_begin:  
24557     \int_gzero:N \g__driver_draw_scope_int  
24558   }  
24559 \cs_new_protected:Npn \__driver_draw_scope_end:  
24560 {  
24561   \prg_replicate:nn  
24562   { \g__driver_draw_scope_int }  
24563   { \__driver_draw_literal:n { </g> } }  
24564   \group_end:  
24565   \int_gset_eq:NN  
24566   \g__driver_draw_scope_int  
24567   \l__driver_draw_scope_int  
24568 }  
24569 \cs_new_protected:Npn \__driver_draw_scope:n #1  
24570 {  
24571   \__driver_draw_literal:n { <g~ #1 > }  
24572   \int_gincr:N \g__driver_draw_scope_int  
24573 }  
24574 \cs_generate_variant:Nn \__driver_draw_scope:n { x }  
24575 \int_new:N \g__driver_draw_scope_int  
24576 \int_new:N \l__driver_draw_scope_int
```

(End definition for `__driver_draw_scope_begin:` and others.)

```
\__driver_draw_moveto:nn  
\__driver_draw_lineto:nn  
  \__driver_draw_rectangle:nnnn  
  \__driver_draw_curveTo:nnnnnn
```

```
\__driver_draw_add_to_path:n
```

```
\g__driver_draw_path_tl
```

```
24577 \cs_new_protected:Npn \__driver_draw_moveto:nn #1#2  
24578 {  
24579   \__driver_draw_add_to_path:n  
24580   { M ~ \dim_to_decimal:n {#1} ~ \dim_to_decimal:n {#2} }  
24581 }  
24582 \cs_new_protected:Npn \__driver_draw_lineto:nn #1#2  
24583 {  
24584   \__driver_draw_add_to_path:n  
24585   { L ~ \dim_to_decimal:n {#1} ~ \dim_to_decimal:n {#2} }  
24586 }  
24587 \cs_new_protected:Npn \__driver_draw_rectangle:nnnn #1#2#3#4  
24588 {  
24589   \__driver_draw_add_to_path:n
```

```

24590    {
24591        M ~ \dim_to_decimal:n {#1} ~ \dim_to_decimal:n {#2}
24592        h ~ \dim_to_decimal:n {#3} ~
24593        v ~ \dim_to_decimal:n {#4} ~
24594        h ~ \dim_to_decimal:n { -#3 } ~
24595        Z
24596    }
24597 }
24598 \cs_new_protected:Npn \__driver_draw_curveto:nnnnnn #1#2#3#4#5#6
24599 {
24600     \__driver_draw_add_to_path:n
24601     {
24602         C ~
24603         \dim_to_decimal:n {#1} ~ \dim_to_decimal:n {#2} ~
24604         \dim_to_decimal:n {#3} ~ \dim_to_decimal:n {#4} ~
24605         \dim_to_decimal:n {#5} ~ \dim_to_decimal:n {#6}
24606     }
24607 }
24608 \cs_new_protected:Npn \__driver_draw_add_to_path:n #1
24609 {
24610     \tl_gset:Nx \g__driver_draw_path_tl
24611     {
24612         \g__driver_draw_path_tl
24613         \tl_if_empty:NF \g__driver_draw_path_tl { \c_space_tl }
24614         #1
24615     }
24616 }
24617 \tl_new:N \g__driver_draw_path_tl

```

(End definition for `__driver_draw_moveto:nn` and others.)

The fill rules here have to be handled as scopes.

```

24618 \cs_new_protected:Npn \__driver_draw_evenodd_rule:
24619     { \__driver_draw_scope:n { fill-rule="evenodd" } }
24620 \cs_new_protected:Npn \__driver_draw_nonzero_rule:
24621     { \__driver_draw_scope:n { fill-rule="nonzero" } }

```

(End definition for `__driver_draw_evenodd_rule:` and `__driver_draw_nonzero_rule::`)

Setting fill and stroke effects and doing clipping all has to be done using scopes. This means setting up the various requirements in a shared auxiliary which deals with the bits and pieces. Clipping paths are reused for path drawing: not essential but avoids constructing them twice. Discarding a path needs a separate function as it's not quite the same.

```

\__driver_draw_path:n
\__driver_draw_closepath:
\__driver_draw_stroke:
\__driver_draw_closestroke:
\__driver_draw_fill:
\__driver_draw_fillstroke:
\__driver_draw_clip:
\__driver_draw_discardpath:
\g__driver_draw_clip_bool
\g__driver_draw_path_int
24622 \cs_new_protected:Npn \__driver_draw_closepath:
24623     { \__driver_draw_add_to_path:n { Z } }
24624 \cs_new_protected:Npn \__driver_draw_path:n #1
24625     {
24626         \bool_if:NTF \g__driver_draw_clip_bool
24627         {
24628             \int_gincr:N \g__driver_clip_path_int
24629             \__driver_draw_literal:x
24630             {
24631                 < clipPath~id = " 13cp \int_use:N \g__driver_clip_path_int " >

```

```

24632      { ?nl }
24633      <path~d=" \g__driver_draw_path_tl "/> { ?nl }
24634      < /clipPath > { ? nl }
24635      <
24636          use~xlink:href =
24637              "\c_hash_str l3path \int_use:N \g__driver_path_int " ~
24638                  #1
24639          />
24640      }
24641      \__driver_draw_scope:x
24642      {
24643          clip-path =
24644              "url( \c_hash_str l3cp \int_use:N \g__driver_clip_path_int)"
24645      }
24646  }
24647  {
24648      \__driver_draw_literal:x
24649          { <path ~ d=" \g__driver_draw_path_tl " ~ #1 /> }
24650  }
24651      \tl_gclear:N \g__driver_draw_path_tl
24652      \bool_gset_false:N \g__driver_draw_clip_bool
24653  }
24654 \int_new:N \g__driver_path_int
24655 \cs_new_protected:Npn \__driver_draw_stroke:
24656     { \__driver_draw_path:n { style="fill:none" } }
24657 \cs_new_protected:Npn \__driver_draw_closestroke:
24658     {
24659         \__driver_draw_closepath:
24660         \__driver_draw_stroke:
24661     }
24662 \cs_new_protected:Npn \__driver_draw_fill:
24663     { \__driver_draw_path:n { style="stroke:none" } }
24664 \cs_new_protected:Npn \__driver_draw_fillstroke:
24665     { \__driver_draw_path:n { } }
24666 \cs_new_protected:Npn \__driver_draw_clip:
24667     { \bool_gset_true:N \g__driver_draw_clip_bool }
24668 \bool_new:N \g__driver_draw_clip_bool
24669 \cs_new_protected:Npn \__driver_draw_discardpath:
24670     {
24671         \bool_if:NT \g__driver_draw_clip_bool
24672         {
24673             \int_gincr:N \g__driver_clip_path_int
24674             \__driver_draw_literal:x
24675             {
24676                 < clipPath~id = " l3cp \int_use:N \g__driver_clip_path_int " >
24677                     { ?nl }
24678                     <path~d=" \g__driver_draw_path_tl "/> { ?nl }
24679                     < /clipPath >
24680             }
24681             \__driver_draw_scope:x
24682             {
24683                 clip-path =
24684                     "url( \c_hash_str l3cp \int_use:N \g__driver_clip_path_int)"
24685             }

```

```

24686     }
24687     \tl_gclear:N \g__driver_draw_path_tl
24688     \bool_gset_false:N \g__driver_draw_clip_bool
24689 }

```

(End definition for `__driver_draw_path:n` and others.)

`__driver_draw_dash:nn` All of these ideas are properties of scopes in SVG. The only slight complexity is converting the dash array properly (doing any required maths).

```

24690 \cs_new_protected:Npn \__driver_draw_dash:nn #1#2
24691 {
24692     \use:x
24693     {
24694         \__driver_draw_dash_aux:nn
24695         { \clist_map_function:nn {#1} \__driver_draw_dash:n }
24696         { \dim_to_decimal:n {#2} }
24697     }
24698 }
24699 \cs_new:Npn \__driver_draw_dash:n #1
24700 { , \dim_to_decimal_in_bp:n {#1} }
24701 \cs_new_protected:Npn \__driver_draw_dash_aux:nn #1#2
24702 {
24703     \__driver_draw_scope:x
24704     {
24705         stroke-dasharray =
24706         "
24707             \tl_if_empty:oTF { \use_none:n #1 }
24708             { none }
24709             { \use_none:n #1 }
24710         " ~
24711         stroke-offset=" #2 "
24712     }
24713 }
24714 \cs_new_protected:Npn \__driver_draw_linewidth:n #1
24715 { \__driver_scope:x { stroke-width=" \dim_to_decimal:n {#1} " } }
24716 \cs_new_protected:Npn \__driver_draw_miterlimit:n #1
24717 { \__driver_scope:x { stroke-miterlimit=" \fp_eval:n {#1} " } }
24718 \cs_new_protected:Npn \__driver_draw_cap_but:
24719 { \__driver_scope:n { stroke-linecap="but" } }
24720 \cs_new_protected:Npn \__driver_draw_cap_round:
24721 { \__driver_scope:n { stroke-linecap="round" } }
24722 \cs_new_protected:Npn \__driver_draw_cap_rectangle:
24723 { \__driver_scope:n { stroke-linecap="square" } }
24724 \cs_new_protected:Npn \__driver_draw_join_miter:
24725 { \__driver_scope:n { stroke-linejoin="miter" } }
24726 \cs_new_protected:Npn \__driver_draw_join_round:
24727 { \__driver_scope:n { stroke-linejoin="round" } }
24728 \cs_new_protected:Npn \__driver_draw_join_bevel:
24729 { \__driver_scope:n { stroke-linejoin="bevel" } }


```

(End definition for `__driver_draw_dash:nn` and others.)

`__driver_draw_color_cmyk:nnnn` SVG only works with RGB colors, so there is some conversion to do. The values also need to be given as percentages, which means a little more maths.

`__driver_draw_color_cmyk_fill:nnnn`

`__driver_draw_color_cmyk_stroke:nnnn`

`__driver_draw_color_gray:n`

`__driver_draw_color_gray_fill:n`

`__driver_draw_color_gray_stroke:n`

`__driver_draw_color_rgb:nnn`

`__driver_draw_color_rgb_fill:nnn`

`__driver_draw_color_rgb_stroke:nnn`

```

24730 \cs_new_protected:Npn \__driver_draw_color_cmyk_aux:Nn nnnnnn #1#2#3#4#5#6
24731 {
24732     \use:x
24733     {
24734         \__driver_draw_color_rgb_auxii:nnn
24735             { \fp_eval:n { -100 * ( #3 ) * ( 1 - ( #6 ) ) - 1 } }
24736             { \fp_eval:n { -100 * ( #4 ) * ( 1 - ( #6 ) ) + #6 - 1 } }
24737             { \fp_eval:n { -100 * ( #5 ) * ( 1 - ( #6 ) ) + #6 - 1 } }
24738     }
24739     #1 #2
24740 }
24741 \cs_new_protected:Npn \__driver_draw_color_cmyk:nnnn
24742     { \__driver_draw_color_cmyk_aux:Nn nnnnnn \c_true_bool \c_true_bool }
24743 \cs_new_protected:Npn \__driver_draw_color_cmyk_fill:nnnn
24744     { \__driver_draw_color_cmyk_aux:Nn nnnnnn \c_false_bool \c_true_bool }
24745 \cs_new_protected:Npn \__driver_draw_color_cmyk_stroke:nnnn
24746     { \__driver_draw_color_cmyk_aux:Nn nnnnnn \c_true_bool \c_false_bool }
24747 \cs_new_protected:Npn \__driver_draw_color_gray_aux:NNn #1#2#3
24748 {
24749     \use:x
24750     {
24751         \__driver_draw_color_gray_aux:nNN
24752             { \fp_eval:n { 100 * (#3) } }
24753     }
24754     #1 #2
24755 }
24756 \cs_new_protected:Npn \__driver_draw_color_gray_aux:nNN #1
24757     { \__driver_draw_color_rgb_auxii:nnnNN {#1} {#1} {#1} }
24758 \cs_generate_variant:Nn \__driver_draw_color_gray_aux:nNN { x }
24759 \cs_new_protected:Npn \__driver_draw_color_gray:n
24760     { \__driver_draw_color_gray_aux:NNn \c_true_bool \c_true_bool }
24761 \cs_new_protected:Npn \__driver_draw_color_gray_fill:n
24762     { \__driver_draw_color_gray_aux:NNn \c_false_bool \c_true_bool }
24763 \cs_new_protected:Npn \__driver_draw_color_gray_stroke:n
24764     { \__driver_draw_color_gray_aux:NNn \c_true_bool \c_false_bool }
24765 \cs_new_protected:Npn \__driver_draw_color_rgb_auxi:NNnnn #1#2#3#4#5
24766 {
24767     \use:x
24768     {
24769         \__driver_draw_color_rgb_auxii:nnnNN
24770             { \fp_eval:n { 100 * (#3) } }
24771             { \fp_eval:n { 100 * (#4) } }
24772             { \fp_eval:n { 100 * (#5) } }
24773     }
24774     #1 #2
24775 }
24776 \cs_new_protected:Npn \__driver_draw_color_rgb_auxii:nnnNN #1#2#3#4#5
24777 {
24778     \__driver_draw_scope:x
24779     {
24780         \bool_if:NT #4
24781             {
24782                 fill =
24783                     "

```

```

24784     rgb
24785     (
24786         #1 \c_percent_str ,
24787         #2 \c_percent_str ,
24788         #3 \c_percent_str
24789     )
24790     "
24791     \bool_if:NT #5 { ~ }
24792 }
24793 \bool_if:NT #5
24794 {
24795     stroke =
24796     "
24797     rgb
24798     (
24799         #1 \c_percent_str ,
24800         #2 \c_percent_str ,
24801         #3 \c_percent_str
24802     )
24803     "
24804 }
24805 }
24806 }
24807 \cs_new_protected:Npn \__driver_draw_color_rgb:nnn
24808     { \__driver_draw_color_rgb_auxi:NNnnn \c_true_bool \c_true_bool }
24809 \cs_new_protected:Npn \__driver_draw_color_rgb_fill:nnn
24810     { \__driver_draw_color_rgb_auxi:NNnnn \c_false_bool \c_true_bool }
24811 \cs_new_protected:Npn \__driver_draw_color_rgb_stroke:nnn
24812     { \__driver_draw_color_rgb_auxi:NNnnn \c_true_bool \c_false_bool }

(End definition for \__driver_draw_color_cmyk:nnnn and others.)

```

__driver_draw_transformcm:nnnnnn

The first four arguments here are floats (the affine matrix), the last two are a displacement vector. Once again, force evaluation to allow for caching.

```

24813 \cs_new_protected:Npn \__driver_draw_transformcm:nnnnnn #1#2#3#4#5#6
24814 {
24815     \__driver_scope:x
24816     {
24817         transform =
24818         "
24819         matrix
24820         (
24821             \fp_eval:n {#1} , \fp_eval:n {#2} ,
24822             \fp_eval:n {#3} , \fp_eval:n {#4} ,
24823             \dim_to_decimal:n {#5} , \dim_to_decimal:n {#6}
24824         )
24825         "
24826     }
24827 }

(End definition for \__driver_draw_transformcm:nnnnnn.)

```

__driver_draw_hbox:Nnnnnnn

No special savings can be made here: simply displace the box inside a scope. As there is nothing to re-box, just make the box passed of zero size.

```

24828 \cs_new_protected:Npn \__driver_draw_hbox:Nnnnnnn #1#2#3#4#5#6#7
24829   {
24830     \__driver_scope_begin:
24831     \__driver_draw_transformcm:nnnnnnn {#2} {#3} {#4} {#5} {#6} {#7}
24832     \__driver_literal:n
24833     {
24834       < g~
24835         stroke="none"~
24836         transform="scale(-1,1)~translate({?x},{?y})~scale(-1,-1)"
24837       >
24838     }
24839   \group_begin:
24840     \box_set_wd:Nn #1 { Opt }
24841     \box_set_ht:Nn #1 { Opt }
24842     \box_set_dp:Nn #1 { Opt }
24843     \box_use:N #1
24844   \group_end:
24845   \__driver_literal:n { </g> }
24846   \__driver_scope_end:
24847 }

(End definition for \__driver_draw_hbox:Nnnnnnn.)
```

24848 </dvisvgm>

24849 </initex | package>

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	
!	<i>186</i>
\!	<i>7697</i>
\"	<i>6665, 6668, 23109</i>
\#	<i>6, 3935, 3967, 6665, 8958</i>
\\$	<i>3934, 6665, 6668, 17907</i>
\%	<i>3936, 6665, 8960</i>
\&	<i>3927, 6665, 6668, 7697, 7698</i>
\&&	<i>186</i>
\'	<i>23109</i>
\(<i>23275</i>
\)	<i>23275</i>
*	<i>186</i>
*	<i>3602, 3625, 6838, 6840, 6844, 6852</i>
**	<i>186</i>
+	<i>186, 186</i>
\,	<i>9672</i>
-	<i>186, 186</i>
\-	<i>205, 286</i>
\.	<i>23109</i>
/	<i>186</i>
\/	<i>285</i>
\:	<i>3933</i>
\:::	<i>33, 299, 314, 2064, 2065, 2066, 2067, 2068, 2069, 2070, 2072, 2074, 2081, 2086, 2092, 2211, 2212, 2213, 2214, 2215, 2216, 2217, 2218, 2219, 2220, 2221, 2222, 2223, 2224, 2225, 2226, 2227, 2228, 2229, 2230, 2231, 2232, 2233, 2234, 2235, 2236, 2237, 2238, 2239, 2241, 2246, 2248, 2253, 2258, 2295, 2296, 2297, 2298, 2299</i>
\:::N	<i>33, 2068, 2221, 2227, 2228, 2232, 2233, 2234, 2298</i>
\:::V	<i>33, 2086, 2217</i>
\:::V_unbraced	<i>2240</i>
\:::c	<i>33, 2070, 2212, 2220, 2222, 2229, 2237, 2238</i>
\:::error	<i>229, 22017</i>
\:::f	<i>33, 2074, 2213, 2214, 2215, 2219, 2297</i>
\:::f_unbraced	<i>2240</i>
\:::n	<i>33, 2067, 2212, 2215, 2216, 2217, 2223, 2227, 2229, 2230, 2233, 2235, 2236, 2238, 2295, 2298</i>
\:::o	<i>33, 2072, 2213, 2216, 2218, 2219, 2220, 2224, 2225, 2227, 2228, 2230, 2231, 2234, 2236, 2239, 2296</i>
\:::o_unbraced	<i>2240, 2295, 2296, 2297, 2298</i>
\:::p	<i>33, 299, 2069</i>
\:::v	<i>33, 2086</i>
\:::v_unbraced	<i>2240</i>
\:::x	<i>33, 2081, 2211, 2221, 2222, 2223, 2224, 2235, 2236, 2237, 2238, 2239</i>
\:::x_unbraced	<i>2240, 2299</i>
<	<i>186</i>
=	<i>186</i>
>	<i>186</i>
?	<i>186</i>
?:	<i>185</i>
\\"	<i>2039, 3929, 6665, 7632, 7650, 7652, 7657, 7658, 7672, 7673, 7676, 7677, 7774, 7864, 7872, 8074, 8082, 8089, 8101, 8102, 8127, 8128, 8135, 8161, 8163, 8164, 8207, 8208, 8209, 8210, 8211, 8410, 8414, 8419, 8963, 9191, 9198, 10562, 10563, 10569, 10588, 10601, 10607, 11059, 11062, 11063, 11064, 11071, 11074, 11075, 16460, 16463, 16488, 16489, 16496, 16497, 16870, 17013, 18398, 18438, 18507, 18508, 18509, 18530, 19751, 19754, 19757, 19760, 19764, 19767, 19802, 19811, 19815, 19820, 19840, 19842, 19843, 19844, 19847, 19850, 19854, 19856, 19860, 19862, 19868, 19870, 19874, 19876, 19880, 19885, 19886, 19928, 19930, 19935, 19937, 19943, 19948, 19949, 19953, 19957, 19964, 19965, 19969, 19977, 20018, 21391, 21392, 21393, 21394</i>
\{	<i>4, 3930, 6665, 7695, 8208, 8410, 8414, 8415, 8957, 17254, 19760, 19767, 19815, 19850, 19886, 19965, 19969, 23287</i>
\}	<i>5, 3931, 6665, 7696, 8208, 8410, 8414, 8415, 8959, 19767, 19851, 19886, 19965, 19969, 23287</i>
\^	<i>7, 10, 103, 194, 195, 196, 197, 1727, 2320, 2324, 2737, 3932, 6665, 6668, 6670, 6676, 6738, 7711, 8887, 8923, 14585, 16571, 16644, 16645, 16648, 16649, 17200, 17205, 17206, 17207, 17208, 17211, 17222, 17259, 17314, 17316, 17318, 17320, 17322, 17324, 17906, 19375,</i>

\^	186	\attributedef	862
_	3938, 6665, 6668	\automatichyphenpenalty	863
\`	23109	\autospacing	1123
\ 	185	\autoxspacing	1124
\~	2758, 3937, 6665, 6668, 8961, 17244, 17248, 17254, 23109		
A		B	
\A	3603, 3626	\badness	298
\AA	23091	\baselineskip	299
\aa	23091	\batchmode	300
\above	287	\begin	197, 200, 12757, 23641, 23644, 24114, 24117
\abovedisplayshortskip	288	begin internal commands:	
\abovedisplayskip	289	__regex_begin	19494
\abovewithdelims	290	\begincsname	864
abs	186	\begingroup	13, 20, 38, 42, 48, 73, 142, 162, 274, 301
\accent	291	\beginL	609
acos	188	\beginR	610
acosd	188	\belowdisplayshortskip	302
acot	189	\belowdisplayskip	303
acotd	189	\binoppenalty	304
acsc	188	\bodydir	937
acscd	188	bool commands:	
\adjdemerits	292	\bool_do_until:Nn	95, 95, 5911
\adjustspacing	950	\bool_do_until:nn	95, 95, 5917
\advance	168, 184, 293	\bool_do_while:Nn	95, 95, 5911
\AE	23092	\bool_do_while:nn	95, 95, 5917
\ae	23092	.bool_gset:N	164, 10108
\afterassignment	294	\bool_gset:Nn	92, 5700, 5743
\aftergroup	295	\bool_gset_eq:NN	
\alignmark	859 92, 5692, 5733, 17191, 18970	
\aligntab	860	\bool_gset_false:N	
asec	188 91, 5680, 5723, 8373, 8394, 18934, 23901, 24204, 24222, 24247, 24270, 24286, 24652, 24688	
asecd	188	.bool_gset_inverse:N	164, 10116
asin	188	\bool_gset_true:N	
asind	188 91, 5680, 5718, 8338, 18977, 23899, 24202, 24273, 24667	
assert commands:		\bool_if:NTF ... 92, 92, 238, 1929, 5750, 5759, 5760, 5906, 5908, 5912, 5914, 7750, 7755, 7760, 7765, 7770, 8202, 8358, 8391, 9152, 9159, 9887, 10080, 10089, 10255, 10280, 10294, 10311, 10345, 10373, 10392, 10394, 10399, 10406, 10430, 10476, 18032, 18041, 18400, 18475, 18480, 18496, 18523, 18635, 18859, 18867, 20000, 20005, 20873, 21275, 23614, 23648, 23766, 23792, 23912, 23917, 23922, 24213, 24217, 24234, 24238, 24242, 24254, 24261, 24265, 24277, 24281, 24626, 24671, 24780, 24791, 24793	
\assert_int:n	18686, 19570		
atan	189		
atand	189		
\AtBeginDocument	491, 8613, 23584, 23722, 23799, 24133, 24528		
\atop	296		
\atopwithdelims	297		
\attribute	861		

\bool_if:nTF 92, 94,
 94, 95, 95, 95, 95, 920, 937, 5770,
 5785, 5873, 5879, 5880, 5888, 5894,
 5896, 5919, 5928, 5932, 5941, 22094
 \bool_if_exist:NTF
 92, 92, 5764, 5781, 9908, 9924
 \bool_if_exist_p:N 92, 92, 5781
 \bool_if_p:N 92, 92, 5750
 \bool_if_p:n 94, 94, 414, 5701, 5703,
 5741, 5746, 5785, 5793, 5898, 5901
 \bool_lazy_all:nTF
 93, 94, 94, 94, 94, 5868, 18274
 \bool_lazy_all_p:n 94, 94, 5868
 \bool_lazy_and:nnTF 93, 94, 94, 94,
 94, 5877, 21451, 22754, 22804, 23257
 \bool_lazy_and_p:nn . 93, 94, 94, 5877
 \bool_lazy_any:nTF
 93, 94, 94, 94, 94, 5883, 22586
 \bool_lazy_any_p:n
 93, 94, 94, 5883, 22757
 \bool_lazy_or:nnTF 93, 94,
 94, 94, 94, 5892, 22662, 22912, 23235
 \bool_lazy_or_p:nn 94, 94, 5892, 23260
 \bool_log:N 92, 92, 5772
 \bool_log:n 92, 92, 5772
 \bool_new:N 91, 91,
 5678, 5777, 5778, 5779, 5780, 7575,
 8336, 8918, 9802, 9803, 9806, 9807,
 9811, 9908, 9924, 17054, 17476,
 18915, 18916, 18918, 18919, 18920,
 20592, 23902, 24205, 24274, 24668
 \bool_not_p:n 94, 94, 5898
 .bool_set:N 164, 10108
 \bool_set:Nn 92, 92, 5700, 5738
 \bool_set_eq:NN
 92, 92, 5692, 5728, 17186, 19123
 \bool_set_false:N 91, 91,
 253, 5680, 5713, 8996, 9136, 9144,
 9154, 9161, 9828, 10265, 10290,
 10302, 10322, 10331, 10380, 18007,
 18180, 18894, 18944, 18957, 19000,
 19064, 20869, 21273, 23622, 23746
 .bool_set_inverse:N 164, 10116
 \bool_set_true:N
 91, 91, 267, 5680, 5708, 9123,
 9823, 10263, 10288, 10304, 10320,
 10326, 10387, 18012, 18184, 18891,
 19062, 19122, 20887, 20902, 20933
 \bool_show:N 92, 92, 408, 5762, 5773
 \bool_show:n 92, 92, 5762, 5775
 \bool_until_do:Nn 95, 95, 5905
 \bool_until_do:nn 95, 95, 5917
 \bool_while_do:Nn 95, 95, 5905
 \bool_while_do:nn 95, 95, 5917
 \bool_xor_p:nn 94, 94, 5899
 \c_false_bool
 20, 91, 285, 309, 406, 410,
 411, 1460, 1500, 1543, 1548, 1592,
 1608, 1874, 1881, 2366, 5678, 5683,
 5687, 5716, 5726, 5841, 5843, 5848,
 5902, 17695, 17708, 17923, 17958,
 18219, 18354, 18370, 18417, 18557,
 19674, 21431, 21483, 21498, 24744,
 24746, 24762, 24764, 24810, 24812
 \g_tmipa_bool 92, 5777
 \l_tmipa_bool 92, 5777
 \g_tmppb_bool 92, 5777
 \l_tmppb_bool 92, 5777
 \c_true_bool
 20, 91, 285, 285, 327, 406,
 410, 411, 1500, 1592, 1607, 1892,
 5681, 5685, 5711, 5721, 5752, 5842,
 5844, 5850, 5903, 17060, 17188,
 17608, 17662, 17706, 17889, 17914,
 17957, 17964, 18352, 18362, 18415,
 18621, 18797, 18808, 18823, 18950,
 18976, 19504, 19580, 19638, 21438,
 21446, 21458, 21465, 21474, 21487,
 21494, 24742, 24744, 24746, 24760,
 24762, 24764, 24808, 24810, 24812
 bool internal commands:
 __bool_!:_Nw 5823
 __bool_&_0:w 5847
 __bool_&_1:w 5845
 __bool_(:_Nw 5825
 __bool_)_0:w 5841
 __bool_)_1:w 5841
 __bool_choose:NNN 5827, 5831, 5832
 __bool_eval_skip_to_end_auxi:_Nw
 5847
 __bool_eval_skip_to_end_-
 auxii:_Nw 5847
 __bool_eval_skip_to_end_-
 auxiii:_Nw 5847
 __bool_get_next:NN
 5811, 5813, 5824, 5828, 5845, 5846
 __bool_if_left_parentheses:wwwn
 5793
 __bool_if_or:wwwn 5793
 __bool_if_parse:NNNww 410, 5800, 5809
 __bool_if_right_parentheses:wwwn
 5793
 __bool_lazy_all:n 5868
 __bool_lazy_any:n 5883
 __bool_p:_Nw 5830
 __bool_S_0:w 5841
 __bool_S_1:w 5841
 __bool_to_str:n 5762

__bool_|_0:w 5845
 __bool_|_1:w 5847
 \botmark 305
 \botmarks 611
 \box 306
 box commands:
 \box_autosize_to_wd_and_ht:Nnn ..
 215, 215, 20510
 \box_autosize_to_wd_and_ht_plus_-
 dp:Nnn 215, 215, 20510
 \box_clear:N 208,
 208, 20033, 20040, 20628, 20688, 20735
 \box_clear:N 208
 \box_clear_new:N 208, 208, 20039
 \box_clip:N . 226, 226, 227, 227, 21563
 \box_dp:N 209, 209,
 11506, 20061, 20068, 20291, 20409,
 20499, 20514, 20756, 20757, 20827,
 20829, 20846, 20860, 21014, 21032,
 21345, 21574, 21581, 21586, 21723,
 23534, 23536, 23685, 23687, 24089,
 24091, 24452, 24454, 24456, 24458
 \box_gclear:N 208, 20033, 20042
 \box_gclear_new:N 208, 20039
 \box_gset_eq:NN 208, 20036, 20045
 \box_gset_eq_clear:NN 208, 208, 20051
 \box_gset_to_last:N 210, 20109
 \box_ht:N 209, 209, 11505,
 20061, 20070, 20290, 20408, 20498,
 20511, 20514, 20683, 20730, 20758,
 20759, 20823, 20825, 20846, 20853,
 21013, 21031, 21343, 21591, 21599,
 21604, 21720, 21722, 23536, 23655,
 23687, 23850, 24091, 24456, 24458
 \box_if_empty:NTF
 210, 210, 20103, 20106, 20107
 \box_if_empty_p:N . 210, 210, 20103
 \box_if_exist:NTF
 . 208, 208, 20040, 20042, 20057, 20140
 \box_if_exist_p:N . 208, 208, 20057
 \box_if_horizontal:NTF
 210, 210, 20091, 20096, 20097
 \box_if_horizontal_p:N 210, 210, 20091
 \box_if_vertical:NTF
 210, 210, 20091, 20100, 20101
 \box_if_vertical_p:N 210, 210, 20091
 \box_log:N 211, 211, 20126
 \box_log:Nnn 211, 211, 20126
 \box_move_down:nn 209, 893, 20080,
 21578, 21586, 21624, 21631, 21702
 \box_move_left:nn 209, 20080
 \box_move_right:nn . 209, 209, 20080
 \box_move_up:nn
 ... 209, 209, 20080, 21052, 21340,
 21595, 21604, 21638, 21651, 24047
 \box_new:N 208, 208, 208,
 20025, 20115, 20116, 20117, 20118,
 20119, 20277, 20570, 20635, 24055
 \box_resize:Nnn . 20565, 21552, 21553
 \box_resize_to_ht:Nn 215, 215, 20428
 \box_resize_to_ht_plus_dp:Nn ...
 215, 215, 20428
 \box_resize_to_wd:Nn 216, 216, 20428
 \box_resize_to_wd_and_ht:Nnn ...
 216, 216, 20428
 \box_resize_to_wd_and_ht_plus_-
 dp:Nnn 216, 216, 20390,
 20565, 20566, 21552, 21553, 21833
 \box_rotate:Nn 216, 216, 20278, 21695
 \box_scale:Nnn 216, 216, 20486, 21855
 \box_set_dp:Nn . 210, 210, 893,
 20067, 20317, 20540, 20543, 21014,
 21032, 21344, 21581, 21589, 21627,
 21632, 21707, 24052, 24425, 24842
 \box_set_eq:NN
 208, 208, 20034, 20045,
 20745, 21034, 21348, 21609, 21656
 \box_set_eq_clear:NN 208, 208, 20051
 \box_set_ht:Nn 210, 210,
 20067, 20316, 20539, 20544, 21013,
 21031, 21342, 21598, 21607, 21641,
 21654, 21705, 24051, 24424, 24841
 \box_set_to_last:N .. 210, 210, 20109
 \box_set_wd:Nn . 210, 210, 20067,
 20318, 20556, 21015, 21033, 21346,
 21708, 23546, 24050, 24423, 24840
 \box_show:N 211, 211, 20120
 \box_show:Nnn 211, 211, 20120
 \box_trim:Nnnnn 227, 227, 21566
 \box_use:N 209,
 209, 20076, 20305, 20320, 20551,
 20560, 21049, 21052, 21130, 21267,
 21337, 21340, 21571, 21579, 21587,
 21596, 21605, 21617, 21625, 21631,
 21639, 21652, 21703, 21710, 23539,
 23564, 23578, 23690, 23704, 23716,
 24047, 24053, 24095, 24109, 24127,
 24426, 24486, 24503, 24522, 24843
 \box_use_clear:N . 209, 209, 20076
 \box_viewport:Nnnnn . 227, 227, 21612
 \box_wd:N 210, 210,
 11504, 20061, 20072, 20292, 20410,
 20500, 20520, 20760, 20761, 20825,
 20829, 20835, 20840, 20982, 21015,
 21033, 21050, 21338, 21347, 21618,
 21722, 21727, 21892, 21899, 23535,

23541, 23656, 23686, 23692, 23851,
 24090, 24097, 24453, 24455, 24490
`\c_empty_box` 210, 211, 20034, 20036, 20115, 21127
`\g_tmpa_box` 211, 20116
`\l_tmpa_box` 211, 20116
`\g_tmpb_box` 211, 20116
`\l_tmpb_box` 211, 20116
`\c_void_box` 208
 box internal commands:
`\l_box_angle_fp` 20266, 20282, 20283, 20284, 20313
`\l_box_autosize:Nnnn` 20510
`\l_box_bottom_dim` 20269,
 20291, 20348, 20352, 20357, 20363,
 20368, 20372, 20381, 20383, 20400,
 20409, 20418, 20451, 20499, 20505
`\l_box_bottom_new_dim`
 20273, 20317, 20349, 20360, 20371,
 20382, 20417, 20504, 20540, 20544
`\l_box_cos_fp` 20267,
 20284, 20296, 20301, 20328, 20340
`\l_box_internal_box` 20277, 20305,
 20306, 20312, 20316, 20317, 20318,
 20320, 20530, 20539, 20540, 20543,
 20544, 20551, 20556, 20560, 21568,
 21576, 21579, 21581, 21584, 21587,
 21589, 21591, 21593, 21596, 21598,
 21599, 21602, 21604, 21605, 21607,
 21609, 21614, 21622, 21625, 21627,
 21630, 21631, 21632, 21636, 21639,
 21641, 21649, 21652, 21654, 21656
`\l_box_left_dim` ... 20269, 20293,
 20348, 20350, 20359, 20363, 20368,
 20374, 20379, 20383, 20411, 20501
`\l_box_left_new_dim` 20273, 20308,
 20319, 20351, 20362, 20373, 20384
`\l_box_log:Nnn` 20126
`\l_box_resize:N` 20390, 20439, 20454, 20466, 20482
`\l_box_resize:NNN` 20390
`\l_box_resize_common:N` 20421, 20508, 20528
`\l_box_resize_set_corners:N` 20390, 20432, 20447, 20462, 20474
`\l_box_right_dim` ... 20269, 20292,
 20346, 20352, 20357, 20361, 20370,
 20372, 20381, 20385, 20396, 20410,
 20416, 20464, 20476, 20500, 20507
`\l_box_right_new_dim` ... 20273,
 20319, 20353, 20364, 20375, 20386,
 20415, 20506, 20548, 20550, 20556
`\l_box_rotate:N` 20278
`\l_box_rotate_quadrant_four:`
 20278, 20377
`\l_box_rotate_quadrant_one:`
 20278, 20344
`\l_box_rotate_quadrant_three:`
 20278, 20366
`\l_box_rotate_quadrant_two:`
 20278, 20355
`\l_box_rotate_x:nnN`
 20278, 20322, 20350, 20352, 20361,
 20363, 20372, 20374, 20383, 20385
`\l_box_rotate_y:nnN`
 20278, 20333, 20346, 20348, 20357,
 20359, 20368, 20370, 20379, 20381
`\l_box_scale_aux:N` 20486, 20525
`\l_box_scale_x_fp` 20388,
 20395, 20416, 20438, 20453, 20463,
 20465, 20475, 20490, 20507, 20520,
 20522, 20523, 20524, 20534, 20546
`\l_box_scale_y_fp`
 20388, 20397, 20418, 20420,
 20433, 20438, 20448, 20453, 20465,
 20477, 20491, 20503, 20505, 20521,
 20522, 20523, 20524, 20535, 20537
`\l_box_show:Nnnn` . 20124, 20134, 20138
`\l_box_sin_fp`
 ... 20267, 20283, 20294, 20329, 20339
`\l_box_top_dim` 20269, 20290, 20346,
 20350, 20359, 20361, 20370, 20374,
 20379, 20385, 20400, 20408, 20420,
 20436, 20451, 20480, 20498, 20503
`\l_box_top_new_dim`
 20273, 20316, 20347, 20358, 20369,
 20380, 20419, 20502, 20539, 20543
`\boxdir` 938
`\boxmaxdepth` 307
`bp` 191
`\brokenpenalty` 308

C

`\c` 23109
`\catcode` 4, 5, 6, 7, 10, 212, 213, 214, 215,
 216, 217, 218, 219, 220, 225, 226,
 227, 228, 229, 230, 231, 232, 233, 309
 catcode commands:
`\c_catcode_active_tl`
 111, 443, 443, 6851, 6911
`\c_catcode_letter_token`
 111, 443, 6833, 6901, 16750
`\c_catcode_other_space_tl` ... 144,
 504, 8897, 8932, 8964, 9028, 9116, 9181
`\c_catcode_other_token`
 111, 443, 6833, 6906, 16748
`\catcodetable` 865

cc 191
 ceil 187
 \char 310, 7016
 char commands:
 \l_char_active_seq
 111, 138, 145, 6663, 8458
 \char_generate:nn
 108, 108, 121, 833, 6689, 8305,
 8897, 17342, 18284, 22485, 22932,
 22933, 22945, 22946, 23066, 23067
 \char_gset_active_eq:NN 107, 6669
 \char_gset_active_eq:nN 107, 6669
 \char_set_active_eq:NN
 107, 107, 6669, 8461
 \char_set_active_eq:nN 107, 107, 6669
 \char_set_catcode:nn 109,
 109, 242, 243, 244, 245, 246, 247,
 248, 249, 250, 6554, 6564, 6566,
 6568, 6570, 6572, 6574, 6576, 6578,
 6580, 6582, 6584, 6586, 6588, 6590,
 6592, 6594, 6596, 6598, 6600, 6602,
 6604, 6606, 6608, 6610, 6612, 6614,
 6616, 6618, 6620, 6622, 6624, 6626
 \char_set_catcode_active:N
 108, 6563,
 6670, 6738, 6852, 7698, 16571, 19375
 \char_set_catcode_active:n
 109, 6595, 6794, 9671, 9672
 \char_set_catcode_alignment:N
 108, 6563, 6840, 19424
 \char_set_catcode_alignment:n
 109, 260, 6595, 6773
 \char_set_catcode_comment:N 108, 6563
 \char_set_catcode_comment:n 109, 6595
 \char_set_catcode_end_line:N
 108, 6563
 \char_set_catcode_end_line:n
 109, 6595
 \char_set_catcode_escape:N 108, 6563
 \char_set_catcode_escape:n 109, 6595
 \char_set_catcode_group_begin:N
 108, 6563, 16644, 16648, 19378
 \char_set_catcode_group_begin:n
 109, 6595, 6766
 \char_set_catcode_group_end:N
 108, 6563, 16645, 16649, 19392
 \char_set_catcode_group_end:n
 109, 6595, 6768
 \char_set_catcode_ignore:N 108, 6563
 \char_set_catcode_ignore:n
 109, 257, 258, 6595
 \char_set_catcode_invalid:N 108, 6563
 \char_set_catcode_invalid:n 109, 6595

 \char_set_catcode_letter:N
 108, 108, 6563, 12873, 12874, 19401
 \char_set_catcode_letter:n
 109, 109, 261, 263, 6595, 6790
 \char_set_catcode_math_subscript:N
 108, 6563, 6844, 19389
 \char_set_catcode_math_subscript:n
 109, 6595, 6785
 \char_set_catcode_math_superscript:N
 108, 6563, 19427
 \char_set_catcode_math_superscript:n
 109, 262, 6595, 6783
 \char_set_catcode_math_toggle:N
 108, 6563, 6838, 19404
 \char_set_catcode_math_toggle:n
 109, 6595, 6771
 \char_set_catcode_other:N
 108, 6563, 16783, 19407
 \char_set_catcode_other:n
 109, 259, 264, 6595, 6741, 6792
 \char_set_catcode_parameter:N
 108, 6563, 19410
 \char_set_catcode_parameter:n
 109, 6595, 6781
 \char_set_catcode_space:N 108, 6563
 \char_set_catcode_space:n
 109, 265, 6595, 6788
 \char_set_lccode:nn
 109, 109, 2743, 2760, 6627,
 6676, 6798, 6799, 7695, 7696, 7697
 \char_set_mathcode:nn 110, 110, 6627
 \char_set_sfcode:nn 110, 110, 6627
 \char_set_uccode:nn 110, 110, 6627
 \char_show_value_catcode:n
 109, 109, 6554
 \char_show_value_lccode:n
 110, 110, 6627
 \char_show_value_mathcode:n
 110, 110, 6627
 \char_show_value_sfcode:n
 111, 111, 6627
 \char_show_value_uccode:n
 110, 110, 6627
 \l_char_special_seq 111, 6663
 \char_value_catcode:n 109,
 109, 242, 243, 244, 245, 246, 247,
 248, 249, 250, 2755, 2785, 6554, 22487
 \char_value_lccode:n 110, 110, 6627
 \char_value_mathcode:n 110, 110, 6627
 \char_value_sfcode:n 111, 111, 6627
 \char_value_uccode:n 110, 110, 6627
 char internal commands:
 __char_generate:nn
 121, 121, 6689, 22895,

22897, 22938, 22940, 22951, 22953
 __char_generate_aux:nn 6689
 __char_generate_aux:nw 6689
 __char_generate_aux:w 6691, 6702
 __char_generate_invalid_-
 catcode: 6689
 \c__char_max_int 6689
 __char_tmp:n
 6796, 6808, 6811, 6813, 6816
 __char_tmp:nN 6671, 6682, 6683
 \l__char_tmp_tl 6689
\chardef 222, 235, 311
chk internal commands:
 __chk_if_exist_cs:N 22,
 22, 1574, 1579, 1584, 1589, 1779, 2328
 __chk_if_exist_var:N
 22, 22, 1767, 2641, 2663,
 2664, 2669, 2670, 2677, 2678, 2679,
 2684, 2685, 2686, 5536, 5710, 5715,
 5720, 5725, 5730, 5735, 5740, 5745
 __chk_if_free_cs:N 22, 22, 441, 465,
 1744, 1793, 1841, 2518, 2524, 2529,
 4093, 4688, 4707, 5424, 6834, 6836,
 6846, 7306, 9232, 9488, 9585, 20028
 __chk_if_free_msg:nn 7583, 7606
 __chk_log:n 22,
 22, 22, 1699, 1761, 2475, 7600, 9991
 __chk_resume_log:
 22, 22, 22, 288, 288, 1699, 20643
 __chk_suspend_log:
 22, 22, 22, 288, 288, 1699, 20636
choice commands:
 .choice: 164, 10124
choices commands:
 .choices:nn 164, 10126
\cite 23280
\leaders 312
\clearmarks 866
clist commands:
 \clist_clear:N
 98, 98, 6016, 6033, 6229, 10247, 10272
 \clist_clear:N 98
 \clist_clear_new:N 98, 98, 6020
 \clist_concat:NNN
 99, 99, 6060, 6113, 6126
 \clist_const:Nn 98, 98, 6013
 \clist_count:N
 103, 103, 105, 6409, 6438, 6470, 21670
 \clist_count:n 103, 6409, 6501, 21661
 \clist_gclear:N 98, 6016, 6035
 \clist_gclear_new:N 98, 6020
 \clist_gconcat:NNN 99, 6060, 6115, 6128
 \clist_get:NN 104, 104, 6138
 \clist_get:NNTF
 104, 104, 6175, 6184, 6185
 \clist_gpop:NN 104, 104, 6149
 \clist_gpop:NNTF
 105, 105, 6175, 6203, 6204
 \clist_gpush:Nn 105, 6206
 \clist_gput_left:Nn
 99, 6112, 6214, 6215,
 6216, 6217, 6218, 6219, 6220, 6221
 \clist_gput_right:Nn 100, 6125
 \clist_gremove_all:Nn 100, 6239
 \clist_gremove_duplicates:N 100, 6223
 \clist_greverse:N 100, 6270
 .clist_gset:N 164, 10136
 \clist_gset:Nn 99, 419, 6106
 \clist_gset_eq:NN 99, 6024, 6226
 \clist_gset_from_seq:NN 99, 6032
 \clist_gsort:Nn 101, 6288, 16239
 \clist_if_empty:NTF
 101, 101, 6069, 6256, 6288, 6333,
 6363, 6383, 6531, 10012, 16247, 21669
 \clist_if_empty:nTF
 101, 101, 6292, 6537
 \clist_if_empty_p:N .. 101, 101, 6288
 \clist_if_empty_p:n .. 101, 101, 6292
 \clist_if_exist:NTF
 99, 99, 6075, 6436, 6531, 8571, 8598
 \clist_if_exist_p:N 99, 99, 6075
 \clist_if_in:NnTF 101,
 101, 6232, 6306, 6322, 6323, 6324, 6325
 \clist_if_in:nnTF
 101, 6306, 6328, 6329, 10932
 \clist_item:Nn
 105, 105, 433, 895, 6467, 21670
 \clist_item:nn .. 105, 895, 6498, 21665
 \clist_log:N 106, 106, 6542
 \clist_log:n 106, 106, 6542
 \clist_map_break:
 102, 102, 6337, 6342,
 6351, 6355, 6371, 6389, 6405, 10445
 \clist_map_break:n
 103, 103, 6405, 10388, 16250
 \clist_map_function:NN 84, 98, 102,
 102, 102, 4120, 4130, 6331, 6414, 6532
 \clist_map_function:Nn 429
 \clist_map_function:nN
 102, 430, 4125,
 4135, 6347, 6539, 10486, 23931, 24293
 \clist_map_function:nn 24695
 \clist_map_inline:Nn
 ... 102, 102, 102, 428, 733, 6230,
 6361, 8600, 8615, 10383, 10436, 16250
 \clist_map_inline:nn 102,
 6361, 9976, 10043, 21426, 23417, 23426

```

\clist_map_variable:NNn  102, 102, 6381
\clist_map_variable:nNn . . . . . 102, 6381
\clist_new:N . 98, 98, 98, 418, 6011,
   6222, 6547, 6548, 6549, 6550, 9799
\clist_pop:NN . . . . . 104, 104, 6149
\clist_pop:NNTF . . . . .
   . . . . . 105, 105, 6175, 6200, 6201
\clist_push:Nn . . . . . 105, 105, 6206
\clist_put_left:Nn . . . . .
   . . . . . 99, 99, 6112, 6206, 6207,
   6208, 6209, 6210, 6211, 6212, 6213
\clist_put_right:Nn . . . . .
   . . . . . 100, 100, 6125, 6233, 10473
\clist_rand_item:N . 227, 227, 21660
\clist_rand_item:n 227, 227, 231, 21660
\clist_remove_all:Nn . 100, 100, 6239
\clist_remove_duplicates:N . . .
   . . . . . 100, 100, 6223
\clist_reverse:N . . . . . 100, 100, 6270
\clist_reverse:n . . .
   . . . . . 100, 100, 426, 426, 6271, 6273, 6276
\clist_set:N . . . . . 164, 10136
\clist_set:Nn . . . . .
   . . . . . 99, 99, 419, 6106, 6113, 6115,
   6126, 6128, 6312, 6377, 6394, 10011
\clist_set_eq:NN . . .
   . . . . . 99, 99, 6024, 6224, 10018, 10368
\clist_set_from_seq:NN . . . . . 99, 99, 6032
\clist_show:N . . . . .
   . . . . . 105, 105, 106, 434, 6528, 6543
\clist_show:n . . . . .
   . . . . . 105, 105, 106, 434, 6528, 6545
\clist_sort:Nn . 101, 101, 6288, 16239
\clist_use:Nn . . . . . 104, 104, 6434
\clist_use:Nnnn . . . . . 103, 103, 373, 6434
\clist_wrap_item:n . . . . . 734, 734, 734
\c_empty_clist . . . . .
   . . . . . 106, 6008, 6140, 6155, 6177, 6193
\l_foo_clist . . . . . 194
\g_tmptpa_clist . . . . . 106, 6547
\l_tmptpa_clist . . . . . 106, 6547
\g_tmptpb_clist . . . . . 106, 6547
\l_tmptpb_clist . . . . . 106, 6547
clist internal commands:
  \__clist_concat:NNNN . . . . . 6060
  \__clist_count:n . . . . . 6409
  \__clist_count:w . . . . . 6409
  \__clist_get:wN . . . . . 6138, 6180
  \__clist_if_empty_n:w . . . . . 6292
  \__clist_if_empty_n:wNw . . . . . 6292
  \__clist_if_in_return:nn . . . . . 6306
\l__clist_internal_clist . . . .
   . . . . . 6009, 6118, 6119, 6131, 6132,
   6312, 6313, 6377, 6378, 6394, 6395
\l__clist_internal_remove_clist . .
   . . . . . 6222, 6229, 6232, 6233, 6235
\__clist_item:nnnN . . . . . 6467, 6500
\__clist_item_n:nw . . . . . 6498
\__clist_item_n_end:n . . . . . 6498
\__clist_item_N_loop:nw . . . . . 6467
\__clist_item_n_loop:nw . . . . . 6498
\__clist_item_n_strip:w . . . . . 6498
\__clist_map_function:Nw . . .
   . . . . . 428, 6331, 6368
\__clist_map_function_n:Nn . . . . . 428, 6347
\__clist_map_unbrace:Nw . . . . . 428, 6347
\__clist_map_variable:Nnw . . . . . 6381
\__clist_pop:NNN . . . . . 6149
\__clist_pop:wN . . . . . 6149
\__clist_pop:wwNNN . . . . . 423, 6149, 6196
\__clist_pop_TF:NNN . . . . . 6175
\__clist_put_left:NNNn . . . . . 6112
\__clist_put_right:NNNn . . . . . 6125
\__clist_rand_item:nn . . . . . 21660
\__clist_remove_all: . . . . . 6239
\__clist_remove_all:NNn . . . . . 6239
\__clist_remove_all:w . . . . . 425, 425, 6239
\__clist_remove_duplicates:NN . . . . . 6223
\__clist_reverse:wwNww . . .
   . . . . . 426, 426, 426, 426, 6276
\__clist_reverse_end:ww . . . . . 426, 6276
\__clist_set_from_seq:NNNN . . . . . 6032
\__clist_set_from_seq:w . . . . . 6032
\__clist_tmp:w . . . . . 425,
   425, 425, 6010, 6245, 6266, 6317, 6319
\__clist_trim_spaces:n . . .
   . . . . . 6014, 6085, 6107, 6109
\__clist_trim_spaces:nn . . . . . 421, 6085
\__clist_trim_spaces_generic:nn . .
   . . . . . 421, 6079
\__clist_trim_spaces_generic:nw . .
   . . . . . 428, 6079, 6087, 6097, 6102, 6349, 6357
\__clist_use:nwwn . . . . . 6434
\__clist_use:nwwwnwn . . . . . 431, 6434
\__clist_use:wwn . . . . . 6434
\__clist_wrap_item:n . . . . . 6032, 16253
\closein . . . . . 313
\closeout . . . . . 314
\clubpenalties . . . . . 612
\clubpenalty . . . . . 315
cm . . . . . 191
code commands:
  .code:n . . . . . 164, 10134
coffin commands:
  \coffin_attach:NnnNnnnn . . .
   . . . . . 219, 219, 886, 21009
  \coffin_attach_mark:NnnNnnnn . .
   . . . . . 21009, 21197, 21218, 21234

```

```

\coffin_clear:N .... 218, 218, 20624
\coffin_display_handles:Nn .. ...
..... 220, 220, 21240
\coffin_dp:N .. ...
. 220, 220, 20756, 21357, 21831, 21857
\coffin_ht:N .. ...
. 220, 220, 20756, 21356, 21831, 21857
\coffin_if_exist:NTF .. ...
. 218, 218, 20601, 20612, 20613, 20617
\coffin_if_exist_p:N 218, 218, 20601
\coffin_join:NnnNnnn 220, 220, 20972
\coffin_log_structure:N .. ...
..... 221, 221, 21368
\coffin_mark_handle:Nnnn .. ...
..... 221, 221, 21185
\coffin_new:N .... 218, 218,
872, 20633, 20750, 20752, 20753,
20754, 20755, 21133, 21134, 21135
\coffin_resize:Nnn .. 227, 227, 21824
\coffin_rotate:Nn ... 227, 227, 21682
\coffin_scale:Nnn ... 227, 227, 21851
\coffin_set_eq:NN .. ...
. 218,
218, 20741, 21006, 21025, 21054, 21261
\coffin_set_horizontal_pole:Nnn ..
..... 219, 219, 20793
\coffin_set_vertical_pole:Nnn ..
..... 219, 219, 20793
\coffin_show_structure:N .. ...
.... 221, 221, 221, 887, 21350, 21369
\coffin_typeset:Nnnn 220, 220, 21125
\coffin_wd:N .. ...
. 220, 220, 20756, 21358, 21827, 21861
\c_empty_coffin ... 221, 20750, 21128
\l_tmpa_coffin .. ...
. 221, 20754
\l_tmpb_coffin .. ...
. 221, 20754
coffin internal commands:
\__coffin_align:NnnNnnnnN .. ...
.. 20974, 21011, 21029, 21037, 21128
\l__coffin_aligned_coffin .. ...
..... 20750, 20975,
20976, 20980, 20986, 20988, 20989,
21005, 21006, 21012, 21013, 21014,
21015, 21016, 21018, 21020, 21024,
21025, 21030, 21031, 21032, 21033,
21034, 21068, 21083, 21129, 21130,
21335, 21342, 21344, 21346, 21348
\l__coffin_aligned_internal_-
coffin .. ...
. 20750, 21047, 21054
\l__coffin_bottom_corner_dim ...
..... 21678, 21702, 21706,
21776, 21787, 21788, 21808, 21816
\l__coffin_bounding_prop .. ...
..... 21676, 21691, 21719,
21721, 21724, 21726, 21732, 21795
\l__coffin_bounding_shift_dim ...
.. 21677, 21700, 21794, 21800, 21801
\__coffin_calculate_intersection:Nnn
..... 20865, 21039, 21042, 21328
\__coffin_calculate_intersection:nnnnnnnn
..... 20865, 21274
\__coffin_calculate_intersection_-
aux:nnnnnN .. ...
. 20865
\c__coffin_corners_prop .. ...
..... 20573, 20640, 20775
\l__coffin_cos_fp .. ...
. 896, 898, 21674, 21685, 21759, 21768
\__coffin_display_attach:Nnnnn 21240
\l__coffin_display_coffin .. ...
.... 21133, 21261, 21267, 21337,
21338, 21343, 21345, 21347, 21348
\l__coffin_display_coord_coffin ..
..... 21133, 21199,
21219, 21235, 21282, 21302, 21321
\l__coffin_display_font_tl .. ...
..... 21178, 21207, 21290
\__coffin_display_handles_-
aux:nnnn .. ...
. 21240
\__coffin_display_handles_-
aux:nnnnnn .. ...
. 21240
\l__coffin_display_handles_prop ..
.. 21136, 21210, 21214, 21293, 21297
\l__coffin_display_offset_dim ...
.. 21173, 21236, 21237, 21322, 21323
\l__coffin_display_pole_coffin ..
.. 21133, 21187, 21198, 21242, 21280
\l__coffin_display_poles_prop ...
..... 21177, 21252,
21257, 21260, 21262, 21264, 21271
\l__coffin_display_x_dim .. ...
..... 21175, 21277, 21332
\l__coffin_display_y_dim .. ...
..... 21175, 21278, 21334
\l__coffin_error_bool .. ...
..... 20592, 20869, 20873,
20887, 20902, 20933, 21273, 21275
\__coffin_find_bounding_shift: ..
..... 21694, 21792
\__coffin_find_bounding_shift_-
aux:nn .. ...
. 21792
\__coffin_find_corner_maxima:N ..
..... 21693, 21772
\__coffin_find_corner_maxima_-
aux:nn .. ...
. 21772
\__coffin_get_pole:NnN .. ...
. 20762, 20867, 20868, 21092, 21093,
21096, 21097, 21254, 21255, 21258
\__coffin_gset_eq_structure>NN 20779

```

__coffin_if_exist:NTF
 20615, 20626, 20648, 20663, 20694,
 20710, 20743, 20795, 20806, 21352
 \l__coffin_internal_box
 20570, 20677,
 20683, 20688, 20724, 20730, 20735,
 21696, 21705, 21707, 21708, 21710
 \l__coffin_internal_dim
 20570, 20981, 20983, 20984,
 21723, 21725, 21727, 21856, 21859
 \l__coffin_internal_tl
 20570, 20579,
 20580, 20581, 20582, 20583, 20584,
 20585, 20586, 20587, 20588, 20589,
 21066, 21067, 21069, 21211, 21212,
 21215, 21216, 21224, 21229, 21294,
 21295, 21298, 21299, 21308, 21313
 \l__coffin_left_corner_dim
 21678, 21700, 21709,
 21777, 21783, 21784, 21807, 21815
 __coffin_mark_handle_aux:nnnnNnn
 21185
 __coffin_offset_corner:Nnnnn . 21075
 __coffin_offset_corners:Nnn . . .
 20994, 20995, 21001, 21002, 21075
 __coffin_offset_pole:Nnnnnnn . 21056
 __coffin_offset_poles:Nnn
 20992, 20993,
 20998, 20999, 21021, 21022, 21056
 \l__coffin_offset_x_dim
 20593, 20978, 20979, 20982,
 20990, 20992, 20994, 21000, 21003,
 21023, 21043, 21051, 21331, 21339
 \l__coffin_offset_y_dim
 20593, 20993, 20995, 21000, 21003,
 21023, 21045, 21052, 21333, 21340
 \l__coffin_pole_a_tl
 20595, 20867, 20872, 21092, 21095,
 21096, 21099, 21254, 21256, 21259
 \l__coffin_pole_b_tl 20595
 20868, 20872, 21093, 21095, 21097,
 21099, 21255, 21256, 21258, 21259
 \c__coffin_poles_prop
 20578, 20642, 20777
 __coffin_reset_structure:N
 20629, 20655, 20674,
 20700, 20721, 20772, 20986, 21016
 __coffin_resize_common:Nnn
 21834, 21837, 21862
 \l__coffin_right_corner_dim
 21678, 21709, 21775, 21785, 21786
 __coffin_rotate_bounding:nnn
 21692, 21729
 __coffin_rotate_corner:Nnnn
 21687, 21729
 __coffin_rotate_pole:Nnnnnnn
 21689, 21741
 __coffin_rotate_vector:nnNN
 21731, 21737, 21743, 21744, 21753
 __coffin_scale_corner:Nnnn
 21840, 21873
 __coffin_scale_pole:Nnnnnnn
 21842, 21873
 __coffin_scale_vector:nnNN
 21866, 21875, 21881
 \l__coffin_scale_x_fp 21820, 21826,
 21843, 21853, 21855, 21861, 21869
 \l__coffin_scale_y_fp 21820,
 21828, 21854, 21855, 21859, 21871
 \l__coffin_scaled_total_height_-
 dim 21822, 21858, 21863
 \l__coffin_scaled_width_dim
 21822, 21860, 21863
 __coffin_set_bounding:N 21690, 21717
 __coffin_set_eq_structure:NN
 20746, 20779
 __coffin_set_pole:Nnn
 20678, 20725, 20793, 21068, 21105,
 21109, 21117, 21121, 21746, 21882
 __coffin_shift_corner:Nnnn
 21712, 21803
 __coffin_shift_pole:Nnnnnnn
 21714, 21803
 \l__coffin_sin_fp
 896, 898, 21674, 21684, 21760, 21767
 \l__coffin_slope_x_fp
 20590, 20927, 20932, 20940, 20946
 \l__coffin_slope_y_fp
 20590, 20929, 20932, 20941, 20946
 \l__coffin_top_corner_dim
 21678, 21706, 21774, 21789, 21790
 __coffin_update_B:nnnnnnnnN . 21090
 __coffin_update_corners:N
 20657, 20676, 20702, 20723, 20820
 __coffin_update_poles:N
 20656, 20675,
 20701, 20722, 20831, 20989, 21020
 __coffin_update_T:nnnnnnnnN . 21090
 __coffin_update_vertical_-
 poles:NNN 21005, 21024, 21090
 \l__coffin_x_dim
 20597, 20876, 20885, 20905, 20908,
 20915, 20924, 20935, 20950, 21040,
 21044, 21063, 21071, 21277, 21329,
 21731, 21733, 21737, 21739, 21743,
 21748, 21875, 21877, 21881, 21884

```

\l__coffin_x_prime_dim ..... 16, 16, 16, 314,
..... 20597, 21040,
21044, 21329, 21332, 21745, 21749
\l__coffin_x_shift_corner:Nnnn ...
..... 21846, 21888
\l__coffin_x_shift_pole:Nnnnnn ...
..... 21848, 21888
\l__coffin_y_dim ..... 20597, 20877, 20890, 20893,
20900, 20917, 20922, 20951, 21041,
21046, 21064, 21071, 21278, 21330,
21731, 21733, 21737, 21739, 21743,
21748, 21875, 21877, 21881, 21884
\l__coffin_y_prime_dim ..... 20597, 21041,
21046, 21330, 21334, 21745, 21750
\color ..... 21193, 21205, 21248, 21288
color commands:
\color_ensure_current: ..... 222, 222, 870, 20652, 20696, 21405
\color_group_begin: ..... 222, 222, 222, 21399
\color_group_end: 222, 222, 222, 21399
\columnwidth ..... 20670, 20716
\copy ..... 316
\copyfont ..... 951
\cos ..... 188
\cosd ..... 188
\cot ..... 188
\cotd ..... 188
\count ..... 164, 166, 167, 168,
172, 173, 175, 176, 179, 181, 182,
183, 184, 188, 189, 191, 192, 317, 7024
\countdef ..... 318
\cr ..... 319
\crampeddisplaystyle ..... 867
\crampedscriptscriptstyle ..... 868
\crampedscriptstyle ..... 869
\crampedtextstyle ..... 870
\crcr ..... 320
\cs ..... 12752
cs commands:
\cs:w ..... 16, 16, 16, 17, 832,
832, 1311, 1331, 1333, 1381, 1643,
1671, 1919, 1979, 2071, 2110, 2124,
2126, 2130, 2131, 2132, 2167, 2173,
2193, 2195, 2200, 2207, 2208, 2271,
2275, 2305, 2510, 4689, 4778, 5480,
5529, 5952, 5954, 9233, 9443, 9489,
9559, 9586, 10423, 10442, 10452,
10466, 10736, 10782, 11463, 11580,
11768, 11800, 12173, 12202, 14210,
14477, 15010, 19326, 19328, 20029
\cs_end: ..... 16, 16, 16, 314,
1311, 1331, 1333, 1337, 1381, 1637,
1643, 1665, 1671, 1851, 1919, 1979,
2071, 2110, 2124, 2126, 2130, 2131,
2132, 2167, 2173, 2193, 2195, 2200,
2207, 2208, 2271, 2275, 2305, 2510,
4689, 4778, 5480, 5485, 5510, 5519,
5529, 5555, 5949, 5955, 5957, 5959,
5961, 5963, 5965, 5967, 5969, 5971,
5973, 5975, 9233, 9443, 9489, 9559,
9586, 10096, 10423, 10443, 10453,
10466, 10736, 10790, 11463, 11583,
11772, 11804, 12173, 12205, 14210,
14477, 15169, 19210, 19341, 20029
\cs_generate_from_arg_count:NNnn
..... 14, 14, 1900, 1937
\cs_generate_variant:Nn ..... 10,
24, 24, 25, 25, 25, 26, 310, 310, 310,
310, 2326, 2521, 2532, 2533, 2538,
2539, 2544, 2545, 2558, 2559, 2576,
2577, 2578, 2579, 2580, 2581, 2598,
2599, 2600, 2601, 2602, 2603, 2604,
2605, 2622, 2623, 2624, 2625, 2626,
2627, 2628, 2629, 2732, 2733, 2734,
2735, 2803, 2804, 2805, 2806, 2868,
2869, 2874, 2875, 2878, 2879, 2880,
2881, 2882, 2883, 2884, 2885, 2894,
2895, 2896, 2897, 2907, 2908, 2909,
2910, 2930, 2931, 2932, 2933, 2952,
2953, 2954, 2963, 2964, 2965, 3008,
3009, 3010, 3011, 3028, 3041, 3057,
3062, 3064, 3073, 3085, 3086, 3109,
3110, 3208, 3219, 3220, 3242, 3252,
3269, 3270, 3271, 3272, 3389, 3395,
3400, 3434, 3449, 3450, 3458, 3513,
3514, 3515, 3516, 3517, 3518, 3519,
3520, 3531, 3532, 3533, 3534, 3557,
3558, 3559, 3560, 3642, 3700, 3777,
3796, 3834, 3849, 3866, 3867, 3868,
3881, 3945, 4096, 4099, 4102, 4105,
4108, 4137, 4138, 4139, 4140, 4141,
4142, 4178, 4179, 4184, 4185, 4207,
4208, 4209, 4210, 4215, 4216, 4217,
4218, 4235, 4236, 4261, 4262, 4279,
4280, 4289, 4290, 4291, 4292, 4312,
4313, 4314, 4315, 4316, 4317, 4346,
4359, 4360, 4375, 4401, 4402, 4407,
4408, 4409, 4410, 4411, 4412, 4421,
4422, 4423, 4424, 4425, 4426, 4427,
4428, 4429, 4430, 4431, 4432, 4456,
4476, 4505, 4516, 4517, 4527, 4550,
4564, 4603, 4606, 4692, 4713, 4729,
4730, 4735, 4736, 4738, 4739, 4741,
4742, 4755, 4756, 4757, 4758, 4767,
```

- 4768, 4769, 4770, 4774, 4775, 4777,
5368, 5372, 5603, 5604, 5634, 5635,
5636, 5637, 5650, 5651, 5652, 5653,
5679, 5688, 5689, 5690, 5691, 5704,
5705, 5758, 5759, 5760, 5761, 5771,
5776, 5909, 5910, 5915, 5916, 6015,
6056, 6057, 6058, 6059, 6073, 6074,
6110, 6111, 6121, 6122, 6123, 6124,
6134, 6135, 6136, 6137, 6148, 6173,
6174, 6184, 6185, 6186, 6200, 6201,
6202, 6203, 6204, 6205, 6237, 6238,
6268, 6269, 6274, 6275, 6322, 6323,
6324, 6325, 6326, 6327, 6328, 6329,
6330, 6346, 6380, 6404, 6417, 6457,
6466, 6490, 6497, 6541, 6546, 6685,
6686, 6687, 6688, 7309, 7312, 7315,
7318, 7321, 7357, 7358, 7359, 7360,
7367, 7368, 7387, 7388, 7389, 7390,
7403, 7424, 7425, 7426, 7427, 7428,
7429, 7443, 7445, 7447, 7449, 7466,
7467, 7477, 7478, 7479, 7480, 7503,
7504, 7505, 7506, 7507, 7508, 7509,
7510, 7520, 7521, 7522, 7523, 7524,
7525, 7540, 7541, 7556, 7567, 7570,
8014, 8356, 8564, 8581, 8645, 8648,
8658, 8659, 8660, 8691, 8709, 8821,
8828, 8845, 8863, 8869, 8872, 8890,
9236, 9242, 9245, 9246, 9251, 9252,
9260, 9261, 9263, 9264, 9266, 9267,
9271, 9272, 9276, 9277, 9442, 9472,
9492, 9498, 9501, 9502, 9507, 9508,
9516, 9517, 9519, 9520, 9522, 9523,
9527, 9528, 9532, 9533, 9558, 9566,
9567, 9569, 9589, 9595, 9599, 9600,
9605, 9606, 9614, 9615, 9617, 9618,
9620, 9621, 9625, 9626, 9630, 9631,
9635, 9637, 9820, 9921, 9937, 9994,
10107, 10240, 10241, 10244, 10252,
10260, 10269, 10277, 10285, 10299,
10317, 10350, 10489, 11049, 11052,
12544, 12748, 12793, 15639, 15680,
15745, 15781, 15785, 15832, 16034,
16041, 16042, 16043, 16046, 16047,
16050, 16051, 16056, 16057, 16064,
16065, 16066, 16067, 16075, 16080,
16217, 16224, 16231, 16238, 16243,
16244, 16931, 17684, 20032, 20037,
20038, 20043, 20044, 20049, 20050,
20055, 20056, 20064, 20065, 20066,
20073, 20074, 20075, 20078, 20079,
20095, 20096, 20097, 20098, 20099,
20100, 20101, 20102, 20105, 20106,
20107, 20108, 20113, 20114, 20122,
20125, 20128, 20137, 20155, 20161,
20162, 20170, 20171, 20180, 20181,
20201, 20202, 20223, 20224, 20232,
20233, 20241, 20242, 20251, 20252,
20262, 20263, 20405, 20442, 20457,
20469, 20485, 20495, 20512, 20515,
20611, 20612, 20613, 20614, 20632,
20645, 20660, 20691, 20707, 20740,
20749, 20817, 20818, 20819, 21008,
21036, 21132, 21239, 21325, 21367,
21370, 21565, 21611, 21658, 21672,
21716, 21836, 21865, 22009, 22010,
22011, 22012, 22029, 22044, 22063,
22086, 22087, 22113, 22198, 22217,
22273, 22274, 22298, 22299, 22313,
22348, 23297, 23299, 23780, 23837,
23860, 24154, 24543, 24574, 24758
\cs_gset:Nn 14, 14, 1914, 1974
\cs_gset:Npn
.... 10, 12, 12, 1366, 1799, 1813,
1815, 4480, 7622, 7624, 7662, 11485
\cs_gset:Npx 12,
1366, 1800, 1813, 1816, 4485, 19218
\cs_gset_eq:NN
.... 15, 15, 15, 1831, 1848, 1856,
2519, 2550, 2551, 2552, 2553, 2671,
3413, 3422, 4094, 4490, 4495, 5685,
5687, 5696, 5697, 5698, 5699, 5721,
5726, 5736, 6683, 7307, 7544, 7552,
8706, 8860, 9220, 16506, 16515,
21435, 21436, 21437, 21438, 21443,
21444, 21445, 21446, 21455, 21456,
21457, 21458, 21462, 21463, 21464,
21465, 21471, 21472, 21473, 21474
\cs_gset_nopar:Nn .. 14, 14, 1914, 1974
\cs_gset_nopar:Npn
.... 12, 12, 1366, 1797, 1805, 1809
\cs_gset_nopar:Npx
.... 12, 1366, 1798, 1805, 1810, 2525,
2530, 2571, 2573, 2575, 2591, 2593,
2595, 2597, 2615, 2617, 2619, 2621
\cs_gset_protected:Nn
.... 14, 14, 1914, 1974
\cs_gset_protected:Npn 12,
12, 1366, 1803, 1825, 1827, 3032,
4990, 6366, 7547, 7593, 8763, 12798,
16146, 16154, 16167, 16580, 16826
\cs_gset_protected:Npx 12,
1366, 1804, 1825, 1828, 4997, 12805
\cs_gset_protected_nopar:Nn
.... 14, 14, 1914, 1974
\cs_gset_protected_nopar:Npn
.... 12, 12, 1366, 1801, 1819, 1821
\cs_gset_protected_nopar:Npx
.... 12, 1366, 1802, 1819, 1822

\cs_if_eq:NNTF 20, 20,
 2006, 2013, 2014, 2017, 2018, 2021,
 2022, 7855, 10063, 11258, 11268,
 11294, 11296, 11298, 11490, 18260
 \cs_if_eq_p:NN ... 20, 20, 2006, 18276
 \cs_if_exist 206
 \cs_if_exist:N
 20, 2560, 2561, 4186, 4188,
 4743, 4745, 5781, 5783, 6075, 6077,
 7468, 7470, 9253, 9255, 9509, 9511,
 9607, 9609, 12606, 12607, 20057, 20059
 \cs_if_exist:NTF 20, 20, 137,
 297, 446, 472, 1623, 1680, 1682,
 1684, 1686, 1688, 1690, 1692, 1694,
 1771, 1781, 2027, 3477, 3900, 3908,
 3954, 3957, 3970, 4062, 4715, 4716,
 4718, 4719, 5401, 5402, 5498, 5505,
 6743, 6744, 6746, 6991, 7008, 7580,
 8434, 8511, 8530, 8633, 8637, 8722,
 8783, 8809, 8813, 9834, 9944, 9990,
 10366, 10408, 10420, 10433, 10439,
 10450, 10464, 10507, 10515, 11483,
 11631, 12593, 15872, 15883, 16144,
 16162, 16165, 16578, 18028, 18430,
 19316, 20603, 20605, 21433, 21441,
 21449, 21469, 21935, 21986, 22129,
 22449, 22478, 22504, 22515, 22522,
 22635, 22857, 22881, 23226, 23504,
 23511, 23517, 23523, 23594, 23603
 \cs_if_exist_p:N 20, 20, 22, 1623, 21452
 \cs_if_exist_use:N
 16, 16, 1679, 10422, 18428
 cs_if_exist_use:N 287
 \cs_if_exist_use:NTF
 16, 16, 1679, 1681,
 1683, 1689, 1691, 10029, 10930,
 11590, 11592, 17291, 17298, 17648,
 17653, 17689, 18084, 18163, 19256,
 21478, 22441, 22474, 23192, 23194
 \cs_if_free:NTF 20, 20, 89,
 472, 1651, 1746, 1756, 2467, 2506, 7886
 \cs_if_free_p:N
 20, 20, 20, 22, 22, 89, 1651
 \cs_log:N .. 16, 16, 136, 298, 298, 2046
 \cs_meaning:N
 15, 15, 275, 1320, 1334, 1342, 2042
 \cs_new:Nn 12, 12, 90, 1914, 1974
 \cs_new:Npn 10,
 11, 11, 14, 89, 89, 738, 1789, 1813,
 1817, 1888, 1890, 1898, 1945, 2011,
 2012, 2013, 2014, 2015, 2016, 2017,
 2018, 2019, 2020, 2021, 2022, 2050,
 2059, 2060, 2064, 2065, 2066, 2067,
 2068, 2069, 2070, 2072, 2074, 2086,
 2092, 2098, 2109, 2111, 2118, 2119,
 2121, 2123, 2125, 2127, 2134, 2136,
 2141, 2146, 2152, 2158, 2164, 2170,
 2176, 2183, 2190, 2197, 2204, 2212,
 2213, 2214, 2215, 2216, 2217, 2218,
 2219, 2220, 2227, 2228, 2229, 2230,
 2231, 2240, 2241, 2246, 2248, 2253,
 2263, 2265, 2267, 2268, 2270, 2272,
 2278, 2284, 2286, 2293, 2295, 2296,
 2297, 2298, 2300, 2302, 2304, 2305,
 2306, 2308, 2313, 2322, 2323, 2393,
 2409, 2414, 2423, 2436, 2451, 2508,
 2730, 2911, 2966, 2967, 2968, 2969,
 2979, 2980, 2985, 2990, 2995, 3000,
 3002, 3012, 3015, 3021, 3023, 3058,
 3060, 3063, 3065, 3074, 3079, 3084,
 3087, 3094, 3101, 3103, 3113, 3125,
 3133, 3139, 3145, 3149, 3156, 3167,
 3176, 3178, 3185, 3191, 3193, 3195,
 3209, 3211, 3213, 3221, 3226, 3231,
 3243, 3244, 3245, 3253, 3299, 3308,
 3339, 3360, 3367, 3375, 3381, 3388,
 3476, 3490, 3498, 3535, 3540, 3545,
 3550, 3555, 3561, 3567, 3572, 3577,
 3582, 3587, 3589, 3596, 3607, 3616,
 3619, 3630, 3639, 3641, 3643, 3651,
 3653, 3660, 3681, 3691, 3693, 3698,
 3699, 3701, 3709, 3711, 3719, 3725,
 3731, 3750, 3752, 3761, 3767, 3773,
 3775, 3778, 3788, 3795, 3797, 3805,
 3810, 3815, 3826, 3833, 3835, 3841,
 3843, 3848, 3850, 3856, 3857, 3862,
 3863, 3864, 3865, 3869, 3874, 3879,
 3882, 3883, 3891, 3896, 3910, 3924,
 4082, 4170, 4176, 4206, 4219, 4274,
 4310, 4344, 4396, 4433, 4435, 4443,
 4449, 4457, 4459, 4461, 4470, 4518,
 4526, 4528, 4551, 4552, 4553, 4560,
 4562, 4619, 4621, 4627, 4645, 4653,
 4661, 4674, 4676, 4683, 4778, 4785,
 4799, 4804, 4810, 4821, 4826, 4833,
 4835, 4837, 4839, 4841, 4843, 4845,
 4855, 4860, 4865, 4870, 4875, 4877,
 4900, 4908, 4916, 4922, 4928, 4936,
 4944, 4950, 4956, 4963, 4977, 5012,
 5026, 5032, 5064, 5096, 5098, 5100,
 5106, 5112, 5124, 5132, 5144, 5152,
 5185, 5218, 5220, 5222, 5224, 5226,
 5231, 5236, 5241, 5246, 5247, 5248,
 5249, 5250, 5251, 5252, 5253, 5254,
 5255, 5256, 5257, 5258, 5259, 5260,
 5261, 5262, 5271, 5272, 5281, 5287,
 5289, 5298, 5305, 5311, 5313, 5315,
 5331, 5342, 5365, 5433, 5460, 5462,

5464, 5479, 5516, 5517, 5526, 5527,
5573, 5579, 5587, 5594, 5601, 5605,
5611, 5643, 5649, 5671, 5769, 5793,
5803, 5805, 5807, 5809, 5813, 5823,
5825, 5830, 5832, 5841, 5842, 5843,
5844, 5845, 5846, 5847, 5849, 5852,
5858, 5864, 5870, 5885, 5898, 5899,
5905, 5907, 5911, 5913, 5917, 5925,
5930, 5938, 5944, 5951, 5953, 5955,
5956, 5958, 5960, 5962, 5964, 5966,
5968, 5970, 5972, 5974, 5976, 5981,
5982, 5983, 5984, 5985, 5986, 5987,
5988, 5989, 5990, 5999, 6001, 6048,
6055, 6079, 6084, 6085, 6092, 6172,
6265, 6267, 6276, 6283, 6286, 6299,
6305, 6331, 6340, 6347, 6353, 6360,
6405, 6407, 6409, 6427, 6434, 6458,
6459, 6462, 6464, 6467, 6475, 6491,
6498, 6506, 6508, 6522, 6527, 6559,
6632, 6641, 6650, 6659, 6689, 6695,
6702, 6748, 6756, 6818, 6938, 6970,
7043, 7055, 7056, 7064, 7073, 7082,
7095, 7096, 7097, 7098, 7148, 7156,
7158, 7160, 7170, 7180, 7266, 7269,
7278, 7287, 7300, 7344, 7391, 7397,
7488, 7494, 7526, 7532, 7557, 7559,
7661, 7747, 7752, 7757, 7762, 7767,
7772, 7859, 7884, 8015, 8293, 8302,
8307, 8316, 8321, 8326, 8331, 8407,
8408, 8412, 8417, 8495, 8747, 8749,
8895, 8904, 8947, 9036, 9045, 9064,
9072, 9078, 9086, 9096, 9101, 9107,
9113, 9177, 9179, 9278, 9283, 9285,
9293, 9301, 9309, 9311, 9323, 9329,
9344, 9346, 9348, 9350, 9352, 9354,
9359, 9364, 9369, 9374, 9376, 9383,
9391, 9399, 9405, 9411, 9419, 9427,
9433, 9439, 9443, 9444, 9451, 9459,
9461, 9463, 9552, 9555, 9559, 9561,
9564, 9632, 9775, 9780, 9782, 9902,
10462, 10480, 10485, 10487, 10490,
10498, 10503, 10620, 10621, 10622,
10623, 10624, 10625, 10626, 10627,
10628, 10629, 10661, 10663, 10665,
10674, 10676, 10688, 10689, 10691,
10701, 10711, 10721, 10731, 10734,
10744, 10748, 10753, 10755, 10757,
10759, 10761, 10768, 10770, 10778,
10780, 10792, 10794, 10796, 10798,
10822, 10824, 10826, 10827, 10828,
10830, 10832, 10834, 10836, 10854,
10869, 10870, 10876, 10891, 10898,
10900, 10907, 11036, 11037, 11038,
11039, 11040, 11041, 11042, 11047,
11050, 11094, 11096, 11098, 11100,
11117, 11119, 11130, 11132, 11141,
11142, 11151, 11164, 11177, 11184,
11198, 11214, 11226, 11236, 11245,
11256, 11266, 11292, 11303, 11312,
11323, 11328, 11348, 11350, 11361,
11366, 11379, 11402, 11403, 11413,
11419, 11436, 11437, 11461, 11470,
11475, 11499, 11528, 11531, 11535,
11537, 11543, 11550, 11557, 11565,
11588, 11602, 11621, 11629, 11644,
11659, 11670, 11680, 11690, 11695,
11704, 11721, 11734, 11739, 11745,
11747, 11754, 11784, 11812, 11828,
11839, 11844, 11862, 11880, 11891,
11906, 11911, 11921, 11931, 11941,
11957, 12005, 12012, 12029, 12037,
12068, 12080, 12094, 12113, 12121,
12130, 12139, 12149, 12152, 12158,
12168, 12169, 12176, 12181, 12228,
12229, 12233, 12250, 12267, 12269,
12280, 12293, 12323, 12325, 12339,
12352, 12367, 12384, 12408, 12410,
12412, 12414, 12424, 12429, 12440,
12451, 12462, 12475, 12495, 12513,
12515, 12531, 12542, 12545, 12555,
12566, 12569, 12602, 12604, 12613,
12632, 12637, 12664, 12665, 12673,
12685, 12691, 12697, 12705, 12713,
12719, 12725, 12733, 12741, 12758,
12777, 12823, 12834, 12858, 12860,
12862, 12864, 12875, 12882, 12884,
12885, 12910, 12916, 12923, 12924,
12932, 12943, 12945, 12947, 12954,
12978, 12980, 12990, 13005, 13014,
13028, 13036, 13044, 13051, 13058,
13066, 13076, 13090, 13101, 13102,
13108, 13125, 13132, 13134, 13141,
13146, 13163, 13164, 13165, 13184,
13190, 13200, 13212, 13219, 13233,
13241, 13279, 13288, 13307, 13309,
13311, 13320, 13331, 13343, 13358,
13371, 13384, 13392, 13409, 13426,
13433, 13441, 13451, 13452, 13461,
13462, 13471, 13481, 13495, 13505,
13516, 13524, 13526, 13537, 13543,
13578, 13599, 13601, 13603, 13605,
13612, 13621, 13626, 13633, 13640,
13660, 13665, 13682, 13692, 13694,
13709, 13710, 13715, 13723, 13724,
13747, 13760, 13767, 13775, 13776,
13777, 13778, 13779, 13780, 13788,
13794, 13796, 13798, 13820, 13825,
13835, 13845, 13855, 13868, 13879,

13884, 13891, 13900, 13902, 13911,
 13920, 13934, 13936, 13938, 13951,
 13961, 13966, 13975, 13983, 13990,
 13996, 14005, 14007, 14019, 14024,
 14032, 14037, 14047, 14053, 14059,
 14066, 14073, 14075, 14080, 14082,
 14087, 14088, 14102, 14112, 14124,
 14129, 14136, 14146, 14148, 14159,
 14173, 14187, 14207, 14220, 14222,
 14227, 14240, 14245, 14253, 14258,
 14268, 14280, 14309, 14310, 14311,
 14313, 14315, 14317, 14331, 14337,
 14346, 14365, 14371, 14381, 14400,
 14408, 14441, 14447, 14456, 14458,
 14472, 14474, 14482, 14484, 14500,
 14516, 14532, 14548, 14564, 14580,
 14585, 14610, 14633, 14662, 14678,
 14688, 14699, 14720, 14735, 14740,
 14745, 14747, 14761, 14763, 14778,
 14786, 14796, 14798, 14840, 14842,
 14844, 14846, 14848, 14863, 14878,
 14893, 14908, 14923, 14938, 14946,
 14960, 14962, 14968, 14980, 14988,
 14995, 15171, 15179, 15190, 15191,
 15193, 15194, 15205, 15212, 15214,
 15220, 15231, 15241, 15248, 15255,
 15270, 15309, 15322, 15353, 15358,
 15363, 15377, 15397, 15399, 15416,
 15431, 15443, 15450, 15455, 15457,
 15466, 15479, 15482, 15503, 15516,
 15531, 15549, 15564, 15574, 15583,
 15596, 15612, 15629, 15634, 15635,
 15636, 15637, 15640, 15645, 15668,
 15676, 15678, 15681, 15686, 15709,
 15734, 15743, 15744, 15746, 15751,
 15761, 15769, 15777, 15779, 15782,
 15784, 15786, 15791, 15796, 15803,
 15817, 15822, 15824, 15834, 15836,
 15838, 15840, 15868, 15870, 15876,
 15885, 15890, 15896, 15902, 15919,
 15924, 15934, 15944, 15946, 15959,
 15972, 15986, 15995, 16000, 16009,
 16011, 16018, 16258, 16260, 16362,
 16375, 16380, 16386, 16387, 16393,
 16399, 16405, 16411, 16417, 16418,
 16420, 16427, 16433, 16530, 16535,
 16536, 16544, 16550, 16719, 16724,
 16731, 16764, 16769, 16784, 16807,
 16812, 16867, 16885, 16890, 16905,
 16907, 16909, 16916, 16956, 17014,
 17043, 17048, 17289, 17295, 17306,
 17312, 17325, 17330, 17345, 17357,
 17367, 17376, 17396, 17499, 17517,
 17525, 17537, 17549, 18017, 18258,
 18272, 18312, 18350, 18502, 18645,
 19007, 19135, 19137, 19143, 19144,
 19151, 19161, 19282, 19616, 19624,
 19993, 21660, 21662, 21667, 21937,
 21943, 21962, 21967, 21972, 21980,
 21992, 22001, 22003, 22005, 22007,
 22013, 22020, 22028, 22030, 22036,
 22045, 22046, 22047, 22056, 22065,
 22067, 22073, 22079, 22108, 22115,
 22138, 22229, 22242, 22248, 22253,
 22265, 22266, 22267, 22314, 22315,
 22316, 22317, 22318, 22319, 22320,
 22328, 22335, 22346, 22349, 22355,
 22367, 22372, 22380, 22394, 22399,
 22410, 22421, 22427, 22432, 22439,
 22451, 22455, 22470, 22480, 22495,
 22496, 22498, 22500, 22502, 22513,
 22539, 22550, 22559, 22573, 22579,
 22582, 22597, 22603, 22605, 22615,
 22621, 22637, 22647, 22654, 22674,
 22684, 22697, 22705, 22711, 22731,
 22740, 22746, 22768, 22781, 22790,
 22796, 22812, 22818, 22823, 22853,
 22855, 23110, 23118, 23125, 23136,
 23148, 23153, 23161, 23175, 23188,
 23190, 23200, 23206, 23220, 23224,
 23233, 23243, 23249, 23291, 23296,
 23298, 23300, 23308, 23320, 23330,
 23339, 23347, 23408, 23409, 23413,
 23414, 23936, 24073, 24298, 24699
 \cs_new:Npx
 .. 11, 308, 309, 1789, 1813, 1818,
 2347, 2490, 6418, 6428, 8941, 11405,
 11513, 12049, 12211, 12949, 14827,
 14833, 15007, 15852, 16744, 17313,
 17315, 17317, 17319, 17321, 17323
 \cs_new_eq:NN 15, 15, 15,
 90, 289, 292, 441, 441, 1528, 1831,
 2049, 2058, 2080, 2318, 2546, 2547,
 2548, 2549, 2550, 2551, 2552, 2553,
 2862, 2863, 3014, 3433, 3447, 3448,
 3595, 3921, 3943, 3944, 4089, 4109,
 4110, 4111, 4112, 4113, 4114, 4115,
 4116, 4565, 4566, 4567, 4568, 4569,
 4570, 4571, 4572, 4573, 4574, 4575,
 4576, 4577, 4578, 4579, 4580, 4581,
 4582, 4583, 4584, 4585, 4586, 4587,
 4588, 4589, 4590, 4614, 4615, 4616,
 4617, 4618, 4720, 4721, 4724, 4776,
 4883, 5011, 5367, 5371, 5413, 5667,
 5675, 5676, 5678, 5692, 5693, 5694,
 5695, 5696, 5697, 5698, 5699, 6008,
 6011, 6012, 6016, 6017, 6018, 6019,
 6020, 6021, 6022, 6023, 6024, 6025,

6026, 6027, 6028, 6029, 6030, 6031,
6206, 6207, 6208, 6209, 6210, 6211,
6212, 6213, 6214, 6215, 6216, 6217,
6218, 6219, 6220, 6221, 6832, 6839,
6841, 6842, 6843, 6845, 6848, 6849,
7091, 7092, 7093, 7322, 7323, 7324,
7325, 7326, 7327, 7328, 7329, 8644,
8719, 8820, 8866, 8896, 8983, 9226,
9227, 9228, 9382, 9441, 9471, 9475,
9476, 9557, 9560, 9563, 9568, 9572,
9573, 9634, 9636, 9640, 9641, 10740,
11197, 11225, 11233, 11234, 11235,
11243, 11244, 12004, 12089, 12090,
12091, 15831, 15833, 15881, 16033,
16044, 16045, 16523, 16524, 16988,
17017, 17136, 17137, 17234, 17242,
17263, 17302, 17303, 17304, 17305,
17490, 18917, 19437, 20061, 20062,
20063, 20076, 20077, 20088, 20089,
20090, 20187, 20199, 20200, 20259,
20260, 20261, 20565, 20566, 20756,
20757, 20758, 20759, 20760, 20761,
21399, 21428, 21429, 21430, 21431,
21480, 21481, 21482, 21483, 21484,
21485, 21486, 21487, 21491, 21492,
21493, 21494, 21495, 21496, 21497,
21498, 22492, 22634, 22703, 22704,
23619, 23669, 23670, 23736, 23743,
23757, 23826, 23859, 24147, 24542
\cs_new_nopar:Nn . . . 13, 13, 1914, 1974
\cs_new_nopar:Npn 11,
11, 289, 291, 1789, 1805, 1811, 2048
\cs_new_nopar:Npx 11, 1789, 1805, 1812
\cs_new_protected:Nn 13, 13, 1914, 1974
\cs_new_protected:Npn
. 11, 11, 1789, 1825,
1829, 1831, 1832, 1833, 1834, 1835,
1836, 1837, 1838, 1839, 1844, 1845,
1846, 1847, 1849, 1900, 1910, 1912,
1922, 1927, 2023, 2025, 2030, 2032,
2034, 2036, 2044, 2046, 2047, 2081,
2211, 2221, 2222, 2223, 2224, 2225,
2226, 2232, 2233, 2234, 2235, 2236,
2237, 2238, 2239, 2258, 2299, 2321,
2326, 2353, 2359, 2364, 2373, 2463,
2500, 2516, 2522, 2527, 2534, 2536,
2540, 2542, 2554, 2556, 2564, 2566,
2568, 2570, 2572, 2574, 2582, 2584,
2586, 2588, 2590, 2592, 2594, 2596,
2606, 2608, 2610, 2612, 2614, 2616,
2618, 2620, 2692, 2694, 2696, 2698,
2720, 2738, 2750, 2752, 2766, 2795,
2797, 2799, 2801, 2807, 2829, 2835,
2864, 2866, 2870, 2872, 2949, 2950,
2951, 3029, 3039, 3042, 3048, 3050,
3105, 3107, 3215, 3217, 3390, 3396,
3398, 3401, 3407, 3416, 3919, 3920,
4091, 4097, 4100, 4103, 4106, 4117,
4122, 4127, 4132, 4143, 4145, 4147,
4180, 4182, 4190, 4198, 4211, 4213,
4221, 4223, 4225, 4237, 4239, 4241,
4263, 4265, 4267, 4318, 4326, 4336,
4347, 4349, 4351, 4353, 4361, 4370,
4376, 4378, 4380, 4477, 4482, 4487,
4493, 4499, 4506, 4597, 4604, 4686,
4693, 4727, 4728, 4731, 4733, 4737,
4740, 4747, 4749, 4751, 4753, 4759,
4761, 4763, 4765, 4771, 4773, 4779,
4986, 4993, 5005, 5369, 5373, 5422,
5434, 5436, 5441, 5477, 5482, 5483,
5494, 5496, 5501, 5566, 5658, 5678,
5680, 5682, 5684, 5686, 5700, 5702,
5762, 5767, 5772, 5774, 6010, 6013,
6032, 6034, 6036, 6060, 6062, 6064,
6106, 6108, 6112, 6114, 6116, 6125,
6127, 6129, 6138, 6146, 6149, 6151,
6153, 6161, 6191, 6223, 6225, 6227,
6239, 6241, 6243, 6270, 6272, 6315,
6361, 6375, 6381, 6392, 6397, 6528,
6534, 6542, 6544, 6554, 6561, 6563,
6565, 6567, 6569, 6571, 6573, 6575,
6577, 6579, 6581, 6583, 6585, 6587,
6589, 6591, 6593, 6595, 6597, 6599,
6601, 6603, 6605, 6607, 6609, 6611,
6613, 6615, 6617, 6619, 6621, 6623,
6625, 6627, 6634, 6636, 6643, 6645,
6652, 6654, 6661, 6673, 6832, 7099,
7101, 7103, 7109, 7126, 7128, 7130,
7144, 7146, 7189, 7304, 7310, 7313,
7316, 7319, 7334, 7336, 7345, 7351,
7361, 7369, 7378, 7430, 7431, 7432,
7451, 7453, 7455, 7542, 7561, 7568,
7583, 7604, 7609, 7611, 7618, 7620,
7627, 7668, 7680, 7685, 7702, 7732,
7738, 7782, 7784, 7853, 7894, 7914,
7915, 7928, 7933, 7959, 7968, 7970,
7972, 7989, 8016, 8018, 8020, 8022,
8029, 8337, 8339, 8351, 8357, 8359,
8376, 8384, 8386, 8455, 8497, 8499,
8507, 8544, 8549, 8558, 8565, 8582,
8584, 8589, 8594, 8644, 8646, 8649,
8661, 8672, 8675, 8692, 8698, 8710,
8712, 8736, 8738, 8751, 8753, 8755,
8761, 8768, 8820, 8823, 8826, 8829,
8846, 8852, 8864, 8867, 8870, 8873,
8879, 8885, 8892, 8894, 8905, 8935,
8953, 8985, 8994, 9006, 9025, 9029,
9117, 9133, 9141, 9150, 9157, 9163,

9214, 9230, 9237, 9243, 9244, 9247,
9249, 9257, 9259, 9262, 9265, 9268,
9270, 9273, 9275, 9473, 9477, 9486,
9493, 9499, 9500, 9503, 9505, 9513,
9515, 9518, 9521, 9524, 9526, 9529,
9531, 9570, 9574, 9583, 9590, 9596,
9598, 9601, 9603, 9611, 9613, 9616,
9619, 9622, 9624, 9627, 9629, 9638,
9642, 9656, 9673, 9679, 9684, 9697,
9703, 9708, 9722, 9725, 9735, 9745,
9767, 9776, 9778, 9812, 9814, 9821,
9826, 9831, 9844, 9849, 9873, 9885,
9906, 9922, 9938, 9940, 9942, 9958,
9968, 9970, 9972, 9988, 9995, 10009,
10022, 10027, 10031, 10039, 10041,
10051, 10078, 10087, 10096, 10097,
10108, 10110, 10112, 10114, 10116,
10118, 10120, 10122, 10124, 10126,
10128, 10130, 10132, 10134, 10136,
10138, 10140, 10142, 10144, 10146,
10148, 10150, 10152, 10154, 10156,
10158, 10160, 10162, 10164, 10166,
10168, 10170, 10172, 10174, 10176,
10178, 10180, 10182, 10184, 10186,
10188, 10190, 10192, 10194, 10196,
10198, 10200, 10202, 10204, 10206,
10208, 10210, 10212, 10214, 10216,
10218, 10220, 10222, 10224, 10226,
10228, 10230, 10232, 10234, 10242,
10245, 10253, 10261, 10267, 10270,
10278, 10286, 10292, 10300, 10306,
10308, 10318, 10324, 10329, 10334,
10351, 10364, 10378, 10404, 10418,
10428, 10471, 10520, 10535, 10546,
10631, 10928, 10945, 10947, 10949,
10951, 10979, 10981, 10983, 10985,
11005, 11007, 11009, 11011, 11013,
11015, 11017, 11019, 11021, 12522,
12529, 12794, 12801, 12813, 15409,
16032, 16035, 16037, 16039, 16048,
16049, 16052, 16054, 16058, 16059,
16060, 16061, 16062, 16068, 16073,
16076, 16078, 16103, 16122, 16130,
16142, 16179, 16186, 16211, 16218,
16225, 16232, 16239, 16241, 16245,
16268, 16277, 16297, 16307, 16317,
16319, 16321, 16322, 16329, 16332,
16342, 16352, 16443, 16450, 16452,
16467, 16500, 16509, 16552, 16562,
16572, 16591, 16600, 16602, 16626,
16633, 16646, 16650, 16653, 16665,
16674, 16688, 16700, 16711, 16817,
16824, 16837, 16848, 16855, 16947,
16965, 16967, 16969, 16971, 16973,
16982, 16989, 16998, 17007, 17012,
17015, 17018, 17020, 17029, 17035,
17041, 17071, 17073, 17074, 17079,
17085, 17093, 17105, 17121, 17138,
17148, 17156, 17158, 17166, 17177,
17195, 17197, 17202, 17210, 17212,
17219, 17224, 17226, 17228, 17235,
17243, 17245, 17247, 17249, 17256,
17261, 17264, 17270, 17493, 17557,
17570, 17580, 17590, 17617, 17646,
17651, 17656, 17676, 17685, 17694,
17696, 17703, 17712, 17714, 17716,
17718, 17724, 17746, 17760, 17787,
17792, 17826, 17861, 17882, 17884,
17894, 17903, 17908, 17917, 17926,
17941, 17954, 17960, 17971, 17984,
17990, 18003, 18023, 18053, 18062,
18075, 18080, 18098, 18106, 18111,
18113, 18115, 18130, 18146, 18148,
18169, 18189, 18210, 18217, 18224,
18236, 18242, 18296, 18331, 18340,
18356, 18374, 18380, 18433, 18443,
18445, 18447, 18454, 18513, 18526,
18542, 18547, 18563, 18581, 18588,
18595, 18597, 18599, 18616, 18630,
18646, 18655, 18669, 18681, 18701,
18710, 18712, 18724, 18733, 18747,
18760, 18767, 18787, 18818, 18852,
18870, 18876, 18885, 18921, 18948,
18967, 18972, 18983, 19013, 19028,
19037, 19048, 19060, 19067, 19069,
19071, 19091, 19096, 19103, 19108,
19113, 19118, 19167, 19187, 19228,
19238, 19254, 19266, 19276, 19309,
19323, 19332, 19338, 19347, 19370,
19376, 19379, 19387, 19390, 19393,
19402, 19405, 19408, 19411, 19416,
19425, 19428, 19431, 19436, 19438,
19443, 19448, 19453, 19459, 19471,
19473, 19477, 19478, 19502, 19510,
19519, 19531, 19540, 19548, 19587,
19636, 19663, 19665, 19667, 19696,
19724, 20010, 20026, 20033, 20035,
20039, 20041, 20045, 20047, 20051,
20053, 20067, 20069, 20071, 20080,
20082, 20084, 20086, 20109, 20111,
20120, 20123, 20126, 20129, 20131,
20138, 20156, 20158, 20160, 20163,
20168, 20172, 20178, 20182, 20188,
20193, 20195, 20197, 20203, 20205,
20207, 20212, 20217, 20222, 20225,
20230, 20234, 20239, 20243, 20249,
20253, 20264, 20278, 20288, 20322,
20333, 20344, 20355, 20366, 20377,

20390, 20406, 20413, 20423, 20428,
20443, 20458, 20470, 20486, 20496,
20510, 20513, 20516, 20528, 20615,
20624, 20633, 20646, 20661, 20692,
20706, 20708, 20739, 20741, 20762,
20772, 20779, 20786, 20793, 20804,
20815, 20820, 20831, 20865, 20880,
20958, 20972, 21009, 21027, 21037,
21056, 21061, 21075, 21080, 21090,
21101, 21113, 21125, 21185, 21232,
21240, 21269, 21318, 21326, 21350,
21368, 21400, 21405, 21504, 21547,
21563, 21566, 21612, 21682, 21717,
21729, 21735, 21741, 21753, 21772,
21781, 21792, 21798, 21803, 21811,
21824, 21837, 21851, 21866, 21873,
21879, 21888, 21895, 21904, 21909,
21917, 21922, 21930, 21932, 22088,
22090, 22092, 22098, 22100, 22102,
22139, 22182, 22193, 22201, 22212,
22269, 22271, 22275, 22291, 22294,
22296, 22300, 22490, 22491, 23370,
23387, 23392, 23399, 23401, 23403,
23410, 23411, 23528, 23543, 23569,
23609, 23620, 23630, 23645, 23664,
23673, 23675, 23677, 23679, 23694,
23707, 23729, 23734, 23738, 23744,
23751, 23755, 23758, 23760, 23771,
23781, 23821, 23827, 23829, 23835,
23838, 23847, 23861, 23863, 23865,
23867, 23869, 23874, 23879, 23889,
23898, 23900, 23903, 23905, 23907,
23909, 23914, 23919, 23924, 23926,
23938, 23943, 23945, 23947, 23949,
23951, 23953, 23955, 23957, 23968,
23973, 23982, 23991, 23996, 24000,
24002, 24004, 24014, 24019, 24024,
24029, 24039, 24058, 24069, 24071,
24081, 24099, 24118, 24140, 24145,
24148, 24152, 24155, 24162, 24168,
24170, 24172, 24177, 24182, 24191,
24201, 24203, 24206, 24208, 24224,
24229, 24249, 24272, 24275, 24288,
24300, 24305, 24307, 24309, 24311,
24313, 24315, 24317, 24319, 24324,
24334, 24347, 24360, 24365, 24367,
24369, 24378, 24390, 24402, 24414,
24433, 24435, 24437, 24439, 24441,
24493, 24506, 24535, 24540, 24544,
24549, 24551, 24559, 24569, 24577,
24582, 24587, 24598, 24608, 24618,
24620, 24622, 24624, 24655, 24657,
24662, 24664, 24666, 24669, 24690,
24701, 24714, 24716, 24718, 24720,
24722, 24724, 24726, 24728, 24730,
24741, 24743, 24745, 24747, 24756,
24759, 24761, 24763, 24765, 24776,
24807, 24809, 24811, 24813, 24828
\cs_new_protected:Npx 11, 308, 309,
313, 1789, 1825, 1830, 1916, 1976,
2336, 2340, 2345, 2482, 2486, 2487,
3456, 6679, 7208, 7793, 7795, 7797,
7799, 7801, 7810, 7812, 7814, 8038,
8040, 8042, 8044, 8046, 8055, 8057,
8059, 17855, 17869, 17871, 23502,
23509, 23515, 23521, 23592, 23601
\cs_new_protected_nopar:Nn
..... 13, 13, 1914, 1974
\cs_new_protected_nopar:Npn
..... 11, 11, 1789, 1806, 1819, 1823
\cs_new_protected_nopar:Npx
..... 11, 1789, 1819, 1824
\cs_set:Nn . 13, 13, 294, 294, 1914, 1974
\cs_set:Npn . 10, 11, 11, 89, 89, 291,
294, 457, 1352, 1381, 1387, 1388,
1389, 1390, 1391, 1392, 1393, 1394,
1395, 1396, 1397, 1398, 1399, 1400,
1401, 1402, 1403, 1404, 1405, 1406,
1407, 1408, 1409, 1410, 1411, 1412,
1413, 1414, 1415, 1416, 1418, 1572,
1577, 1582, 1587, 1594, 1600, 1601,
1613, 1617, 1619, 1621, 1679, 1681,
1683, 1685, 1687, 1689, 1691, 1693,
1742, 1789, 1805, 1813, 1813, 1914,
1974, 2773, 2838, 2958, 3111, 3479,
4629, 4637, 5539, 5547, 6245, 6317,
6739, 7200, 7206, 7338, 7613, 7615,
8911, 9761, 10954, 10962, 10971,
10988, 10996, 11024, 16202, 16976,
17274, 17275, 17276, 17586, 17587,
18119, 18120, 18135, 18136, 18401,
18425, 18463, 18951, 19230, 23420
\cs_set:Npx 11, 299, 534, 1352, 1603,
1813, 1814, 2840, 7113, 7118, 7135,
7136, 8957, 8958, 8959, 8960, 8961,
9748, 9754, 10004, 17031, 17037, 18632
\cs_set_eq:NN . 15, 15, 15, 90, 292,
300, 1526, 1831, 2340, 2363, 2486,
2546, 2547, 2548, 2549, 2665, 4269,
4270, 4272, 4382, 4383, 4394, 5486,
5681, 5683, 5692, 5693, 5694, 5695,
5711, 5716, 5731, 6682, 6876, 7107,
7111, 7132, 7134, 7211, 8963, 8964,
8965, 8973, 9999, 10014, 10046,
10057, 10067, 16445, 16446, 16451,
16629, 16661, 16685, 18396, 18402,
18422, 18954, 19589, 21557, 21558
\cs_set_nopar:Nn . . . 13, 13, 1914, 1974

\cs_set_nopar:Npn
 ... 10, 11, 11, 112, 291, 291, 291,
 1352, 1380, 1805, 1807, 9960, 10025
\cs_set_nopar:Npx
 ... 11, 1352, 1384, 1805, 1808, 2083,
 2260, 2565, 2567, 2569, 2583, 2585,
 2587, 2589, 2607, 2609, 2611, 2613
\cs_set_protected:Nn 13, 13, 1914, 1974
\cs_set_protected:Npn
 ... 10, 11, 11, 254, 291, 1352,
 1368, 1370, 1372, 1374, 1376, 1378,
 1382, 1420, 1422, 1424, 1426, 1428,
 1433, 1435, 1437, 1439, 1441, 1446,
 1458, 1473, 1490, 1507, 1513, 1519,
 1525, 1527, 1529, 1541, 1555, 1695,
 1697, 1700, 1701, 1702, 1706, 1707,
 1711, 1713, 1716, 1718, 1720, 1721,
 1722, 1725, 1738, 1740, 1744, 1754,
 1765, 1769, 1779, 1787, 1791, 1825,
 1825, 1858, 1879, 2632, 2639, 2661,
 2667, 2675, 2682, 3429, 3452, 3964,
 3975, 3991, 3998, 4002, 4012, 4015,
 4029, 4044, 4047, 4056, 4064, 4298,
 5534, 5708, 5713, 5718, 5723, 5728,
 5733, 5738, 5743, 6671, 6796, 6966,
 6984, 7716, 7779, 8025, 8027, 8291,
 8891, 8893, 9004, 9084, 9175, 9544,
 9992, 11417, 11481, 12010, 12078,
 12092, 12231, 12248, 12278, 12291,
 12350, 12365, 12382, 16447, 17867,
 17892, 17901, 18383, 18390, 18392,
 18394, 18397, 18399, 18403, 18405,
 18407, 18409, 18414, 18416, 18418,
 18420, 18423, 19110, 19111, 19475,
 20697, 20718, 22883, 22887, 22890,
 22916, 22927, 23058, 23061, 23083,
 23368, 23429, 23803, 23807, 23812
\cs_set_protected:Npx
 11, 240, 1352, 1709, 1825, 1826, 7901
\cs_set_protected_nopar:Nn
 ... 14, 14, 1914, 1974
\cs_set_protected_nopar:Npn
 ... 12, 12, 291, 1352, 1819, 1819
\cs_set_protected_nopar:Npx
 ... 12, 1352, 1819, 1820
\cs_show:N 16, 16, 16, 20, 136,
 297, 298, 298, 2036, 2046, 2047
\cs_to_str:N 4, 17,
 17, 41, 47, 284, 285, 285, 355, 1594,
 1607, 3927, 3928, 3929, 3930, 3931,
 3932, 3933, 3934, 3935, 3936, 3937,
 3938, 8896, 12525, 17045, 18365, 23287
\cs_undefine:N 15,
 15, 399, 535, 1847, 8063, 8064, 8065

cs internal commands:

__cs_count_signature:N . 22, 22, 1888
__cs_count_signature:nnN 1888
__cs_generate_from_signature:n .
 ... 1932, 1945
__cs_generate_from_signature:NNn .
 ... 1918, 1922
__cs_generate_from_signature:nnNNNn .
 ... 1924, 1927
__cs_generate_internal_variant:n .
 ... 2470, 2482
__cs_generate_internal_variant:wwNnn .
 ... 2484, 2500
__cs_generate_internal_variant:wwnw .
 ... 2482
__cs_generate_internal_variant_-
 loop:n 2482
__cs_generate_variant:N . 2329, 2336
__cs_generate_variant:nnNN
 ... 2332, 2364
__cs_generate_variant:Nnnw
 ... 2371, 2373
__cs_generate_variant:ww 2336
__cs_generate_variant:wwNN
 ... 310, 311, 311, 311, 2380, 2463
__cs_generate_variant:wwNw 2336
__cs_generate_variant_loop:nNnw
 ... 310, 311, 2381, 2393
__cs_generate_variant_loop_-
 end:nwwwNNnn
 ... 310, 311, 311, 2383, 2393
__cs_generate_variant_loop_-
 invalid:NNwNNnn 311, 2393
__cs_generate_variant_loop_-
 long:wNNnn 311, 2386, 2393
__cs_generate_variant_loop_-
 same:w 311, 2393
__cs_generate_variant_same:N
 ... 311, 2412, 2451
__cs_get_function_name:N 22, 22, 1619
__cs_get_function_signature:N
 ... 22, 22, 1619
__cs_parm_from_arg_count:nnTF
 ... 1448, 1858, 1902
__cs_parm_from_arg_count_-
 test:nnTF 1858
__cs_split_function:NN 22,
 22, 285, 1430, 1443, 1534, 1535,
 1603, 1620, 1622, 1889, 1924, 2330
__cs_split_function_auxi:w .. 1603
__cs_split_function_auxii:w .. 1603
__cs_tmp:w ... 23, 308, 313, 1789,
 1805, 1807, 1808, 1809, 1810, 1811,
 1812, 1813, 1814, 1815, 1816, 1817,

1818, 1819, 1820, 1821, 1822, 1823,
 1824, 1825, 1826, 1827, 1828, 1829,
 1830, 1914, 1950, 1951, 1952, 1953,
 1954, 1955, 1956, 1957, 1958, 1959,
 1960, 1961, 1962, 1963, 1964, 1965,
 1966, 1967, 1968, 1969, 1970, 1971,
 1972, 1973, 1974, 1982, 1983, 1984,
 1985, 1986, 1987, 1988, 1989, 1990,
 1991, 1992, 1993, 1994, 1995, 1996,
 1997, 1998, 1999, 2000, 2001, 2002,
 2003, 2004, 2005, 2340, 2363, 2471,
 2486, 2632, 2645, 2647, 9544, 9554
 __cs_to_str:N 284, 1594
 __cs_to_str:w 284, 284, 1594
 csc 188
 cscd 188
 \csname 14, 21, 39, 43, 49,
 68, 90, 92, 98, 123, 146, 150, 222, 321
 \currentgrouplevel 613
 \currentgroupstype 614
 \currentifbranch 615
 \currentiflevel 616
 \currentiftype 617

D

\day 322
 dd 191
 \deadcycles 323
 \def 74,
 75, 76, 105, 122, 124, 125, 143, 144,
 147, 163, 178, 207, 211, 236, 275, 324
 default commands:
 .default:n 165, 10144
 \defaulthyphenchar 325
 \defaultskewchar 326
 deg 190
 \delcode 327
 \delimiter 328
 \delimiterfactor 329
 \delimitershortfall 330
 deprecation commands:
 .deprecation_error: 1, 892, 21547
 deprecation internal commands:
 __deprecation_error:Nnn
 ... 21504, 21549, 21550, 21551,
 21552, 21553, 21554, 21555, 21556
 \detokenize 68, 222, 618
 \DH 23093
 \dh 23093
 dim commands:
 \dim_abs:n 148, 148, 9278
 \dim_add:Nn 148, 148, 9268
 \dim_case:nn 151, 151, 9354
 \dim_case:nnn 21517
 \dim_case:nnTF
 ... 151, 151, 9354, 9359, 9364, 21517
 \dim_compare:nNnTF
 ... 149, 149, 151, 151,
 151, 152, 9313, 9378, 9413, 9421,
 9430, 9436, 20883, 20886, 20889,
 20898, 20901, 20904, 20913, 20920,
 20978, 20983, 20990, 21103, 21115,
 21574, 21591, 21620, 21634, 21644
 \dim_compare:nTF
 ... 80, 150, 150, 152, 152, 152,
 152, 156, 9318, 9385, 9393, 9402, 9408
 \dim_compare_p:n 150, 150, 9318
 \dim_compare_p:nNn 149, 149, 9313
 \dim_const:Nn 147, 147, 9237,
 9479, 9480, 23659, 23661, 23852, 23854
 \dim_do_until:nn 152, 152, 9383
 \dim_do_until:nNnn 151, 151, 9411
 \dim_do_while:nn 152, 152, 9383
 \dim_do_while:nNnn 151, 151, 9411
 \dim_eval:n 149, 150, 152, 152, 152,
 161, 9357, 9362, 9367, 9372, 9439,
 9474, 20682, 20729, 20799, 20810,
 20823, 20825, 20827, 20829, 20835,
 20840, 20846, 20853, 20860, 21086,
 21087, 21356, 21357, 21358, 21627,
 21651, 21720, 21722, 21727, 21807,
 21808, 21815, 21816, 21892, 21899
 \dim_gadd:Nn 148, 9268
 .dim_gset:N 165, 10152
 \dim_gset:Nn 148, 9240, 9257
 \dim_gset_eq:NN 148, 9262
 \dim_gsub:Nn 148, 9268
 \dim_gzero:N 147, 9243, 9250
 \dim_gzero_new:N 147, 9247
 \dim_if_exist:NTF 147,
 147, 9248, 9250, 9253, 23632, 23840
 \dim_if_exist_p:N 147, 147, 9253
 \dim_log:N 154, 154, 9475
 \dim_log:n 154, 154, 9475
 \dim_max:nn 148, 148, 9278, 21786, 21790
 \dim_min:nn
 ... 148, 148, 9278, 21784, 21788, 21801
 \dim_new:N 147, 147, 147, 9229, 9239,
 9248, 9250, 9481, 9482, 9483, 9484,
 20269, 20270, 20271, 20272, 20273,
 20274, 20275, 20276, 20571, 20593,
 20594, 20597, 20598, 20599, 20600,
 21173, 21175, 21176, 21677, 21678,
 21679, 21680, 21681, 21822, 21823
 \dim_ratio:nn
 ... 149, 149, 149, 517, 9309, 9468
 .dim_set:N 165, 10152

\dim_set:Nn . 148, 148, [9257](#), 20290, 20291, 20292, 20324, 20335, 20408, 20409, 20410, 20425, 20498, 20499, 20500, 20502, 20504, 20506, 20667, 20713, 20885, 20890, 20900, 20905, 20915, 20922, 20935, 20961, 20981, 21040, 21041, 21043, 21045, 21063, 21064, 21174, 21277, 21278, 21329, 21330, 21331, 21333, 21723, 21755, 21763, 21774, 21775, 21776, 21777, 21783, 21785, 21787, 21789, 21794, 21800, 21856, 21858, 21860, 21868, 21870, 23655, 23656, 23850, 23851
\dim_set_eq:NN 148, 148, [9262](#), 20669, 20670, 20715, 20716, 23634, 23636, 23842, 23843
\dim_show:N 154, 154, [9471](#)
\dim_show:n 154, 154, 517, [9473](#), 9478
\dim_sub:Nn 148, 148, [9268](#)
\dim_to_decimal:n . 153, 153, [9444](#), 9460, 9465, 24452, 24453, 24454, 24455, 24456, 24458, 24580, 24585, 24591, 24592, 24593, 24594, 24603, 24604, 24605, 24696, 24715, 24823
\dim_to_decimal_in_bp:n 153, 153, [9459](#), 23534, 23535, 23536, 23685, 23686, 23687, 23872, 23877, 23883, 23884, 23885, 23893, 23894, 23933, 23937, 23941, 24035, 24089, 24090, 24091, 24175, 24180, 24186, 24187, 24195, 24196, 24197, 24295, 24299, 24303, 24409, 24700
\dim_to_decimal_in_sp:n 153, 153, 153, [9461](#)
\dim_to_decimal_in_unit:nn 153, 153, 153, [9463](#)
\dim_to_fp:n 154, 154, 588, 605, [9471](#), 15796, 20328, 20329, 20339, 20340, 20396, 20399, 20400, 20426, 20435, 20436, 20450, 20451, 20464, 20476, 20479, 20480, 20928, 20930, 20940, 20941, 20942, 20943, 20965, 20966, 20967, 20968, 21759, 21760, 21767, 21768, 21827, 21830, 21831, 21869, 21871
\dim_until_do:nn 152, 152, [9383](#)
\dim_until_do:nNnn ... 151, 151, [9411](#)
\dim_use:N 152, 152, 152, 152, 9281, 9287, 9288, 9289, 9295, 9296, 9297, 9321, 9342, 9440, 9447, 9447, 21071, 21725, 21727, 21733, 21739, 21748, 21749, 21750, 21877, 21884
\dim_while_do:nn 152, 152, [9383](#)
\dim_while_do:nNnn ... 152, 152, [9411](#)
\dim_zero:N .. 147, 147, [9243](#), 9248, 20293, 20411, 20501, 20876, 20877
\dim_zero:N 147
\dim_zero_new:N 147, 147, [9247](#)
\c_max_dim 154, 157, 5445, 5450, 5452, [9479](#), 9577, 21774, 21775, 21776, 21777, 21794
\g_tmipa_dim 154, [9481](#)
\l_tmipa_dim 154, [9481](#)
\g_tmppb_dim 154, [9481](#)
\l_tmppb_dim 154, [9481](#)
\c_zero_dim 154, 9243, [9479](#), 9576, 20194, 20214, 20559, 20883, 20886, 20889, 20898, 20901, 20904, 20913, 20920, 20978, 20983, 20990, 21578, 21589, 21595, 21607, 21620, 21624, 21632, 21634, 21638, 21644, 21654, 23546
dim internal commands:
__dim_abs:N 9278
__dim_case:nnTF 9354
__dim_case:nw 9354
__dim_case_end:nw 9354
__dim_compare:w 9318
__dim_compare:wNN 513, [9318](#)
__dim_compare_!:_w 9318
__dim_compare_<:_w 9318
__dim_compare_=:_w 9318
__dim_compare_>:_w 9318
__dim_compare_end:w 9326, 9352
__dim_eval:w 161, 161, 516, 588, [9226](#), 9258, 9269, 9274, 9281, 9287, 9288, 9289, 9295, 9296, 9297, 9312, 9315, 9321, 9342, 9347, 9440, 9447, 9462, 11541, 11554, 12117, 20068, 20070, 20072, 20081, 20083, 20085, 20087, 20165, 20190, 20209, 20236, 20265, 21570, 21572, 21616, 21618, 21699, 24043
__dim_eval_end: 161, 161, 9226, 9258, 9269, 9274, 9281, 9291, 9299, 9312, 9315, 9440, 9447, 9462, 20068, 20070, 20072, 20081, 20083, 20085, 20087, 20165, 20190, 20209, 20236, 20265, 21570, 21572, 21616, 21618, 21701, 24043
__dim_maxmin:wwN 9278
__dim_ratio:n 9309
__dim_to_decimal:w 9444
\dimen 331, 7023
\dimendef 332
\dimexpr 619
\directlua 16, 23, 53, 59, 61, 871
\disablecjktoken 1158

\discretionary 333
\displayindent 334
\displaylimits 335
\displaystyle 336
\displaywidowpenalties 620
\displaywidowpenalty 337
\displaywidth 338
\divide 339
\DJ 23094
\dj 23094
\doublehyphendemerits 340
\dp 341
\draftmode 952
driver internal commands:
 __driver_absolute_lengths:n
 24073, 24086
 __driver_box_use_clip:N 240,
 240, 21564, 23528, 23679, 24081, 24441
 __driver_box_use_rotate:Nn 240,
 240, 20311, 23543, 23694, 24099, 24493
 __driver_box_use_scale:Nnn
 241, 241, 20532,
 23569, 23707, 24114, 24118, 24506
 \g__driver_clip_path_int
 24441, 24628,
 24631, 24644, 24673, 24676, 24684
 \l__driver_color_current_tl
 23581, 23598, 23719, 23731,
 23809, 24130, 24142, 24525, 24537
 __driver_color_ensure_current: ..
 241, 241, 21406,
 23592, 23729, 23798, 24140, 24535
 __driver_color_reset: ..
 .. 23592, 23729, 23798, 24140, 24535
 \l__driver_color_stack_int
 23591, 23597, 23606
 \l__driver_cos_fp 23543
 __driver_draw_add_to_path:n ..
 24577, 24623
 __driver_draw_begin: ..
 241, 241, 23861, 24155, 24544
 __driver_draw_cap_but: ..
 .. 244, 244, 244, 23926, 24288, 24690
 __driver_draw_cap_rectangle: ..
 244, 23926, 24288, 24690
 __driver_draw_cap_round: ..
 244, 23926, 24288, 24690
 __driver_draw_clip: ..
 243, 243, 23903, 24206, 24622
 \g__driver_draw_clip_bool
 24206, 24622
 __driver_draw_closepath: ..
 242, 242, 23903, 24206, 24622
 __driver_draw_closestroke: ..
 242, 242, 23903, 24206, 24622
 __driver_draw_color_cmyk:nnnn ..
 244, 244, 23957, 24319, 24730
 __driver_draw_color_cmyk_- aux:nnnn 23957
 __driver_draw_color_cmyk_- aux:NNnnnn
 24730, 24742, 24744, 24746
 __driver_draw_color_cmyk_- fill:nnnn 244, 23957, 24319, 24730
 __driver_draw_color_cmyk_- stroke:nnnn 244, 23957, 24319, 24730
 __driver_draw_color_gray:n ..
 ... 244, 244, 244, 23957, 24319, 24730
 __driver_draw_color_gray_aux:n ..
 23957
 __driver_draw_color_gray_- aux:NNn .. 24747, 24760, 24762, 24764
 __driver_draw_color_gray_- aux:nNN .. 24751, 24756, 24758
 __driver_draw_color_gray_fill:n ..
 244, 23957, 24319, 24730
 __driver_draw_color_gray_- stroke:n .. 244, 23957, 24319, 24730
 __driver_draw_color_reset: .. 24319
 __driver_draw_color_rgb:nnn ..
 244, 23957, 24319, 24730
 __driver_draw_color_rgb_aux:nnn ..
 23957
 __driver_draw_color_rgb_- auxi:NNnnn ..
 24765, 24808, 24810, 24812
 __driver_draw_color_rgb_- auxii:nnm .. 24734
 __driver_draw_color_rgb_- auxii:nnnNN .. 24757, 24769, 24776
 __driver_draw_color_rgb_- fill:nnn .. 244, 23957, 24319, 24730
 __driver_draw_color_rgb_- stroke:nnn .. 244, 23957, 24319, 24730
 __driver_draw_curveto:nnnnnn ..
 242, 242, 23869, 24172, 24577
 __driver_draw_dash:n ..
 23926, 24288, 24690
 __driver_draw_dash:nn ..
 243, 243, 23926, 24288, 24690
 __driver_draw_dash_aux:nn .. 24690
 __driver_draw_discardpath: ..
 243, 243, 23903, 24206, 24622
 __driver_draw_end: ..
 241, 241, 23861, 24155, 24544

```

\g__driver_draw_eor_bool . 23898,
  23912, 23917, 23922, 24201, 24217,
  24234, 24242, 24254, 24265, 24281
\__driver_draw_evenodd_rule: ...
  .... 243, 23898, 24201, 24618
\__driver_draw_fill: ...
  .... 243, 243, 23903, 24206, 24622
\__driver_draw_fillstroke: ...
  .... 243, 23903, 24206, 24622
\__driver_draw_hbox:Nnnnnn ...
  .... 245, 245, 24039, 24414, 24828
\__driver_draw_join_bevel: ...
  .... 244, 23926, 24288, 24690
\__driver_draw_join_miter: ...
  .... 244, 23926, 24288, 24690
\__driver_draw_join_round: ...
  .... 244, 23926, 24288, 24690
\__driver_draw_lineto:nn ...
  .... 242, 242, 23869, 24172, 24577
\__driver_draw_linewidth:n ...
  .... 243, 243, 23926, 24288, 24690
\__driver_draw_literal:n ...
  .... 23859, 23866, 23868,
  23871, 23876, 23881, 23891, 23904,
  23906, 23908, 23911, 23916, 23921,
  23925, 23928, 23940, 23944, 23946,
  23948, 23950, 23952, 23954, 23956,
  23970, 23975, 23984, 23998, 24001,
  24003, 24016, 24021, 24026, 24031,
  24152, 24169, 24171, 24174, 24179,
  24184, 24193, 24207, 24210, 24211,
  24212, 24215, 24221, 24231, 24232,
  24237, 24240, 24246, 24251, 24252,
  24257, 24258, 24259, 24260, 24263,
  24269, 24279, 24285, 24290, 24302,
  24306, 24308, 24310, 24312, 24314,
  24316, 24318, 24321, 24322, 24326,
  24336, 24349, 24362, 24366, 24368,
  24371, 24380, 24392, 24404, 24542,
  24563, 24571, 24629, 24648, 24674
\__driver_draw_miterlimit:n ...
  .... 244, 244, 23926, 24288, 24690
\__driver_draw_move:nn ...
  .... 242
\__driver_draw_moveto:nn ...
  .... 242, 23869, 24172, 24577
\__driver_draw_nonzero_rule: ...
  .... 243, 243, 23898, 24201, 24618
\__driver_draw_path:n ...
  .... 24622
\g__driver_draw_path_int ...
  .... 24622
\g__driver_draw_path_tl ...
  .... 24577,
  24633, 24649, 24651, 24678, 24687
\__driver_draw_rectangle:nnnn ...
  .... 242, 242, 23869, 24172, 24577
\__driver_draw_scope:n ...
  .... 24547, 24551,
  24619, 24621, 24641, 24681, 24703,
  24715, 24717, 24719, 24721, 24723,
  24725, 24727, 24729, 24778, 24815
\__driver_draw_scope_begin: 241,
  241, 23862, 23865, 24168, 24546, 24551
\__driver_draw_scope_end: ...
  .... 241,
  241, 23864, 23865, 24168, 24550, 24551
\g__driver_draw_scope_int ...
  .... 24551
\l__driver_draw_scope_int ...
  .... 24551
\__driver_draw_stroke: ...
  .... 242, 242, 243, 23903, 24206, 24622
\__driver_draw_transformcm:nnnnn ...
  .... 245, 245, 24029,
  24045, 24402, 24418, 24813, 24831
\l__driver_image_attr_tl ...
  .... 23608, 23612, 23623, 23632, 23635,
  23637, 23657, 23659, 23661, 23667
\__driver_image_getbb_auxi:n ...
  .... 23609
\__driver_image_getbb_auxi:nN ...
  .... 23821
\__driver_image_getbb_auxii:n ...
  .... 23609
\__driver_image_getbb_auxii:nn ...
  .... 23821
\__driver_image_getbb_auxii:nnN ...
  .... 23821
\__driver_image_getbb_auxiii:nNn ...
  .... 23821
\__driver_image_getbb_auxvi:nNm ...
  .... 23845, 23847
\__driver_image_getbb_eps:n ...
  .... 23736, 24147
\__driver_image_getbb_jpg:n ...
  .... 23609, 23736, 23821
\__driver_image_getbb_pdf:n ...
  .... 23609, 23736, 23821
\__driver_image_getbb_png:n ...
  .... 23609, 23736, 23821
\__driver_image_include_auxi:nn ...
  .... 23751
\__driver_image_include_auxii:nnn ...
  .... 23751
\__driver_image_include_auxiii:nn ...
  .... 23751
\__driver_image_include_auxii:nnn ...
  .... 23781
\__driver_image_include_eps:n ...
  .... 23751, 24148
\__driver_image_include_jpg:n ...
  .... 23664, 23751
\__driver_image_include_pdf:n ...
  .... 23664, 23751
\__driver_image_include_png:n ...
  .... 23664, 23751
\g__driver_image_int ...
  .... 23750, 23783, 23784

```

__driver_literal:
 23502, 23531, 23673, 23682,
 23859, 24058, 24084, 24102, 24121,
 24433, 24436, 24438, 24440, 24444,
 24446, 24463, 24542, 24832, 24845
__driver_matrix:n 23521, 23551, 23572
\g__driver_path_int 24637, 24654
__driver_scope_begin: 965, 23509,
 23530, 23545, 23571, 23675, 23681,
 23696, 23709, 24044, 24069, 24083,
 24101, 24120, 24416, 24435, 24830
__driver_scope_begin:n 24439,
 24465, 24473, 24477, 24495, 24508
__driver_scope_end: 23509,
 23540, 23565, 23579, 23675, 23691,
 23705, 23717, 24048, 24069, 24096,
 24110, 24128, 24429, 24435, 24487,
 24488, 24489, 24504, 24523, 24846
\l__driver_sin_fp 23543
__driver_tmp:w 23803, 23816
\l__driver_tmp_box 24039
\dtou 1125
\dump 342
\dvientension 872
\dvifeedback 873
\dvivivable 874

E

\edef 4, 106, 131, 209, 343
\efcode 779
\else 15, 22, 44, 46,
 91, 94, 95, 99, 100, 161, 165, 180, 344
else commands:
\else: 21, 79,
 79, 79, 79, 90, 96, 96, 145, 145,
 160, 217, 217, 217, 279, 286, 308,
 326, 326, 338, 338, 625, 1297, 1339,
 1495, 1627, 1630, 1639, 1645, 1655,
 1658, 1667, 1673, 1719, 1853, 1874,
 1883, 1894, 1947, 1948, 2009, 2104,
 2341, 2397, 2398, 2399, 2455, 2458,
 2890, 2903, 2916, 2926, 2942, 2975,
 3236, 3265, 3288, 3304, 3312, 3322,
 3335, 3351, 3502, 3511, 3524, 3529,
 3685, 3738, 3741, 3744, 3756, 3770,
 3979, 3985, 3995, 4021, 4033, 4038,
 4052, 4285, 4322, 4331, 4628, 4649,
 4665, 4668, 4723, 4824, 4851, 4888,
 4896, 5182, 5215, 5266, 5404, 5489,
 5512, 5521, 5557, 5583, 5622, 5630,
 5754, 5789, 5818, 5837, 5992, 5994,
 5996, 5998, 6142, 6157, 6179, 6195,
 6706, 6710, 6713, 6721, 6727, 6763,
 6858, 6863, 6868, 6873, 6880, 6887,

6892, 6897, 6902, 6907, 6912, 6917,
6922, 6927, 6944, 6951, 6957, 6960,
6995, 6998, 7060, 7069, 7077, 7086,
7152, 7166, 7175, 7185, 7195, 7498,
8728, 9284, 9305, 9316, 9326, 9353,
9540, 9687, 9711, 9729, 9740, 9750,
9760, 9771, 10669, 10679, 10680,
10695, 10705, 10786, 10848, 10851,
10865, 10883, 10887, 11104, 11107,
11155, 11168, 11188, 11216, 11217,
11238, 11252, 11275, 11276, 11318,
11336, 11371, 11375, 11423, 11440,
11446, 11450, 11454, 11576, 11608,
11616, 11649, 11653, 11665, 11675,
11685, 11716, 11729, 11764, 11774,
11793, 11806, 11819, 11823, 11834,
11857, 11874, 11886, 11900, 11916,
11924, 11926, 11936, 11947, 11963,
11979, 11985, 11990, 11997, 12021,
12044, 12071, 12189, 12193, 12200,
12220, 12240, 12256, 12302, 12333,
12342, 12357, 12373, 12393, 12434,
12448, 12481, 12483, 12489, 12504,
12617, 12628, 12649, 12652, 12655,
12658, 12669, 12678, 12681, 12827,
12840, 12843, 12850, 12868, 12882,
12899, 12960, 12963, 12972, 12984,
12995, 13009, 13022, 13062, 13096,
13116, 13153, 13171, 13174, 13180,
13194, 13229, 13247, 13250, 13253,
13256, 13315, 13388, 13456, 13457,
13466, 13501, 13584, 13588, 13592,
13654, 13688, 13915, 13944, 13948,
14107, 14116, 14168, 14179, 14195,
14203, 14262, 14341, 14352, 14357,
14391, 14404, 14416, 14422, 14595,
14602, 14624, 14652, 14667, 14671,
14693, 14724, 14727, 14752, 14755,
14792, 14802, 14806, 14809, 14859,
14874, 14889, 14904, 14919, 14934,
14955, 15000, 15226, 15264, 15265,
15274, 15318, 15370, 15384, 15385,
15386, 15489, 15511, 15526, 15544,
15592, 15608, 15758, 15807, 15812,
15909, 15930, 15952, 16135, 16136,
16607, 16610, 16613, 16623, 16638,
16669, 16695, 16707, 16739, 16747,
16749, 16751, 16752, 16753, 16754,
16755, 16757, 16775, 16796, 16800,
16873, 16877, 17125, 17126, 17130,
17131, 17145, 17152, 17351, 17361,
17408, 17418, 17430, 17431, 17433,
17435, 17438, 17439, 17442, 17443,
17452, 17454, 17456, 17459, 17460,

\etex_eTeXrevision:D	623
\etex_eTeXversion:D	624
\etex_everyeof:D	625, 2709, 22280, 22305
\etex_firstmarks:D	626
\etex_fontchardp:D	627
\etex_fontcharht:D	628
\etex_fontcharic:D	629
\etex_fontcharwd:D	630
\etex_glueexpr:D .	631, 9514, 9525, 9530, 9549, 9556, 9562, 9565, 15801
\etex_glueshrink:D	632, 22120
\etex_glueshrinkorder:D	633
\etex_gluestretch:D	634, 16792, 16798, 22119
\etex_gluestretchorder:D	635
\etex_gluetomu:D	636
\etex_ifcsname:D	637, 1312
\etex_ifdefined:D .	638, 795, 1169, 1178, 1181, 1187, 1241, 1242, 1249, 1261, 1268, 1281, 1289, 1311, 1347
\etex_iffontchar:D	639
\etex_interactionmode:D	640, 20130, 20133, 20135
\etex_interlinepenalties:D	641
\etex_lastlinefit:D	642
\etex_lastnodedtype:D	643
\etex_marks:D	644
\etex_middle:D	645
\etex_muexpr:D	646, 9612, 9623, 9628, 9633
\etex_mutoglu:D	647
\etex_numexpr:D	648, 3953, 4615
\etex_pagediscards:D	649
\etex_parshapedimen:D	650
\etex_parshapeindent:D	651
\etex_parshapelength:D	652
\etex_predisplaydirection:D ..	653
\etex_protected:D	654, 1354, 1356, 1358, 1359, 1360, 1361, 1362, 1363, 1364, 1365, 1373, 1375, 1377, 1379, 21506
\etex_readline:D	655, 8743
\etex_savinghyphcodes:D	656
\etex_savingvdiscards:D	657
\etex_scantokens:D	658, 2726, 2776, 2791
\etex_showgroups:D	659
\etex_showifs:D	660
\etex_showtokens:D	341, 485, 661, 1285, 8401
\etex_splitbotmarks:D	662
\etex_splitediscards:D	663
\etex_splitfirstmarks:D	664

\etex_TeXXeTstate:D 665
 \etex_topmarks:D 666
 \etex_tracingassigns:D 667
 \etex_tracinggroups:D 668
 \etex_tracingifs:D 669
 \etex_tracingnesting:D 670
 \etex_tracingscantokens:D 671
 \etex_unexpanded:D 672, 1278, 1317,
 2304, 2307, 2310, 2315, 3197, 3223,
 3247, 4812, 6777, 22231, 22322, 23112
 \etex_unless:D 673, 1302
 \etex_widowpenalties:D 674
 \eTeXrevision 623
 \eTeXversion 624
 \etoksapp 875
 \etokspre 876
 \euc 1126
 \everycr 357
 \everydisplay 358
 \everyeof 625
 \everyhbox 359
 \everyjob 66, 67, 360
 \everymath 361
 \everypar 362
 \everyvbox 363
 ex 191
 \exhyphenpenalty 364
 exp 187
 exp commands:
 \exp:w
 . 31, 31, 31, 32, 32, 32, 32, 32, 32,
 279, 284, 300, 300, 301, 301, 308,
 334, 334, 340, 349, 416, 416, 482,
 578, 578, 578, 578, 580, 581, 581,
 585, 602, 606, 607, 914, 1318, 1417,
 1419, 2077, 2089, 2095, 2135, 2139,
 2144, 2150, 2156, 2162, 2174, 2179,
 2181, 2188, 2244, 2251, 2256, 2264,
 2266, 2269, 2276, 2282, 2291, 2307,
 2311, 2316, 2318, 2982, 2987, 2992,
 2997, 3199, 3357, 3537, 3542, 3547,
 3552, 3569, 3574, 3579, 3584, 3668,
 3677, 3729, 4857, 4862, 4867, 4872,
 5946, 6691, 6697, 7194, 8295, 9050,
 9325, 9356, 9361, 9366, 9371, 10708,
 10909, 11131, 11249, 11250, 11251,
 11252, 11363, 11381, 11422, 11466,
 11473, 11478, 11487, 11503, 11511,
 11561, 11574, 11575, 11584, 11596,
 11614, 11615, 11635, 11648, 11652,
 11674, 11702, 11715, 11728, 11752,
 11763, 11773, 11792, 11805, 11818,
 11821, 11833, 11856, 11885, 11899,
 11915, 11935, 11946, 11952, 11962,
 12008, 12018, 12033, 12041, 12058,
 12074, 12081, 12118, 12127, 12136,
 12141, 12143, 12163, 12166, 12173,
 12174, 12223, 12239, 12261, 12269,
 12276, 12299, 12330, 12372, 12390,
 12460, 12472, 12501, 12503, 12507,
 12509, 12519, 12539, 12611, 12625,
 12635, 12744, 12745, 12746, 12893,
 12896, 12904, 12928, 12936, 13888,
 14415, 14437, 14597, 14770, 14976,
 15368, 15369, 15373, 15643, 15647,
 15684, 15688, 15730, 15749, 15753,
 15789, 15793, 15859, 15955, 15983,
 15990, 16010, 16026, 22226, 22233,
 22324, 22361, 23114, 23142, 23328
 \exp_after:wN 30, 30, 31, 32, 32, 279,
 299, 301, 339, 390, 462, 485, 554,
 556, 556, 556, 578, 578, 578, 578,
 578, 578, 578, 578, 578, 580, 580,
 581, 581, 638, 638, 695, 695, 695,
 745, 769, 834, 1315, 1331, 1333,
 1338, 1340, 1417, 1419, 1463, 1476,
 1494, 1496, 1546, 1551, 1558, 1598,
 1602, 1605, 1606, 1638, 1640, 1643,
 1666, 1668, 1671, 1852, 1854, 1862,
 1882, 1884, 1919, 1979, 2054, 2064,
 2071, 2073, 2076, 2077, 2084, 2088,
 2089, 2094, 2095, 2100, 2105, 2107,
 2110, 2118, 2120, 2122, 2124, 2126,
 2129, 2130, 2131, 2135, 2138, 2143,
 2148, 2149, 2150, 2154, 2155, 2156,
 2160, 2161, 2162, 2166, 2167, 2168,
 2172, 2173, 2174, 2178, 2179, 2180,
 2181, 2185, 2186, 2187, 2188, 2192,
 2193, 2194, 2199, 2200, 2201, 2202,
 2206, 2207, 2208, 2209, 2243, 2244,
 2247, 2250, 2251, 2255, 2256, 2264,
 2266, 2267, 2269, 2271, 2274, 2275,
 2280, 2281, 2285, 2288, 2289, 2290,
 2294, 2301, 2303, 2304, 2305, 2307,
 2310, 2315, 2330, 2331, 2332, 2333,
 2338, 2342, 2369, 2376, 2396, 2510,
 2635, 2724, 2725, 2774, 2775, 2789,
 2790, 2849, 2850, 2851, 2856, 2900,
 2913, 2914, 2972, 2973, 3063, 3170,
 3197, 3228, 3233, 3234, 3235, 3237,
 3250, 3260, 3279, 3301, 3311, 3314,
 3331, 3332, 3342, 3347, 3348, 3363,
 3364, 3365, 3494, 3598, 3621, 3655,
 3656, 3667, 3668, 3676, 3684, 3686,
 3692, 3695, 3713, 3714, 3715, 3727,
 3728, 3755, 3757, 3763, 3769, 3782,
 3802, 3813, 3829, 3837, 3845, 3852,
 3859, 3871, 3978, 3980, 3986, 4067,

4181, 4183, 4195, 4203, 4302, 4340,
4352, 4363, 4364, 4365, 4386, 4387,
4434, 4463, 4464, 4465, 4535, 4536,
4539, 4623, 4628, 4631, 4632, 4639,
4640, 4656, 4657, 4678, 4679, 4796,
4801, 4806, 4829, 4831, 4958, 4959,
4960, 5154, 5182, 5187, 5215, 5228,
5238, 5265, 5267, 5268, 5276, 5293,
5337, 5480, 5487, 5520, 5522, 5529,
5576, 5582, 5584, 5608, 5824, 5827,
5831, 5947, 6143, 6158, 6180, 6196,
6253, 6261, 6266, 6441, 6442, 6445,
6446, 6691, 6692, 6697, 6698, 6699,
6751, 6759, 6777, 6778, 6821, 6822,
6823, 6934, 6955, 7002, 7038, 7067,
7068, 7070, 7076, 7079, 7151, 7153,
7163, 7164, 7165, 7167, 7173, 7174,
7176, 7183, 7184, 7186, 7192, 7193,
7196, 7273, 7282, 7291, 7341, 7535,
7725, 8296, 8297, 8298, 8299, 8311,
8401, 8402, 8427, 8440, 8673, 8824,
8992, 9033, 9042, 9045, 9048, 9049,
9051, 9090, 9147, 9178, 9280, 9284,
9287, 9288, 9295, 9296, 9320, 9325,
9338, 9341, 9446, 9548, 9665, 9666,
9675, 9677, 9694, 9699, 9701, 9718,
9727, 9731, 9732, 9739, 9741, 9742,
9751, 9769, 9772, 9781, 9847, 9889,
10035, 10036, 10343, 10423, 10424,
10443, 10453, 10466, 10467, 10500,
10668, 10670, 10671, 10682, 10683,
10684, 10694, 10696, 10704, 10706,
10713, 10714, 10715, 10716, 10717,
10718, 10723, 10724, 10725, 10726,
10727, 10728, 10729, 10800, 10802,
10829, 10833, 10858, 10862, 10879,
10886, 10888, 10909, 10968, 10976,
10993, 11002, 11048, 11131, 11200,
11201, 11202, 11254, 11264, 11283,
11289, 11315, 11316, 11319, 11330,
11334, 11341, 11342, 11353, 11354,
11363, 11370, 11372, 11373, 11381,
11407, 11422, 11440, 11441, 11444,
11445, 11447, 11448, 11452, 11453,
11455, 11456, 11465, 11466, 11473,
11478, 11487, 11501, 11502, 11507,
11508, 11510, 11517, 11518, 11520,
11539, 11540, 11544, 11547, 11552,
11553, 11555, 11559, 11560, 11562,
11571, 11572, 11573, 11574, 11577,
11578, 11579, 11582, 11596, 11613,
11614, 11624, 11625, 11635, 11647,
11651, 11664, 11666, 11674, 11684,
11686, 11692, 11697, 11699, 11701,
11707, 11708, 11712, 11714, 11726,
11727, 11749, 11751, 11757, 11760,
11762, 11766, 11771, 11776, 11777,
11787, 11788, 11790, 11791, 11794,
11798, 11803, 11817, 11820, 11832,
11841, 11848, 11849, 11850, 11851,
11853, 11855, 11866, 11867, 11868,
11869, 11871, 11873, 11875, 11876,
11877, 11883, 11884, 11894, 11898,
11899, 11901, 11902, 11903, 11908,
11914, 11925, 11927, 11934, 11935,
11937, 11938, 11945, 11951, 11961,
12014, 12015, 12016, 12017, 12031,
12032, 12034, 12039, 12040, 12055,
12057, 12074, 12081, 12115, 12116,
12117, 12123, 12124, 12125, 12126,
12132, 12133, 12134, 12135, 12142,
12154, 12155, 12161, 12162, 12164,
12165, 12171, 12172, 12187, 12188,
12190, 12191, 12198, 12199, 12201,
12204, 12216, 12218, 12219, 12221,
12222, 12224, 12237, 12238, 12239,
12241, 12242, 12243, 12253, 12254,
12255, 12260, 12262, 12273, 12274,
12275, 12283, 12284, 12286, 12296,
12297, 12298, 12301, 12303, 12304,
12305, 12328, 12329, 12332, 12334,
12335, 12336, 12355, 12356, 12358,
12359, 12360, 12370, 12371, 12372,
12374, 12375, 12376, 12388, 12389,
12392, 12394, 12395, 12396, 12416,
12417, 12418, 12419, 12420, 12421,
12422, 12432, 12433, 12435, 12436,
12437, 12443, 12453, 12454, 12455,
12456, 12457, 12458, 12459, 12460,
12465, 12466, 12467, 12468, 12469,
12470, 12471, 12487, 12488, 12490,
12491, 12498, 12499, 12500, 12505,
12506, 12508, 12517, 12518, 12533,
12535, 12536, 12537, 12610, 12624,
12634, 12642, 12643, 12743, 12826,
12828, 12867, 12869, 12891, 12892,
12895, 12902, 12903, 12906, 12907,
12914, 12919, 12926, 12927, 12934,
12935, 12968, 12969, 12970, 12972,
12983, 13008, 13010, 13016, 13017,
13021, 13024, 13046, 13048, 13061,
13063, 13069, 13071, 13074, 13080,
13082, 13084, 13085, 13086, 13088,
13093, 13095, 13097, 13101, 13104,
13110, 13111, 13115, 13117, 13118,
13119, 13127, 13129, 13130, 13137,
13143, 13150, 13151, 13156, 13157,
13158, 13159, 13178, 13179, 13180,

13186, 13187, 13188, 13193, 13195,
13203, 13205, 13207, 13208, 13210,
13221, 13223, 13225, 13226, 13231,
13282, 13283, 13290, 13291, 13293,
13295, 13297, 13299, 13301, 13303,
13305, 13314, 13316, 13322, 13324,
13326, 13327, 13328, 13334, 13336,
13338, 13339, 13340, 13361, 13362,
13365, 13373, 13375, 13379, 13380,
13381, 13382, 13387, 13389, 13395,
13398, 13401, 13404, 13412, 13415,
13418, 13421, 13428, 13430, 13436,
13444, 13446, 13448, 13465, 13467,
13474, 13476, 13479, 13485, 13487,
13489, 13490, 13491, 13493, 13507,
13508, 13511, 13529, 13531, 13533,
13545, 13548, 13551, 13554, 13557,
13560, 13563, 13566, 13570, 13582,
13586, 13590, 13593, 13608, 13614,
13616, 13618, 13628, 13652, 13655,
13667, 13669, 13673, 13674, 13675,
13677, 13678, 13680, 13686, 13687,
13693, 13696, 13697, 13698, 13699,
13712, 13717, 13719, 13726, 13729,
13732, 13735, 13738, 13741, 13749,
13750, 13762, 13770, 13772, 13782,
13784, 13791, 13800, 13802, 13805,
13808, 13811, 13814, 13827, 13829,
13837, 13839, 13847, 13849, 13858,
13861, 13864, 13871, 13886, 13887,
13904, 13906, 13907, 13964, 13977,
13979, 13985, 13998, 14000, 14002,
14026, 14040, 14042, 14049, 14051,
14091, 14092, 14093, 14095, 14096,
14097, 14099, 14100, 14106, 14108,
14109, 14115, 14117, 14118, 14119,
14120, 14132, 14138, 14140, 14175,
14182, 14189, 14209, 14210, 14212,
14214, 14216, 14229, 14234, 14235,
14236, 14237, 14238, 14242, 14247,
14249, 14255, 14261, 14263, 14264,
14270, 14271, 14272, 14273, 14274,
14275, 14276, 14277, 14282, 14284,
14286, 14288, 14290, 14294, 14296,
14298, 14300, 14302, 14304, 14322,
14326, 14334, 14335, 14340, 14342,
14351, 14354, 14355, 14356, 14358,
14359, 14360, 14368, 14374, 14386,
14389, 14390, 14392, 14393, 14417,
14418, 14421, 14423, 14439, 14443,
14444, 14445, 14461, 14467, 14476,
14477, 14478, 14486, 14502, 14518,
14534, 14550, 14566, 14592, 14596,
14597, 14601, 14603, 14639, 14645,
14646, 14648, 14650, 14651, 14653,
14654, 14664, 14665, 14668, 14669,
14670, 14672, 14673, 14674, 14691,
14692, 14694, 14695, 14701, 14703,
14706, 14709, 14712, 14715, 14723,
14726, 14728, 14731, 14738, 14742,
14750, 14751, 14754, 14756, 14758,
14762, 14766, 14771, 14772, 14780,
14781, 14782, 14783, 14810, 14811,
14812, 14948, 14949, 14950, 14952,
14970, 14971, 14972, 14973, 14974,
14975, 14982, 14991, 14998, 14999,
15009, 15173, 15174, 15175, 15196,
15197, 15207, 15216, 15224, 15225,
15227, 15228, 15233, 15234, 15243,
15250, 15259, 15260, 15273, 15275,
15303, 15304, 15313, 15316, 15341,
15347, 15348, 15367, 15373, 15375,
15401, 15402, 15404, 15418, 15419,
15427, 15437, 15471, 15474, 15484,
15485, 15488, 15490, 15496, 15510,
15512, 15553, 15556, 15576, 15638,
15642, 15647, 15650, 15671, 15672,
15679, 15683, 15688, 15691, 15720,
15721, 15727, 15728, 15729, 15736,
15744, 15748, 15753, 15756, 15772,
15773, 15784, 15788, 15793, 15798,
15799, 15800, 15806, 15808, 15811,
15813, 15855, 15857, 15858, 15879,
15899, 15905, 15908, 15911, 15914,
15921, 15927, 15931, 15941, 15950,
15955, 15957, 15978, 15988, 15989,
15997, 16004, 16013, 16014, 16022,
16126, 16264, 16265, 16274, 16293,
16294, 16309, 16310, 16311, 16532,
16538, 16540, 16551, 16565, 16620,
16621, 16622, 16623, 16629, 16630,
16647, 16661, 16662, 16692, 16721,
16736, 16738, 16740, 16761, 16771,
16779, 16789, 16799, 16801, 16803,
16832, 16862, 16872, 16875, 16876,
16878, 16879, 16887, 16888, 16902,
16951, 16959, 16962, 17000, 17019,
17032, 17033, 17038, 17039, 17042,
17045, 17082, 17089, 17096, 17102,
17109, 17117, 17151, 17153, 17162,
17281, 17310, 17327, 17350, 17352,
17353, 17360, 17363, 17364, 17388,
17410, 17420, 17520, 17522, 17528,
17531, 17533, 17540, 17543, 17545,
17552, 17554, 17560, 17563, 17566,
17864, 17936, 17948, 18069, 18072,
18240, 18248, 18319, 18365, 18575,
18873, 18979, 19003, 19056, 19082,

19146, 19147, 19155, 19158, 19288,
 19289, 19292, 19293, 19301, 19303,
 19304, 19313, 19326, 19328, 19385,
 19620, 19653, 19655, 21410, 21939,
 21940, 22015, 22016, 22039, 22051,
 22060, 22066, 22069, 22231, 22251,
 22283, 22284, 22322, 22359, 22376,
 22377, 22533, 22534, 22555, 22556,
 22604, 22625, 22658, 22750, 22800,
 22894, 22896, 22902, 22924, 22937,
 22939, 22950, 22952, 23074, 23112,
 23140, 23157, 23158, 23202, 23203,
 23253, 23323, 23325, 23326, 23327,
 23333, 23334, 23336, 23342, 23343,
 23377, 23382, 23384, 23390, 23816
`\exp_args:N` 30
`\exp_args:cc` 27, 1330, 1499, 1510, 1516, 1522, 2123
`\exp_args:Nc` 27, 27, 296, 1330, 1334,
 1342, 1766, 1788, 1806, 1832, 1837,
 1844, 1899, 1911, 1978, 2011, 2012,
 2013, 2014, 2031, 2035, 2123, 2818,
 3034, 3434, 3458, 5498, 5536, 8758,
 10526, 11272, 11497, 12247, 12266,
 12290, 12309, 12311, 12313, 12315,
 12317, 12319, 12321, 12364, 12380,
 12381, 12400, 12402, 16821, 18195
`\exp_args:Ncc` 28, 1834, 1838,
 1846, 2019, 2020, 2021, 2022, 2123
`\exp_args:Nccc` 28, 2123
`\exp_args:Ncco` 28, 2183
`\exp_args:Nccx` 29, 2227
`\exp_args:Ncf` 28, 2146
`\exp_args:NcNc` 28, 2183
`\exp_args:NcNo` 28, 2183
`\exp_args:Ncnx` 29, 2227
`\exp_args:Nco` 28, 303, 2146
`\exp_args:Ncx` 28, 2212, 8304
`\exp_args:Nf` 27, 27,
 2134, 3369, 3370, 3386, 3645, 3647,
 3703, 3705, 3721, 4437, 4438, 4454,
 4858, 4863, 4868, 4873, 5028, 5097,
 5099, 5117, 5126, 5137, 5146, 5284,
 5301, 5463, 6430, 6481, 6495, 6516,
 8309, 9082, 9357, 9362, 9367, 9372,
 12787, 18729, 21661, 21956, 21969,
 21994, 21996, 22250, 22707, 22715,
 22770, 22820, 23304, 23312, 23316
`\exp_args:Nff` .. 28, 2212, 5438, 23310
`\exp_args:Nfo` 28, 2212
`\exp_args:NNc` 28, 28,
 275, 1833, 1836, 1845, 1913, 2015,
 2016, 2017, 2018, 2045, 2123, 4989,
 4996, 8311, 8673, 8824, 12797, 12804
`\exp_args:Nnc` 28, 2212
`\exp_args:NNf` 28, 2146,
 4982, 8672, 8823, 8979, 14942, 14943
`\exp_args:Nnf` 28, 2212
`\exp_args:Nnnc` 28, 2227
`\exp_args:NNNo` 28,
 2118, 2715, 19527, 22286, 22309
`\exp_args:NNno` 28, 2227
`\exp_args:Nnno` 28, 2227
`\exp_args:NNNV` 28, 2183
`\exp_args:NNNx` 29, 2227
`\exp_args:NNnx` 29, 29, 2227
`\exp_args:Nnnx` 29, 2227
`\exp_args:NNo`
 24, 24, 24, 28, 2118, 5016, 7335
`\exp_args:Nno` 28, 2212, 2729,
 6385, 7265, 9328, 10953, 10961,
 10970, 10987, 10995, 11023, 22290
`\exp_args:NNoo` 28, 28, 2227
`\exp_args:NNox` 29, 2227
`\exp_args:Nnox` 29, 2227
`\exp_args:NNV` 28, 2146
`\exp_args:NNv` 28, 2146
`\exp_args:NnV` 28, 2212
`\exp_args:NNx` 28, 28, 2040, 2212, 16858
`\exp_args:Nnx` 28, 2212
`\exp_args:No` 27, 27,
 734, 2024, 2118, 2709, 2714, 2949,
 2950, 2951, 2966, 2967, 2968, 2969,
 3022, 3040, 3049, 3106, 3108, 3216,
 3218, 3244, 3253, 3388, 3641, 3652,
 3699, 3710, 3776, 3795, 3833, 3848,
 4173, 5016, 5103, 5109, 6082, 6260,
 6271, 6273, 6308, 6313, 6510, 6514,
 8773, 8877, 8908, 8989, 9554, 10131,
 10149, 10177, 10199, 16253, 16841,
 16906, 16908, 16985, 17680, 17730,
 17751, 18343, 18358, 18703, 19093,
 19225, 19261, 19299, 19673, 19702,
 20130, 21964, 22280, 23296, 23298
`\exp_args:Noc` 28, 2212
`\exp_args:Nof` 28, 2212
`\exp_args:Noo` 28, 28, 2212, 17773, 18716
`\exp_args:Nooo` 28, 2227
`\exp_args:Nox` 29, 2227
`\exp_args:Nox` 28, 2212
`\exp_args:NV` 27, 27, 2134, 9116, 9181,
 10129, 10147, 10175, 10197, 17187
`\exp_args:Nv` 27, 27, 2134
`\exp_args:NVV` 28, 2146, 9028
`\exp_args:Nx`
 27, 27, 1860, 2211, 6800, 7687,
 8341, 8378, 8913, 10133, 10151,
 10179, 10201, 17906, 17907, 18251

\exp_args:Nxo 28, [2212](#)
 \exp_args:Nxx 28, [2212](#)
 \exp_end: 31,
 31, 31, 31, 31, 32, 279, 281, 284,
 301, 301, 334, 334, 340, 349, 349,
 417, 482, 578, 578, 606, 912,
 1319, 1500, 1511, 1517, 1523, 2107,
 2116, [2318](#), 3013, 3177, 3363, 3364,
 3698, 5978, 5981, 5982, 5983, 5984,
 5985, 5986, 5987, 5988, 5989, 5990,
 6700, 6733, 6748, 6751, 6756, 6759,
 6818, 6822, 8299, 11254, 12151,
 14439, 15375, 15957, [22352](#), [23336](#)
 \exp_end_continue_f:nw . . . 32, 32, [2318](#)
 \exp_end_continue_f:w
 32, 32, 32, 32, 32, 32,
 300, 300, 580, 581, 581, 2077, 2135,
 2150, 2174, 2244, 2269, 2307, [2318](#),
 7194, 9050, 9325, 10708, 10909,
 11363, 11381, 11402, 11473, 11478,
 11487, 11503, 11561, 11596, 11604,
 11635, 12008, 12033, 12074, 12081,
 12118, 12160, 12163, 12173, 12174,
 12501, 12503, 12507, 12509, 12611,
 12625, 12635, 12744, 12745, 12746,
 12893, 12904, 12928, 12936, 13888,
 14597, 14770, 14976, 15643, 15647,
 15684, 15688, 15730, 15749, 15753,
 15789, 15793, 15860, 15990, [22226](#)
 \exp_last_two_unbraced:Nnn
 29, 29, [2300](#), 20870, 21094, 21098
 \exp_last_unbraced:Nn 29, 29, 29,
 29, [2263](#), 2786, 5115, 5135, 6043,
 6525, 6825, 12998, 16252, 17341,
 18469, 21223, 21228, 21306, 21312
 \exp_last_unbraced:NNn
 29, [2263](#), 3184, 6335, 6368,
 7528, 8910, 21068, 22367, 22426, 23148
 \exp_last_unbraced:Nnn
 29, 29, [2263](#), 7393, 7483, 22032
 \exp_last_unbraced:NNNn 29, [2263](#)
 \exp_last_unbraced:NnNn 29, [2263](#)
 \exp_not:N 30, 30, 30, 30, 144, 191,
 301, 301, 319, 321, 321, 337, 338,
 445, 451, 586, 734, 746, 754, 827,
 830, 912, [1315](#), 1467, 1533, 1536,
 1605, 1606, 1607, 1608, 1609, 1613,
 1614, 1918, 1919, 1978, 1979, [2064](#),
 2100, [2304](#), 2305, 2338, 2339, 2340,
 2341, 2342, 2343, 2344, 2345, 2347,
 2348, 2349, 2353, 2354, 2359, 2360,
 2361, 2380, 2389, 2418, 2419, 2420,
 2421, 2471, 2484, 2485, 2486, 2488,
 2491, 2494, 2495, 2500, 2501, 2502,
 2510, 2709, 2771, 2842, 3257, 3260,
 3276, 3279, 3310, 3317, 3457, 3891,
 4172, 4174, 4510, 5001, 5264, 6420,
 6423, 6431, 6432, 6680, 6765, 6769,
 6803, 6853, 6857, 6862, 6867, 6872,
 6879, 6886, 6891, 6896, 6901, 6906,
 6911, 6921, 6926, 6931, 6934, 6935,
 6936, 6938, 6939, 6950, 6955, 6970,
 6971, 6988, 6993, 6994, 6995, 6996,
 6997, 6998, 7000, 7002, 7003, 7004,
 7005, 7010, 7011, 7032, 7035, 7036,
 7038, 7039, 7040, 7043, 7044, 7046,
 7047, 7049, 7050, 7052, 7115, 7120,
 7138, 7162, 7165, 7172, 7173, 7182,
 7183, 7212, 7436, 7459, 7794, 7796,
 7798, 7800, 7805, 7806, 7811, 7813,
 7815, 8039, 8041, 8043, 8045, 8050,
 8051, 8056, 8058, 8060, 8343, 8381,
 9451, 9749, 9756, 9902, 9903, 9911,
 9913, 9927, 9929, 9981, 9982, 10102,
 10103, 10541, 11407, 11408, 11409,
 11413, 11414, 11439, 11440, 11520,
 11521, 11522, 11523, 11524, 11528,
 11529, 11531, 11533, 11606, 11646,
 11650, 11672, 11765, 11797, 11882,
 11896, 11913, 11923, 11933, 11970,
 11973, 12051, 12052, 12054, 12055,
 12056, 12057, 12058, 12059, 12062,
 12065, 12183, 12185, 12213, 12214,
 12216, 12217, 12218, 12219, 12220,
 12221, 12222, 12223, 12224, 12225,
 12226, 12354, 12369, 12809, 12951,
 14829, 14830, 14831, 14835, 14836,
 14837, 15854, 15855, 15857, 15858,
 15859, 15860, 15861, 15863, 16208,
 16215, 16222, 16608, 16611, 16746,
 16747, 16748, 16749, 16750, 16751,
 16752, 16753, 16754, 16755, 16756,
 16757, 16760, 16761, 16762, 17314,
 17316, 17318, 17320, 17322, 17324,
 17668, 17670, 17857, 17859, 17870,
 17874, 18033, 18438, 19087, 19377,
 19388, 19613, 19736, 21508, 21511,
 22104, 22281, 22305, 22895, 22897,
 22938, 22940, 22951, 22953, 23373,
 23374, 23597, 23598, 23599, 23606
 \exp_not:n 30,
 30, 30, 30, 30, 31, 31, 31, 31, 43,
 43, 44, 45, 46, 59, 63, 64, 103, 104,
 105, 124, 144, 227, 229, 230, 232,
 234, 236, 236, 288, 316, 319, 319,
 324, 324, 324, 324, 333, 365, 420,
 423, 426, 426, 458, 740, 740, 740,
 745, 745, 751, 754, 754, 765, 769,

827, 827, 832, 832, 835, 935, [1315](#),
 1468, 1470, 1538, 1712, 1714, 1862,
[2064](#), 2260, [2304](#), 2425, 2440, 2525,
 2555, 2557, 2565, 2567, 2571, 2573,
 2583, 2585, 2587, 2589, 2591, 2593,
 2595, 2597, 2607, 2609, 2611, 2613,
 2615, 2617, 2619, 2621, 2639, 2641,
 2642, 2680, 2687, 2731, 2773, 2780,
 2839, 2843, 2844, 2845, 3102, 3104,
 3385, 3509, 3997, 4194, 4195, 4202,
 4203, 4219, 4251, 4271, 4274, 4277,
 4345, 4398, 4453, 4511, 4551, 4561,
 5002, 6052, 6053, 6068, 6070, 6101,
 6167, 6260, 6267, 6287, 6424, 6430,
 6458, 6463, 6494, 6527, 6804, 7007,
 7116, 7121, 7135, 7139, 7211, 7213,
 7400, 7437, 7440, 7441, 7460, 7464,
 7788, 7805, 7903, 8033, 8050, 8354,
 8380, 8481, 8492, 8743, 8795, 8871,
 8888, 9690, 9692, 9714, 9716, 9749,
 9757, 9758, 9782, 9948, 9984, 10006,
 10104, 10249, 10274, 10475, 10477,
 11515, 12810, 12952, 15009, 16036,
 16038, 16040, 16364, 16636, 16733,
 16759, 16870, 17668, 17670, 18254,
 18634, 18857, 19076, 19088, 19143,
 19326, 19328, 19335, 19377, 19385,
 19399, 19414, 19451, 19628, 19633,
 22046, 22059, [23409](#), 23412, 23414

\exp_stop_f: [31](#), 31, 31, 32, 32, 32,
 300, 336, 365, 365, 365, 551, 619,
 740, 746, [2074](#), 3500, 3510, 3523,
 3529, 3683, 3696, 3735, 3736, 3742,
 3754, 3770, 3997, 4020, 4031, 4039,
 4051, 4206, 4625, 4635, 4643, 4823,
 4828, 6704, 6707, 6708, 6715, 6716,
 6717, 6718, 6723, 6724, 6744, 6825,
 6942, 8311, 8673, 8824, 9038, 9052,
 9336, 9538, 10620, 10841, 10856,
 10881, 11131, 11135, 11139, 11141,
 11145, 11149, 11157, 11162, 11175,
 11182, 11195, 11206, 11207, 11218,
 11219, 11228, 11231, 11241, 11276,
 11332, 11337, 11421, 11451, 11570,
 11612, 11663, 11683, 11710, 11724,
 11759, 11786, 11795, 11814, 11830,
 11846, 11864, 11924, 11943, 11959,
 11974, 11988, 12186, 12197, 12482,
 12486, 12656, 12662, 12664, 12667,
 12675, 12676, 12848, 12913, 12918,
 12961, 12967, 12982, 13019, 13092,
 13114, 13168, 13169, 13177, 13510,
 13528, 13581, 13585, 13589, 13607,
 13642, 13643, 13644, 13645, 13646,
 13672, 13684, 13701, 13941, 13942,
 14039, 14131, 14164, 14177, 14182,
 14191, 14193, 14319, 14348, 14353,
 14383, 14420, 14460, 14479, 14591,
 14637, 14638, 14643, 14649, 14667,
 14690, 14722, 14725, 14768, 14788,
 14794, 14803, 14814, 14850, 14865,
 14880, 14895, 14910, 14925, 14953,
 14997, 15223, 15386, 15388, 15425,
 15436, 15468, 15509, 15518, 15533,
 15552, 15585, 15598, 15648, 15689,
 15737, 15740, 15754, 15966, 15970,
 15977, 16125, 16383, 16583, 16618,
 16677, 16678, 16684, 16702, 16705,
 16706, 16708, 16726, 16773, 16792,
 16798, 16871, 16874, 16961, 17124,
 17125, 17126, 17131, 17406, 17427,
 17428, 17432, 17436, 17437, 17440,
 17441, 17449, 17450, 17453, 17457,
 17458, 17461, 17508, 17599, 17830,
 17835, 17863, 17933, 17934, 17973,
 18064, 18364, 18639, 18657, 18684,
 18694, 18749, 18762, 18773, 18789,
 18840, 19023, 19153, 19157, 19206,
 19258, 19270, 19286, 19291, 19297,
 19340, 19381, 19395, 19418, 19626,
 19631, 19652, 19726, 19738, 19743,
 22483, 22495, 22708, 22723, 22778,
 23322, 23332, 23341, 23351, 23352,
 23358, 23373, 23374, 23375, 23376

exp internal commands:
 __exp_arg_last_unbraced:nn .. [2240](#)
 __exp_arg_next:Nnn [2064](#), [2071](#)
 __exp_arg_next:nnn
 .. [2064](#), [2073](#), [2076](#), [2084](#), [2088](#), [2094](#)
 __exp_eval_error_msg:w [2098](#)
 __exp_eval_register:N [2089](#),
[2095](#), [2098](#), [2139](#), [2144](#), [2156](#), [2162](#),
[2180](#), [2181](#), [2188](#), [2251](#), [2256](#), [2264](#),
[2266](#), [2276](#), [2282](#), [2291](#), [2311](#), [2316](#)
 \l__exp_internal_t1
 [32](#), [277](#), [1380](#), [1384](#),
[1385](#), [2064](#), [2083](#), [2084](#), [2260](#), [2261](#)
 __exp_last_two_unbraced:nnN . [2300](#)
 \expandafter [13](#),
[14](#), [21](#), [38](#), [39](#), [42](#), [43](#), [48](#), [49](#), [66](#), [67](#),
[90](#), [92](#), [98](#), [123](#), [146](#), [154](#), [169](#), [185](#), [365](#)
 \expanded [878](#)
 \expandglyphsinfont [953](#)
 \expansionERROR [2322](#)
 \ExplFileVersion [6](#)
 \ExplFileDescription [6](#)
 \ExplFileName [6](#)
 \ExplFileVersion [6](#)

\explicithyphenpenalty 877
 \ExplSyntaxOff 3, 6,
 6, 6, 6, 7, 207, 240, 251, 252, 252, 254
 \ExplSyntaxOn 3, 6,
 6, 6, 6, 7, 236, 251, 252, 252, 320, 437

F

false 191
 \fam 366
 \fi 17, 35, 41,
 51, 70, 71, 72, 95, 96, 97, 100, 101,
 130, 139, 152, 153, 170, 186, 204, 367
 fi commands:
 \fi: . 21, 79, 79, 79, 90, 96, 96, 145,
 160, 217, 217, 217, 279, 284, 286,
 324, 326, 326, 349, 351, 353, 381,
 390, 418, 560, 585, 600, 600, 624,
 683, 745, 745, 1297, 1341, 1464,
 1477, 1497, 1547, 1552, 1559, 1597,
 1602, 1632, 1633, 1641, 1647, 1660,
 1661, 1669, 1675, 1723, 1763, 1777,
 1855, 1875, 1885, 1896, 1948, 2009,
 2055, 2103, 2106, 2113, 2114, 2349,
 2370, 2377, 2386, 2401, 2402, 2403,
 2417, 2437, 2439, 2460, 2461, 2636,
 2855, 2856, 2892, 2905, 2918, 2928,
 2945, 2957, 2961, 2977, 3171, 3224,
 3229, 3238, 3240, 3267, 3290, 3306,
 3315, 3324, 3330, 3337, 3342, 3353,
 3357, 3365, 3504, 3511, 3524, 3529,
 3612, 3616, 3617, 3635, 3688, 3698,
 3740, 3746, 3747, 3758, 3770, 3771,
 3792, 3830, 3856, 3981, 3987, 3999,
 4000, 4025, 4037, 4041, 4042, 4054,
 4068, 4247, 4250, 4287, 4303, 4324,
 4334, 4385, 4390, 4628, 4651, 4668,
 4669, 4671, 4717, 4726, 4781, 4789,
 4816, 4824, 4830, 4853, 4890, 4898,
 5183, 5216, 5264, 5269, 5406, 5491,
 5514, 5523, 5559, 5561, 5577, 5585,
 5609, 5624, 5632, 5748, 5756, 5791,
 5818, 5837, 5992, 5994, 5996, 5998,
 6000, 6002, 6144, 6159, 6182, 6198,
 6712, 6715, 6716, 6717, 6718, 6723,
 6724, 6729, 6730, 6731, 6732, 6769,
 6827, 6829, 6858, 6863, 6868, 6873,
 6880, 6887, 6892, 6897, 6902, 6907,
 6912, 6917, 6922, 6927, 6946, 6951,
 6962, 6963, 7010, 7011, 7062, 7071,
 7080, 7088, 7154, 7168, 7177, 7187,
 7197, 7500, 7536, 7602, 8730, 9031,
 9040, 9061, 9070, 9074, 9081, 9089,
 9284, 9307, 9316, 9335, 9339, 9353,
 9542, 9695, 9719, 9733, 9743, 9763,

9764, 9773, 10672, 10681, 10685,
 10697, 10707, 10765, 10768, 10769,
 10774, 10788, 10826, 10827, 10828,
 10829, 10830, 10831, 10832, 10833,
 10834, 10835, 10836, 10837, 10850,
 10852, 10863, 10866, 10880, 10885,
 10889, 11028, 11110, 11111, 11112,
 11137, 11138, 11147, 11148, 11159,
 11160, 11161, 11172, 11173, 11174,
 11181, 11192, 11193, 11194, 11204,
 11205, 11209, 11210, 11218, 11221,
 11222, 11230, 11240, 11253, 11276,
 11320, 11339, 11340, 11349, 11355,
 11375, 11376, 11404, 11425, 11442,
 11449, 11457, 11458, 11504, 11505,
 11506, 11569, 11585, 11610, 11611,
 11618, 11626, 11655, 11656, 11659,
 11661, 11662, 11667, 11677, 11680,
 11682, 11687, 11718, 11731, 11736,
 11742, 11745, 11746, 11780, 11781,
 11808, 11809, 11822, 11825, 11836,
 11859, 11878, 11888, 11904, 11918,
 11924, 11928, 11933, 11939, 11954,
 11965, 11984, 11994, 11996, 12002,
 12023, 12046, 12075, 12192, 12196,
 12206, 12207, 12226, 12244, 12259,
 12263, 12271, 12285, 12306, 12337,
 12347, 12361, 12377, 12397, 12438,
 12449, 12462, 12464, 12484, 12485,
 12492, 12510, 12619, 12630, 12642,
 12643, 12644, 12651, 12653, 12654,
 12660, 12661, 12664, 12670, 12671,
 12679, 12680, 12682, 12683, 12829,
 12842, 12852, 12853, 12858, 12859,
 12860, 12861, 12862, 12863, 12870,
 12879, 12884, 12908, 12911, 12913,
 12920, 12965, 12966, 12973, 12986,
 13001, 13011, 13025, 13055, 13064,
 13098, 13120, 13138, 13155, 13172,
 13173, 13175, 13176, 13181, 13196,
 13229, 13258, 13259, 13260, 13261,
 13262, 13275, 13317, 13390, 13455,
 13457, 13458, 13468, 13497, 13500,
 13501, 13512, 13532, 13595, 13596,
 13597, 13609, 13648, 13649, 13650,
 13651, 13657, 13660, 13662, 13672,
 13689, 13701, 13917, 13921, 13923,
 13927, 13934, 13935, 13945, 13946,
 13949, 14041, 14110, 14121, 14133,
 14163, 14170, 14181, 14197, 14204,
 14265, 14321, 14331, 14333, 14343,
 14361, 14362, 14394, 14397, 14406,
 14408, 14410, 14424, 14438, 14462,
 14482, 14483, 14496, 14512, 14528,

14544, 14560, 14576, 14590, 14598,
 14604, 14615, 14618, 14621, 14630,
 14640, 14642, 14648, 14655, 14658,
 14667, 14675, 14696, 14729, 14730,
 14757, 14759, 14773, 14774, 14793,
 14808, 14814, 14815, 14816, 14860,
 14875, 14890, 14905, 14920, 14935,
 14938, 14940, 14957, 15002, 15229,
 15265, 15266, 15276, 15317, 15318,
 15342, 15374, 15380, 15381, 15384,
 15386, 15387, 15392, 15404, 15423,
 15428, 15435, 15438, 15470, 15480,
 15481, 15491, 15513, 15528, 15546,
 15554, 15557, 15585, 15593, 15609,
 15647, 15665, 15688, 15706, 15739,
 15753, 15759, 15814, 15815, 15915,
 15916, 15932, 15956, 15967, 15968,
 15984, 16127, 16139, 16140, 16196,
 16266, 16275, 16286, 16295, 16304,
 16339, 16349, 16359, 16467, 16469,
 16542, 16546, 16550, 16586, 16597,
 16615, 16616, 16617, 16624, 16640,
 16647, 16651, 16659, 16671, 16679,
 16686, 16697, 16709, 16728, 16741,
 16759, 16777, 16785, 16787, 16790,
 16797, 16802, 16880, 16881, 16960,
 16996, 17005, 17083, 17090, 17091,
 17097, 17100, 17103, 17110, 17111,
 17114, 17118, 17119, 17128, 17129,
 17133, 17134, 17146, 17154, 17163,
 17164, 17190, 17308, 17310, 17334,
 17338, 17354, 17365, 17398, 17402,
 17421, 17423, 17430, 17433, 17434,
 17438, 17442, 17443, 17444, 17445,
 17454, 17455, 17459, 17462, 17463,
 17464, 17512, 17515, 17523, 17534,
 17535, 17546, 17547, 17555, 17567,
 17568, 17577, 17578, 17588, 17607,
 17608, 17613, 17614, 17662, 17671,
 17695, 17706, 17708, 17772, 17820,
 17823, 17824, 17839, 17842, 17865,
 17931, 17932, 17937, 17964, 17965,
 17976, 17980, 18008, 18013, 18021,
 18055, 18060, 18065, 18073, 18103,
 18160, 18176, 18215, 18222, 18233,
 18303, 18320, 18324, 18338, 18368,
 18576, 18627, 18643, 18667, 18689,
 18698, 18758, 18765, 18785, 18803,
 18814, 18816, 18846, 18849, 18874,
 18955, 18980, 19004, 19005, 19026,
 19057, 19083, 19156, 19199, 19211,
 19263, 19273, 19274, 19294, 19305,
 19329, 19345, 19383, 19385, 19397,
 19399, 19420, 19508, 19566, 19583,
 19584, 19628, 19629, 19633, 19634,
 19656, 19661, 19694, 19732, 19740,
 19741, 19745, 19746, 20092, 20094,
 20104, 22040, 22160, 22168, 22176,
 22847, 22848, 22851, 23324, 23337,
 23344, 23356, 23362, 23363, 23373,
 23374, 23375, 23385, 23392, 23394

file commands:

- \file_add_path:nN
- 138, 138, 145, 8497, 8539, 8651, 8663
- \g_file_current_name_tl
- 138, 7637, 8425, 8443, 8575, 8576, 8579
- \file_if_exist:nTF
- 138, 138, 138, 138, 145,
 8537, 8551, 21906, 21911, 21919, 21924
- \file_if_exist_input:n 228, 228
- \file_if_exist_input:nTF
- 228, 228, 21904, 21909, 21917
- \file_input:n
- . . . 138, 138, 138, 228, 228, 8544
- \file_list: 139, 139, 8594
- \file_path_include:n
- 138, 138, 138, 228, 8582
- \file_path_remove:n .. 139, 139, 8582

file internal commands:

- __file_add_path:nN 8497
- __file_add_path_search:nN . . . 8497
- __file_if_exist:nTF
- 145, 8544, 22277, 22302
- __file_input:n
- . . . 8544, 21907, 21914, 21920, 21927
- __file_input_aux:n 8544
- \g_file_internal_ior
- 145, 8501, 8502, 8505, 8522, 8523, 8778
- \l__file_internal_name_tl
- 145, 487, 8447, 8463, 8464,
 8465, 8470, 8477, 8478, 8480, 8481,
 8487, 8492, 8539, 8540, 8547, 8651,
 8652, 8654, 8663, 8664, 8667, 21907,
 21914, 21920, 21927, 22285, 22308
- \l__file_internal_seq 8452,
 8515, 8517, 8596, 8602, 8607, 8609
- \l__file_internal_tl
- 8446, 8460, 8461, 8578, 8579
- __file_name_sanitize:nn
- 145, 145, 8455, 8498,
 8554, 8583, 8591, 8647, 8657, 8827
- __file_name_sanitize_aux:n .. 8455
- __file_path_include:n 8582
- \g_file_record_seq 490,
 491, 491, 8438, 8568, 8573, 8596, 8616
- \l__file_saved_search_path_seq ..
- 8449, 8513, 8533

\l_file_search_path_seq
 8448, 8514, 8516,
 8517, 8520, 8532, 8586, 8587, 8592
 \g_file_stack_seq
 490, 8437, 8575, 8578
 \finalhyphendemerits 368
 \firstmark 369
 \firstmarks 626
 \firstvalidlanguage 879
 flag commands:
 \flag_clear:n 82, 82, 82,
 5482, 5495, 5534, 18245, 19590, 19591
 \flag_clear_new:n 82, 82, 5494
 \flag_height:n 83, 83, 5499,
 5516, 5530, 5547, 19602, 19607, 19608
 \flag_if_exist:n 83
 \flag_if_exist:nTF 83, 5495, 5503, 5541
 \flag_if_exist_p:n 83, 5503
 \flag_if_raised:n 83, 5552
 \flag_if_raised:nTF 83, 5508, 18253
 \flag_if_raised_p:n 83, 5508
 \flag_log:n 82, 82, 5496
 \flag_new:n 82,
 82, 82, 5477, 5495, 10924, 10925,
 10926, 10927, 18235, 19494, 19495
 \flag_raise:n 83, 83, 5527,
 10958, 10967, 10975, 10992, 11001,
 11032, 18268, 18290, 19627, 19632
 \flag_show:n 82, 82, 5496
 flag fp commands:
 \flag_fp_division_by_zero . 183, 10924
 \flag_fp_invalid_operation . 183, 10924
 \flag_fp_overflow 183, 10924
 \flag_fp_underflow 183, 10924
 flag internal commands:
 __flag_chk_exist:n 5532
 __flag_clear:wn 5482, 5537
 __flag_height_end:wn 5516
 __flag_height_loop:wn 5516, 5550
 \floatingpenalty 370
 \floor 187
 \fmtname 145
 \font 371
 \fontchardp 627
 \fontcharht 628
 \fontcharic 629
 \fontcharwd 630
 \fontdimen 372
 \fontid 880
 \fontname 373
 \forcecjktoken 1160
 \formatname 881
 fp commands:
 \c_e_fp 182, 184, 16081
 \fp_abs:n
 186, 191, 191, 721, 15834, 20426,
 20503, 20505, 20507, 21859, 21861
 \fp_add:Nn 177, 177, 721, 16058
 \fp_compare:nNnTF 179, 179,
 180, 180, 180, 180, 12621, 12716,
 12722, 12727, 12735, 12779, 12785,
 15977, 15991, 16020, 20294, 20296,
 20301, 20522, 20537, 20546, 20931,
 21843, 23548, 23554, 23700, 24104
 \fp_compare:nTF
 179, 179, 180, 180, 180, 181,
 186, 12608, 12688, 12694, 12699, 12707
 \fp_compare_p:n 179, 179, 12608
 \fp_compare_p:nNn 179, 179, 12621
 \fp_const:Nn 176,
 176, 16035, 16081, 16082, 16083, 16084
 \fp_do_until:nn 180, 180, 12685
 \fp_do_until:nNnn 180, 180, 12713
 \fp_do_while:nn 180, 180, 12685
 \fp_do_while:nNnn 180, 180, 12713
 \fp_eval:n 177, 177, 179, 185, 185,
 186, 186, 186, 186, 186, 186,
 186, 186, 186, 186, 187, 187,
 187, 187, 187, 187, 187, 187, 188,
 188, 188, 188, 188, 188, 188, 188,
 188, 188, 188, 188, 188, 188, 188,
 188, 188, 188, 188, 188, 189, 189,
 189, 189, 189, 189, 189, 189, 189,
 190, 190, 190, 191, 191, 728, 15831,
 23559, 23574, 23576, 23702, 23713,
 23714, 23944, 23962, 23963, 23964,
 23965, 23977, 23978, 23986, 23987,
 23994, 24001, 24003, 24009, 24010,
 24011, 24022, 24027, 24033, 24034,
 24106, 24123, 24124, 24306, 24328,
 24329, 24340, 24341, 24353, 24354,
 24366, 24368, 24373, 24384, 24396,
 24407, 24408, 24500, 24515, 24516,
 24717, 24735, 24736, 24737, 24752,
 24770, 24771, 24772, 24821, 24822
 \fp_format:nn 192
 \fp_function:Nw 12515
 \fp_gadd:Nn 177, 16058
 .fp_gset:N 165, 10160
 \fp_gset:Nn 176, 16035, 16059, 16061
 \fp_gset_eq:NN 177, 16044, 16049
 \fp_gsub:Nn 177, 16058
 \fp_gzero:N 176, 16048, 16055
 \fp_gzero_new:N 176, 16052
 \fp_if_exist:NTF
 178, 178, 12606, 16053, 16055, 16070
 \fp_if_exist_p:N 178, 178, 12606
 \fp_if_nan:nTF 192

\fp_log:N 183, 183, [16076](#)
 \fp_log:n 183, 183, [16076](#)
 \fp_max:nn 191, 191, [15836](#)
 \fp_min:nn 191, [15836](#)
 \fp_new:N 176, 176, 176,
 [16032](#), 16053, 16055, 16085, 16086,
 16087, 16088, 20266, 20267, 20268,
 20388, 20389, 20590, 20591, 21674,
 21675, 21820, 21821, 23567, 23568
 \fp_new_function:Npn [12522](#)
 .fp_set:N 165, [10160](#)
 \fp_set:Nn . 176, 176, [16035](#), 16058,
 16060, 20282, 20283, 20284, 20395,
 20397, 20433, 20448, 20463, 20475,
 20477, 20490, 20491, 20520, 20521,
 20927, 20929, 21684, 21685, 21826,
 21828, 21853, 21854, 23547, 23550
 \fp_set_eq:NN 177, 177, [16044](#), 16048,
 20438, 20453, 20465, 20523, 20524
 \fp_show:N . 183, 183, 728, [16068](#), 16077
 \fp_show:n . 183, 183, 728, [16068](#), 16079
 \fp_step_function:nnnN
 181, 181, [12741](#), 12816
 \fp_step_inline:nnnn 181, 181, [12794](#)
 \fp_step_variable:nnnNn
 181, 181, [12794](#)
 \fp_sub:Nn 177, 177, [16058](#)
 \fp_to_decimal:N 177, 177,
 178, [10917](#), [15678](#), 15696, 15780, 15831
 \fp_to_decimal:n
 177, 177, 177, 177, 178, 720, [15678](#),
 15783, 15833, 15835, 15837, 15839
 \fp_to_dim:N 177, 177, [15779](#)
 \fp_to_dim:n 177, 177, 182,
 [15779](#), 20326, 20337, 20426, 20937,
 20963, 21757, 21765, 21869, 21871
 \fp_to_int:N 178, 178, [15784](#)
 \fp_to_int:n 178,
 178, 725, [15784](#), 15992, 15993, 21959
 \fp_to_int_dispatch:w 720
 \fp_to_scientific:N
 178, 178, [10918](#), [15637](#), 15655, 15662
 \fp_to_scientific:n
 178, 178, 178, [15637](#)
 \fp_to_tl:N 178, 178, [15744](#), 16071
 \fp_to_tl:n 178, 178,
 10634, 10957, 10966, 10991, 11000,
 11029, 12771, 12782, [15744](#), 16074
 \fp_trap:nn 183, 183, 183,
 564, [10928](#), 11043, 11044, 11045, 11046
 \fp_until_do:nn 180, 180, [12685](#)
 \fp_until_do:nNnn 180, 180, [12713](#)
 \fp_use:N
 178, 178, [15831](#), 23553, 23557, 23562

\fp_while_do:nn 181, 181, [12685](#)
 \fp_while_do:nNnn 180, 180, [12713](#)
 \fp_zero:N 176, 176, [16048](#), 16053, 23549
 fp_zero:N 176
 \fp_zero_new:N 176, 176, [16052](#)
 \c_inf_fp 182, 190, [10643](#),
 12083, 13272, 13352, 14390, 14413,
 14620, 14623, 14627, 14651, 14942
 \c_nan_fp 190, 567, 590, [10643](#), 10968,
 10976, 11048, 11264, 11283, 11289,
 11473, 11478, 11487, 11596, 11635,
 12074, 12084, 12286, 12552, 12775,
 14594, 15373, 15879, 15941, 15955
 \c_one_fp 182, 12087, 12467, 12488,
 12867, 13693, 14384, 14589, 14641,
 14866, 14896, 15367, 15950, [16081](#)
 \c_pi_fp 182, 190, 599, [12085](#), [16083](#)
 \g_tmipa_fp 182, [16085](#)
 \l_tmipa_fp 182, [16085](#)
 \g_tmrb_fp 182, [16085](#)
 \l_tmrb_fp 182, [16085](#)
 \c_zero_fp
 181, 628, 727, [10643](#), 10690, 12088,
 12479, 12491, 12869, 13101, 13268,
 14393, 14414, 14617, 14654, 15590,
 15662, 15806, 16033, 16048, 16049,
 20294, 20296, 20301, 20537, 20546,
 21843, 23548, 23554, 23700, 24104

fp internal commands:

__fp_&_o:ww 617, 625, [12872](#)
 __fp_*_o:ww [13233](#)
 __fp+_o:ww
 627, 627, 627, 627, 627, 655, [12954](#)
 __fp_,_o:ww 609, [12248](#)
 __fp-_o:ww 627, 627, 627, [12949](#)
 __fp/_o:ww 636, 636, 675, [13343](#)
 __fp^_o:ww 14585
 __fp_acos_o:w 712, 715, [15531](#)
 __fp_acot_o:Nw 14841, 14843, [15353](#)
 __fp_acotii_o:Nww 15356, 15361, [15377](#)
 __fp_acotii_o:ww 708
 __fp_acsc_normal_o:NnwNnw
 714, 15589, 15604, [15612](#)
 __fp_acsc_o:w [15583](#)
 __fp_add:NNNn [16058](#)
 __fp_add_big_i:wNww 629
 __fp_add_big_i_o:wNww
 627, 630, 13021, [13028](#)
 __fp_add_big_ii:wNww 629
 __fp_add_big_ii_o:wNww 13024, [13028](#)
 __fp_add_inf_o:Nww 12970, [12990](#)
 __fp_add_normal_o:Nww
 629, 12969, [13005](#)

```

\__fp_add_npos_o:NnwNnw . . . . .
    ..... 629, 13008, 13014
\__fp_add_return_ii_o:Nww . . . .
    ..... 12972, 12978, 12983
\__fp_add_significand_carry_-
    o:wwwNN . . . . 631, 13061, 13076
\__fp_add_significand_no_carry_-
    o:wwwNN . . . . 631, 13063, 13066
\__fp_add_significand_o:NnnwnnnnN
    ..... 630, 630, 13031, 13039, 13044
\__fp_add_significand_pack:NNNNNNN
    ..... 13044
\__fp_add_significand_test_o:N 13044
\__fp_add_zeros_o:Nww . 12968, 12980
\__fp_and_return:wNw . . . . 12872
\__fp_array_count:n . . . .
    ..... 10891, 11248, 15366, 15949
\__fp_array_count_loop:Nw . . . . 10891
\__fp_array_to_clist:n . . .
    .. 11309, 12344, 12345, 15840, 15982
\__fp_array_to_clist_loop:Nw . 15840
\__fp_asec_o:w . . . . 15596
\__fp_asin_auxi_o:NnNww . . .
    .. 713, 713, 715, 15561, 15564, 15623
\__fp_asin_isqrt:wn . . . . 15564
\__fp_asin_normal_o:NnwNnnnnw . .
    ..... 15522, 15538, 15549
\__fp_asin_o:w . . . . 15516
\__fp_atan_auxi:ww . 710, 15441, 15455
\__fp_atan_auxii:w . . . . 15455
\__fp_atan_combine_aux:ww . . . . 15482
\__fp_atan_combine_o:NwwwwN . .
    ..... 709, 709, 15401, 15418, 15482
\__fp_atan_dispatch_o:NNnNw . . 15353
\__fp_atan_div:wnwwnw . . .
    ..... 709, 15429, 15431
\__fp_atan_inf_o:NNNw . . .
    ..... 708, 708, 15389,
        15390, 15391, 15399, 15534, 15607
\__fp_atan_near:wwnn . . . . 15431
\__fp_atan_near_aux:wwn . . . . 15431
\__fp_atan_normal_o:NNnwNnw . .
    ..... 708, 15393, 15409
\__fp_atan_o:Nw . 14845, 14847, 15353
\__fp_atan_Taylor_break:w . . . . 15466
\__fp_atan_Taylor_loop:www . .
    ..... 710, 15461, 15466
\__fp_atan_test_o:NwwNwwN . .
    ..... 714, 15412, 15416, 15571
\__fp_atanii_o:Nww 15356, 15361, 15377
\__fp_basics_pack_high:NNNNNw . .
    .. 631, 647, 10759, 13069, 13221,
        13322, 13334, 13474, 13667, 14138
\__fp_basics_pack_high_carry:w . .
    ..... 557, 10759
\__fp_basics_pack_low:NNNNNw . .
    .. 638, 647,
        10759, 13071, 13223, 13324, 13336,
        13476, 13616, 13618, 13669, 14140
\__fp_basics_pack_weird_high:NNNNNNNNw
    ..... 193, 10770, 13080, 13485
\__fp_basics_pack_weird_low:NNNNw
    ..... 193, 10770, 13082, 13487
\c__fp_big_leading_shift_int . .
    .. 10745, 13546, 13828, 13838, 13848
\c__fp_big_middle_shift_int . .
    .. 10745, 13549, 13552, 13555,
        13558, 13561, 13564, 13568, 13830,
        13840, 13850, 13859, 13862, 13865
\c__fp_big_trailing_shift_int . .
    .. 10745, 13572, 13872
\c__fp_Bigg_leading_shift_int . .
    .. 10750, 13396, 13413
\c__fp_Bigg_middle_shift_int . .
    .. 10750, 13399, 13402, 13416, 13419
\c__fp_Bigg_trailing_shift_int . .
    .. 10750, 13405, 13422
\c__fp_block_int . . . . 10648, 14090
\__fp_case_return:nw . . . .
    .. 10827, 10857, 10860, 10865, 11369,
        14349, 15389, 15390, 15391, 15649,
        15690, 15755, 15757, 15758, 15806
\__fp_case_return_i_o:ww . 10834,
    12971, 12985, 12994, 13266, 15380
\__fp_case_return_ii_o:ww . . .
    .. 10834, 13267, 14639, 14657, 15381
\__fp_case_return_o:Nw . . . .
    .. 561, 10828, 14384, 14389,
        14392, 14589, 14594, 14617, 14620,
        14623, 14866, 14896, 15590, 15592
\__fp_case_return_o:Nww . . . .
    .. 10832, 13268, 13269,
        13272, 13273, 14641, 14650, 14653
\__fp_case_return_same_o:w . . .
    .. 561, 10830,
        13497, 13501, 13685, 13688, 14169,
        14396, 14614, 14851, 14859, 14874,
        14889, 14904, 14911, 14919, 14934,
        15519, 15527, 15545, 15591, 15608
\__fp_case_use:nw . . . .
    .. 10826, 12996, 13264,
        13265, 13270, 13271, 13351, 13354,
        13499, 14162, 14165, 14625, 14852,
        14857, 14867, 14872, 14882, 14887,
        14897, 14902, 14912, 14917, 14927,
        14932, 15521, 15524, 15534, 15536,
```

15542, 15586, 15588, 15599, 15602,
 15607, 15652, 15659, 15693, 15700
 $\backslash\text{--fp_chk:w}$ 550, 550, 550, 550, 552,
 552, 552, 552, 599, 599, 627, 629,
 629, 629, 631, 637, 637, 640, 640,
10630, 10643, 10644, 10645, 10646,
 10647, 10657, 10662, 10664, 10665,
 10686, 10689, 10691, 10701, 10714,
 10733, 10838, 10854, 11024, 11029,
 11266, 11312, 11321, 11323, 12097,
 12613, 12638, 12639, 12758, 12771,
 12775, 12831, 12832, 12835, 12846,
 12847, 12855, 12856, 12864, 12875,
 12878, 12890, 12916, 12955, 12975,
 12976, 12978, 12979, 12980, 12988,
 12991, 13002, 13003, 13005, 13014,
 13090, 13242, 13276, 13277, 13280,
 13359, 13495, 13503, 13505, 13682,
 13690, 13692, 13694, 13698, 14159,
 14171, 14173, 14381, 14398, 14400,
 14586, 14605, 14607, 14608, 14611,
 14628, 14631, 14634, 14659, 14660,
 14662, 14678, 14763, 14776, 14778,
 14782, 14786, 14848, 14861, 14863,
 14876, 14878, 14891, 14893, 14906,
 14908, 14921, 14923, 14936, 14946,
 15378, 15394, 15395, 15399, 15410,
 15516, 15529, 15531, 15547, 15550,
 15560, 15583, 15594, 15596, 15610,
 15612, 15617, 15645, 15666, 15669,
 15686, 15707, 15710, 15751, 15767,
 15770, 15827, 15828, 15959, 15961
 $\backslash\text{--fp_compare:wNNNW}$ 12408
 $\backslash\text{--fp_compare_aux:wn}$ 12621
 $\backslash\text{--fp_compare_back:ww}$
 . 618, 619, 12482, 12634, 12637, 12845
 $\backslash\text{--fp_compare_nan:w}$ 619, 12637
 $\backslash\text{--fp_compare_npos:nwnw}$ 617,
 619, 619, 12648, 12665, 13092, 13941
 $\backslash\text{--fp_compare_return:w}$ 12608
 $\backslash\text{--fp_compare_significand:nnnnnnnn}$
 12665
 $\backslash\text{--fp_cos_o:w}$ 14863
 $\backslash\text{--fp_cot_o:w}$ 694, 14923
 $\backslash\text{--fp_cot_zero_o:Nnw}$
 693, 695, 14881, 14923
 $\backslash\text{--fp_csc_o:w}$ 14878
 $\backslash\text{--fp_decimate:nNnnnn}$ 561,
10780, 10845, 10872, 11325, 13030,
 13038, 13117, 14427, 14431, 15716
 $\backslash\text{--fp_decimate:Nnnnn}$ 10792
 $\backslash\text{--fp_decimate_auxi:Nnnnn}$ 10796
 $\backslash\text{--fp_decimate_auxii:Nnnnn}$ 10796
 $\backslash\text{--fp_decimate_auxiii:Nnnnn}$.. 10796
 $\backslash\text{--fp_decimate_auxiv:Nnnnn}$... 10796
 $\backslash\text{--fp_decimate_auxix:Nnnnn}$... 10796
 $\backslash\text{--fp_decimate_auxv:Nnnnn}$... 10796
 $\backslash\text{--fp_decimate_auxvi:Nnnnn}$... 10796
 $\backslash\text{--fp_decimate_auxvii:Nnnnn}$.. 10796
 $\backslash\text{--fp_decimate_auxviii:Nnnnn}$.. 10796
 $\backslash\text{--fp_decimate_auxx:Nnnnn}$... 10796
 $\backslash\text{--fp_decimate_auxxi:Nnnnn}$... 10796
 $\backslash\text{--fp_decimate_auxxii:Nnnnn}$.. 10796
 $\backslash\text{--fp_decimate_auxxiii:Nnnnn}$.. 10796
 $\backslash\text{--fp_decimate_auxxiv:Nnnnn}$.. 10796
 $\backslash\text{--fp_decimate_auxxv:Nnnnn}$... 10796
 $\backslash\text{--fp_decimate_auxxvi:Nnnnn}$.. 10796
 $\backslash\text{--fp_decimate_pack:nnnnnnnnnnw}$..
 559, 10803, 10822
 $\backslash\text{--fp_decimate_pack:nnnnnnw}$
 10823, 10824
 $\backslash\text{--fp_decimate_tiny:Nnnnn}$ 10792
 $\backslash\text{--fp_div_npos_o:Nnw}$
 639, 640, 13348, 13358
 $\backslash\text{--fp_div_significand_calc:wwnnnnnnn}$
 643,
 643, 13375, 13384, 13430, 14242, 14249
 $\backslash\text{--fp_div_significand_calc_i:wnnnnnnnn}$.. 13384
 $\backslash\text{--fp_div_significand_calc_i:wnnnnnnnn}$.. 13384
 $\backslash\text{--fp_div_significand_i_o:wnnw}$..
 640, 643, 13365, 13371
 $\backslash\text{--fp_div_significand_ii:wnn}$...
 645, 13379, 13380, 13381, 13426
 $\backslash\text{--fp_div_significand_iii:wwnnnnn}$
 645, 13382, 13433
 $\backslash\text{--fp_div_significand_iv:wwnnnnnnn}$
 646, 13436, 13441
 $\backslash\text{--fp_div_significand_large_o:wwwNNNNwN}$... 648, 13467, 13481
 $\backslash\text{--fp_div_significand_pack:NNN}$..
 647, 647,
 647, 677, 677, 677, 677, 677, 677,
 677, 13428, 13461, 14229, 14247, 14255
 $\backslash\text{--fp_div_significand_small_o:wwwNNNNwN}$... 647, 13465, 13471
 $\backslash\text{--fp_div_significand_test_o:w}$...
 647, 647, 13373, 13462
 $\backslash\text{--fp_div_significand_v:NN}$
 13446, 13448, 13451
 $\backslash\text{--fp_div_significand_v:NNw}$.. 13441
 $\backslash\text{--fp_div_significand_vi:Nw}$...
 646, 13441
 $\backslash\text{--fp_division_by_zero_o:Nnw}$...
 564, 10988, 11036, 14166, 14942, 14943
 $\backslash\text{--fp_division_by_zero_o:NNww}$...
 564, 10996, 11036, 13352, 13355, 14627

```

\__fp_ep_compare:www . 13936, 15425
\__fp_ep_compare_aux:www . 13936
\__fp_ep_div:wwwn . . . . .
. . . . . 705, 13966, 14077,
15343, 15440, 15444, 15453, 15620
\__fp_ep_div_eps_pack:NNNNw . 13996
\__fp_ep_div_epsilon:wnNNNNn . . .
667
\__fp_ep_div_epsilon:wnNNNNNn . . .
. . . . . 13993, 13996
\__fp_ep_div_epsiloni:wwnNNNNNn . 13996
\__fp_ep_div_esti:wwwn . . .
. . . . . 667, 13972, 13975
\__fp_ep_div_estii:wwnnwwn . 13975
\__fp_ep_div_estiii:NNNNNwwnn . 13975
\__fp_ep_inv_to_float_o:wN . . .
695
\__fp_ep_inv_to_float_o:wwN . .
. . . . . 703, 14073, 14081, 14885, 14900
\__fp_ep_isqrt:wwn . . . . . 14019, 15581
\__fp_ep_isqrt_aux:wwn . . . . . 14019
\__fp_ep_isqrt_auxi:wwn 14022, 14024
\__fp_ep_isqrt_auxii:wwnnnnw . 14019
\__fp_ep_isqrt_epsilon:wN . . .
. . . . . 670, 14056, 14059
\__fp_ep_isqrt_epsiloni:wwN . . . . . 14059
\__fp_ep_isqrt_esti:wwwnnwn . .
. . . . . 14034, 14037
\__fp_ep_isqrt_estii:wwwnnwn . 14037
\__fp_ep_isqrt_estiii:NNNNNwwn . .
. . . . . 14037
\__fp_ep_mul:wwwn . . .
. . . . . 13951, 15300, 15330, 15568, 15579
\__fp_ep_mul_raw:wwwN . . .
. . . . . 13951, 14964, 15250
\__fp_ep_to_ep:wwN . 13902, 13953,
13956, 13968, 13971, 14021, 15569
\__fp_ep_to_ep_end:www . . . . . 13902
\__fp_ep_to_ep_loop:N . . .
. . . . . 703, 13902, 15251
\__fp_ep_to_ep_zero:ww . . . . . 13902
\__fp_ep_to_fixed:wwn . . .
. . . . . 13884, 14961, 15447, 15456, 15566
\__fp_ep_to_fixed_auxi:www . . . . . 13884
\__fp_ep_to_fixed_auxii:nnnnnnwn .
. . . . . 13884
\__fp_ep_to_float_o:wN . . . . . 695
\__fp_ep_to_float_o:wwN . . . . . 692,
703, 14073, 14085, 14855, 14870, 15349
\__fp_error:nnnn . . .
10957, 10965, 10974, 10991, 10999,
11027, 11050, 11122, 11259, 11261,
11282, 11287, 12343, 12770, 12781
\__fp_exp_after_?f:nw . . . . . 587, 11461
\__fp_exp_after_array_f:w 10734,
12054, 12894, 12905, 12929, 12937
\__fp_exp_after_f:nw . . .
. . . . . 587, 10691, 12096, 12178
\__fp_exp_after_mark_f:nw . 587, 11461
\__fp_exp_after_normal:nNnw . .
. . . . . 10694, 10704, 10721
\__fp_exp_after_normal:Nwwww . .
. . . . . 10723, 10731
\__fp_exp_after_o:w . . . . . 554, 10691,
10831, 10835, 10837, 11319, 11363,
11381, 12155, 12502, 12863, 12880,
12979, 13696, 14775, 14780, 16026
\__fp_exp_after_special:nNnw . .
. . . . . 10696, 10706, 10711
\__fp_exp_after_stop_f:nw . . . . . 10734
\__fp_exp_large:w . . . . . 14474
\__fp_exp_large:_wN . . . . . 14474
\__fp_exp_large_after:wwn . . . . . 14474
\__fp_exp_large_i:wN . . . . . 14474
\__fp_exp_large_ii:wN . . . . . 14474
\__fp_exp_large_iii:wN . . . . . 14474
\__fp_exp_large_iv:wN . . . . . 14474
\__fp_exp_large_v:wN . . . . . 14474, 14762
\__fp_exp_normal_o:w . . . . . 14386, 14400
\__fp_exp_o:w . . . . . 14147, 14381
\__fp_exp_overflow>NN . . . . . 14400
\__fp_exp_pos_large:NnnNwn . .
. . . . . 14432, 14474
\__fp_exp_pos_o:NNnw . . .
. . . . . 14403, 14405, 14408
\__fp_exp_pos_o:Nnnwn . . . . . 14400
\__fp_exp_Taylor:Nnnwn . .
. . . . . 14428, 14447, 14582
\__fp_exp_Taylor_break:Nww . . . . . 14447
\__fp_exp_Taylor_ii:ww . . . . . 14453, 14456
\__fp_exp_Taylor_loop:www . . . . . 14447
\__fp_expand:n . . . . . 722, 10900, 15844
\__fp_expand_loop:nwnN . . . . . 10900
\__fp_exponent:w . . . . . 10665
\c_fp_five_int . . . . . 11129,
11153, 11166, 11179, 11186, 11238
\__fp_fixed_add:nnNnnnnw . . . . . 13778
\__fp_fixed_add:Nnnnnwnn . . . . . 13778
\__fp_fixed_add:wwn . . .
. . . . . 656, 656, 13778, 14017, 14323,
14331, 14342, 14360, 15452, 15512
\__fp_fixed_add_after:NNNNNwn . 13778
\__fp_fixed_add_one:wN . . .
. . . . . 13710, 14010, 14464, 14473, 15578
\__fp_fixed_add_pack:NNNNNwn . 13778
\__fp_fixed_continue:wn . . .
. . . . . 13709, 13954, 13959, 13969, 14486,
14502, 14518, 14534, 14550, 14566,
14738, 14999, 15288, 15570, 15579
\__fp_fixed_div_int:wnN . . . . . 13747

```

$\backslash\text{--fp_fixed_div_int:wwN}$
 13747, 14322, 14463, 15471
 $\backslash\text{--fp_fixed_div_int_after:Nw}$
 658, 13747
 $\backslash\text{--fp_fixed_div_int_auxi:wnn}$. 13747
 $\backslash\text{--fp_fixed_div_int_auxii:wnn}$
 658, 13747
 $\backslash\text{--fp_fixed_div_int_pack:Nw}$
 658, 658, 658, 658, 658, 13747
 $\backslash\text{--fp_fixed_div_myriad:wn}$
 13715, 14014
 $\backslash\text{--fp_fixed_inv_to_float_o:wn}$
 14080, 14405, 14674
 $\backslash\text{--fp_fixed_mul:nnnnnnnw}$ 13798
 $\backslash\text{--fp_fixed_mul:wnn}$
 656, 657, 702, 704, 13798,
 13963, 13994, 14009, 14011, 14015,
 14068, 14071, 14084, 14324, 14334,
 14374, 14465, 14483, 14583, 14684,
 15257, 15311, 15459, 15492, 15494
 $\backslash\text{--fp_fixed_mul_add:nnnnnnnn}$
 13866, 13868
 $\backslash\text{--fp_fixed_mul_add:nnnnwnnn}$
 13873, 13879
 $\backslash\text{--fp_fixed_mul_add:Nwnnnnnnn}$
 13831, 13841, 13851, 13855
 $\backslash\text{--fp_fixed_mul_add:wwn}$ 13825
 $\backslash\text{--fp_fixed_mul_after:wwn}$
 660, 13717, 13723, 13726,
 13800, 13827, 13837, 13847, 14701
 $\backslash\text{--fp_fixed_mul_one_minus_}$
 mul:wwn 13825
 $\backslash\text{--fp_fixed_mul_short:wwn}$. 13724,
 13992, 14013, 14055, 14057, 15505
 $\backslash\text{--fp_fixed_mul_sub_back:wwnn}$
 13825,
 14069, 15278, 15280, 15281, 15282,
 15283, 15284, 15285, 15286, 15287,
 15291, 15293, 15294, 15295, 15296,
 15297, 15298, 15299, 15324, 15326,
 15327, 15328, 15329, 15332, 15334,
 15335, 15336, 15337, 15472, 15480
 $\backslash\text{--fp_fixed_one_minus_mul:wwn}$
 661, 662, 13845
 $\backslash\text{--fp_fixed_sub:wwn}$
 13778, 14061, 14340, 14356,
 14368, 15003, 15453, 15510, 15576
 $\backslash\text{--fp_fixed_to_float_o:Nw}$
 14087, 14349
 $\backslash\text{--fp_fixed_to_float_o:wn}$
 656, 711, 14074, 14087,
 14369, 14379, 14403, 14670, 15500
 $\backslash\text{--fp_fixed_to_float_pack:ww}$
 14119, 14129

$\backslash\text{--fp_fixed_to_float_rad_o:wN}$
 14082, 15500
 $\backslash\text{--fp_fixed_to_float_round_}$
 up:wnnnnw 14132, 14136
 $\backslash\text{--fp_fixed_to_float_zero:w}$
 14115, 14124
 $\backslash\text{--fp_fixed_to_loop:N}$
 14092, 14102, 14106
 $\backslash\text{--fp_fixed_to_loop_end:w}$
 14108, 14112
 $\backslash\text{--fp_from_dim:wNNnnnnnn}$ 15796
 $\backslash\text{--fp_from_dim:wnnnnwNn}$. 15823, 15824
 $\backslash\text{--fp_from_dim:wnnnnwNw}$ 15796
 $\backslash\text{--fp_from_dim:wNw}$ 15796
 $\backslash\text{--fp_from_dim_test:ww}$
 720, 11539, 11552, 12115, 15796
 $\backslash\text{--fp_function_apply:nw}$
 .. 615, 615, 616, 12517, 12533, 12555
 $\backslash\text{--fp_function_args:Nwn}$.. 615, 12522
 $\backslash\text{--fp_function_store:wwNwnn}$
 616, 12555
 $\backslash\text{--fp_function_store_end:wnnn}$
 616, 12555
 $\backslash\text{c_fp_half_prec_int}$
 10648, 11756, 11788
 $\backslash\text{--fp_inf_fp:N}$ 10661, 11012
 $\backslash\text{--fp_int:wTF}$ 10838, 15961
 $\backslash\text{--fp_int:p:w}$ 10838
 $\backslash\text{--fp_invalid_operation:nnw}$
 564, 564, 10954, 11036,
 11048, 15654, 15661, 15695, 15702
 $\backslash\text{--fp_invalid_operation_o:nw}$ 564,
 11047, 13499, 14162, 14858, 14873,
 14888, 14903, 14918, 14933, 15525,
 15543, 15559, 15587, 15600, 15616
 $\backslash\text{--fp_invalid_operation_o:Nww}$
 564, 10962,
 11036, 12998, 13270, 13271, 14769
 $\backslash\text{--fp_invalid_operation_tl_o:nn}$
 .. 564, 10971, 11036, 11307, 15981
 $\backslash\text{c_fp_leading_shift_int}$
 10741, 13718,
 13727, 13801, 14702, 15198, 15235
 $\backslash\text{--fp_ln_c:NwNw}$ 678, 679, 14306, 14337
 $\backslash\text{--fp_ln_div_after:Nw}$
 677, 677, 14209, 14258
 $\backslash\text{--fp_ln_div_i:w}$ 14231, 14240
 $\backslash\text{--fp_ln_div_ii:wwn}$
 .. 14234, 14235, 14236, 14237, 14245
 $\backslash\text{--fp_ln_div_vi:wwn}$.. 14238, 14253
 $\backslash\text{--fp_ln_exponent:wn}$ 679, 14185, 14346
 $\backslash\text{--fp_ln_exponent_one:ww}$ 14351, 14365
 $\backslash\text{--fp_ln_exponent_small:NNww}$
 14354, 14358, 14371

```

\c_fp_ln_i_fixed_t1 ..... 14150
\c_fp_ln_ii_fixed_t1 ..... 14150
\c_fp_ln_iii_fixed_t1 ..... 14150
\c_fp_ln_iv_fixed_t1 ..... 14150
\c_fp_ln_ix_fixed_t1 ..... 14150
\c_fp_ln_npos_o:w ..... 673, 673, 14171, 14173
\c_fp_ln_o:w .. 673, 688, 14149, 14159
\c_fp_ln_significand>NNNNnnnn .. 674, 14184, 14187, 14682
\c_fp_ln_square_t_after:w ..... 14282, 14313
\c_fp_ln_square_t_pack:NNNNNw ... 14284, 14286, 14288, 14290, 14311
\c_fp_ln_t_large:Nw ..... 678, 14263, 14270, 14280
\c_fp_ln_t_small:Nw ... 14261, 14268
\c_fp_ln_t_small:w ..... 678
\c_fp_ln_Taylor:wwNw 678, 14314, 14315
\c_fp_ln_Taylor_break:w 14320, 14331
\c_fp_ln_Taylor_loop:www ..... 14316, 14317, 14326
\c_fp_ln_twice_t_after:w 14294, 14310
\c_fp_ln_twice_t_pack:Nw .. 14296, 14298, 14300, 14302, 14304, 14309
\c_fp_ln_vi_fixed_t1 ..... 14150
\c_fp_ln_vii_fixed_t1 ..... 14150
\c_fp_ln_viii_fixed_t1 ..... 14150
\c_fp_ln_x_fixed_t1 ..... 14150, 14368, 14375
\c_fp_ln_x_ii:wnnnn .. 14189, 14207
\c_fp_ln_x_iii:NNNNNnw .. 14216, 14220
\c_fp_ln_x_iii_var:NNNNNw ..... 14214, 14222
\c_fp_ln_x_iv:wnnnnnnn .. 676, 14212, 14227
\c_fp_max_exp_exponent_int ... 10654, 14411
\c_fp_max_exponent_int .. 10652, 10658, 10679, 13925, 14126, 14737
\c_fp_middle_shift_int ..... 10741, 13730, 13733, 13736, 13739, 13803, 13806, 13809, 13812, 14704, 14707, 14710, 14713, 15201, 15208, 15238, 15244
\c_fp_minmax_auxi:ww ..... 12839, 12851, 12858
\c_fp_minmax_auxii:ww ..... 12841, 12849, 12858
\c_fp_minmax_break_o:w .. 12832, 12862
\c_fp_minmax_loop:Nw .. 623, 12826, 12828, 12834
\c_fp_minmax_o:Nw ..... 617, 12603, 12605, 12823
\c_fp_minus_min_exponent_int ... 10652, 10680
\c_fp_mul_cases_o:NnNnww ..... 639, 13235, 13241, 13345
\c_fp_mul_cases_o:nNnww .. 13241
\c_fp_mul_npos_o:Nww ..... 636, 637, 639, 720, 720, 13238, 13279, 15826
\c_fp_mul_significand_drop:NNNNNw ... 638, 13288
\c_fp_mul_significand_keep:NNNNNw ... 13288
\c_fp_mul_significand_large_f:NwwNNNN .. 13316, 13320
\c_fp_mul_significand_o:nnnnNnnn .. 637, 637, 13286, 13288
\c_fp_mul_significand_small_f:NNNwwN .. 13314, 13331
\c_fp_mul_significand_test_f:NNN .. 638, 13290, 13311
\c_fp_myriad_int ..... 10651, 13713, 13744, 13745, 13822, 13882
\c_fp_neg_sign:N ..... 627, 10674, 12952, 13105
\c_fp_new_function:NNnnn .. 12522
\c_fp_not_o:w .. 617, 12028, 12864
\c_fp_one_fixed_t1 ..... 13707, 14322, 14478, 14738, 14762, 15405, 15471, 15576
\c_fp_overflow:w ..... 554, 564, 566, 566, 10682, 11036, 14413
\c_fp_overflowing_fp ..... 10655, 15655, 15696
\c_fp_pack:NNNNNw .. 10741, 13719, 13729, 13732, 13735, 13738, 13741, 13802, 13805, 13808, 13811, 13814, 14703, 14706, 14709, 14712, 14715
\c_fp_pack_big:NNNNNNw .. 10745, 13548, 13551, 13554, 13557, 13560, 13563, 13566, 13570, 13829, 13839, 13849, 13858, 13861, 13864, 13871
\c_fp_pack_Bigg:NNNNNNw ..... 10750, 13398, 13401, 13404, 13415, 13418, 13421
\c_fp_pack_eight:wNNNNNNNN .. 634, 10757, 13214, 13519, 13893, 14970, 14971
\c_fp_pack_twice_four:wNNNNNNNN .. 10755, 11356, 11357, 13156, 13157, 13894, 13895, 13896, 13928, 13929, 13930, 14117, 14118, 14450, 14451, 14452, 14972, 14973, 15186, 15819
\c_fp_parse:n .. 588, 606, 616, 621, 721, 727, 728, 11387, 11536, 12139, 12156, 12557, 12611, 12625, 12635,

```

12746, 12789, 15643, 15684, 15749,
 15789, 15835, 15837, 15839, 15979,
 16015, 16036, 16038, 16040, 16063
 $\backslash__fp_parse_after:w\text{w}$ 12139
 $\backslash__fp_parse_apply_binary:NwNwN$..
 580, 581, 581,
 581, 611, 12169, 12274, 12297, 12348
 $\backslash__fp_parse_apply_compare:NwNNNNNwN$
 12466, 12475
 $\backslash__fp_parse_apply_compare_{-}$
 aux:NwN 12487, 12490, 12495
 $\backslash__fp_parse_apply_juxtapose:NwwN$
 611, 12325
 $\backslash__fp_parse_apply_unary:NNNwN$...
 12005, 12014, 12123, 12132
 $\backslash__fp_parse_caseless_inf:N$... 12089
 $\backslash__fp_parse_caseless_infinity:N$..
 12089
 $\backslash__fp_parse_caseless_nan:N$... 12089
 $\backslash__fp_parse_compare:NNNNNNN$.. 12408
 $\backslash__fp_parse_compare_auxi:NNNNNNN$
 12408
 $\backslash__fp_parse_compare_auxii:NNNNNN$..
 12408
 $\backslash__fp_parse_compare_end:NNNNw$. 12408
 $\backslash__fp_parse_continue:NwN$
 580, 581, 581, 581, 581, 607, 12158,
 12171, 12505, 12902, 12926, 12934
 $\backslash__fp_parse_continue_compare:NNwNN$
 12498, 12513
 $\backslash__fp_parse_digits_N$ 11417
 $\backslash__fp_parse_digits_i:N$ 11417
 $\backslash__fp_parse_digits_ii:N$ 11417
 $\backslash__fp_parse_digits_iii:N$ 11417
 $\backslash__fp_parse_digits_iv:N$ 11417
 $\backslash__fp_parse_digits_v:N$ 11417
 $\backslash__fp_parse_digits_vi:N$
 11417, 11714, 11762
 $\backslash__fp_parse_digits_vii:N$
 593, 11417, 11701, 11751
 $\backslash__fp_parse_excl_error:$ 12408
 $\backslash__fp_parse_expand:w$
 584, 585, 11402, 11404,
 11426, 11466, 11511, 11544, 11548,
 11586, 11619, 11657, 11659, 11678,
 11680, 11702, 11719, 11732, 11752,
 11782, 11810, 11826, 11837, 11860,
 11889, 11899, 11906, 11919, 11935,
 11955, 11966, 12024, 12047, 12059,
 12128, 12137, 12145, 12225, 12239,
 12262, 12301, 12332, 12372, 12392,
 12460, 12473, 12520, 12540, 12898
 $\backslash__fp_parse_exponent:N$
 597, 11510, 11693, 11842, 11909, 11911
 $\backslash__fp_parse_exponent:Nw$
 11717, 11730,
 11779, 11807, 11858, 11887, 11906
 $\backslash__fp_parse_exponent_aux:N$... 11911
 $\backslash__fp_parse_exponent_body:N$..
 11937, 11941
 $\backslash__fp_parse_exponent_digits:N$...
 11945, 11957
 $\backslash__fp_parse_exponent_keep:N$.. 11968
 $\backslash__fp_parse_exponent_keep:NTF$...
 11948, 11968
 $\backslash__fp_parse_exponent_sign:N$...
 11927, 11931
 $\backslash__fp_parse_function:NNN$
 11095, 11097, 11099,
 11113, 12121, 12603, 12605, 14841,
 14843, 14845, 14847, 15869, 15871
 $\backslash__fp_parse_infix:NN$
 .. 587, 589, 604, 608, 616, 11465,
 11597, 11636, 12056, 12076, 12081,
 12096, 12118, 12178, 12181, 12237
 $\backslash__fp_parse_infix_!:N$ 12408
 $\backslash__fp_parse_infix_&:Nw$ 12365
 $\backslash__fp_parse_infix_(:N$ 12323
 $\backslash__fp_parse_infix_):N$ 12231
 $\backslash__fp_parse_infix_*:N$ 12350
 $\backslash__fp_parse_infix_+:N$
 616, 11402, 12291
 $\backslash__fp_parse_infix_,:N$ 12248
 $\backslash__fp_parse_infix_{-}:N$ 12291
 $\backslash__fp_parse_infix_{/}:N$ 12291
 $\backslash__fp_parse_infix_::N$.. 12382, 12887
 $\backslash__fp_parse_infix_<:N$ 12408
 $\backslash__fp_parse_infix_=:N$ 12408
 $\backslash__fp_parse_infix_>:N$ 12408
 $\backslash__fp_parse_infix_?:N$ 12382
 $\backslash__fp_parse_infix_^:N$ 12291
 $\backslash__fp_parse_infix_after_operand:NwN$
 589, 11501, 11559, 12031, 12176
 $\backslash__fp_parse_infix_and:N$ 12291, 12381
 $\backslash__fp_parse_infix_check:NNN$...
 12201, 12211
 $\backslash__fp_parse_infix_comma:w$ 12248
 $\backslash__fp_parse_infix_comma_error:w$..
 12248
 $\backslash__fp_parse_infix_end:N$
 606, 609, 12146, 12150, 12229
 $\backslash__fp_parse_infix_juxtapose:N$...
 611, 12191, 12199, 12324, 12325
 $\backslash__fp_parse_infix_mark:NNN$
 12188, 12228
 $\backslash__fp_parse_infix_mul:N$ 12291, 12358
 $\backslash__fp_parse_infix_or:N$. 12291, 12380
 $\backslash__fp_parse_infix_|:Nw$ 12365

`_fp_parse_large:N` 592, 11664, [11747](#)
`_fp_parse_large_leading:wwNN ..`
`.....` 11749, [11754](#)
`_fp_parse_large_round:NN ..`
`.....` 596, 11790, [11862](#)
`_fp_parse_large_round_aux:wNN ..`
`.....` [11862](#)
`_fp_parse_large_round_test:NN ..`
`.....` [11862](#)
`_fp_parse_large_trailing:wwNN ..`
`.....` 11760, [11784](#)
`_fp_parse_letters:N ..`
`.....` 589, 590, 11574, [11588](#)
`_fp_parse_lparen_after:NwN ..` [12037](#)
`_fp_parse_o:n ..`
`.....` 576, [12152](#), 12744, 12745, 15945, 16010
`_fp_parse_one:Nw ..` 580, 580, 581,
`.....` 582, 582, 582, 582, 583, 591, 604,
`.....` 607, [11402](#), [11437](#), 11641, 12004, 12164
`_fp_parse_one_digit:NN ..`
`.....` 603, [11453](#), [11557](#)
`_fp_parse_one_fp:NN ..`
`.....` 586, [11445](#), [11461](#)
`_fp_parse_one_other:NN` [11456](#), [11565](#)
`_fp_parse_one_register:NN ..`
`.....` [11448](#), [11499](#)
`_fp_parse_one_register_aux:Nw ..`
`.....` [11499](#)
`_fp_parse_one_register_-`
`auxii:wwwNw ..` [11499](#)
`_fp_parse_one_register_dim:ww ..`
`.....` [11499](#)
`_fp_parse_one_register_int:www ..`
`.....` [11499](#)
`_fp_parse_one_register_mu:www ..`
`.....` [11499](#)
`_fp_parse_one_register_wd:Nw` [11499](#)
`_fp_parse_one_register_wd:w ..` [11499](#)
`_fp_parse_operand:Nw ..` 580,
`.....` 580, 580, 582, 582, 582, 583,
`.....` 583, 606, 615, 615, [11402](#), 12020,
`.....` 12022, 12043, 12045, [12128](#), [12137](#),
`.....` 12144, [12158](#), 12261, 12300, 12331,
`.....` 12391, [12473](#), 12520, 12540, 12897
`_fp_parse_pack_carry:w ..` 594, [11734](#)
`_fp_parse_pack_leading>NNNNNww ..`
`.....` 11697, [11734](#), [11757](#)
`_fp_parse_pack_trailing>NNNNNNww ..`
`.....` [11707](#), [11734](#), [11776](#), [11787](#), [11794](#)
`_fp_parse_prefix:NNN ..` [11577](#), [11621](#)
`_fp_parse_prefix_!Nw ..` [12010](#)
`_fp_parse_prefix_(Nw ..` [12037](#)
`_fp_parse_prefix_)Nw ..` [12068](#)
`_fp_parse_prefix_+Nw ..` [12004](#)
`_fp_parse_prefix_-Nw ..` [12010](#)
`_fp_parse_prefix_.Nw ..` [12029](#)
`_fp_parse_prefix_unknown>NNN ..` [11621](#)
`_fp_parse_return_semicolon:w ..`
`.....` [11403](#), [11424](#), [11617](#),
`.....` 11824, 11835, 11917, 11949, 11964
`_fp_parse_round:Nw ..` 892, [11100](#), [21557](#)
`_fp_parse_round_after:wN ..`
`.....` 598, [11839](#), [11844](#), [11894](#)
`_fp_parse_round_deprecation_error:Nw ..` [11100](#), [21557](#)
`_fp_parse_round_loop:N ..` 597, 597,
`.....` 598, [11812](#), [11855](#), [11873](#), [11898](#)
`_fp_parse_round_up:N ..` [11812](#)
`_fp_parse_small:N ..` 592, [11684](#), [11695](#)
`_fp_parse_small_leading:wwNN ..`
`.....` 11699, [11704](#), [11766](#)
`_fp_parse_small_round:NN ..`
`.....` 11726, [11844](#), [11883](#)
`_fp_parse_small_trailing:wwNN ..`
`.....` 11712, [11721](#), [11798](#)
`_fp_parse_strim_end:w ..` [11670](#)
`_fp_parse_strim_zeros:N ..`
`.....` 592, 603, [11651](#), [11670](#), [12035](#)
`_fp_parse_trim_end:w ..` [11644](#)
`_fp_parse_trim_zeros:N ..` [11563](#), [11644](#)
`_fp_parse_unary_function>NNN ..`
`.....` [12121](#), [12944](#), [12946](#),
`.....` 12948, [14147](#), [14149](#), [14829](#), [14835](#)
`_fp_parse_word:Nw ..` 589, [11571](#), [11588](#)
`_fp_parse_word_abs:N ..` [12943](#)
`_fp_parse_word_acos:N ..` [14821](#)
`_fp_parse_word_acosd:N ..` [14821](#)
`_fp_parse_word_acot:N ..` [14840](#)
`_fp_parse_word_acotd:N ..` [14840](#)
`_fp_parse_word_acsc:N ..` [14821](#)
`_fp_parse_word_acsd:N ..` [14821](#)
`_fp_parse_word_asec:N ..` [14821](#)
`_fp_parse_word_asecd:N ..` [14821](#)
`_fp_parse_word_asin:N ..` [14821](#)
`_fp_parse_word_asind:N ..` [14821](#)
`_fp_parse_word_atan:N ..` [14840](#)
`_fp_parse_word_atand:N ..` [14840](#)
`_fp_parse_word_bp:N ..` [12092](#)
`_fp_parse_word_cc:N ..` [12092](#)
`_fp_parse_word_ceil:N ..` [11094](#)
`_fp_parse_word_cm:N ..` [12092](#)
`_fp_parse_word_cos:N ..` [14821](#)
`_fp_parse_word_cosd:N ..` [14821](#)
`_fp_parse_word_cot:N ..` [14821](#)
`_fp_parse_word_cotd:N ..` [14821](#)
`_fp_parse_word_csc:N ..` [14821](#)
`_fp_parse_word_cscd:N ..` [14821](#)
`_fp_parse_word_dd:N ..` [12092](#)

`_fp_parse_word_deg:N` 12078
`_fp_parse_word_em:N` 12111
`_fp_parse_word_ex:N` 12111
`_fp_parse_word_exp:N` 14146
`_fp_parse_word_false:N` 12078
`_fp_parse_word_floor:N` 11094
`_fp_parse_word_in:N` 12092
`_fp_parse_word_inf:N`
 12078, 12089, 12090
`_fp_parse_word_ln:N` 14146
`_fp_parse_word_max:N` 12602
`_fp_parse_word_min:N` 12602
`_fp_parse_word_mm:N` 12092
`_fp_parse_word_nan:N` . 12078, 12091
`_fp_parse_word_nc:N` 12092
`_fp_parse_word_nd:N` 12092
`_fp_parse_word_pc:N` 12092
`_fp_parse_word_pi:N` 12078
`_fp_parse_word_pt:N` 12092
`_fp_parse_word_rand:N` 15868
`_fp_parse_word_randint:N` ... 15868
`_fp_parse_word_round:N` 11100
`_fp_parse_word_sec:N` 14821
`_fp_parse_word_secd:N` 14821
`_fp_parse_word_sign:N` 12943
`_fp_parse_word_sin:N` 14821
`_fp_parse_word_sind:N` 14821
`_fp_parse_word_sp:N` 12092
`_fp_parse_word_sqrt:N` 12943
`_fp_parse_word_tan:N` 14821
`_fp_parse_word_tand:N` 14821
`_fp_parse_word_true:N` 12078
`_fp_parse_word_trunc:N` 11094
`_fp_parse_zero:`
 592, 11666, 11686, 11690
`_fp_pow_B:wwN` 14685, 14720
`_fp_pow_C_neg:w` 14723, 14740
`_fp_pow_C_overflow:w`
 14728, 14735, 14756
`_fp_pow_C_pack:w` 14742, 14750, 14761
`_fp_pow_C_pos:w` 14726, 14745
`_fp_pow_C_pos_loop:wN`
 14746, 14747, 14754
`_fp_pow_exponent:Nnnnnnnw`
 14691, 14694, 14699
`_fp_pow_exponent:wnN` . 14683, 14688
`_fp_pow_neg:www` .. 690, 14596, 14763
`_fp_pow_neg_aux:wNN` ... 690, 14763
`_fp_pow_neg_case:w` .. 14765, 14786
`_fp_pow_neg_case_aux:nnnnn` . 14786
`_fp_pow_neg_case_aux:w` 14786
`_fp_pow_normal_o:ww`
 685, 686, 14601, 14633
`_fp_pow_npos_aux>NNnww`
 14668, 14672, 14678
`_fp_pow_npos_o:Nww` 687, 14645, 14662
`_fp_pow_zero_or_inf:ww`
 685, 14603, 14610
`\c_fp_prec_and_int` ... 11387, 12320
`\c_fp_prec_colon_int`
 11387, 12403, 12897
`\c_fp_prec_comma_int` ... 11387,
 12043, 12070, 12252, 12257, 12261
`\c_fp_prec_comp_int`
 11387, 12431, 12473
`\c_fp_prec_end_int` 606, 11387, 12144
`\c_fp_prec_func_int` .. 11387, 12128
`\c_fp_prec_funcii_int`
 .. 11387, 12042, 12137, 12520, 12540
`\c_fp_prec_hat_int` ... 11387, 12310
`\c_fp_prec_hatii_int` . 11387, 12310
`\c_fp_prec_int` 10648, 10784, 10845,
 10872, 11325, 14431, 14800, 14804,
 15714, 15716, 15722, 15764, 15965
`\c_fp_prec_not_int`
 603, 11387, 12027, 12028
`\c_fp_prec_or_int` 11387, 12322
`\c_fp_prec_paren_int`
 11387, 12045, 12235
`\c_fp_prec_plus_int`
 579, 11387, 12316, 12318
`\c_fp_prec_quest_int`
 11387, 12386, 12401
`\c_fp_prec_times_int`
 .. 11387, 12312, 12314, 12327, 12331
`\c_fp_rand_eight_int`
 15885, 15904, 15907
`\c_fp_rand_four_int`
 .. 15885, 15910, 15913, 15926, 15929
`_fp_rand_myriads:n`
 723, 725, 15890, 15945, 16015
`_fp_rand_myriads_get:w` 15890
`_fp_rand_myriads_last:` 15890
`_fp_rand_myriads_last:w` 15890
`_fp_rand_myriads_loop:nn` ... 15890
`_fp_rand_o:` 15934
`_fp_rand_o:Nw`
 15869, 15876, 15881, 15934
`_fp_rand_o:w` 15934
`\c_fp_rand_size_int`
 903, 903, 15885, 15991,
 16002, 21953, 21954, 21965, 21974
`_fp_rand_uniform:` 15885,
 15900, 15922, 15989, 16005, 16023
`_fp_randint_badarg:w` 15946
`_fp_randint_e:w` 15946
`_fp_randint_e:wnn` 15946

```

\__fp_randint_e:wwNnn . . . . . 15946
\__fp_randint_e:wwwNnn . . . . . 725, 15946
\__fp_randint_narrow_e:nnnn . . 15946
\__fp_randint_o:Nw 15871, 15881, 15946
\__fp_randint_wide_e:nnnn . . . 15946
\__fp_randint_wide_e:wnnn . . . 15946
\__fp_reverse_args:Nww . . . . . 714,
    715, 10626, 15341, 15427, 15539, 15605
\__fp_round:NNN 570, 570, 572, 639,
    654, 11130, 11200, 13073, 13084,
    13326, 13338, 13478, 13489, 13673
\__fp_round:Nwn . 11250, 11303, 15794
\__fp_round:Nww . 11251, 11272, 11303
\__fp_round:Nwww . . . . . 11252, 11266
\__fp_round_digit:Nw . . . . . 559,
    638, 639, 654, 10802, 11214, 13087,
    13230, 13329, 13341, 13492, 13678
\__fp_round_name_from_cs:N . . .
    . . . . . 11262, 11288, 11292, 11308
\__fp_round_neg:NNN . . . . . 570,
    635, 635, 11225, 13192, 13207, 13225
\__fp_round_no_arg_o:Nw 11249, 11256
\__fp_round_normal:NnnwNNnn . 11303
\__fp_round_normal>NNwNnn . 11303
\__fp_round_normal:NwNNnw . 11303
\__fp_round_normal_end:wwNnn . 11303
\__fp_round_o:Nw . . . . . .
    . . . . . 11095, 11097, 11099, 11114, 11245
\__fp_round_pack:Nw . . . . . 11303
\__fp_round_return_one: . . .
    . . . . . 570, 11130, 11136,
    11146, 11154, 11158, 11167, 11171,
    11180, 11187, 11191, 11229, 11239
\__fp_round_s>NNNw . . . . .
    . . . . . 570, 597, 11198, 11848, 11866
\__fp_round_special:NwwNnn . . 11303
\__fp_round_special_aux:Nw . . 11303
\__fp_round_to_nearest:NNN . .
    . . . . . 573, 574, 11114, 11118, 11120,
    11130, 11234, 11258, 11268, 15794
\__fp_round_to_nearest_neg:NNN 11225
\__fp_round_to_nearest_ninf:NNN .
    . . . . . 574, 11130, 11243
\__fp_round_to_nearest_ninf_-
    neg:NNN . . . . . 11225
\__fp_round_to_nearest_pinf:NNN .
    . . . . . 574, 11130, 11235
\__fp_round_to_nearest_pinf_-
    neg:NNN . . . . . 11225
\__fp_round_to_nearest_zero:NNN .
    . . . . . 574, 11130
\__fp_round_to_nearest_zero_-
    neg:NNN . . . . . 11225

```

```

__fp_sqrt_Newton_o:wwn . . . . .
..... 650, 13514, 13525, 13526
\__fp_sqrt_npos_auxi_o:wwnnN . 13505
\__fp_sqrt_npos_auxii_o:wNNNNNNNN . 13505
\__fp_sqrt_npos_o:w . . . . . 13502, 13505
\__fp_sqrt_o:w . . . . . 12948, 13495
\__fp_step:NNnnnn . . . . . 12794
\__fp_step:NnnnnN . . . . . 12741
\__fp_step:wwwN . . . . . 12741
\__fp_sub_back_far_o:NnnwnnnnN . . .
..... 634, 13119, 13165
\__fp_sub_back_near_after:wNNNNw . . .
..... 13125, 13203
\__fp_sub_back_near_o:nnnnnnnnnN . . .
..... 632, 13115, 13125
\__fp_sub_back_near_pack:NNNNNNw . . .
..... 13125, 13205
\__fp_sub_back_not_far_o:wwwNN . . .
..... 13180, 13200
\__fp_sub_back_quite_far_ii:NN 13184
\__fp_sub_back_quite_far_o:wwNN . . .
..... 13178, 13184
\__fp_sub_back_shift:wnnnn . . .
..... 633, 13137, 13141
\__fp_sub_back_shift_ii:ww . . . 13141
\__fp_sub_back_shift_iii:NNNNNNNNw . . .
..... 13141
\__fp_sub_back_shift_iv:nnnnw . 13141
\__fp_sub_back_very_far_ii_-o:nnNwwNN . . .
..... 13212
\__fp_sub_back_very_far_o:wwwNN . . .
..... 13179, 13212
\__fp_sub_eq_o:Nnwnw . . . . . 13090
\__fp_sub_npos_i_o:Nnwnw . . .
..... 632, 13095, 13104, 13108
\__fp_sub_npos_ii_o:Nnwnw . . . 13090
\__fp_sub_npos_o:Nnwnw . . .
..... 631, 13010, 13090
\__fp_tan_o:w . . . . . 14908
\__fp_tan_series_aux_o:Nnwww . 15309
\__fp_tan_series_o:NNwww . . .
..... 694, 695, 14915, 14930, 15309
\__fp_ternary:NwwN . 617, 12401, 12885
\__fp_ternary_auxi:NwwN . 617, 12885
\__fp_ternary_auxii:NwwN . . .
..... 617, 12403, 12885
\__fp_ternary_break_point:n . 12885
\__fp_ternary_loop:Nw . . . . . 12885
\__fp_ternary_loop_break:w . . . . . 12885
\__fp_ternary_map_break: . . . . . 12885
\__fp_tmp:w . . . . . 559, 610, 10796,
10806, 10807, 10808, 10809, 10810,
10811, 10812, 10813, 10814, 10815,
10816, 10817, 10818, 10819, 10820,
10821, 11417, 11429, 11430, 11431,
11432, 11433, 11434, 11435, 11481,
11497, 12010, 12027, 12028, 12078,
12083, 12084, 12085, 12086, 12087,
12088, 12092, 12100, 12101, 12102,
12103, 12104, 12105, 12106, 12107,
12108, 12109, 12110, 12231, 12247,
12248, 12266, 12278, 12290, 12291,
12309, 12311, 12313, 12315, 12317,
12319, 12321, 12350, 12364, 12365,
12380, 12381, 12382, 12400, 12402
\__fp_to_decimal_dispatch:w . . .
..... 717, 719,
720, 12787, 15679, 15683, 15686, 15793
\__fp_to_decimal_huge:wnnnn . . . 15686
\__fp_to_decimal_large:Nnnw . . . 15686
\__fp_to_decimal_normal:wnnnnn . . .
..... 15686, 15765
\__fp_to_int_dispatch:w . . .
..... 15784, 15788, 15791
\__fp_to_scientific_dispatch:w . . .
..... 716, 717, 719, 15638, 15642, 15645
\__fp_to_scientific_normal:wnnnnn . . .
..... 15645
\__fp_to_scientific_normal:wNw 15645
\__fp_to_tl_dispatch:w . . .
..... 719, 15744, 15748, 15751, 15861
\__fp_to_tl_normal:nnnnn . . . . . 15751
\__fp_to_tl_scientific:wnnnnn . 15751
\__fp_to_tl_scientific:wNw . . . . . 15751
\c_fp_trailing_shift_int . . .
..... 10741, 13720,
13742, 13815, 14716, 15201, 15238
\__fp_trap_division_by_zero_set:N . . .
..... 10979
\__fp_trap_division_by_zero_set_error: . . .
..... 10979
\__fp_trap_division_by_zero_set_flag: . . .
..... 10979
\__fp_trap_division_by_zero_set_none: . . .
..... 10979
\__fp_trap_invalid_operation_set:N . . .
..... 10945
\__fp_trap_invalid_operation_set_error: . . .
..... 10945
\__fp_trap_invalid_operation_set_flag: . . .
..... 10945
\__fp_trap_invalid_operation_set_none: . . .
..... 10945
\__fp_trap_overflow_set:N . . . . . 11005
\__fp_trap_overflow_set:NnNn . 11005
\__fp_trap_overflow_set_error: 11005
\__fp_trap_overflow_set_flag: . . . . . 11005

```

```

\__fp_trap_overflow_set_none: . 11005
\__fp_trap_underflow_set:N . 11005
\__fp_trap_underflow_set_error: .
. . . . . 11005
\__fp_trap_underflow_set_flag: 11005
\__fp_trap_underflow_set_none: 11005
\__fp_trig>NNNNNwn . 14854, 14869,
14884, 14899, 14914, 14929, 14946
\__fp_trig_inverse_two_pi: .
. . . . . 701, 15007, 15176
\__fp_trig_large:ww . 14954, 15171
\__fp_trig_large_auxi:wwwww . 15171
\__fp_trig_large_auxii:ww . 15171
\__fp_trig_large_auxiii:wNNNNNNNN .
. . . . . 15171
\__fp_trig_large_auxiv:wN . 15171
\__fp_trig_large_auxix:Nw . 15214
\__fp_trig_large_auxv:www .
. . . . . 15187, 15194
\__fp_trig_large_auxvi:wnnnnnnnn .
. . . . . 15194
\__fp_trig_large_auxvii:w .
. . . . . 15197, 15214
\__fp_trig_large_auxviii:w . 15214
\__fp_trig_large_auxviii:ww .
. . . . . 15216, 15220
\__fp_trig_large_auxxx:wNNNNN . 15214
\__fp_trig_large_auxxi:w . 15214
\__fp_trig_large_pack:NNNNNw .
. . . . . 15194, 15243
\__fp_trig_small:ww .
. . . . . 696, 703, 14956, 14960, 14966, 15253
\__fp_trigd_large:ww . 14954, 14968
\__fp_trigd_large_auxi:nnnnwNNNN .
. . . . . 14968
\__fp_trigd_large_auxii:wNw . 14968
\__fp_trigd_large_auxiii:www . 14968
\__fp_trigd_small:ww .
. . . . . 697, 14956, 14962, 15005
\__fp_trim_zeros:w .
. . . . . 15629, 15727, 15736, 15778
\__fp_trim_zeros_dot:w . 15629
\__fp_trim_zeros_end:w . 15629
\__fp_trim_zeros_loop:w . 15629
\__fp_type_from_scan:N .
. . . . . 586, 10736, 11405, 11463
\__fp_type_from_scan:w . 11405
\__fp_underflow:w .
. . . . . 554, 564, 566, 566, 10683, 11036, 14414
\__fp_use_i:ww .
. . . . . 664, 713, 10628, 13931, 15558
\__fp_use_i:www . . . . . 10628
\__fp_use_i_until_s:nw .
. . . . . 702, 10623, 10670, 10846, 10905,
12283, 14998, 15196, 15202, 15233
\__fp_use_ii_until_s:nnw 10623, 10668
\__fp_use_none_stop_f:n .
. . . . . 10620, 14095, 14096, 14097
\__fp_use_none_until_s:w .
. . . . . 10623, 13531, 14772, 15553, 15556
\__fp_use_s:n .
. . . . . 10621
\__fp_use_s:nn .
. . . . . 10621
\__fp_zero_fp:N . 10661, 11020, 11370
\__fp_l_o:ww . . . . . 617, 12872
\__fp . . . . . 12875, 12882
\futurelet . . . . . 374

G
\gdef . . . . . 375
\GetIdInfo . . . . . 6, 6, 6
\gleaders . . . . . 887
\global . . . . . 166, 167,
181, 182, 183, 194, 195, 196, 197,
198, 199, 200, 201, 202, 205, 272, 376
\globaldefs . . . . . 377
\glueexpr . . . . . 631
\glueshrink . . . . . 632
\glueshrinkorder . . . . . 633
\gluestretch . . . . . 634
\gluestretchorder . . . . . 635
\gluetomu . . . . . 636
group commands:
\group_align_safe_begin/end: 418, 748
\group_align_safe_begin: . 97, 97,
97, 324, 328, 410, 2837, 3151, 5795,
5999, 7123, 7141, 16555, 22330, 23120
\group_align_safe_end: . . . . .
. . . . . 97, 97, 97, 324,
328, 2860, 3177, 5843, 5844, 5999,
7105, 7115, 7120, 7138, 16559, 22351
\group_begin: . . . . . 9,
9, 9, 888, 1324, 2038, 2045, 2469,
2702, 2708, 2736, 2936, 3428, 3451,
3601, 3624, 3947, 3966, 4296, 6669,
6675, 6736, 6809, 6833, 6851, 6875,
6965, 6983, 7199, 7694, 7715, 7778,
8024, 8290, 8457, 8920, 8955, 9670,
12872, 16188, 16554, 16570, 16643,
16782, 16857, 16975, 17181, 18878,
19369, 19512, 19521, 19533, 19542,
19550, 19669, 19698, 20142, 20157,
20159, 20166, 20176, 20191, 20194,
20204, 20206, 20210, 20215, 20220,
20228, 20237, 20247, 21399, 22279,
22304, 22880, 22889, 22911, 23060,
23082, 23367, 24422, 24556, 24839

```

\c_group_begin_token	45, 111, 338, 442, 3284, 3319, 6833, 6857, 16608, 20175, 20246	\hbox_set_end:	212, 212, 20172, 20699
\group_end:	9, 9, 9, 9, 366, 888, 1324, 2041, 2045, 2472, 2704, 2716, 2794, 2940, 2943, 3446, 3471, 3606, 3629, 3973, 4078, 4306, 4311, 6677, 6684, 6812, 6830, 6850, 6854, 6882, 6982, 7028, 7264, 7701, 7726, 7883, 8069, 8306, 8491, 8924, 8979, 9721, 12883, 16207, 16447, 16560, 16590, 16652, 16806, 16859, 16977, 17192, 18882, 18883, 19430, 19517, 19528, 19612, 19675, 19735, 20148, 20157, 20159, 20166, 20184, 20191, 20194, 20204, 20206, 20210, 20215, 20220, 20228, 20237, 20256, 21403, 22286, 22309, 22903, 22910, 23075, 23081, 23106, 23391, 24427, 24564, 24844	\hbox_set_to_wd:Nnn	212, 212, 20163
\c_group_end_token	111, 442, 6833, 6862, 16611, 20185, 20257	\hbox_to_wd:nn	212, 212, 20188, 20548
\group_insert_after:N	9, 9, 9, 1329, 17189, 23599, 23732, 23810, 24143, 24538	\hbox_to_zero:n	212, 212, 20188, 20196, 20198
groups commands:		\hbox_unpack:N	213, 213, 20199, 20980, 21127
.groups:n	165, 10168	\hbox_unpack_clear:N	213, 213, 20199
H			
\H	23109	\hcoffin commands:	
\halign	378	\hcoffin_set:Nn	218, 218, 20646, 21187, 21199, 21242, 21282
\hangafter	379	\hcoffin_set:Nw	218, 218, 20692
\hangindent	380	\hcoffin_set_end:	218, 218, 20692
\hbadness	381	\hfil	383
\hbox	382	\hfill	384
hbox commands:		\hfilneg	385
\hbox:n	212, 212, 20156, 20309, 20557, 21190, 21245	\hfuzz	386
\hbox_gset:Nn	212, 20158	\hjcode	882
\hbox_gset:Nw	212, 20172	\hoffset	387
\hbox_gset_end:	212, 20172	\holdinginserts	388
\hbox_gset_to_wd:Nnn	212, 20163	\hpack	883
\hbox_overlap_left:n	212, 212, 20195	\hrule	389
\hbox_overlap_right:n	212, 212, 20195, 23539, 23578, 23690, 23716, 24095, 24127, 24522	\hsize	390
\hbox_set:Nn	212, 212, 212, 20158, 20280, 20305, 20306, 20392, 20430, 20445, 20460, 20472, 20488, 20518, 20530, 20650, 20751, 20976, 21047, 21335, 21564, 21568, 21576, 21584, 21593, 21602, 21614, 21622, 21630, 21636, 21649, 21696, 21710, 23653, 23849, 24041	\hskip	391
\hbox_set:Nw	212, 212, 20172, 20696	\hss	392
I			
\I	195	\ht	393
\i	198, 23104	hundred commands:	
\if commands:		\c_one_hundred	78, 5394
\if:w	21, 107, 107, 107, 283, 284, 284, 284, 311, 1297, 1597, 1947, 1948, 2395, 2398, 2399, 2453, 2456, 3993, 5264, 7075, 9747, 9751, 9769, 11646, 11650, 11672, 11765, 11797, 11816, 11882, 11896, 11913, 11933, 12354, 12369, 14667	\hyphenation	394
\if_bool:N	96, 96, 96, 5675, 5707, 7592	\hyphenationbounds	884
\if_box_empty:N	217, 217, 20088, 20104	\hyphenationmin	885
\if_case:w	79, 79, 349, 349, 351, 390, 560, 639, 683, 683, 690, 1863, 3696, 3770, 4614, 5155, 5188, 6824, 9052, 10678, 10841, 10856, 11247, 11276, 12442, 12482, 12957,	\hyphenchar	395
		\hyphenpenalty	396
		\hyphenpenaltytymode	886

13092, 13167, 13192, 13244, 13684,
 13701, 13940, 14164, 14191, 14348,
 14383, 14486, 14502, 14518, 14534,
 14550, 14566, 14591, 14643, 14765,
 14788, 14850, 14865, 14880, 14895,
 14910, 14925, 15365, 15382, 15435,
 15518, 15533, 15585, 15598, 15648,
 15689, 15754, 15948, 15974, 16619,
 16792, 17141, 17409, 18156, 18172,
 18229, 18639, 18694, 19023, 19325
`\if_catcode:w` 21, 329, 338, 339, 451,
1297, 2972, 3275, 3317, 3329, 3346,
 6857, 6862, 6867, 6872, 6879, 6886,
 6891, 6896, 6901, 6906, 6911, 6921,
 6950, 7157, 7162, 11439, 11606,
 11923, 11970, 12183, 12341, 16608,
 16611, 16748, 16750, 16752, 16753,
 16754, 16755, 16756, 23373, 23374
`\if_charcode:w` 21, 107, 337, 338, 353,
 451, 753, 1297, 3256, 3310, 3853,
 6926, 7159, 10844, 16667, 16690,
 16734, 17349, 17359, 17816, 19195
`\if_cs_exist:N`
 21, 1311, 1628, 1656, 6958, 7084
`\if_cs_exist:w` 21, 1311, 1337, 1637,
 1665, 1851, 5485, 5510, 5519, 5555
`\if_dim:w`
 160, 160, 9226, 9303, 9315, 9340
`\if_eof:w` 145, 145, 8719, 8726
`\if_false:` 21, 91, 324,
 328, 336, 369, 381, 418, 745, 745,
 834, 1297, 2855, 2856, 2957, 2961,
 3224, 3229, 3240, 3330, 3342, 3357,
 3365, 4247, 4250, 4385, 4390, 4801,
 6000, 6767, 9031, 9070, 9074, 9081,
 9089, 9325, 16546, 16597, 16647,
 16651, 17308, 17310, 17334, 17338,
 17398, 17402, 17588, 17607, 17608,
 17614, 17662, 17695, 17706, 17708,
 17931, 17964, 17976, 17980, 18008,
 18013, 18021, 18055, 18060, 18065,
 18103, 18303, 18320, 18324, 19385,
 19399, 19628, 19633, 19740, 19745
`\if_hbox:N` 217, 217, 20088, 20092
`\if_int_compare:w` 21,
 79, 79, 381, 381, 382, 382, 1327,
 3500, 3508, 3523, 3528, 3683, 3735,
 3736, 3742, 3754, 3770, 3996, 4020,
 4031, 4032, 4039, 4051, 4614, 4647,
 4717, 4781, 4834, 4836, 4838, 4840,
 4842, 4844, 4846, 4849, 6000, 6002,
 6704, 6707, 6708, 6715, 6716, 6717,
 6718, 6723, 6724, 6942, 7066, 9038,
 9536, 10679, 10680, 10784, 10881,
 11135, 11145, 11153, 11166, 11179,
 11186, 11207, 11219, 11228, 11238,
 11332, 11337, 11421, 11451, 11567,
 11569, 11605, 11610, 11663, 11683,
 11710, 11724, 11759, 11786, 11814,
 11830, 11846, 11864, 11923, 11943,
 11959, 11972, 11986, 12019, 12042,
 12070, 12184, 12194, 12196, 12235,
 12252, 12257, 12295, 12327, 12386,
 12431, 12623, 12653, 12656, 12667,
 12670, 12675, 12676, 12679, 12682,
 12844, 12918, 12961, 12982, 13019,
 13114, 13168, 13169, 13172, 13175,
 13245, 13254, 13455, 13528, 13581,
 13585, 13589, 13607, 13642, 13643,
 13644, 13645, 13646, 13672, 13942,
 13945, 14039, 14131, 14177, 14193,
 14319, 14353, 14411, 14420, 14460,
 14636, 14638, 14649, 14667, 14690,
 14722, 14725, 14768, 14800, 14804,
 14953, 14997, 15386, 15424, 15433,
 15468, 15552, 15555, 15737, 15904,
 15910, 15926, 15964, 16124, 16194,
 16262, 16270, 16281, 16284, 16302,
 16337, 16347, 16357, 16583, 16657,
 16677, 16684, 16702, 16726, 16773,
 16788, 16871, 16874, 16958, 16993,
 17002, 17081, 17087, 17088, 17095,
 17098, 17101, 17107, 17108, 17112,
 17115, 17116, 17124, 17125, 17126,
 17131, 17160, 17161, 17406, 17427,
 17428, 17429, 17432, 17436, 17437,
 17440, 17441, 17449, 17450, 17453,
 17457, 17458, 17461, 17508, 17530,
 17542, 17551, 17559, 17562, 17572,
 17575, 17599, 17666, 17765, 17830,
 17835, 17863, 17930, 17962, 18064,
 18334, 18364, 18574, 18657, 18684,
 18749, 18762, 18773, 18789, 18840,
 18872, 19001, 19002, 19051, 19078,
 19153, 19206, 19258, 19268, 19270,
 19286, 19340, 19381, 19395, 19418,
 19553, 19581, 19626, 19631, 19652,
 19726, 19738, 19743, 22156, 22164,
 22172, 22825, 22826, 22831, 23322,
 23332, 23341, 23351, 23352, 23358
`\if_int_odd:w` 79, 79,
 705, 4614, 4714, 4886, 4894, 5400,
 6714, 6722, 6742, 11157, 11204,
 11216, 12478, 13228, 13510, 14803,
 14814, 15223, 15262, 15272, 15315,
 15339, 15509, 16798, 17150, 17519,
 17527, 17539, 17935, 18212, 23372
`\if_meaning:w`

21	326, 326, 338, 462, 624	
1297	1460, 1475, 1493, 1543, 1548	
1557	1625, 1643, 1653, 1671, 1881	
1892	2008, 2053, 2100, 2101, 2338	
2366	2375, 2634, 2888, 2900, 2913	
2924	2939, 3169, 3233, 3301, 3610	
3633	3790, 3828, 3984, 4066, 4283	
4301	4320, 4328, 4628, 4663, 4668	
4669	4816, 5575, 5581, 5607, 5620	
5628	5752, 5818, 5837, 6140, 6155	
6177	6193, 6916, 6955, 6993, 6996	
7058	7150, 7191, 7496, 7534, 9284	
9333	9686, 9710, 9727, 9738, 10667	
10681	10693, 10703, 10763, 10772	
10863	10878, 10880, 11028, 11102	
11105	11108, 11134, 11144, 11156	
11169	11170, 11189, 11190, 11204	
11205	11216, 11217, 11275, 11314	
11349	11352, 11368, 11375, 11440	
11443	11504, 11505, 11506, 11623	
11736	11742, 11971, 12213, 12282	
12479	12497, 12615, 12642, 12643	
12644	12645, 12646, 12647, 12825	
12837	12838, 12866, 12877, 12887	
12958	12993, 13007, 13053, 13060	
13136	13148, 13248, 13251, 13262	
13313	13386, 13454, 13457, 13464	
13497	13498, 13501, 13913, 13924	
14104	14114, 14161, 14260, 14339	
14388	14402, 14588, 14600, 14613	
14616	14619, 14622, 14648, 14749	
14753	15265, 15318, 15380, 15381	
15383	15384, 15404, 15421, 15487	
15585	15647, 15688, 15753, 15805	
15810	15963, 16135, 16136, 16605	
16635	16746, 17188, 17507, 17812	
17815	18219, 18621, 18797, 18808	
18823	18950, 18976, 19298, 19504	
19580	19638, 19674, 22038, 23375	
\if_mode_horizontal:	21, 1307, 5994	
\if_mode_inner:	21, 1307, 5996	
\if_mode_math:	21, 1307, 5998	
\if_mode_vertical:	21, 1307, 5992	
\if_predicate:w	89, 91, 96, 96, 5675, 5787	
\if_true:	21, 91, 326, 326, 1297	
\if_vbox:N	217, 217, 20088, 20094	
\ifabsdim	954	
\ifabsnum	955	
\ifcase	398	
\ifcat	399	
\ifcsname	637	
\ifdbox	1127	
\ifddir	1128	
\ifdefined	158, 638	
\ifdim	400	
\ifeof	401	
\iffalse	402	
\iffontchar	639	
\ifhbox	403	
\ifhmode	404	
\ifincsname	780	
\ifinner	405	
\ifmdir	1129	
\ifmmode	406	
\ifnum	45, 60, 89, 95, 100, 164, 179, 407	
\ifodd	408	
\ifpdfabsdim	742	
\ifpdfabsnum	743	
\ifpdfprimitive	744	
\ifprimitive	856	
\iftbox	1130	
\iftdir	1131	
\iftrue	409	
\ifvbox	410	
\ifvmode	411	
\ifvoid	412	
\ifx	14, 21, 39, 43, 49, 90, 92, 98, 123, 145, 146, 413	
\ifybox	1132	
\ifydir	1133	
\ignoreligaturesinfont	956	
\ignorespaces	414	
\IJ	23095	
\ij	23095	
image internal commands:		
__image_extract_bb:n	23741, 23747	
\l__image_ht_dim	23634, 23655, 23660, 23842, 23850, 23853	
\l__image_interpolate_bool	23614, 23622, 23648, 23766, 23792	
\l__image_interpolate_t1	23746	
\l__image_page_int	23611, 23625, 23626, 23650, 23651, 23740, 23764, 23765, 23789, 23790, 23823, 23831, 23832	
__image_read_bb:n	23736, 24147	
\l__image_tmp_box	23653, 23655, 23656, 23849, 23850, 23851	
\l__image_wd_dim	23636, 23656, 23662, 23843, 23851, 23855	
\immediate	415	
in	191	
\indent	416	
inf	190	
inherit commands:		
.inherit:n	165, 10170	
\inhibitglue	1134	

\inhibitxspcode 1135
 \initcatcodetable 888
 initial commands:
 .initial:n 166, [10172](#)
 \input 50, [159](#), [160](#), [417](#)
 \inputlineno 418
 \insert 419
 \inserttht 957
 \insertpenalties 420
 int commands:
 \c_eight 78, [5375](#)
 \c_eleven 78, [5375](#)
 \c_fifteen 78, [5375](#)
 \c_five 78, [5375](#)
 \l_foo_int 204
 \c_four 78, [5375](#)
 \c_fourteen 78, [5375](#)
 \int_abs:n 68, [68](#), [4621](#), [5445](#)
 \int_add:Nn [70](#), [70](#), [4747](#), [9137](#), [17132](#),
 [18214](#), [18779](#), [18780](#), [18985](#), [19075](#)
 \int_case:nn
 72, [72](#), [390](#), [4855](#), [5018](#), [5024](#)
 \int_case:nnm 21518
 \int_case:nnTF 23, [72](#),
 72, [4532](#), [4855](#), [4860](#), [4865](#), [6438](#),
 [18470](#), [19998](#), [21518](#), [22717](#), [22772](#)
 \int_compare:nNnTF 70, [71](#),
 71, [72](#), [72](#), [73](#), [73](#), [2710](#), [2740](#), [2754](#),
 2758, [2783](#), [3377](#), [3384](#), [3662](#), [3664](#),
 3673, [3912](#), [4445](#), [4452](#), [4695](#), [4701](#),
 [4847](#), [4879](#), [4930](#), [4938](#), [4947](#), [4953](#),
 4965, [4968](#), [4979](#), [5014](#), [5102](#), [5108](#),
 5114, [5134](#), [5288](#), [5307](#), [5309](#), [5351](#),
 5428, [5450](#), [5452](#), [5901](#), [6477](#), [6479](#),
 6484, [6493](#), [6513](#), [8466](#), [8785](#), [8875](#),
 9455, [12547](#), [15614](#), [15712](#), [15714](#),
 16002, [16111](#), [16918](#), [17168](#), [17179](#),
 17332, [17635](#), [17637](#), [18376](#), [18952](#),
 19170, [19178](#), [19352](#), [19601](#), [21477](#),
 21664, [21945](#), [21952](#), [21953](#), [21954](#),
 21974, [22058](#), [22457](#), [22459](#), [22462](#),
 22482, [22607](#), [22639](#), [22642](#), [22676](#),
 22679, [22686](#), [22699](#), [22814](#), [23210](#),
 23625, [23650](#), [23764](#), [23789](#), [23831](#)
 \int_compare:nTF 71,
 71, [73](#), [73](#), [73](#), [80](#), [179](#), [513](#),
 [4794](#), [4902](#), [4910](#), [4919](#), [4925](#), [5443](#),
 5445, [5466](#), [8700](#), [8724](#), [8854](#), [15763](#),
 [19787](#), [19984](#), [19985](#), [19990](#), [19991](#)
 \int_compare_p:n 71, [71](#), [4794](#)
 \int_compare_p:nNn
 21, [71](#), [71](#), [4847](#), [18279](#), [18280](#),
 21453, [22664](#), [22759](#), [22760](#), [22761](#),
 22806, [23236](#), [23237](#), [23261](#), [23262](#)
 \int_const:Nn 69, [69](#), [2319](#), [4693](#),
 5317, [5318](#), [5319](#), [5320](#), [5321](#), [5322](#),
 5323, [5324](#), [5325](#), [5326](#), [5327](#), [5328](#),
 5329, [5330](#), [5375](#), [5376](#), [5377](#), [5378](#),
 5379, [5380](#), [5381](#), [5382](#), [5383](#), [5384](#),
 5385, [5386](#), [5387](#), [5388](#), [5389](#), [5390](#),
 5391, [5392](#), [5393](#), [5394](#), [5395](#), [5396](#),
 5397, [5398](#), [5416](#), [6745](#), [6764](#), [8620](#),
 8780, [8781](#), [10648](#), [10649](#), [10650](#),
 10651, [10652](#), [10653](#), [10654](#), [10741](#),
 10742, [10743](#), [10745](#), [10746](#), [10747](#),
 10750, [10751](#), [10752](#), [11129](#), [11387](#),
 11388, [11389](#), [11390](#), [11391](#), [11392](#),
 11393, [11394](#), [11395](#), [11396](#), [11397](#),
 11398, [11399](#), [11400](#), [11401](#), [15887](#),
 15888, [15889](#), [16177](#), [17067](#), [17068](#),
 17069, [17070](#), [17468](#), [17469](#), [17470](#),
 17471, [17472](#), [17473](#), [17477](#), [17478](#),
 17479, [17480](#), [17481](#), [17482](#), [17483](#),
 17484, [17485](#), [17486](#), [17487](#), [17488](#),
 17489, [21419](#), [21421](#), [21423](#), [21424](#),
 21425, [22141](#), [22179](#), [23657](#), [23784](#)
 \int_decr:N 70, [70](#),
 [4759](#), [16290](#), [16291](#), [16292](#), [16335](#),
 16336, [16345](#), [16346](#), [16355](#), [16356](#),
 16598, [19025](#), [19342](#), [19396](#), [19582](#)
 \int_div_round:Nn 68, [68](#), [4653](#)
 \int_div_truncate:Nn
 69, [69](#), [69](#), [4653](#), [5029](#), [5127](#), [5147](#),
 21422, [22836](#), [22842](#), [22844](#), [22854](#)
 \int_do_until:Nn 73, [73](#), [4900](#)
 \int_do_until:nNn 72, [72](#), [4928](#)
 \int_do_while:Nn 73, [73](#), [4900](#)
 \int_do_while:nNn 72, [72](#), [4928](#)
 \int_eval:n 14,
 26, [26](#), [68](#), [68](#), [68](#), [68](#), [69](#), [70](#),
 71, [71](#), [72](#), [79](#), [80](#), [137](#), [293](#), [393](#),
 484, [578](#), [619](#), [641](#), [641](#), [643](#), [830](#),
 912, [1889](#), [1905](#), [3076](#), [3081](#), [3370](#),
 3378, [3386](#), [3733](#), [3780](#), [3817](#), [4438](#),
 4446, [4454](#), [4520](#), [4619](#), [4858](#), [4863](#),
 4868, [4873](#), [4983](#), [5011](#), [5097](#), [5099](#),
 5229, [5239](#), [5274](#), [5285](#), [5291](#), [5302](#),
 5333, [5370](#), [5427](#), [5435](#), [5439](#), [5461](#),
 5463, [6411](#), [6420](#), [6471](#), [6481](#), [6495](#),
 6502, [6517](#), [8686](#), [8840](#), [9082](#), [9462](#),
 10658, [12526](#), [16472](#), [16473](#), [18404](#),
 18406, [18408](#), [18412](#), [18596](#), [18598](#),
 18608, [18622](#), [18623](#), [18625](#), [18626](#),
 18863, [18893](#), [19791](#), [19836](#), [19837](#),
 20004, [20124](#), [20134](#), [21957](#), [21976](#),
 21983, [22022](#), [22061](#), [22255](#), [22821](#),
 22854, [22856](#), [23313](#), [23317](#), [23349](#)
 \int_from_alpha:n 76, [76](#), [5272](#)

```

\int_from_base:nn ..... 77, 77, 5289, 5312, 5314, 5316
\int_from_bin:n .. 76, 76, 5311, 21519
\int_from_binary:n ..... 21519
\int_from_hex:n .. 77, 77, 5311, 21520
\int_from_hexadecimal:n ..... 21520
\int_from_oct:n .. 77, 77, 5311, 21521
\int_from_octal:n ..... 21521
\int_from_roman:n ..... 77, 77, 5331
\int_gadd:Nn ..... 70, 4747
\int_gdecr:N .. 70, 3037, 4497, 4759,
      5009, 6372, 7551, 8766, 12817, 16835
\int_gincr:N ..... 70, 3031,
      4489, 4759, 4988, 4995, 5425, 6365,
      7546, 8757, 12796, 12803, 16820,
      23783, 24443, 24572, 24628, 24673
\int_gset:N ..... 166, 10180
\int_gset:Nn 70, 4698, 4704, 4771, 7745
\int_gset_eq:NN ..... 69, 4737, 24565
\int_gsub:Nn ..... 70, 4747
\int_gzero:N .. 69, 4727, 4734, 24557
\int_gzero_new:N ..... 69, 4731
\int_if_even:nTF ..... 72, 4884
\int_if_even_p:n ..... 72, 4884
\int_if_exist:NTF 70, 70, 4732, 4734,
      4743, 5345, 5349, 18152, 18193, 23773
\int_if_exist_p:N ..... 70, 70, 4743
\int_if_odd:nTF .. 72, 72, 4884, 14028
\int_if_odd_p:n .. 72, 72, 4884, 18517
\int_incr:N ..... 70, 70, 4759,
      9978, 16198, 16300, 16301, 16639,
      16670, 16682, 16694, 16992, 17001,
      17025, 17026, 18059, 18435, 18614,
      18706, 18986, 19022, 19024, 19101,
      19330, 19382, 19524, 19579, 19648
\int_log:N ..... 77, 77, 5371
\int_log:n ..... 77, 77, 5373
\int_max:nn ..... 69, 69, 721, 4621,
      13889, 14977, 18730, 19730, 19731
\int_min:nn ..... 69, 69, 4621
\int_mod:nn 69, 69, 4653, 5019, 5118,
      5138, 8468, 15998, 21420, 21970, 22856
\int_new:N ..... 69, 69, 4685, 4697, 4703, 4732,
      4734, 5408, 5409, 5410, 5411, 5421,
      6004, 8898, 8901, 8903, 8915, 9797,
      16092, 16093, 16094, 16095, 16096,
      16097, 16098, 16099, 16100, 16101,
      16102, 16525, 16526, 16527, 16528,
      16944, 16945, 17051, 17052, 17053,
      17065, 17466, 17467, 17474, 17475,
      17492, 18534, 18536, 18537, 18538,
      18541, 18897, 18898, 18899, 18900,
      18901, 18902, 18903, 18904, 18905,
      18906, 18909, 18910, 18911, 19131,
      19493, 19496, 19497, 19498, 23591,
      23750, 24492, 24575, 24576, 24654
\int_rand:nn ..... 228,
      228, 231, 723, 906, 21665, 21670,
      21935, 21989, 22052, 22111, 23294
\int_set:N ..... 166, 10180
\int_set:Nn 70, 70, 2039, 2711, 2768,
      4771, 8742, 8744, 8881, 8883, 8899,
      8909, 8921, 8956, 8962, 8976, 9982,
      16105, 16107, 16109, 16132, 16133,
      16148, 16156, 16157, 16169, 16170,
      16181, 16182, 16183, 16200, 16203,
      16564, 16593, 16658, 17013, 18220,
      18535, 18591, 18593, 18672, 18726,
      18727, 18738, 18750, 18774, 18792,
      18841, 18926, 18941, 18961, 19050,
      19085, 19529, 19678, 19703, 20133,
      20135, 20143, 20144, 20145, 20146
\int_set_eq:NN ..... 69, 69, 4737,
      16149, 16191, 17123, 17573, 17576,
      17584, 17585, 17621, 17673, 17966,
      18058, 18067, 18154, 18552, 18566,
      18612, 18613, 18663, 18771, 18772,
      18824, 18879, 18928, 18931, 18938,
      18940, 18943, 18956, 18959, 18987,
      18988, 18992, 19125, 19639, 24553
\int_show:N ..... 77, 77, 5367
\int_show:n 77, 77, 395, 484, 5369, 5374
\int_step_... ..... 206
\int_step_function:nnnn .. 74, 74,
      386, 621, 4956, 5008, 6808, 6813,
      6816, 8797, 18993, 19595, 19706, 19714
\int_step_inline:nnnn 74, 74, 622,
      4986, 5430, 8629, 8805, 18935, 20012
\int_step_variable:nnnNn 74, 74, 4986
\int_sub:Nn ... 70, 70, 4747, 9145,
      17127, 17771, 18827, 18835, 18844
\int_to_Alph:n ..... 75, 75, 76, 5032
\int_to_alpha:n 75, 75, 75, 75, 76, 5032
\int_to_arabic:n ..... 74, 74, 5011
\int_to_Base:n ..... 76
\int_to_base:n ..... 76
\int_to_Base:nn ... 76, 77, 5096, 5223
\int_to_base:nn ..... 76, 76, 77, 5096, 5219, 5221, 5225
\int_to_bin:n 75, 75, 76, 76, 5218, 21522
\int_to_binary:n ..... 21522
\int_to_Hex:n .. 76, 76, 77, 5218, 19767
\int_to_hex:n 76, 76, 76, 77, 5218, 21523
\int_to_hexadecimal:n ..... 21523
\int_to_oct:n .. 76, 76, 77, 5218, 21524
\int_to_octal:n ..... 21524
\int_to_Roman:n .... 76, 76, 77, 5226

```

```

\int_to_roman:n .. 76, 76, 76, 77, 5226
\int_to_symbols:nnn ..... 75, 75, 75, 5012, 5034, 5066
\int_until_do:nn ..... 73, 73, 4900
\int_until_do:nNnn ..... 73, 73, 4928
\int_use:N ..... 68, 70, 70, 70,
    572, 576, 3033, 3035, 4490, 4496,
    4776, 4991, 4998, 6367, 6369, 7545,
    7553, 7661, 8153, 8744, 8759, 8877,
    9983, 12799, 12806, 16822, 16952,
    16963, 17601, 17667, 17731, 17742,
    17752, 17756, 17767, 17768, 17774,
    17775, 17781, 17782, 17949, 18517,
    18584, 18586, 18605, 18606, 18607,
    18704, 18717, 18718, 19040, 19086,
    19116, 19208, 19220, 19300, 19529,
    19769, 23626, 23651, 23667, 23765,
    23776, 23788, 23790, 24445, 24475,
    24631, 24637, 24644, 24676, 24684
\int_while_do:nn ..... 73, 73, 4900
\int_while_do:nNnn ..... 73, 73, 4928
\int_zero:N ... 69, 69, 4727, 4732,
    9000, 9975, 16594, 16595, 16596,
    16683, 16978, 16979, 17583, 17769,
    18177, 18466, 18551, 18925, 18939,
    19191, 19523, 23611, 23740, 23823
int_zero:N ..... 69
\int_zero_new:N ..... 69, 69, 4731
\c_max_int ... 78, 768, 818, 5397,
    17098, 17112, 18988, 20121, 20127
\c_nine ..... 78, 5375
\c_one ... 78, 397, 4760, 4762, 5375
\c_seven ..... 78, 5375
\c_six ..... 78, 5375
\c_sixteen ..... 78, 5375
\c_ten ..... 78, 5375
\c_thirteen ..... 78, 5375
\c_three ..... 78, 5375
\g_tmpa_int ..... 78, 5408
\l_tmpa_int ..... 2, 78, 199, 5408
\g_tmpb_int ..... 78, 5408
\l_tmpb_int ..... 2, 78, 5408
\c_twelve ..... 78, 5375
\c_two ..... 78, 5375
\c_zero ..... 78,
    276, 284, 284, 284, 912, 1346,
    1600, 1602, 4695, 4727, 4728, 4781,
    4789, 4965, 4968, 5375, 6000, 6002
int internal commands:
\__int_abs:N ..... 4621
\__int_case:nnTF ..... 4855
\__int_case:nw ..... 4855
\__int_case_end:nw ..... 4855
\__int_compare:nnN ..... 382, 4794
\__int_compare>NNw ..... 381, 382, 4794
\__int_compare:Nw .. 381, 381, 382, 4794
\__int_compare:w ..... 381, 4794
\__int_compare_!=NNw ..... 4794
\__int_compare_<NNw ..... 4794
\__int_compare_<=NNw ..... 4794
\__int_compare_=NNw ..... 4794
\__int_compare_==NNw ..... 4794
\__int_compare_>NNw ..... 4794
\__int_compare_>=NNw ..... 4794
\__int_compare_end_=NNw .. 382, 4794
\__int_constdef:Nw ..... 4693
\__int_div_truncate:NwNw ..... 4653
\__int_eval:w ..... 80,
    80, 376, 376, 381, 570, 572, 572,
    572, 585, 600, 629, 637, 638, 638,
    641, 645, 683, 1863, 3656, 3669,
    3687, 3692, 3715, 3716, 3728, 3764,
    4614, 4620, 4624, 4632, 4633, 4640,
    4641, 4655, 4657, 4658, 4675, 4678,
    4679, 4680, 4709, 4748, 4750, 4772,
    4797, 4831, 4849, 4886, 4894, 4959,
    4960, 4961, 5155, 5182, 5188, 5215,
    5488, 5520, 5948, 6556, 6557, 6560,
    6629, 6630, 6633, 6638, 6639, 6642,
    6647, 6648, 6651, 6656, 6657, 6660,
    6692, 6693, 6699, 9034, 9043, 9067,
    9079, 10675, 10784, 10787, 10893,
    11203, 11207, 11219, 11220, 11248,
    11331, 11335, 11374, 11563, 11568,
    11609, 11698, 11709, 11758, 11789,
    11795, 11796, 11842, 11852, 11854,
    11870, 11872, 11895, 11897, 12035,
    12195, 12443, 12626, 13018, 13026,
    13047, 13049, 13070, 13072, 13081,
    13083, 13112, 13118, 13128, 13130,
    13204, 13206, 13222, 13224, 13228,
    13244, 13284, 13292, 13294, 13296,
    13298, 13300, 13302, 13304, 13323,
    13325, 13335, 13337, 13363, 13366,
    13374, 13376, 13396, 13399, 13402,
    13405, 13413, 13416, 13419, 13422,
    13429, 13431, 13437, 13445, 13447,
    13449, 13455, 13475, 13477, 13486,
    13488, 13509, 13530, 13534, 13546,
    13549, 13552, 13555, 13558, 13561,
    13564, 13567, 13571, 13583, 13587,
    13591, 13594, 13615, 13617, 13619,
    13629, 13668, 13670, 13679, 13713,
    13718, 13720, 13727, 13730, 13733,
    13736, 13739, 13742, 13751, 13763,
    13771, 13773, 13783, 13785, 13792,
    13801, 13803, 13806, 13809, 13812,
    13815, 13828, 13830, 13838, 13840,
```

13848, 13850, 13859, 13862, 13865,
 13872, 13887, 13905, 13908, 13964,
 13978, 13980, 13986, 13999, 14001,
 14003, 14027, 14043, 14050, 14051,
 14074, 14090, 14094, 14139, 14141,
 14183, 14194, 14213, 14215, 14217,
 14230, 14243, 14248, 14250, 14256,
 14273, 14274, 14275, 14276, 14277,
 14278, 14283, 14285, 14287, 14289,
 14291, 14295, 14297, 14299, 14301,
 14303, 14305, 14327, 14335, 14419,
 14468, 14681, 14702, 14704, 14707,
 14710, 14713, 14716, 14732, 14758,
 14768, 14784, 14811, 14812, 14813,
 14951, 14983, 14992, 15174, 15175,
 15198, 15208, 15217, 15235, 15244,
 15251, 15262, 15272, 15305, 15315,
 15340, 15349, 15366, 15403, 15420,
 15422, 15434, 15435, 15475, 15486,
 15497, 15555, 15673, 15774, 15809,
 15906, 15912, 15928, 15949, 16265,
 16539, 16704, 16762, 16772, 16776,
 16780, 16803, 16963, 17150, 17409,
 18212, 18773, 19094, 19147, 19148,
 19159, 19169, 19272, 19657, 21940,
 21941, 22140, 23326, 23334, 23343
 $\backslash\text{--int_eval_end}$:
 80, 80, 80, 1863, 4614, 4620,
 4624, 4659, 4675, 4681, 4709, 4748,
 4750, 4772, 4849, 4886, 4894, 5155,
 5182, 5188, 5215, 5948, 6556, 6557,
 6560, 6629, 6630, 6633, 6638, 6639,
 6642, 6647, 6648, 6651, 6656, 6657,
 6660, 10675, 10775, 10896, 11248,
 11345, 11349, 12443, 12626, 13228,
 13263, 13451, 13773, 13908, 14732,
 14784, 14984, 14993, 15262, 15272,
 15315, 15340, 15366, 15435, 15907,
 15913, 15929, 15949, 16776, 17150,
 17411, 18212, 19185, 19272, 22140
 $\backslash\text{--int_from_alph:N}$ 392, 5272
 $\backslash\text{--int_from_alph:nN}$ 392, 5272
 $\backslash\text{--int_from_base:N}$ 393, 5289
 $\backslash\text{--int_from_base:nnN}$ 393, 5289
 $\backslash\text{--int_from_roman:NN}$ 5331
 $\backslash\text{c_int_from_roman_C_int}$ 5317
 $\backslash\text{c_int_from_roman_c_int}$ 5317
 $\backslash\text{c_int_from_roman_D_int}$ 5317
 $\backslash\text{c_int_from_roman_d_int}$ 5317
 $\backslash\text{--int_from_roman_error:w}$ 5331
 $\backslash\text{c_int_from_roman_I_int}$ 5317
 $\backslash\text{c_int_from_roman_i_int}$ 5317
 $\backslash\text{c_int_from_roman_L_int}$ 5317
 $\backslash\text{c_int_from_roman_l_int}$ 5317
 $\backslash\text{c_int_from_roman_M_int}$ 5317
 $\backslash\text{c_int_from_roman_m_int}$ 5317
 $\backslash\text{c_int_from_roman_V_int}$ 5317
 $\backslash\text{c_int_from_roman_v_int}$ 5317
 $\backslash\text{c_int_from_roman_X_int}$ 5317
 $\backslash\text{c_int_from_roman_x_int}$ 5317
 $\backslash\text{--int_maxmin:wwN}$ 4621
 $\backslash\text{--int_mod:ww}$ 4653
 $\backslash\text{--int_pass_signs:wn}$
 392, 5262, 5276, 5293
 $\backslash\text{--int_pass_signs_end:wn}$ 5262
 $\backslash\text{--int_rand:ww}$ 21935
 $\backslash\text{--int_rand_narrow:n}$ 21935
 $\backslash\text{--int_rand_narrow:nn}$
 903, 21956, 21962, 21975
 $\backslash\text{--int_rand_narrow:nnn}$ 21935
 $\backslash\text{--int_rand_narrow:nnnn}$ 903, 21935
 $\backslash\text{--int_show:nn}$ 5367
 $\backslash\text{--int_step:NnnnN}$ 4956
 $\backslash\text{--int_step:NNnnnN}$ 4986
 $\backslash\text{--int_step:wwwN}$ 4956
 $\backslash\text{--int_to_Base:nn}$ 5096
 $\backslash\text{--int_to_base:nn}$ 5096
 $\backslash\text{--int_to_Base:nnN}$ 5096
 $\backslash\text{--int_to_base:nnN}$ 5096
 $\backslash\text{--int_to_Base:nnnnN}$ 5096
 $\backslash\text{--int_to_base:nnnnN}$ 5096
 $\backslash\text{--int_to_Letter:n}$ 5096
 $\backslash\text{--int_to_letter:n}$ 5096
 $\backslash\text{--int_to_roman:N}$ 5226
 $\backslash\text{--int_to_roman:w}$
 79, 79, 381, 391, 1327,
 4614, 4807, 5229, 5239, 6803, 6826,
 10787, 11770, 11802, 14210, 14477
 $\backslash\text{--int_to_Roman_aux:N}$ 5238, 5241, 5244
 $\backslash\text{--int_to_Roman_c:w}$ 5226
 $\backslash\text{--int_to_roman_c:w}$ 5226
 $\backslash\text{--int_to_Roman_d:w}$ 5226
 $\backslash\text{--int_to_roman_d:w}$ 5226
 $\backslash\text{--int_to_Roman_i:w}$ 5226
 $\backslash\text{--int_to_roman_i:w}$ 5226
 $\backslash\text{--int_to_Roman_l:w}$ 5226
 $\backslash\text{--int_to_roman_l:w}$ 5226
 $\backslash\text{--int_to_Roman_m:w}$ 5226
 $\backslash\text{--int_to_roman_m:w}$ 5226
 $\backslash\text{--int_to_Roman_Q:w}$ 5226
 $\backslash\text{--int_to_roman_Q:w}$ 5226
 $\backslash\text{--int_to_Roman_v:w}$ 5226
 $\backslash\text{--int_to_roman_v:w}$ 5226
 $\backslash\text{--int_to_Roman_x:w}$ 5226
 $\backslash\text{--int_to_roman_x:w}$ 5226
 $\backslash\text{--int_to_symbols:nnnn}$ 5012
 $\backslash\text{--int_value:w}$ 80, 80, 80, 381,
 411, 513, 553, 559, 572, 578, 578,

578, 578, 578, 578, 585, 588, 593,
 593, 600, 619, 627, 635, 643, 705,
 717, 746, 746, 1602, 3656, 3657,
 3669, 3687, 3692, 3714, 3715, 3716,
 3728, 3764, 4614, 4620, 4623, 4624,
 4631, 4632, 4633, 4639, 4640, 4641,
 4655, 4657, 4658, 4675, 4678, 4679,
 4680, 4797, 4801, 4831, 4959, 4960,
 4961, 5182, 5215, 5450, 5461, 5488,
 5520, 5530, 5828, 5831, 5948, 6692,
 6693, 6699, 9034, 9043, 9312, 10725,
 10726, 10727, 10728, 10729, 10801,
 10862, 10880, 10893, 11203, 11317,
 11331, 11333, 11335, 11338, 11374,
 11509, 11540, 11541, 11554, 11563,
 11693, 11698, 11700, 11709, 11713,
 11750, 11758, 11761, 11767, 11778,
 11789, 11795, 11796, 11799, 11842,
 11852, 11854, 11870, 11872, 11895,
 11909, 11987, 11989, 12035, 12117,
 12537, 12641, 12968, 12969, 12970,
 12972, 13018, 13021, 13024, 13047,
 13049, 13070, 13072, 13081, 13083,
 13087, 13105, 13112, 13118, 13128,
 13130, 13144, 13152, 13160, 13204,
 13206, 13222, 13224, 13227, 13230,
 13284, 13292, 13294, 13296, 13298,
 13300, 13302, 13304, 13323, 13325,
 13329, 13335, 13337, 13341, 13363,
 13366, 13374, 13376, 13379, 13380,
 13381, 13382, 13396, 13399, 13402,
 13405, 13413, 13416, 13419, 13422,
 13429, 13431, 13437, 13445, 13447,
 13449, 13475, 13477, 13486, 13488,
 13492, 13509, 13530, 13534, 13546,
 13549, 13552, 13555, 13558, 13561,
 13564, 13567, 13571, 13583, 13587,
 13591, 13594, 13615, 13617, 13619,
 13629, 13653, 13656, 13668, 13670,
 13676, 13679, 13700, 13713, 13718,
 13720, 13727, 13730, 13733, 13736,
 13739, 13742, 13751, 13763, 13771,
 13773, 13783, 13785, 13792, 13801,
 13803, 13806, 13809, 13812, 13815,
 13828, 13830, 13838, 13840, 13848,
 13850, 13859, 13862, 13865, 13872,
 13887, 13905, 13908, 13964, 13978,
 13980, 13986, 13999, 14001, 14003,
 14027, 14043, 14050, 14051, 14094,
 14096, 14097, 14098, 14139, 14141,
 14176, 14183, 14190, 14211, 14213,
 14215, 14217, 14230, 14234, 14235,
 14236, 14237, 14238, 14243, 14248,
 14250, 14256, 14273, 14274, 14275,
 14276, 14277, 14278, 14283, 14285,
 14287, 14289, 14291, 14295, 14297,
 14299, 14301, 14303, 14305, 14327,
 14335, 14351, 14356, 14360, 14419,
 14468, 14479, 14666, 14702, 14704,
 14707, 14710, 14713, 14716, 14723,
 14726, 14728, 14732, 14754, 14756,
 14784, 14811, 14812, 14813, 14951,
 14983, 14992, 15174, 15175, 15176,
 15198, 15208, 15217, 15235, 15244,
 15251, 15261, 15305, 15314, 15349,
 15403, 15420, 15475, 15486, 15497,
 15673, 15736, 15774, 15801, 15809,
 15811, 15813, 15900, 15906, 15912,
 15922, 15928, 15989, 15998, 16005,
 16023, 16265, 16539, 16551, 16722,
 16760, 16762, 16772, 16780, 16799,
 16801, 16809, 17328, 17800, 17806,
 17838, 17840, 17849, 17850, 17973,
 18353, 18367, 19094, 19147, 19148,
 19159, 19621, 19653, 19655, 20605,
 20637, 20638, 20639, 20641, 20765,
 20774, 20776, 20781, 20782, 20783,
 20784, 20788, 20789, 20790, 20791,
 20816, 20822, 20824, 20826, 20828,
 20833, 20838, 20843, 20850, 20857,
 20988, 21018, 21019, 21058, 21077,
 21083, 21253, 21362, 21686, 21688,
 21711, 21713, 21738, 21778, 21805,
 21813, 21839, 21841, 21845, 21847,
 21876, 21890, 21897, 21940, 21941,
 22052, 22061, 23326, 23334, 23343
 intarray internal commands:
 __intarray_count:N 81,
 81, 81, 5433, 5443, 5457, 5466, 5470
 \g__intarray_font_int 5421, 5425, 5426
 __intarray_gset:Nnn 81, 81, 81, 5434
 __intarray_gset_aux:Nnn 5434
 __intarray_gset_fast:Nnn 81, 81,
 5434, 18937, 19015, 19017, 19019,
 19042, 19045, 19099, 19555, 19557,
 19563, 19571, 19573, 19576, 19642,
 19644, 19646, 19659, 19664, 19666
 __intarray_item:Nn . 81, 81, 81, 5460
 __intarray_item_aux:Nn 5460
 __intarray_item_fast:Nn . 81, 81,
 5460, 19010, 19031, 19034, 19052,
 19079, 19140, 19141, 19164, 19165,
 19171, 19174, 19175, 19179, 19182,
 19183, 19233, 19234, 19560, 19688
 __intarray_new:Nn
 81, 81, 5422, 17062, 17063, 17064,
 18912, 18913, 19499, 19500, 19501
 \interactionmode 640

\interlinepenalties 641
 \interlinepenalty 421
 ior commands:
 \ior_close:N
 . 139, 139, 140, 140, 8505, 8677, 8698
 \ior_get>NN
 . 140, 140, 141, 141, 145, 8736, 8752
 \ior_get_str>NN 9214, 21551
 \ior_if_eof:N 496
 \ior_if_eof:NTF
 142, 142, 8502, 8523, 8720, 8764, 8771
 \ior_if_eof_p:N 142, 142, 8720
 \ior_list_streams:
 . 140, 140, 903, 8710, 21931
 \ior_log_streams: 228, 228, 21930
 \ior_map_break: . 141, 141, 8747, 8765
 \ior_map_break:n 142, 142, 8747
 \ior_map_inline:Nn ... 141, 141, 8751
 \ior_new:N 139, 139, 8644, 8778
 \ior_open:Nn 139, 139, 8646
 \ior_open:NnTF
 . 139, 139, 8656, 8658, 8659
 \ior_str_get>NN ... 141, 141, 141,
 8738, 8754, 9219, 9220, 9221, 21551
 \ior_str_map_inline:Nn 141, 141, 8751
 \c_term_iow
 . 145, 8620, 8644, 8700, 8706, 8724
 ior internal commands:
 \l_ior_internal_tl 8751
 __ior_list_streams:Nn ... 8710, 8866
 __ior_map_inline>NNn 8751
 __ior_map_inline>NNNn 8751
 __ior_map_inline_loop>NNN ... 8751
 __ior_new:N 493, 8671, 8685
 __ior_open:Nn
 145, 145, 8501, 8522, 8654, 8667, 8675
 __ior_open_aux:Nn 8646
 __ior_open_aux:NnTF 8656
 __ior_open_stream:Nn 8675
 \l_ior_stream_tl
 . 8626, 8678, 8686, 8694
 \g_ior_streams_prop
 . 8627, 8695, 8703, 8711
 \g_ior_streams_seq
 . 8621, 8678, 8704, 8705
 iow commands:
 \iow_char:N 143, 143,
 8161, 8163, 8164, 8460, 8896, 14585,
 16460, 16463, 16488, 16489, 16496,
 16497, 17314, 17316, 17318, 17320,
 17322, 17324, 17906, 17907, 18398,
 18507, 18508, 18509, 18530, 19751,
 19754, 19757, 19760, 19764, 19767,
 19802, 19811, 19815, 19820, 19840,
 19842, 19843, 19844, 19847, 19850,
 19851, 19854, 19856, 19860, 19862,
 19868, 19870, 19874, 19876, 19880,
 19885, 19886, 19928, 19930, 19935,
 19937, 19943, 19948, 19953, 19957,
 19964, 19965, 19969, 19977, 20018
 \iow_close:N 139, 140, 140, 8831, 8852
 \iow_indent:n
 . 144, 144, 144, 502, 503, 8102,
 8935, 8965, 8973, 10563, 11060, 11072
 \l_iow_line_count_int
 . 144, 144, 144, 503,
 759, 8898, 8977, 9002, 16920, 16924
 \iow_list_streams:
 . 140, 140, 903, 8864, 21933
 \iow_log:n .. 22, 142, 142, 288, 485,
 1695, 1706, 7734, 7735, 7736, 7880,
 8358, 8393, 8608, 8609, 8610, 8891
 \iow_log_streams: ... 228, 228, 21932
 \iow_new:N 139, 139, 8820
 \iow_newline:
 143, 143, 143, 143, 143, 146, 468,
 485, 500, 7690, 7706, 7708, 8895, 8974
 \iow_now:Nn
 142, 142, 142, 142, 142, 143, 143,
 8885, 8891, 8892, 8893, 8894, 22195
 \iow_open:Nn 139, 139, 8826
 \iow_shipout:Nn 143,
 143, 143, 143, 500, 8870, 22214
 \iow_shipout_x:Nn
 . 143, 143, 143, 143, 143, 500, 8867
 \iow_term:n 142, 142, 1695,
 7704, 7740, 7741, 7742, 8358, 8891
 \iow_wrap:nnnn 56,
 133, 134, 134, 143, 143, 144, 144,
 144, 144, 144, 144, 297, 483, 484,
 485, 503, 7682, 7683, 7735, 7741,
 7878, 8341, 8385, 8938, 8950, 8953
 \c_log_iow
 . 145, 497, 8780, 8854, 8891, 8892
 \c_term_iow ... 145, 497, 497, 8780,
 8797, 8820, 8854, 8860, 8893, 8894
 iow internal commands:
 __iow_indent:n 502, 8935, 8965
 __iow_indent_error:n 502, 8935, 8973
 \l_iow_indent_int
 . 8914, 9000, 9018, 9129, 9137, 9145
 \l_iow_indent_tl 8914,
 9001, 9017, 9128, 9138, 9146, 9147
 \l_iow_line_break_bool
 . 8918, 8996, 9123,
 9136, 9144, 9152, 9154, 9159, 9161
 \l_iow_line_part_tl ... 505, 505,
 506, 506, 8916, 8998, 9010, 9031,

- __iow_wrap_newline: 9150
 __iow_wrap_newline:n 9150
 \c__iow_wrap_newline_marker_tl ..
 8920, 8963
 __iow_wrap_next:nw . 9004, 9082, 9124
 __iow_wrap_next_line:w .. 9076, 9117
 __iow_wrap_set:Nn 8953
 __iow_wrap_start:w 8985
 __iow_wrap_store_do:n ..
 9075, 9153, 9160, 9163
 \l__iow_wrap_tl ... 503, 503, 504,
 508, 508, 8919, 8968, 8971, 8980,
 8987, 8989, 8992, 8999, 9167, 9169
 __iow_wrap_trim:N ..
 508, 9105, 9153, 9160, 9175
 __iow_wrap_trim:w 9175
 __iow_wrap_unindent: 9133
 __iow_wrap_unindent:n 9141
 \c__iow_wrap_unindent_marker_tl ..
 8920, 8945
- J**
- \J 197
 \j 23105
 \jcharwidowpenalty 1136
 \jfam 1137
 \jfont 1138
 \jis 1139
 job commands:
 \c_job_name_tl 21516
 \jobname 422
- K**
- \k 23109
 \kanjiskip 1140
 \kansuji 1141
 \kansujichar 1142
 \kcatcode 1143
 \kchar 1161
 \kchardef 1162
 \kern 423
 kernel internal commands:
 \l_kernel_expl_bool
 7, 235, 238, 253, 267
 __kernel_primitive>NN ..
 . 253, 275, 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, 603,
604, 605, 606, 607, 608, 609, 610,
611, 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,
695, 696, 697, 698, 699, 700, 701,
702, 703, 704, 705, 706, 707, 708,
709, 710, 711, 712, 713, 714, 715,
716, 717, 718, 719, 720, 721, 722,
723, 724, 725, 726, 727, 728, 729,
730, 731, 732, 733, 734, 735, 736,
737, 738, 739, 740, 741, 742, 743,
744, 745, 746, 747, 748, 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, 788, 793, 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, 901, 902, 903, 904, 905,
906, 907, 908, 909, 910, 911, 912,
913, 914, 915, 916, 917, 918, 919,
920, 921, 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, 966, 967, 968,
969, 970, 971, 972, 973, 974, 975,
976, 977, 978, 979, 980, 981, 982,
983, 984, 985, 986, 987, 988, 989,
990, 991, 992, 993, 994, 995, 996,
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, 1035, 1036, 1037, 1038,
1039, 1040, 1041, 1042, 1043, 1044,
1045, 1046, 1047, 1048, 1049, 1050,
1051, 1052, 1053, 1054, 1055, 1056,
1057, 1058, 1059, 1060, 1061, 1062,
1063, 1064, 1065, 1066, 1067, 1068,
1069, 1070, 1071, 1072, 1073, 1074,
1075, 1076, 1077, 1078, 1079, 1080,
1081, 1082, 1083, 1084, 1085, 1086,
1087, 1088, 1089, 1090, 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, 1151, 1152, 1153, 1154, 1155, 1156, 1157, 1158, 1159, 1160, 1161, 1162, 1163, 1164
 _kernel_register_log:N 23, 23, 2032, 5371, 9475, 9476, 9572, 9573, 9640, 9641
 _kernel_register_show:N 23, 23, 23, 297, 341, 2023, 2033, 5367, 9471, 9568, 9636
 _kernel_register_show_aux:n . 2023
 _kernel_register_show_aux:nN .. 2024, 2025

keys commands:

- \l_keys_choice_int 164, 166, 168, 168, 168, 169, 9797, 9975, 9978, 9982, 9983
- \l_keys_choice_tl 164, 166, 168, 168, 169, 9797, 9981
- \keys_define:nn 163, 163, 163, 9812, 10561
- \keys_if_choice_exist:nnnTF 172, 172, 10512
- \keys_if_choice_exist_p:nnn 172, 172, 10512
- \keys_if_exist:nnTF 172, 172, 547, 10505, 10522
- \keys_if_exist_p:nn . 172, 172, 10505
- \l_keys_key_tl ... 170, 170, 9800, 9917, 9933, 10354, 10440, 10442, 10475
- \keys_log:nn 172, 172, 10546
- \l_keys_path_tl 170, 170, 9804, 9840, 9859, 9868, 9877, 9881, 9895, 9910, 9912, 9914, 9926, 9928, 9930, 9945, 9948, 9952, 9960, 9961, 9962, 9965, 9979, 10000, 10005, 10014, 10018, 10025, 10029, 10033, 10040, 10047, 10058, 10064, 10068, 10083, 10092, 10100, 10135, 10336, 10343, 10366, 10369, 10408, 10412, 10420, 10422, 10423, 10434, 10437, 10457, 10482, 10483
- \keys_set:nn 162, 166, 170, 170, 170, 170, 171, 10035, 10040, 10232, 10256, 10264, 10312, 10321
- \keys_set_filter:nnn 172, 172, 10267

\keys_set_filter:nnnN 172, 172, 172, 10267
 \keys_set_groups:nnn 172, 172, 10267
 \keys_set_known:nn .. 171, 171, 10242
 \keys_set_known:nnN 171, 171, 171, 171, 542, 10242
 \keys_show:nn 172, 172, 172, 548, 10520, 10547
 \l_keys_value_tl 170, 170, 9810, 10083, 10411, 10414, 10416, 10424, 10444, 10453, 10467, 10477

keys internal commands:

- __keys_bool_set:Nn 9906, 10109, 10111, 10113, 10115
- __keys_bool_set_inverse:Nn 9922, 10117, 10119, 10121, 10123
- __keys_check_groups: . 10370, 10378
- __keys_choice_find:n .. 9939, 10480
- __keys_choice_make: 9909, 9925, 9938, 9969, 10125
- __keys_choice_make:N 9938
- __keys_choice_make_aux:N 9938
- __keys_choices_make:nn 9968, 10127, 10129, 10131, 10133
- __keys_choices_make:Nnn 9968
- __keys_cmd_set:nn .. 9910, 9912, 9914, 9926, 9928, 9930, 9961, 9962, 9979, 9988, 10033, 10040, 10100, 10135
- \c__keys_code_root_tl 9790, 9990, 9992, 10029, 10420, 10423, 10440, 10442, 10450, 10452, 10464, 10466, 10508, 10516, 10527
- \c__keys_default_root_tl 9790, 10000, 10005, 10408, 10412
- __keys_default_set:n 9919, 9935, 9995, 10145, 10147, 10149, 10151
- __keys_define:n 9817, 9821
- __keys_define:nn 9817, 9821
- __keys_define:nnn 9812
- __keys_define_aux:nn 9821
- __keys_define_code:n 9835, 9885
- __keys_define_code:w 9885
- __keys_execute: 10347, 10374, 10396, 10400, 10418
- __keys_execute:nn 10418, 10482, 10483
- __keys_execute_unknown: 10418
- \l__keys_filtered_bool 9806, 10280, 10288, 10290, 10294, 10302, 10304, 10373, 10394, 10399
- __keys_find_key_module:w 10324
- \l__keys_groups_clist 9799, 10011, 10012, 10019, 10368, 10383
- \c__keys_groups_root_tl 9790, 10014, 10018, 10366, 10369

```

\__keys_groups_set:n . . . 10009, 10169
\__keys_inherit:n . . . . 10022, 10171
\c__keys_inherit_root_tl . . . .
. . . . . 9790, 10025, 10434, 10437
\__keys_initialise:n . . .
. . . . . 10027, 10173, 10175, 10177, 10179
\__keys_keys_set_known:nn . . . 10242
\__keys_meta_make:n . . . 10031, 10189
\__keys_meta_make:nn . . . 10031, 10191
\l__keys_module_tl . . .
. . . . . 9801, 9813, 9816, 9818, 9861,
9862, 9868, 10036, 10233, 10236,
10238, 10327, 10332, 10342, 10348,
10356, 10358, 10450, 10452, 10457
\__keys_multichoice_find:n . .
. . . . . 9941, 10480
\__keys_multichoice_make: . .
. . . . . 9938, 9971, 10193
\__keys_multichoice_make:nn . .
. . . . . 9968, 10195, 10197, 10199, 10201
\l__keys_no_value_bool . . .
. . . . . 9802, 9823, 9828, 9887, 10080,
10089, 10326, 10331, 10406, 10476
\l__keys_only_known_bool . . .
. . . . . 9803, 10255, 10263, 10265, 10430
\__keys_parent:n . . .
. . . . . 9945, 9948, 9952, 10434, 10437, 10487
\__keys_parent:w . . .
. . . . . 10487
\__keys_property_find:n . . 9833, 9844
\__keys_property_find:w . . .
. . . . . 9844
\__keys_property_search:w . .
. . . . . 9869, 9873, 9882
\l__keys_property_tl . . .
. . . . . 9805,
9834, 9837, 9840, 9846, 9847, 9853,
9865, 9878, 9890, 9891, 9894, 9898
\c__keys_props_root_tl . . . 9796,
9834, 9891, 9898, 10108, 10110,
10112, 10114, 10116, 10118, 10120,
10122, 10124, 10126, 10128, 10130,
10132, 10134, 10136, 10138, 10140,
10142, 10144, 10146, 10148, 10150,
10152, 10154, 10156, 10158, 10160,
10162, 10164, 10166, 10168, 10170,
10172, 10174, 10176, 10178, 10180,
10182, 10184, 10186, 10188, 10190,
10192, 10194, 10196, 10198, 10200,
10202, 10204, 10206, 10208, 10210,
10212, 10214, 10216, 10218, 10220,
10222, 10224, 10226, 10228, 10230
\__keys_remove_spaces:n 9816, 9846,
9979, 10236, 10340, 10482, 10498,
10508, 10516, 10525, 10527, 10531
\__keys_remove_spaces:w . . .
. . . . . 10498
\l__keys_selective_bool . . .
. . . . . 9806, 10311, 10320, 10322, 10345
\l__keys_selective_seq . . .
. . . . . 9808, 10307, 10310, 10315, 10381
\__keys_set:n . . .
. . . . . 10237, 10324
\__keys_set:nn . . .
. . . . . 10237, 10324
\__keys_set:nnn . . .
. . . . . 10232
\__keys_set_aux: . . .
. . . . . 10324
\__keys_set_aux:nnn . . .
. . . . . 10324
\__keys_set_filter:nnn . . .
. . . . . 10267
\__keys_set_filter:nnnnN . . .
. . . . . 10267
\__keys_set_groups:nnn . . .
. . . . . 10267
\__keys_set_known:nn . . .
. . . . . 10257, 10261
\__keys_set_known:nnnN . . .
. . . . . 10242
\__keys_set_selective: . . .
. . . . . 10324
\__keys_set_selective:nn . . .
. . . . . 10267
\__keys_set_selective:nnn . . .
. . . . . 10267
\__keys_set_selective:nnnn . . .
. . . . . 10267
\__keys_show:N . . .
. . . . . 10520
\__keys_store_unused: . . .
. . . . . 10375, 10395, 10401, 10418
\l__keys_tmp_bool . . .
. . . . . 9811, 10380, 10387, 10392
\c__keys_type_root_tl . . .
. . . . . 9790, 9945, 9948, 9960
\__keys_undefine: . . 10024, 10041, 10227
\l__keys_unused_clist . . .
. . . . . 541,
9809, 10243, 10247, 10249, 10250,
10268, 10272, 10274, 10275, 10473
\__keys_validate_cleanup:w . . 10051
\__keys_validate_forbidden: . . 10051
\__keys_validate_required: . . 10051
\c__keys_validate_root_tl . . .
. . . . . 9790, 10058, 10064, 10068, 10422
\__keys_value_or_default:n . .
. . . . . 10344, 10404
\__keys_value_requirement:nn . .
. . . . . 10051, 10229, 10231
\__keys_variable_set:NnnN . . .
. . . . . 10097, 10137, 10139, 10141,
10143, 10153, 10155, 10157, 10159,
10161, 10163, 10165, 10167, 10181,
10183, 10185, 10187, 10203, 10205,
10207, 10209, 10211, 10213, 10215,
10217, 10219, 10221, 10223, 10225
keyval commands:
\keyval_parse>NNn . . .
. . . . . 174, 174, 174, 9656, 9817, 10237
keyval internal commands:
\__keyval_action: . . .
. . . . . 9735
\__keyval_def:Nn . . .
. . . . . 9737, 9753, 9778
\__keyval_def_aux:n . . .
. . . . . 9778
\__keyval_def_aux:w . . .
. . . . . 9778
\__keyval_empty_key: . . .
. . . . . 9772, 9776

```

\l_keyval_key_tl	
.....	9653, 9737, 9738, 9749, 9757
_keyval_loop>NNw	9659, 9665, 9725
_keyval_sanitise_aux:w	9669
_keyval_sanitise_comma:	9664, 9669
_keyval_sanitise_comma_auxi:w	9669
_keyval_sanitise_comma_auxii:w	9669
_keyval_sanitise_equals:	9663, 9669
_keyval_sanitise_equals_auxi:w	9669
_keyval_sanitise_equals_-auxii:w	9669
\l_keyval_sanitise_t1	9655, 9662,
9666, 9675, 9677, 9681, 9688, 9690,	9699, 9701, 9705, 9712, 9714, 9723
_keyval_split>NNw	9730, 9735
_keyval_split_tidy:w	9735
_keyval_split_value>NNw	9735
\l_keyval_value_t1	9653, 9753, 9758
\kuten	1144, 1163
L	
\L	23096
\l	23096
13kernel	23444
\13kernel.charcat	239
13kernel.charcat	239, 23459
\13kernelstrcmp	239
13kernelstrcmp	239, 23449
\label	23280
\language	424
\lastallocatedread	3954, 3955
\lastallocatedtoks	16163
\lastbox	425
\lastkern	426
\lastlinefit	642
\lastnamedcs	889
\lastnodedtype	643
\lastpenalty	427
\lastsavedboxresourceindex	958
\lastsavedimageresourceindex	959
\lastsavedimageresourcepages	960
\lastskip	428
\lastxpos	961
\lastypos	962
\latelua	890
LaTeX3 error commands:	
\LaTeX3_error:	482, 482
\lccode	166,
181, 194, 196, 198, 200, 202, 205, 429	
\leaders	430
\left	431
left commands:	
\c_left_brace_str	55, 3927, 17359,
17718, 17722, 17742, 17756, 17780,	18224, 18301, 19278, 19313, 19334
\leftghost	939
\lefthyphenmin	432
\leftmarginkern	781
\leftskip	433
\leqno	434
\let	1, 40, 272, 273, 435
\letcharcode	891
\letterspacefont	782
\limits	436
\LineBreak	80, 81, 82, 83, 84,
85, 86, 87, 105, 112, 113, 114, 122, 124	
\linedir	940
\linepenalty	437
\lineskip	438
\lineskiplimit	439
\linewidth	20669, 20715
\ln	14691, 14694
\ln	187
\localbrokenpenalty	941
\localinterlinepenalty	942
\localleftbox	943
\localrightbox	944
\loccount	8637, 8813
\loctoks	16135, 16136, 16162
\long	275, 440, 7018, 7022
\LongText	76, 110, 134
\looseness	441
\lower	442
\lowercase	443
\lpcode	783
lua commands:	
\lua_escape:n	239, 239, 23408
\lua_escape_x:n	239, 239, 239, 23408
\lua_now:n	238, 238, 238, 23408
\lua_now_x:n	238, 238, 238, 23408
\lua_shipout:n	238, 238, 238, 23408
\lua_shipout_x:n	238, 238, 23408
\luaescapestring	892
\luafunction	893
luatex commands:	
\luatex_alignmark:D	859, 1188
\luatex_aligntab:D	860, 1189
\luatex_attribute:D	861, 1190
\luatex_attributedef:D	862, 1191
\luatex_automatichyphenpenalty:D	863
\luatex_begincsname:D	864
\luatex_bodydir:D	937, 1223, 1286
\luatex_boxdir:D	938, 1224
\luatex_catcodetable:D	865, 1192
\luatex_clearmarks:D	866, 1193

\luatex_crampeddisplaystyle:D
 867, 1194
 \luatex_crampedscriptscriptstyle:D
 868, 1195
 \luatex_crampedscriptstyle:D
 869, 1196
 \luatex_crampedtextstyle:D 870, 1197
 \luatex_directlua:D
 ... 871, 1181, 1182, 3481, 6743,
 6746, 6752, 8783, 22145, 22184, 23408
 \luatex_dviextension:D
 872
 \luatex_dvifeedback:D
 873
 \luatex_dvivariable:D
 874
 \luatex_etoksapp:D
 875
 \luatex_etokspre:D
 876
 \luatex_expanded:D
 878, 1279, 3494
 \luatex_explithyphenpenalty:D 877
 \luatex_firstvalidlanguage:D
 879
 \luatex_fontid:D
 880, 1198
 \luatex_formatname:D
 881, 1199
 \luatex_gleaders:D
 887, 1200
 \luatex_hjcode:D
 882
 \luatex_hpack:D
 883
 \luatex_hyphenationbounds:D
 884
 \luatex_hyphenationmin:D
 885
 \luatex_hyphenpenaltymode:D
 886
 \luatex_if_engine:TF
 ... 21526, 21527, 21528
 \luatex_if_engine_p:
 21525
 \luatex_initcatcodetable:D 888, 1201
 \luatex_lastnamedcs:D
 889
 \luatex_latelua:D
 ... 890, 1202, 22203, 23410
 \luatex_leftghost:D
 939, 1225
 \luatex_letcharcode:D
 891
 \luatex_linedir:D
 940
 \luatex_localbrokenpenalty:D
 ... 941, 1226
 \luatex_localinterlinepenalty:D
 ... 942, 1227
 \luatex_localleftbox:D
 943, 1228
 \luatex_localrightbox:D
 944, 1229
 \luatex_luaescapestring:D
 ... 344,
 892, 1203, 3492, 22187, 22206, 23413
 \luatex_luafunction:D
 893, 1204
 \luatex_luatexbanner:D
 894
 \luatex_luatexdatestamp:D
 895
 \luatex_luatexrevision:D
 896
 \luatex_luatexversion:D
 ... 897, 1242, 1261,
 1347, 3477, 4715, 5401, 8785, 21433
 \luatex_mathdir:D
 945, 1230
 \luatex_mathdisplayskipmode:D
 898
 \luatex_matheqnogapstep:D
 899
 \luatex_mathnolimitsmode:D
 900
 \luatex_mathoption:D
 901
 \luatex_mathrulesfam:D
 902
 \luatex_mathscriptsmode:D
 903
 \luatex_mathstyle:D
 ... 904, 1205
 \luatex_mathsurroundmode:D
 905
 \luatex_mathsurroundskip:D
 906
 \luatex_nohrule:D
 907
 \luatex_nokerns:D
 908, 1206
 \luatex_noligs:D
 909, 1207
 \luatex_nospaces:D
 910
 \luatex_no rule:D
 911
 \luatex_outputbox:D
 912, 1208
 \luatex_pagebottomoffset:D 913, 1231
 \luatex_pagedir:D
 ... 946, 1232, 1287
 \luatex_pageleftoffset:D
 ... 914, 1209
 \luatex_pagerightoffset:D
 ... 915, 1234
 \luatex_pagetopoffset:D
 ... 916, 1210
 \luatex_pardir:D
 ... 947, 1236
 \luatex_pdfextension:D
 ... 917, 23504, 23505,
 23511, 23512, 23517, 23518, 23523,
 23524, 23594, 23595, 23603, 23604
 \luatex_pdffeedback:D
 918
 \luatex_pdfvariable:D
 919
 \luatex_postexhyphenchar:D 920, 1211
 \luatex_posthyphenchar:D
 ... 921, 1212
 \luatex_predisplaygapfactor:D
 ... 922
 \luatex_preexhyphenchar:D
 ... 923, 1213
 \luatex_prehyphenchar:D
 ... 924, 1214
 \luatex_rightghost:D
 948, 1237
 \luatex_savecatcodetable:D 925, 1215
 \luatex_scantexttokens:D
 926, 1216
 \luatex_setfontid:D
 927
 \luatex_shapemode:D
 928
 \luatex_suppressifcsnameerror:D
 ... 929, 1217
 \luatex_suppresslongerror:D 930, 1218
 \luatex_suppressmathparerror:D
 ... 931, 1219
 \luatex_suppressoutererror:D
 ... 932, 1220
 \luatex_textdir:D
 949, 1238
 \luatex_toksapp:D
 933
 \luatex_tokspre:D
 934
 \luatex_tpack:D
 935
 \luatex_vpack:D
 936
 \luatexalignmark
 1188
 \luatexaligntab
 1189
 \luatexattribute
 1190
 \luatexattributedef
 1191
 \luatexbanner
 894
 \luatexbodydir
 1223
 \luatexboxdir
 1224

\luatexcatcodetable	1192	\marks	644
\luatexclearmarks	1193	math commands:	
\luatexcrampeddisplaystyle	1194	\c_math_subscript_token	
\luatexcrampedscriptscriptstyle ..	1195 111, 443, 6833, 6891, 16755	
\luatexcrampedscriptstyle	1196	\c_math_superscript_token	
\luatexcrampedtextstyle	1197 111, 443, 6833, 6886, 16754	
\luatexdatestamp	895	\c_math_toggle_token	
\luatexfontid	1198 111, 442, 6833, 6867, 16752	
\luatexformatname	1199	\mathaccent	446
\luatexleaders	1200	\mathbin	447
\luatexinitcatcodetable	1201	\mathchar	448, 7017
\luatexlatelua	1202	\mathchardef	449
\luatexleftghost	1225	\mathchoice	450
\luatexlocalbrokenpenalty	1226	\mathclose	451
\luatexlocalinterlinepenalty	1227	\mathcode	452
\luatexlocalleftbox	1228	\mathdir	945
\luatexlocalrightbox	1229	\mathdisplayskipmode	898
\luatexluaescapestring	1203	\matheqnogapstep	899
\luatexluafunction	1204	\mathinner	453
\luatexmathdir	1230	\mathnolimitsmode	900
\luatexmathstyle	1205	\mathop	454
\luatexnokerns	1206	\mathopen	455
\luatexnoligs	1207	\mathoption	901
\luatexoutputbox	1208	\mathord	456
\luatexpagebottomoffset	1231	\mathpunct	457
\luatexpagedir	1232	\mathrel	458
\luatexpageheight	1233	\mathrulesfam	902
\luatexpageleftoffset	1209	\mathscriptsmode	903
\luatexpagerightoffset	1234	\mathstyle	904
\luatexpagetopoffset	1210	\mathsurround	459
\luatexpagewidth	1235	\mathsurroundmode	905
\luatexpardir	1236	\mathsurroundskip	906
\luatexpostexhyphenchar	1211	max	187
\luatexposthyphenchar	1212	max commands:	
\luatexpreehyphenchar	1213	\c_max_char_int 78, 5398, 17332, 19769	
\luatexprehyphenchar	1214	\c_max_register_int	
\luatexrevision	896 78, 206, 731, 731, 731,	
\luatexrightghost	1237	731, 1347, 4614, 8153, 16107, 16133,	
\luatexsavecatcodetable	1215	16170, 16178, 16182, 16993, 17002	
\luatexscantextokens	1216	max internal commands:	
\luatexsuppressfontnotfounderror	1187, 1222	\c__max_constdef_int	4693
\luatexsuppressifcsnameerror	1217	\maxdeadcycles	460
\luatexsuppresslongerror	1218	\maxdepth	461
\luatexsuppressmathparerror	1219	\meaning	462
\luatexsuppressoutererror	1220	\medmuskip	463
\luatextextdir	1238	\message	464
\luatextracingfonts	1183	\MessageBreak	122
\luatexUchar	1221	meta commands:	
\luatexversion	45, 100, 897	.meta:n	166, 10188
		.meta:nn	166, 10190
M		\middle	645
\mag	444	min	187
\mark	445		

minus commands:

- \c_minus_inf_fp 182,
190, [10643](#), [13273](#), [13355](#), [14166](#), [14943](#)
- \c_minus_zero_fp 181, [10643](#), [13269](#), [15592](#)

\mkern 465

mm 191

mode commands:

- \mode_if_horizontal:TF 96, 96, [5993](#)
- \mode_if_horizontal_p: 96, 96, [5993](#)
- \mode_if_inner:TF 96, 96, [5995](#)
- \mode_if_inner_p: 96, 96, [5995](#)
- \mode_if_math:TF 96, 96, [5997](#)
- \mode_if_math_p: 96, [5997](#)
- \mode_if_vertical:TF 96, 96, [5991](#)
- \mode_if_vertical_p: 96, 96, [5991](#)

\month 466

\moveleft 467

\moveright 468

msg commands:

- \msg_critical:nn 130, [7828](#)
- \msg_critical:nnn 130, [7828](#)
- \msg_critical:nnnn 130, [7828](#)
- \msg_critical:nnnnn 130, [7828](#)
- \msg_critical:nnnnnn 130, [7828](#)
- \msg_critical_text:n 129, 129, [7747](#), [7831](#)
- \msg_error:nn 131, [7839](#)
- \msg_error:nnn 131, [7839](#)
- \msg_error:nnnn 131, [7839](#)
- \msg_error:nnnnn 131, [7839](#)
- \msg_error:nnnnnn 131, 131, [228](#), [7839](#)
- \msg_error_text:n 129, 129, [7747](#), [7846](#)
- \msg_expandable_error:nn 229, [21992](#)
- \msg_expandable_error:nnn 229, [21992](#)
- \msg_expandable_error:nnnn 229, [21992](#)
- \msg_expandable_error:nnnnn 229, [21992](#)
- \msg_expandable_error:nnnnnn 229, 229, [21992](#)
- \msg_fatal:nn 130, [7817](#)
- \msg_fatal:nnn 130, [7817](#)
- \msg_fatal:nnnn 130, [7817](#)
- \msg_fatal:nnnnn 130, [7817](#)
- \msg_fatal:nnnnnn 130, 130, [7817](#)
- \msg_fatal_text:n 129, 129, [7747](#), [7820](#)
- \msg_gset:nnn 128, [7604](#)
- \msg_gset:nnnn 128, [7604](#)
- \msg_if_exist:nnTF 129, 129, [7578](#), [7585](#), [7595](#), [7896](#)
- \msg_if_exist_p:nn 129, 129, [7578](#)
- \msg_info:nn 131, [7868](#)
- \msg_info:nnn 131, [7868](#)
- \msg_info:nnnn 131, [7868](#)

\msg_info:nnnnn 131, [7868](#)

\msg_info:nnnnnn 131, [7868](#)

\msg_info:nnnnnnn 131, [7868](#)

\msg_line_context: 129, 129, 466,
[1742](#), [1761](#), [2478](#), [7600](#), [7661](#), [7750](#),
[7755](#), [7760](#), [7765](#), [7770](#), [8205](#), [9991](#)

\msg_line_number: 129, 129, [7661](#), [9784](#)

\msg_log:n 134, 134, [7732](#), [7870](#)

\msg_log:nn 131, [7876](#)

\msg_log:nnn 131, [7876](#)

\msg_log:nnnn 131, [7876](#)

\msg_log:nnnnn 131, [7876](#)

\msg_log:nnnnnn 131, 131, [7876](#)

\msg_new:nnn 128, [7604](#), [8019](#)

\msg_new:nnnn 128, 128, 465, 465, [7604](#), [8017](#)

\msg_none:nn 132, [7882](#)

\msg_none:nnn 132, [7882](#)

\msg_none:nnnn 132, [7882](#)

\msg_none:nnnnn 132, [7882](#)

\msg_none:nnnnnn 132, 132, [7882](#)

\msg_redirect_class:nn 132, 132, [7968](#)

\msg_redirect_module:nnn 133, 133, [7968](#)

\msg_redirect_name:nnn 133, 133, [7959](#)

\msg_see_documentation_text:n 130, 130, [7772](#), [7823](#), [7834](#), [7849](#)

\msg_set:nnn 128, [7604](#), [8023](#)

\msg_set:nnnn 128, 128, [7604](#), [8021](#)

\msg_term:n 134, 134, [7732](#), [7862](#)

\msg_warning:nn 131, [7860](#)

\msg_warning:nnn 131, [7860](#)

\msg_warning:nnnn 131, [7860](#)

\msg_warning:nnnnn 131, [7860](#)

\msg_warning:nnnnnn 131, 131, [7860](#), [8067](#)

\msg_warning_text:n 129, 129, [7747](#), [7864](#)

msg internal commands:

- __msg_class_chk_exist:nTF 7884, [7898](#), [7964](#), [7974](#), [7979](#)
- __msg_class_loop_seq 475, [7893](#),
[7983](#), [7991](#), [8001](#), [8002](#), [8005](#), [8007](#)
- __msg_class_new:nn 472, 476, [7778](#), [7817](#),
[7828](#), [7839](#), [7860](#), [7868](#), [7876](#), [7882](#)
- __msg_class_t1 473,
475, [7889](#), [7905](#), [7917](#), [7938](#), [7942](#),
[7945](#), [7953](#), [7992](#), [7994](#), [7996](#), [8010](#)
- \c__msg_coding_error_text_t1 137, [7629](#), [8073](#), [8081](#), [8107](#),

8125, 8134, 8146, 8160, 8169, 8190,
 8197, 8219, 8226, 8233, 8241, 10560,
 10581, 10587, 10594, 21374, 21385
`\c__msg_continue_text_tl` . 7629, 7673
`\c__msg_critical_text_tl` . 7629, 7836
`\l__msg_current_class_tl`
 475, 7889, 7900,
 7937, 7942, 7945, 7953, 7982, 7996
`__msg_error:Nnnnnn` 7839
`__msg_error_code:nnnnnn` 8066
`__msg_expandable_error:n`
 136, 136, 482, 8290, 8309
`__msg_expandable_error:w` . 482, 8290
`__msg_expandable_error_module:nn`
 21992
`__msg_fatal_code:nnnnnn` 8062
`\c__msg_fatal_text_tl` 7629, 7825
`\c__msg_help_text_tl` 7629, 7677
`\l__msg_hierarchy_seq`
 473, 474, 7892, 7920, 7930, 7935
`\l__msg_internal_tl`
 485, 7574, 8389, 8390, 8393, 8402
`__msg_interrupt_more_text:n`
 467, 7680
`__msg_interrupt_text:n` 7683, 7694
`__msg_interrupt_wrap:nn`
 7672, 7676, 7680
`__msg_kernel_class_new:nN`
 477, 8024, 8062, 8066, 8067, 8068
`__msg_kernel_class_new_aux:nN` . 8024
`__msg_kernel_error:nn`
 135, 1725, 8062, 9762, 9777,
 16331, 17309, 17565, 17594, 17636,
 17639, 18307, 19350, 19419, 20875
`__msg_kernel_error:nnn` . 135, 1461,
 1503, 1544, 1549, 1725, 1773, 1783,
 1934, 1941, 2367, 2811, 5662, 7887,
 8062, 8371, 8485, 8555, 8653, 9854,
 9916, 9932, 10074, 10091, 10633,
 10940, 16454, 17336, 17400, 17600,
 17789, 18165, 18205, 18325, 19207,
 19214, 19433, 20151, 20620, 23431
`__msg_kernel_error:nnnn`
 135, 1452, 1481, 1563,
 1725, 1748, 1758, 1904, 2428, 7587,
 7597, 7912, 8062, 9839, 9893, 9951,
 9964, 10082, 10456, 10936, 16324,
 17766, 17831, 18047, 19354, 20767
`__msg_kernel_error:nnnnn`
 135, 5456,
 8062, 8937, 16470, 19605, 19728, 21511
`__msg_kernel_error:nnnnnn`
 135, 135, 2443, 5448, 8062
`__msg_kernel_expandable_-`
`error:nn` 136, 4084,
 5979, 6705, 6709, 6711, 6719, 6725,
 7301, 8307, 11472, 12272, 15878, 21982
`__msg_kernel_expandable_-`
`error:nnn` . 136, 2115, 3069, 4546,
 4790, 4970, 5543, 6453, 8307, 11477,
 11491, 11493, 11594, 11633, 11639,
 11976, 11981, 11992, 11999, 12062,
 12072, 12214, 12236, 12768, 23422
`__msg_kernel_expandable_-`
`error:nnnn` 136,
 8307, 12405, 12426, 12900, 21947
`__msg_kernel_expandable_-`
`error:nnnnn`
 136, 5469, 8307, 8949, 11051,
 12550, 15371, 15939, 15953, 21508
`__msg_kernel_expandable_-`
`error:nnnnnn` 136, 136, 8307
`__msg_kernel_fatal:nn`
 135, 8062, 8681, 8835
`__msg_kernel_fatal:nnn` 135, 8062
`__msg_kernel_fatal:nnnn` 135, 8062
`__msg_kernel_fatal:nnnnn` 135, 8062
`__msg_kernel_fatal:nnnnnn`
 135, 135, 8062
`__msg_kernel_info:nn` 135, 8067
`__msg_kernel_info:nnn` 135, 8067
`__msg_kernel_info:nnnn` 135, 8067
`__msg_kernel_info:nnnnn` 135, 8067
`__msg_kernel_info:nnnnnn`
 135, 135, 8067
`__msg_kernel_new:nnn`
 134, 8016, 8112, 8114, 8116,
 8118, 8120, 8138, 8246, 8248, 8250,
 8252, 8254, 8256, 8258, 8265, 8272,
 8279, 10610, 11079, 11081, 11083,
 11085, 11087, 11089, 12572, 12574,
 12576, 12578, 12580, 12582, 12584,
 12586, 12588, 12590, 12595, 12819,
 12821, 15874, 16934, 21389, 21988
`__msg_kernel_new:nnnn` . . 134, 134,
 8016, 8070, 8078, 8086, 8093, 8104,
 8122, 8131, 8143, 8150, 8157, 8166,
 8175, 8181, 8187, 8194, 8204, 8216,
 8223, 8230, 8238, 9182, 9188, 9195,
 9202, 9207, 9783, 10548, 10551,
 10557, 10566, 10572, 10578, 10584,
 10591, 10598, 10604, 10913, 11053,
 11068, 16459, 16476, 16483, 16492,
 19750, 19756, 19763, 19771, 19778,
 19784, 19794, 19800, 19824, 19831,
 19839, 19846, 19853, 19859, 19867,
 19873, 19879, 19888, 19895, 19904,

19907, 19915, 19921, 19927, 19934,
 19941, 19951, 19962, 19972, 19981,
 19987, 21371, 21379, 21382, 23436
`_msg_kernel_set:nnn` 134, 8016
`_msg_kernel_set:nnnn` 134, 134, 8016
`_msg_kernel_warning:nn`
 135, 8067, 18071
`_msg_kernel_warning:nnn`
 135, 8067, 17997,
 17998, 18037, 18087, 18123, 18139
`_msg_kernel_warning:nnnn`
 135, 8067, 17699, 17845
`_msg_kernel_warning:nnnnn` 135,
 3409, 3418, 8067, 9216, 16502, 16511
`_msg_kernel_warning:nnnnnn`
 135, 135, 7999, 8067
`\l_msg_line_context_bool`
 7575, 7750, 7755, 7760, 7765, 7770
`_msg_log_next:`
 136, 136, 136, 136, 298,
 2033, 2046, 2047, 3399, 3402, 4605,
 5374, 5502, 5773, 5775, 6543, 6545,
 7569, 8336, 9478, 9575, 9643, 10547,
 16077, 16079, 21369, 21931, 21933
`\g_msg_log_next_bool`
 484, 485, 8336, 8358, 8373, 8391, 8394
`_msg_log_wrap:n` 484
`\c_msg_more_text_prefix_tl`
 7576, 7615, 7624, 7842
`\c_no_info_text_tl` .. 7629, 7672
`_msg_no_more_text:nnn` 7839
`\c_on_line_text_tl` .. 7629, 7664
`_msg_redirect:nnn` 7968
`_msg_redirect_loop_chk:nnn`
 7968, 8010
`_msg_redirect_loop_list:n` .. 7968
`\l_msg_redirect_prop`
 7891, 7917, 7962, 7965
`\c_return_text_tl`
 7629, 8076, 8084, 8091
`_msg_show_item:n` 137, 137,
 137, 484, 485, 4601, 6532, 6539, 8408
`_msg_show_item:nn`
 137, 137, 137, 463, 7565, 8408
`_msg_show_item_unbraced:nn`
 137, 137, 494, 8408, 8717, 10541, 21363
`_msg_show_pre:nnnnnn`
 136, 136, 6536, 8339, 8365, 8714,
 10524, 10530, 16842, 16851, 21354
`_msg_show_pre_aux:n` 483, 8339
`_msg_show_variable:NNNnn` . 136,
 137, 137, 137, 297, 374, 433, 463,
 484, 728, 2027, 3392, 4599, 5498,
 5764, 6530, 7563, 8359, 16070, 18430
`_msg_show_wrap:n`
 136, 137, 137, 137, 341,
 484, 484, 484, 484, 485, 485, 485,
 485, 547, 2042, 3397, 6538, 6562,
 6635, 6644, 6653, 6662, 8368, 8378,
 8384, 8716, 10532, 10539, 16860, 21359
`_msg_show_wrap:Nn`
 136, 137, 137, 5370,
 5768, 8376, 9474, 9571, 9639, 16074
`_msg_show_wrap_aux:n` 8384
`_msg_show_wrap_aux:w` 8384
`\c_text_prefix_tl` 7576,
 7580, 7613, 7622, 7822, 7833, 7848,
 7865, 7873, 7879, 8312, 8343, 21997
`_msg_tmp:w` 8291, 8304
`\c_trouble_text_tl` 7629
`_msg_use:nnnnnn` 7788, 7894
`_msg_use_code:` 473, 473, 7894
`_msg_use_hierarchy:nwwN` 7894
`_msg_use_redirect_module:n`
 474, 7894
`_msg_use_redirect_name:n` 7894
`\mskip` 469
`\muexpr` 646
multichoice commands:
`.multichoice:` 166, 10192
multicoices commands:
`.multicoices:nn` 166, 10192
`\multiply` 470
`\muskip` 471, 7025
muskip commands:
`\c_max_muskip` 160, 9644
`\muskip_add:Nn` 158, 158, 9622
`\muskip_const:Nn`
 158, 158, 9590, 9644, 9645
`\muskip_eval:n` 159, 159, 159, 9632, 9639
`\muskip_gadd:Nn` 158, 9622
`\muskip_gset:Nn` 159, 9593, 9611
`\muskip_gset_eq:NN` 159, 9616
`\muskip_gsub:Nn` 159, 9622
`\muskip_gzero:N` 158, 9596, 9604
`\muskip_gzero_new:N` 158, 9601
`\muskip_if_exist:NTF`
 158, 158, 9602, 9604, 9607
`\muskip_if_exist_p:N` 158, 158, 9607
`\muskip_log:N` 160, 160, 9640
`\muskip_log:n` 160, 160, 9640
`\muskip_new:N`
 158, 158, 158, 9582, 9592,
 9602, 9604, 9646, 9647, 9648, 9649
`\muskip_set:Nn` 159, 159, 9611
`\muskip_set_eq:NN` 159, 159, 9616
`\muskip_show:N` 159, 159, 9636
`\muskip_show:n` 159, 159, 524, 9638, 9643

\muskip_sub:Nn	159, 159, 9622	\normalshowtokens	1285
\muskip_use:N	159, 159, 159, 159, 9633, 9634	\normalunexpanded	1278
\muskip_zero:N	158, 9596, 9602	\normalvcenter	1277
muskip_zero:N	158	\normalvoffset	1284
\muskip_zero_new:N	158, 158, 9601	\nospaces	910
\g_tmpa_muskip	160, 9646	notexpanded commands:	
\l_tmpa_muskip	160, 9646	\notexpanded: <token>	119
\g_tmpb_muskip	160, 9646	\novrule	911
\l_tmpb_muskip	160, 9646	\nulldelimiterspace	481
\c_zero_muskip	160, 9597, 9644	\nullfont	482
\muskipdef	472	\number	55, 483
\mutoglu	647	\numexpr	167, 181, 648
O			
N			
nan	190	\o	23098
nc	191	\oe	23098
nd	191	\OE	23099
\newbox	377	\oe	23099
\newcatcodetable	52	\omit	484
\newcount	377	one commands:	
\newdimen	377	\c_minus_one	5412, 21554
\newlinechar	103, 473	\c_one_degree_fp	182, 190, 12086, 16083
\next	74, 106, 118, 118, 119, 131, 140, 144, 147, 155	\openin	485
\NG	23097	\openout	486
\ng	23097	\or	487
\noalign	474	or commands:	
\noautospacing	1145	\or: 79, 79, 79, 349, 349, 349, 349, 351, 351, 683, 1297, 1865, 1866, 1867, 1868, 1869, 1870, 1871, 1872, 1873, 3696, 3771, 4614, 5157, 5158, 5159, 5160, 5161, 5162, 5163, 5164, 5165, 5166, 5167, 5168, 5169, 5170, 5171, 5172, 5173, 5174, 5175, 5176, 5177, 5178, 5179, 5180, 5181, 5190, 5191, 5192, 5193, 5194, 5195, 5196, 5197, 5198, 5199, 5200, 5201, 5202, 5203, 5204, 5205, 5206, 5207, 5208, 5209, 5210, 5211, 5212, 5213, 5214, 6765, 6769, 6772, 6776, 6780, 6782, 6784, 6786, 6787, 6789, 6791, 6793, 6795, 9054, 9055, 9056, 9057, 9058, 9059, 9060, 10682, 10683, 10684, 10843, 10858, 10859, 11250, 11251, 11276, 12445, 12446, 12447, 12483, 12969, 12970, 12971, 13094, 13179, 13265, 13266, 13267, 13268, 13269, 13270, 13271, 13272, 13273, 13350, 13353, 13686, 13687, 13701, 13947, 14167, 14192, 14198, 14199, 14200, 14201, 14202, 14350, 14385, 14387, 14395, 14482, 14486, 14487, 14488, 14489, 14490, 14491, 14492, 14493, 14494, 14495, 14502, 14503, 14504, 14505, 14506, 14507, 14508, 14509, 14510,	

14511, 14518, 14519, 14520, 14521,	\parfillskip	504
14522, 14523, 14524, 14525, 14526,	\parindent	505
14527, 14534, 14535, 14536, 14537,	\parshape	506
14538, 14539, 14540, 14541, 14542,	\parshapedimen	650
14543, 14550, 14551, 14552, 14553,	\parshapeindent	651
14554, 14555, 14556, 14557, 14558,	\parshapelength	652
14559, 14566, 14567, 14568, 14569,	\parskip	507
14570, 14571, 14572, 14573, 14574,	\patterns	508
14575, 14593, 14644, 14647, 14656,	\pausing	509
14767, 14790, 14791, 14852, 14857,	pc	191
14867, 14872, 14882, 14887, 14897,	\pdfadjustspacing	745
14902, 14912, 14917, 14927, 14932,	\pdfannot	675
15369, 15390, 15391, 15435, 15520,	\pdfcatalog	676
15523, 15535, 15541, 15588, 15590,	\pdfcolorstack	678
15591, 15601, 15607, 15650, 15651,	\pdfcolorstackinit	679
15658, 15691, 15692, 15699, 15756,	\pdfcompresslevel	677
15757, 15951, 15980, 16621, 16622,	\pdfcopyfont	746
16794, 16795, 17142, 17143, 17144,	\pdfcreationdate	680
17145, 17413, 17414, 17415, 17416,	\pdfdecimaldigits	681
17417, 18641, 18696, 19024, 19025	\pdfdest	682
\outer	\pdfdestmargin	683
\output	\pdfdraftmode	747
\outputbox	\pdfeachlinedepth	748
\outputmode	\pdfeachlineheight	749
\outputpenalty	\pdfendlink	684
\over	\pdfendthread	685
\overfullrule	\pdfextension	917
\overline	\pdffeedback	918
\overwithdelims	\pdffirstlineheight	750
	\pdffontattr	686
	\pdffontexpand	751
	\pdffontname	687
\PackageError	\pdffontobjnum	688
\pagebottomoffset	\pdffontsize	752
\pagedepth	\pdfgamma	689
\pagedir	\pdffgentounicode	692
\pagediscards	\pdffglyptounicode	693
\pagefillstretch	\pdfhorigin	694
\pagefillstretch	\pdfignoreddimen	753
\pagefilstretch	\pdfimageapplygamma	690
\pagegoal	\pdfimagegamma	691
\pageheight	\pdfimagehicolor	695
\pageleftoffset	\pdfimageresolution	696
\pagerightoffset	\pdfinincludechars	697
\pageshrink	\pdfinclusioncopyfonts	698
\pagestretch	\pdfinclusionerrorlevel	699
\pagetopoffset	\pdfinfo	700
\pagetotal	\pdfinsertht	754
\pagewidth	\pdflastannot	701
\par	\pdflastlinedepth	755
10, 11, 11, 11, 12, 12, 12, 13, 13, 13,	\pdflastlink	702
14, 14, 14, 140, 140, 292, 292, 503,	\pdflastobj	703
856, 20204, 20206, 20210, 20215,	\pdflastxform	704
20220, 20228, 20237, 20255, 21402	\pdflastximage	705
\pardir		

\pdflastximagecolordepth	706	\pdftex_if_engine:TF	
\pdflastximagepages	707	21530, 21531, 21532	
\pdflastxpos	756	\pdftex_if_engine_p:	21529
\pdflastypos	757	\pdftex_ifabsdim:D	742, 954
\pdflinkmargin	708	\pdftex_ifabsnum:D	743, 955
\pdfliteral	709	\pdftex_ifincname:D	780
\pdfmapfile	758	\pdftex_ifprimitive:D	744, 856
\pdfmapline	759	\pdftex_ignoredimen:D	753
\pdfminorversion	710	\pdftex_ignoreligaturesinfont:D	956
\pdfnames	711	\pdftex_insertht:D	754, 957
\pdfnoligatures	760	\pdftex_lastlinedepth:D	755
\pdfnormaldeviate	761	\pdftex_lastxpos:D	756, 961
\pdfobj	712	\pdftex_lastypos:D	757, 962
\pdfobjcompresslevel	713	\pdftex_leftmarginkern:D	781
\pdfoutline	714	\pdftex_letterspacefont:D	782
\pdfoutput	715	\pdftex_lpcode:D	783
\pdfpageattr	716	\pdftex_mapfile:D	758, 1244
\pdfpagebox	717	\pdftex_mapline:D	759, 1245
\pdfpageheight	762	\pdftex_noligatures:D	760
\pdfpageref	718	\pdftex_normaldeviate:D	761, 963
\pdfpageresources	719	\pdftex_pageheight:D	762, 965, 1233
\pdfpagesattr	720	\pdftex_pagewidth:D	763, 1235
\pdfpagewidth	763	\pdftex_pagewith:D	966
\pdfpkmode	764	\pdftex_pdfannot:D	675
\pdfpkresolution	765	\pdftex_pdfcatalog:D	676
\pdfprimitive	766	\pdftex_pdfcolorstack:D	
\pdfprotrudechars	767	678, 23596, 23605	
\pdfpxdimen	768	\pdftex_pdfcolorstackinit:D	679
\pdfrandomseed	769	\pdftex_pdfcompresslevel:D	677
\pdfrefobj	721	\pdftex_pdfcreationdate:D	680
\pdfrefxform	722	\pdftex_pdfdecimaldigits:D	681
\pdfrefximage	723	\pdftex_pdfdest:D	682
\pdfrestore	724	\pdftex_pdfdestmargin:D	683
\pdfretval	725	\pdftex_pdfendlink:D	684
\pdfsave	726	\pdftex_pdfendthread:D	685
\pdfsavepos	770	\pdftex_pdffontattr:D	686
\pdfsetmatrix	727	\pdftex_pdffontname:D	687
\pdfsetrandomseed	772	\pdftex_pdffontobjnum:D	688
\pdfshellescape	773	\pdftex_pdfgamma:D	689
\pdfstartlink	728	\pdftex_pdgentounicode:D	692
\pdfstartthread	729	\pdftex_pdglyphounicode:D	693
\pdfstrcmp	40, 771	\pdftex_pdforigin:D	694
\pdfstrcmp	344	\pdftex_pdfimageapplygamma:D	690
\pdfsuppressptexinfo	730	\pdftex_pdfimagegamma:D	691
pdftex commands:		\pdftex_pdfimagehicolor:D	695
\pdftex_adjustspacing:D	745, 950	\pdftex_pdfimageresolution:D	696
\pdftex_copyfont:D	746, 951	\pdftex_pdfincludechars:D	697
\pdftex_draftmode:D	747, 952	\pdftex_pdfinclusioncopyfonts:D	698
\pdftex_eachlinedepth:D	748	\pdftex_pdfinclusionerrorlevel:D	699
\pdftex_eachlineheight:D	749	\pdftex_pdfinfo:D	700
\pdftex_efcode:D	779	\pdftex_pdflastannot:D	701
\pdftex_firstlineheight:D	750	\pdftex_pdflastlink:D	702
\pdftex_fontexpand:D	751, 953	\pdftex_pdflastobj:D	703
\pdftex_fontsize:D	752	\pdftex_pdflastxform:D	704, 958

```

\pdftex_pdflastximage:D ..... 705, 959, 23654, 23658
\pdftex_pdflastximagecolordepth:D ..... 706
\pdftex_pdflastximagepages:D 707, 960
\pdftex_pdfslinkmargin:D ..... 708
\pdftex_pdfliteral:D .... 709, 23506
\pdftex_pdfminorversion:D ..... 710
\pdftex_pdfnames:D ..... 711
\pdftex_pdfobj:D ..... 712
\pdftex_pdfobjcompresslevel:D .. 713
\pdftex_pdfoutline:D ..... 714
\pdftex_pdfoutput:D .. 715, 964, 21478
\pdftex_pdfpageattr:D ..... 716
\pdftex_pdfpagebox:D ..... 717
\pdftex_pdfpageref:D ..... 718
\pdftex_pdfpageresources:D .... 719
\pdftex_pdfpagesattr:D ..... 720
\pdftex_pdfrefobj:D ..... 721
\pdftex_pdfrefxform:D .... 722, 970
\pdftex_pdfrefximage:D ..... 723, 971, 23654, 23666
\pdftex_pdfrestore:D .... 724, 23519
\pdftex_pdfretval:D ..... 725
\pdftex_pdfsave:D ..... 726, 23513
\pdftex_pdfsetmatrix:D ... 727, 23525
\pdftex_pdfstartlink:D ..... 728
\pdftex_pdfstartthread:D ..... 729
\pdftex_pdfsuppressptexinfo:D .. 730
\pdftex_pdftexbanner:D .... 776, 1262
\pdftex_pdftexrevision:D .. 777, 1263
\pdftex_pdftexversion:D ..... 273, 778, 1241, 1264, 21441
\pdftex_pdfthread:D ..... 731
\pdftex_pdfthreadmargin:D ..... 732
\pdftex_pdftrailer:D ..... 733
\pdftex_pdfuniqueresname:D .... 734
\pdftex_pdfvorigin:D ..... 735
\pdftex_pdfxform:D .... 736, 973
\pdftex_pdfxformattr:D ..... 737
\pdftex_pdfxformname:D ..... 738
\pdftex_pdfxformresources:D ... 739
\pdftex_pdfximage:D .. 740, 974, 23647
\pdftex_pdfximagebbox:D .. 741, 23643
\pdftex_pkmode:D ..... 764
\pdftex_pkresolution:D ..... 765
\pdftex_primitive:D ..... 766, 857
\pdftex_protrudechars:D .... 767, 967
\pdftex_pxdimen:D ..... 768, 968
\pdftex_quitvmode:D ..... 784
\pdftex_randomseed:D .. 769, 969, 22138
\pdftex_rightmarginkern:D ..... 785
\pdftex_rpcode:D ..... 786
\pdftex_savepos:D ..... 770, 972
\pdftex_setrandomseed:D ..... 772, 975, 22140
\pdftex_shellescape:D 773, 858, 22151
\pdftex_strcmp:D ..... 771, 3476
\pdftex_synctex:D ..... 787
\pdftex_tagcode:D ..... 788
\pdftex_tracingfonts:D ..... 774, 976, 1177, 1179, 1183
\pdftex_uniformdeviate:D ... 723, 723, 775, 977, 15872, 15883, 15886, 21935, 21965, 21986, 22129
\pdftexbanner ..... 776
\pdftexrevision ..... 777
\pdftexversion ..... 95, 778
\pdfthread ..... 731
\pdfthreadmargin ..... 732
\pdftracingfonts ..... 774, 1178, 1179
\pdftrailer ..... 733
\pdfuniformdeviate ..... 775
\pdfuniqueresname ..... 734
\pdfvariable ..... 919
\pdfvorigin ..... 735
\pdfxform ..... 736
\pdfxformattr ..... 737
\pdfxformname ..... 738
\pdfxformresources ..... 739
\pdfximage ..... 740
\pdfximagebbox ..... 741
peek commands:
\peek_after:Nw ..... 97, 115, 115, 115, 116, 7099, 7124, 7142, 7192
\peek_catcode:NTF .... 116, 116, 7216
\peek_catcode_ignore_spaces:NTF .
..... 116, 116, 7216
\peek_catcode_remove:NTF .....
..... 116, 116, 7216
\peek_catcode_remove_ignore_-
spaces:NTF ..... 116, 116, 7216
\peek_charcode:NTF ... 116, 116, 7232
\peek_charcode_ignore_spaces:NTF .
..... 117, 117, 7232
\peek_charcode_remove:NTF .....
..... 117, 117, 7232
\peek_charcode_remove_ignore_-
spaces:NTF ..... 117, 117, 7232
\peek_gafter:Nw ... 116, 116, 116, 7099
\peek_meaning:NTF .... 117, 117, 7248
\peek_meaning_ignore_spaces:NTF .
..... 117, 117, 7248
\peek_meaning_remove:NTF .....
..... 117, 117, 7248
\peek_meaning_remove_ignore_-
spaces:NTF ..... 118, 118, 7248

```

\peek_N_type:TF	1148
. 237, 237, 23367, 23401, 23403	
peek internal commands:	
__peek_def:nnnn 7199,	653
7216, 7220, 7224, 7228, 7232, 7236,	922
7240, 7244, 7248, 7252, 7256, 7260	512
__peek_def:nnnnn 7199	513
__peek_execute_branches:	923
. 452, 7196, 7211	924
__peek_execute_branches_-	514
catcode: 7156, 7219, 7221, 7227, 7229	515
__peek_execute_branches_-	516
catcode_aux: 7156	prg commands:
__peek_execute_branches_-	
catcode_auxii:N 7156	
__peek_execute_branches_-	
catcode_auxiii: 7156	
__peek_execute_branches_-	
charcode:	
. 7156, 7235, 7237, 7243, 7245	
__peek_execute_branches_-	
meaning: 7148, 7251, 7253, 7259, 7261	
__peek_execute_branches_N_type:	
. 23367	
__peek_false:w 937, 937, 7095, 7118,	
7136, 7153, 7176, 7186, 23384, 23397	
__peek_get_prefix_arg_replacement:wN	
. 7265	
__peek_ignore_spaces_execute_branches:	
. 7189,	
7223, 7231, 7239, 7247, 7255, 7263	
__peek_N_type:w 23367	
__peek_N_type_aux:nw 23367	
\l_peek_search_t1	
449, 451, 7094, 7112, 7133, 7173, 7183	
\l_peek_search_token	
. 449, 7093, 7111, 7132, 7150	
__peek_tmp:w 7095, 7107, 23368, 23390	
__peek_token_generic:NNTF . 937,	
7109, 7126, 7128, 23400, 23402, 23404	
__peek_token_remove_generic:NNTF	
. 7130, 7144, 7146	
__peek_true:w	
. 937, 937, 7095, 7113, 7134,	
7151, 7174, 7184, 23382, 23396, 23397	
__peek_true_aux:w 7095, 7106, 7135	
__peek_true_remove:w 7103, 7134	
\penalty	510
pi	190
\pm	12750, 12751
\postbreakpenalty	1147
\postdisplaypenalty	511
\postexhyphenchar	920
\posthyphenchar	921
\prebreakpenalty	1148
\predisplaydirection	653
\predisplaygapfactor	922
\predisplaypenalty	512
\predisplaysize	513
\preexhyphenchar	923
\prehyphenchar	924
\pretolerance	514
\prevdepth	515
\prevgraf	516
\prg_break_point:Nn	87
\prg_do_nothing: 9, 9, 97, 319,	
362, 362, 433, 566, 720, 783, 828,	
1534, 1535, 2048, 2058, 2405, 2410,	
2697, 2725, 2775, 2790, 4157, 4164,	
4404, 4406, 6166, 6507, 6511, 6518,	
8798, 9053, 9728, 10903, 10946,	
10980, 11006, 11014, 12454, 15829,	
16229, 16236, 16312, 16445, 16446,	
16692, 16736, 17596, 17633, 17634,	
17641, 17642, 19205, 19349, 22284	
\prg_new_conditional:Nnn 89,	
89, 1433, 5677, 22154, 22162, 22170	
\prg_new_conditional:Npnn	
. 89, 89, 89,	
442, 452, 1420, 2006, 2876, 2886,	
2898, 2920, 2922, 2970, 3254, 3273,	
3292, 3327, 3344, 3355, 3506, 3521,	
3526, 4281, 4794, 4847, 4884, 4892,	
5503, 5508, 5618, 5626, 5638, 5644,	
5677, 5750, 5785, 5868, 5877, 5883,	
5892, 5991, 5993, 5995, 5997, 6292,	
6855, 6860, 6865, 6870, 6877, 6883,	
6889, 6894, 6899, 6904, 6909, 6914,	
6919, 6924, 6931, 6948, 6953, 6988,	
7032, 7472, 7481, 7578, 8720, 9313,	
9318, 9534, 9546, 10505, 10512,	
10838, 11968, 12608, 12621, 17404,	
17425, 17447, 17505, 20091, 20093,	
20103, 20601, 22131, 22135, 22219	
\prg_new_eq_conditional>NNn	
. 90, 90,	
1525, 2560, 2561, 3472, 3473, 3474,	
3475, 4186, 4188, 4591, 4592, 4593,	
4594, 4595, 4596, 4743, 4745, 5677,	
5781, 5783, 6075, 6077, 6288, 6290,	
7468, 7470, 9253, 9255, 9509, 9511,	
9607, 9609, 12606, 12607, 20057, 20059	
\prg_new_protected_conditional:Nnn	
. 89, 89, 1433, 5677	
\prg_new_protected_conditional:Npnn	
. 89, 89, 1420, 2934, 2955,	
4293, 4403, 4405, 4413, 4415, 4417,	

4419, 5677, 6175, 6187, 6189, 6306,
 6310, 7404, 7414, 7511, 8537, 8656,
 17810, 19461, 19466, 19479, 19481
 \backslash prg_replicate:nn 96,
 96, 416, 416, 416, 5944, 8253, 8913,
 13889, 14743, 14977, 15181, 15182,
 15184, 15186, 15199, 15236, 15722,
 15730, 17022, 17602, 18283, 18648,
 18674, 18825, 18833, 19210, 19628,
 19633, 19640, 19739, 19744, 24561
 \backslash prg_return_false: 55,
 90, 91, 91, 91, 337, 366, 381, 381,
 427, 838, 1416, 1626, 1631, 1644,
 1649, 1657, 1674, 2009, 2891, 2904,
 2917, 2927, 2944, 2960, 2976, 3266,
 3289, 3305, 3313, 3323, 3336, 3350,
 3364, 3503, 3511, 3524, 3529, 4286,
 4307, 4330, 4792, 4824, 4829, 4852,
 4889, 4895, 5506, 5513, 5558, 5623,
 5631, 5677, 5755, 5790, 5874, 5880,
 5881, 5887, 5896, 5992, 5994, 5996,
 5998, 6178, 6194, 6295, 6320, 6858,
 6863, 6868, 6873, 6880, 6887, 6892,
 6897, 6902, 6907, 6912, 6917, 6922,
 6927, 6945, 6951, 6956, 6961, 6994,
 6997, 7036, 7061, 7078, 7087, 7412,
 7422, 7475, 7497, 7518, 7581, 8541,
 8665, 8729, 9316, 9337, 9352, 9353,
 9541, 9549, 10510, 10518, 10849,
 10851, 11983, 11995, 12616, 12629,
 17419, 17430, 17433, 17438, 17442,
 17443, 17451, 17454, 17459, 17462,
 17511, 17514, 17817, 17822, 19507,
 20092, 20094, 20104, 20607, 20609,
 22136, 22157, 22167, 22175, 22225
 \backslash prg_return_true/false: 344
 \backslash prg_return_true:
 55, 90, 91, 91, 91, 338, 338, 366,
 459, 837, 838, 1416, 1629, 1646,
 1654, 1659, 1672, 1677, 2009, 2889,
 2902, 2915, 2925, 2941, 2960, 2974,
 3264, 3287, 3303, 3321, 3334, 3352,
 3363, 3501, 3511, 3524, 3529, 4284,
 4311, 4333, 4824, 4850, 4887, 4897,
 5506, 5511, 5556, 5621, 5629, 5677,
 5753, 5788, 5872, 5880, 5889, 5895,
 5896, 5992, 5994, 5996, 5998, 6181,
 6197, 6296, 6320, 6858, 6863, 6868,
 6873, 6880, 6887, 6892, 6897, 6902,
 6907, 6912, 6917, 6922, 6927, 6943,
 6951, 6959, 7059, 7085, 7410, 7420,
 7475, 7499, 7516, 7581, 8542, 8668,
 8727, 8732, 8734, 9316, 9353, 9539,
 9550, 10509, 10517, 10842, 10847,
 11978, 12001, 12618, 12627, 17407,
 17422, 17430, 17433, 17438, 17442,
 17454, 17459, 17462, 17509, 17813,
 17819, 19505, 20092, 20094, 20104,
 20606, 22132, 22159, 22165, 22173
 \backslash prg_set_conditional:Nnn
 89, 1433, 5677
 \backslash prg_set_conditional:Npnn
 89, 90, 91, 1420,
 1623, 1635, 1651, 1663, 5552, 5677
 \backslash prg_set_eq_conditional:NNn
 90, 1525, 5677
 \backslash prg_set_protected_conditional:Nnn
 89, 1433, 5677
 \backslash prg_set_protected_conditional:Npnn
 89, 1420, 5677
prg internal commands:
 \backslash _prg_break:
 97, 825, 2058, 3383, 4440,
 6005, 7501, 10894, 15847, 16377,
 16566, 16789, 16863, 16894, 16895,
 16896, 16897, 16898, 16899, 16959,
 17305, 19128, 19155, 22070, 22076
 \backslash _prg_break:n 97, 97, 97, 2058,
 3385, 4311, 4453, 6005, 7400, 18720
 \backslash _prg_break_point: 97, 97, 740, 907,
 2058, 3373, 4308, 4441, 6005, 7395,
 7486, 10895, 15848, 16370, 16377,
 16567, 16716, 16864, 16901, 16953,
 17282, 18721, 18999, 19149, 22071
 \backslash _prg_break_point:Nn 40, 40,
 62, 63, 97, 97, 97, 97, 102, 103, 141,
 142, 298, 298, 298, 371, 386, 2049,
 3019, 3037, 3046, 4468, 4503, 4514,
 5009, 6005, 6337, 6351, 6371, 6389,
 7530, 7549, 8765, 12817, 16835, 22034
 \backslash _prg_case_end:nw
 23, 23, 2980, 3595, 4883, 9382
 \backslash _prg_compare_error: 80, 80, 380,
 381, 513, 4779, 4797, 4799, 9321, 9323
 \backslash _prg_compare_error:Nw
 80, 80, 380, 381, 382, 4779, 4819
 \backslash _prg_generate_conditional:nnNnnnn
 1430, 1449, 1458
 \backslash _prg_generate_conditional:nnnnnnw
 1458
 \backslash _prg_generate_conditional_-
 count:nnNnn 1433
 \backslash _prg_generate_conditional_-
 count:nnNnnnn 1433
 \backslash _prg_generate_conditional_-
 parm:nnNpnn 1420
 \backslash _prg_generate_F_form:wnnnnnn 1490
 \backslash _prg_generate_p_form:wnnnnnn 1490

__prg_generate_T_form:wnnnnnn [1490](#)
 __prg_generate_TF_form:wnnnnnn [1490](#)
 __prg_map_1:w [97](#)
 __prg_map_2:w [97](#)
 __prg_map_break:Nn
 [97](#), [97](#), [298](#), [331](#), [430](#), [463](#),
 [2049](#), [3059](#), [3061](#), [4458](#), [4460](#), [6005](#),
 [6406](#), [6408](#), [7558](#), [7560](#), [8748](#), [8750](#)
 \g__prg_map_int . [97](#), [97](#), [330](#), [386](#),
 [757](#), [3031](#), [3033](#), [3035](#), [3037](#), [4489](#),
 [4490](#), [4496](#), [4497](#), [4988](#), [4991](#), [4995](#),
 [4998](#), [5009](#), [6004](#), [6365](#), [6367](#), [6369](#),
 [6372](#), [7545](#), [7546](#), [7551](#), [7553](#), [8757](#),
 [8759](#), [8766](#), [12796](#), [12799](#), [12803](#),
 [12806](#), [12817](#), [16820](#), [16822](#), [16835](#)
 __prg_replicate:N [5944](#)
 __prg_replicate_ [5944](#)
 __prg_replicate_0:n [5944](#)
 __prg_replicate_1:n [5944](#)
 __prg_replicate_2:n [5944](#)
 __prg_replicate_3:n [5944](#)
 __prg_replicate_4:n [5944](#)
 __prg_replicate_5:n [5944](#)
 __prg_replicate_6:n [5944](#)
 __prg_replicate_7:n [5944](#)
 __prg_replicate_8:n [5944](#)
 __prg_replicate_9:n [5944](#)
 __prg_replicate_first:N [5944](#)
 __prg_replicate_first_-:n [5944](#)
 __prg_replicate_first_0:n [5944](#)
 __prg_replicate_first_1:n [5944](#)
 __prg_replicate_first_2:n [5944](#)
 __prg_replicate_first_3:n [5944](#)
 __prg_replicate_first_4:n [5944](#)
 __prg_replicate_first_5:n [5944](#)
 __prg_replicate_first_6:n [5944](#)
 __prg_replicate_first_7:n [5944](#)
 __prg_replicate_first_8:n [5944](#)
 __prg_replicate_first_9:n [5944](#)
 __prg_set_eq_conditional:NNNn [1525](#)
 __prg_set_eq_conditional:nnNnnNNw
 [1533](#), [1541](#)
 __prg_set_eq_conditional_F_-
 form:nnn [1541](#)
 __prg_set_eq_conditional_F_-
 form:wnnnnn [1587](#)
 __prg_set_eq_conditional_-
 loop:nnnnNw [1541](#)
 __prg_set_eq_conditional_p_-
 form:nnn [1541](#)
 __prg_set_eq_conditional_p_-
 form:wnnnnn [1572](#)
 __prg_set_eq_conditional_T_-
 form:nnn [1541](#)
 __prg_set_eq_conditional_T_-
 form:wNnnn [1582](#)
 __prg_set_eq_conditional_TF_-
 form:nnn [1541](#)
 __prg_set_eq_conditional_TF_-
 form:wNnnn [1577](#)
 \primitive [857](#)
 prop commands:
 \c_empty_prop
 [127](#), [456](#), [7303](#), [7307](#), [7311](#), [7314](#), [7474](#)
 \prop_clear:N
 [122](#), [122](#), [7310](#), [7317](#), [18958](#), [20987](#)
 \prop_clear:N [122](#)
 \prop_clear_new:N
 [122](#), [122](#), [7316](#), [20637](#), [20638](#)
 \prop_count:N . [229](#), [229](#), [22020](#), [22052](#)
 \prop_gclear:N [122](#), [7310](#), [7320](#)
 \prop_gclear_new:N [122](#), [7316](#)
 \prop_get:Nn [97](#), [21533](#), [21534](#)
 \prop_get:NnN [84](#), [85](#), [123](#), [123](#),
 [124](#), [7361](#), [21210](#), [21214](#), [21293](#), [21297](#)
 \prop_get:NnNTF [123](#),
 [124](#), [125](#), [125](#), [7511](#), [7520](#), [7521](#),
 [7523](#), [7524](#), [7917](#), [7937](#), [7992](#), [20764](#)
 \prop_gpop:NnN [123](#), [123](#), [7369](#)
 \prop_gpop:NnNTF
 [123](#), [125](#), [125](#), [7404](#), [7427](#), [7428](#)
 \prop_gput:Nnn
 [123](#), [7430](#), [8641](#), [8695](#), [8817](#), [8849](#)
 \prop_gput_if_new:Nnn [123](#), [7451](#)
 \prop_gremove:Nn [124](#), [7345](#), [8703](#), [8857](#)
 \prop_gset_eq:NN [122](#),
 [7314](#), [7322](#), [20639](#), [20641](#), [20788](#), [20790](#)
 \prop_if_empty:NTF [124](#), [124](#),
 [7472](#), [7478](#), [7479](#), [7564](#), [8715](#), [22049](#)
 \prop_if_empty_p:N [124](#), [124](#), [7472](#)
 \prop_if_exist:NTF
 [124](#), [124](#), [7317](#), [7320](#), [7468](#), [7564](#)
 \prop_if_exist_p:N [124](#), [124](#), [7468](#)
 \prop_if_in:NnTF
 [124](#), [124](#), [7481](#), [7505](#), [7506](#), [7507](#), [7508](#)
 \prop_if_in_p:Nn [124](#), [7481](#)
 \prop_item:Nn
 [124](#), [124](#), [229](#), [7391](#), [21533](#), [21534](#)
 \prop_log:N [126](#), [126](#), [7568](#)
 \prop_map_break: [126](#), [126](#), [905](#),
 [7530](#), [7535](#), [7549](#), [7557](#), [22034](#), [22039](#)
 \prop_map_break:n [126](#), [126](#), [7557](#)
 \prop_map_function:NN
 [125](#), [125](#), [229](#), [461](#),
 [905](#), [7526](#), [7565](#), [8717](#), [21361](#), [22025](#)
 \prop_map_inline:Nn
 [125](#), [125](#), [7542](#), [19650](#),
 [21058](#), [21077](#), [21262](#), [21271](#), [21686](#),

21688, 21691, 21711, 21713, 21778,
 21795, 21839, 21841, 21845, 21847
`\prop_map_tokens:Nn . 229, 229, 22030`
`\prop_new:N 122, 122, 122, 7304, 7317,`
`7320, 7330, 7331, 7332, 7333, 7781,`
`7891, 8627, 8803, 18907, 18908,`
`20573, 20578, 21136, 21177, 21676`
`\prop_pop:NnN 123, 123, 7369`
`\prop_pop:NnTF`
`123, 125, 125, 7404, 7424, 7425`
`\prop_put:Nnn 123, 123,`
`127, 310, 455, 7430, 7965, 7981,`
`7998, 19115, 20574, 20575, 20576,`
`20577, 20580, 20581, 20582, 20584,`
`20585, 20586, 20587, 20588, 20589,`
`20816, 20822, 20824, 20826, 20828,`
`20833, 20838, 20843, 20850, 20857,`
`21082, 21137, 21139, 21141, 21143,`
`21145, 21147, 21149, 21151, 21153,`
`21155, 21157, 21159, 21161, 21163,`
`21165, 21167, 21169, 21171, 21719,`
`21721, 21724, 21726, 21732, 21738,`
`21805, 21813, 21876, 21890, 21897`
`\prop_put_if_new:Nnn . 123, 123, 7451`
`\prop_rand_key_value:N 229, 229, 22045`
`\prop_remove:Nn 124, 124,`
`7345, 7962, 7977, 21257, 21260, 21264`
`\prop_set_eq:NN`
`122, 122, 7311, 7322, 19126, 20774,`
`20776, 20781, 20783, 21017, 21252`
`\prop_show:N 126, 126, 464, 7561, 7569`
`\g_tmpa_prop 127, 7330`
`\l_tmpa_prop 127, 7330`
`\g_tmpb_prop 127, 7330`
`\l_tmpb_prop 127, 7330`
 prop internal commands:
`_prop_count:nn 22020`
`_prop_if_in:N 461, 7481`
`_prop_if_in:nwwn 461, 7481`
`\l_prop_internal_tl 127, 459,`
`460, 7302, 7434, 7440, 7441, 7457, 7464`
`_prop_item_Nn:nwn 458`
`_prop_item_Nn:nwwn 7391`
`_prop_map_function:Nwwn 7526`
`_prop_map_tokens:nwwn 905, 22030`
`_prop_pair:wn 127, 127, 127,`
`455, 455, 455, 457, 457, 461, 463,`
`463, 7300, 7339, 7342, 7394, 7397,`
`7436, 7459, 7484, 7488, 7529, 7532,`
`7545, 7547, 7552, 22033, 22036, 22056`
`_prop_put>NNnn 7430`
`_prop_put_if_new>NNnn 7451`
`_prop_rand:NN 22045`
`_prop_rand:nNn 22045, 22046`
`_prop_rand_item:Nw 22045`
`_prop_split:NnTF`
`127, 127, 459, 459, 459, 460,`
`461, 7334, 7347, 7353, 7363, 7371,`
`7380, 7406, 7416, 7439, 7462, 7513`
`_prop_split_aux:NnTF 7334`
`_prop_split_aux:w 457, 457, 7334`
`\protect 11488, 22553, 22580`
`\protected 207,`
`209, 211, 236, 503, 654, 7020, 7022`
`\protrudechars 967`
`\ProvidesExplClass 6`
`\ProvidesExplFile 6, 23479`
`\ProvidesExplPackage 6, 6`
`pt 191`
 ptex commands:
`\ptex_automspacing:D 1123`
`\ptex_autoxspacing:D 1124`
`\ptex_dtou:D 1125`
`\ptex_euc:D 1126`
`\ptex_ifdbox:D 1127`
`\ptex_ifddir:D 1128`
`\ptex_ifmdir:D 1129`
`\ptex_iftbox:D 1130`
`\ptex_iftdir:D 1131`
`\ptex_ifybox:D 1132`
`\ptex_ifydir:D 1133`
`\ptex_inhibitglue:D 1134`
`\ptex_inhibitxspcode:D 1135`
`\ptex_jcharwidowpenalty:D 1136`
`\ptex_jfam:D 1137`
`\ptex_jfont:D 1138`
`\ptex_jis:D 1139, 4717, 21453`
`\ptex_kanjiskip:D 1140, 16578, 21449`
`\ptex_kansuji:D 1141`
`\ptex_kansujichar:D 1142`
`\ptex_kcatcode:D 1143`
`\ptex_kuten:D 1144`
`\ptex_noautospacing:D 1145`
`\ptex_noautoxspacing:D 1146`
`\ptex_postbreakpenalty:D 1147`
`\ptex_preambrepenalty:D 1148`
`\ptex_showmode:D 1149`
`\ptex_sjis:D 1150`
`\ptex_tate:D 1151`
`\ptex_tbaselineshift:D 1152`
`\ptex_tfont:D 1153`
`\ptex_xkanjiskip:D 1154`
`\ptex_xspcode:D 1155`
`\ptex_ybaselineshift:D 1156`
`\ptex_yoko:D 1157`
`\pxdimen 968`

Q

quark commands:

```

\q_mark    23, 23, 46, 46, 85, 285, 285,
            285, 309, 310, 323, 323, 330, 330,
            333, 333, 333, 333, 347, 347, 354,
            410, 421, 421, 423, 423, 425, 425,
            425, 426, 426, 431, 431, 431, 457,
            525, 739, 739, 739, 740, 741, 746,
            1607, 1608, 1614, 1616, 1617, 2344,
            2345, 2347, 2354, 2360, 2384, 2393,
            2407, 2415, 2418, 2427, 2442, 2464,
            2485, 2488, 2501, 2796, 2798, 2800,
            2802, 3001, 3012, 3090, 3091, 3094,
            3097, 3098, 3104, 3118, 3119, 3125,
            3129, 3131, 3134, 3556, 3588, 3599,
            3616, 3860, 3862, 4540, 4541, 4555,
            4558, 4807, 4810, 4876, 5567, 5797,
            5798, 5799, 5800, 5803, 5804, 5805,
            5806, 5807, 5808, 5809, 6089, 6098,
            6103, 6158, 6168, 6172, 6196, 6248,
            6254, 6267, 6279, 6280, 6281, 6284,
            6285, 6286, 6295, 6296, 6305, 6350,
            6358, 6447, 6448, 6460, 6461, 6524,
            7339, 7341, 7342, 7922, 7923, 7928,
            7931, 9094, 9101, 9113, 9375, 9659,
            9666, 9676, 9700, 9722, 9732, 9742,
            11408, 11409, 11414, 16383, 16418,
            16420, 16423, 16427, 16430, 16433,
            16435, 16438, 23379, 23380, 23387
\q_nil ..... 19, 19,
            85, 85, 85, 278, 323, 326, 326,
            326, 326, 333, 402, 404, 404, 404,
            410, 410, 410, 1401, 1404, 2822,
            2900, 2901, 2913, 2914, 3117, 3121,
            3139, 3142, 3145, 3233, 3234, 5567,
            5620, 5641, 5643, 5796, 5798, 5799,
            5803, 5804, 5805, 5806, 5807, 5808,
            9676, 9686, 9700, 9710, 18249, 18264
\q_no_value ..... .
            . 58, 59, 59, 59, 59, 59, 64, 64,
            64, 84, 85, 85, 85, 85, 104, 123, 123,
            123, 138, 367, 368, 402, 403, 423,
            458, 458, 489, 4321, 4329, 4341,
            4365, 5567, 5628, 5647, 5649, 5855,
            6141, 6156, 7365, 7376, 7385, 8509
\quark_if_nil:n ..... 404, 404, 404
\quark_if_nil:NTF ..... .
            . 85, 85, 5618, 18268, 18288
\quark_if_nil:nTF ..... .
            . 85, 85, 403, 2820, 5638, 5652, 5653
\quark_if_nil_p:N ..... 85, 85, 5618
\quark_if_nil_p:n ..... 85, 85, 5638
\quark_if_no_value:NTF ..... 85,
            85, 5618, 5635, 5636, 5860, 8540,
```

8652, 8664, 21212, 21216, 21295, 21299
\quark_if_no_value:nTF . . 85, 85, 5638
\quark_if_no_value_p:N . . 85, 85, 5618
\quark_if_no_value_p:n . . 85, 85, 5638
\quark_if_recursion_tail_break:N
..... 21535
\quark_if_recursion_tail_break:n
..... 21536
\quark_if_recursion_tail_stop:N .
.... 86, 86, 5344, 5573, 6400, 23085
\quark_if_recursion_tail_stop:n .
. 86, 86, 341, 5587, 6094, 6430, 22918
quark_if_recursion_tail_stop:n . 403
\quark_if_recursion_tail_stop_-
do:Nn 86, 86,
3898, 5283, 5300, 5347, 5573, 22374,
22382, 22541, 22561, 23155, 23163
\quark_if_recursion_tail_stop_-
do:nn 86, 86, 5587, 5872, 5887, 23208
\quark_new:N 84, 84, 5566, 5567, 5568,
5569, 5570, 5571, 5572, 5654, 5655
\q_recursion_stop . . 19, 19, 86, 86,
86, 86, 86, 87, 278, 282, 402, 1403,
1406, 1470, 1538, 2334, 2658, 3876,
3883, 3888, 4076, 5278, 5295, 5338,
5365, 5571, 5869, 5884, 6090, 6388,
6424, 18249, 18264, 22332, 22335,
22344, 22355, 22365, 22372, 22378,
22397, 22399, 22408, 22410, 22417,
22418, 22421, 22424, 22437, 22535,
22557, 22579, 22605, 22610, 22613,
22615, 22618, 22621, 22626, 22647,
22650, 22652, 22654, 22659, 22740,
22744, 22746, 22751, 22765, 22790,
22794, 22796, 22801, 22809, 23057,
23103, 23122, 23125, 23134, 23136,
23146, 23153, 23159, 23181, 23185,
23188, 23189, 23204, 23220, 23222,
23243, 23247, 23249, 23254, 23269
\q_recursion_tail 86, 86, 86, 86, 86,
86, 86, 86, 86, 87, 87, 87, 282, 402,
403, 428, 461, 462, 462, 905, 1470,
1475, 1538, 1557, 2634, 2658, 3018,
3036, 3045, 3372, 3876, 4066, 4076,
5278, 5295, 5338, 5571, 5575, 5581,
5590, 5597, 5602, 5607, 5614, 5869,
5884, 6090, 6336, 6350, 6370, 6388,
6424, 7485, 7496, 7529, 7534, 9659,
9666, 9727, 22033, 22038, 22332,
22378, 22412, 22535, 22557, 22588,
23056, 23102, 23122, 23159, 23204
\q_stop . . 19, 19, 23, 23, 29, 44, 84,
85, 85, 85, 278, 285, 310, 323, 332,
347, 351, 381, 392, 392, 402, 426,

426, 426, 431, 445, 447, 457, 482, 740, 740, 741, 1402, 1405, 1486, 1491, 1508, 1514, 1520, 1568, 1572, 1577, 1582, 1587, 1609, 1614, 1616, 1617, 2348, 2361, 2388, 2415, 2419, 2423, 2431, 2437, 2446, 2464, 2491, 2502, 2821, 2973, 2979, 3001, 3012, 3092, 3094, 3099, 3101, 3123, 3145, 3224, 3226, 3243, 3261, 3280, 3302, 3556, 3588, 3599, 3608, 3614, 3616, 3622, 3639, 3658, 3717, 3773, 3785, 3823, 3839, 3846, 3854, 3856, 3860, 3862, 3986, 3991, 3994, 3998, 4016, 4029, 4040, 4044, 4047, 4049, 4050, 4053, 4056, 4341, 4344, 4352, 4354, 4434, 4435, 4542, 4555, 4558, 4560, 4786, 4802, 4804, 4808, 4821, 4876, 5271, 5277, 5294, 5567, 5801, 5809, 5855, 5858, 6143, 6146, 6158, 6161, 6169, 6172, 6180, 6196, 6254, 6281, 6284, 6285, 6297, 6305, 6449, 6460, 6461, 6462, 6488, 6522, 6936, 6939, 6971, 7005, 7040, 7044, 7050, 7073, 7267, 7274, 7283, 7292, 7339, 7342, 7924, 8390, 8407, 9022, 9079, 9117, 9130, 9178, 9179, 9326, 9352, 9375, 9550, 9552, 9730, 9745, 9767, 9781, 9782, 9847, 9849, 9869, 9873, 9882, 9890, 9903, 10343, 10351, 10361, 10488, 10490, 10495, 11409, 11414, 11524, 11529, 15894, 15965, 16371, 16380, 16384, 16386, 16418, 16419, 16420, 16425, 16427, 16431, 16433, 16441, 22017, 22053, 22066, 22067, 22069, 22073, 22077, 22079, 22084, 23368, 23381, 23390, 23803, 23816

quark internal commands:

- \s_fp 550, 550, 550, 550, 552, 552, 552, 583, 584, 587, 599, 599, 601, 625, 627, 627, 629, 629, 629, 631, 637, 637, 640, 640, 720, 10630, 10643, 10644, 10645, 10646, 10647, 10657, 10662, 10664, 10665, 10686, 10689, 10691, 10701, 10713, 10733, 10826, 10828, 10830, 10831, 10832, 10834, 10835, 10836, 10838, 10854, 11024, 11029, 11266, 11312, 11321, 11323, 11973, 12097, 12282, 12613, 12638, 12639, 12758, 12771, 12775, 12831, 12832, 12835, 12846, 12847, 12855, 12856, 12858, 12859, 12860, 12862, 12863, 12864, 12875, 12878, 12890, 12916, 12949, 12952, 12955, 12975, 12976, 12978, 12979, 12980, 12988, 12991, 13002, 13003, 13005, 13014, 13090, 13242, 13276, 13277, 13280, 13359, 13495, 13503, 13505, 13682, 13690, 13692, 13694, 13697, 14159, 14171, 14173, 14381, 14398, 14400, 14586, 14605, 14607, 14608, 14611, 14628, 14631, 14634, 14659, 14660, 14662, 14678, 14763, 14776, 14778, 14781, 14786, 14848, 14861, 14863, 14876, 14878, 14891, 14893, 14906, 14908, 14921, 14923, 14936, 14946, 15378, 15394, 15395, 15399, 15410, 15516, 15529, 15531, 15547, 15550, 15560, 15583, 15594, 15596, 15610, 15612, 15617, 15645, 15666, 15669, 15686, 15707, 15710, 15751, 15767, 15770, 15827, 15828, 15959, 15961
- \s_fp_division 10638
- \s_fp_exact 10638, 10643, 10644, 10645, 10646, 10647, 12831
- \s_fp_invalid 10638
- \s_fp_mark 563, 583, 583, 587, 606, 608, 616, 10636, 10904, 10905, 10907, 10911, 12146, 12185, 12560, 12561, 12564, 12567, 12568
- \s_fp_overflow 10638, 10664
- \s_fp_stop 10636, 12054, 12147, 12150, 12562, 12567, 12568, 12570, 12894, 12905, 12929, 12937
- \s_fp_underflow 10638, 10662
- \s_prop 127, 127, 455, 455, 455, 455, 457, 457, 462, 463, 905, 7299, 7300, 7303, 7339, 7342, 7394, 7397, 7437, 7460, 7484, 7488, 7529, 7532, 7547, 22033, 22036, 22056, 22061
- _quark_if_nil:w 404, 404, 5638
- _quark_if_no_value:w 5638
- _quark_if_recursion_tail:w 403, 5587, 5614
- _quark_if_recursion_tail_-break>NN 87, 3053, 5605
- _quark_if_recursion_tail_-break:nN 87, 87, 3025, 3383, 5605, 6342, 6355
- \s_seq 67, 359, 362, 363, 363, 368, 373, 733, 906, 4082, 4090, 4120, 4125, 4130, 4135, 4168, 4194, 4202, 4206, 4385, 4435, 4552, 5672, 16215, 16222, 19594, 22067, 22073
- \s_stop 88, 88, 88, 740, 5670, 5671, 14457, 14472, 15632, 15636, 16384, 16386

\s__tl 744,
 745, 745, 745, 745, 745, 745, 745,
 745, 745, 745, 745, 746, 746,
 746, 747, 747, 747, 747, 747, 747,
 754, 754, 755, 755, 16519, 16521,
 16733, 16760, 16766, 16791, 16809,
 16814, 16826, 16834, 16863, 16867
\q__tl_act_mark
 334, 334, 334, 3149, 3152, 3169, 5654
\q__tl_act_stop
 334, 3149, 3152, 3156, 3165, 3167,
 3173, 3178, 3181, 3185, 3188, 5654
\quitvmode 784

R

\r 23109
\radical 517
\raise 518
rand 190
randint 190
\randomseed 969
\read 519
\readline 655
\ref 23280
regex commands:
 \c_foo_regex 197
\regex_(g)set:Nn 203
\regex_const:Nn 197, 203, 203, 19438
\regex_count:NnN 204, 19471
\regex_count:nnN 204, 204, 838, 19471
\regex_extract_all:NnN 19475
\regex_extract_all:nnN 204, 764, 19475
\regex_extract_all:NnNTF 204, 19475
\regex_extract_all:nnNTF
 204, 204, 19475
\regex_extract_once:NnN 19475
\regex_extract_once:nnN
 204, 204, 19475
\regex_extract_once:NnNTF 204, 19475
\regex_extract_once:nnNTF
 201, 204, 204, 19475
\regex_gset:Nn 203, 203, 19438
\regex_match:NnTF 203, 19461
\regex_match:nnTF 203, 203, 19461
\regex_new:N 203, 203, 766, 19436
\regex_replace_all:NnN 19475
\regex_replace_all:nnN 205, 19475
\regex_replace_all:NnNTF 205, 19475
\regex_replace_all:nnNTF
 205, 205, 19475
\regex_replace_once:NnN 19475
\regex_replace_once:nnN 205, 19475
\regex_replace_once:NnNTF 205, 19475

\regex_replace_once:nnNTF
 205, 205, 19475
\regex_set:Nn 203, 203, 19438
\regex_show:N 203, 19453
\regex_show:n 203, 203, 19453
\regex_split:NnN 19475
\regex_split:nnN 205, 19475
\regex_split:NnNTF 205, 19475
\regex_split:nnNTF 205, 205, 19475

regex internal commands:

__regex_action_cost:n 806, 809,
 818, 18636, 18637, 18645, 19065, 19091
__regex_action_free:n
 206, 807, 817, 18659, 18665,
 18666, 18677, 18739, 18743, 18769,
 18794, 18798, 18801, 18829, 18837,
 18847, 18861, 18892, 19063, 19067
__regex_action_free_aux:nn .. 19067
__regex_action_free_group:n 206,
 807, 817, 18687, 18809, 18812, 19067
__regex_action_start_wildcard: ..
 806, 18556, 19060
__regex_action_submatch:n
 807, 18763, 18764, 18890, 19111, 19113
__regex_action_success:
 206, 806, 18559, 18575, 19118
__regex_action_wildcard: 821
\c__regex_all_catcodes_int
 17477, 17584, 17666, 18221
__regex_anchor:N
 777, 816, 17889, 18402, 18852
\c__regex_ascii_lower_int
 17070, 17127, 17132
\c__regex_ascii_max_control_int ..
 17067, 17239
\c__regex_ascii_max_int
 17067, 17232, 17240, 17429
\c__regex_ascii_min_int
 17067, 17231, 17238
__regex_assertion:Nn 777,
 816, 17889, 17914, 17923, 18399, 18852
__regex_b_test:
 777, 816, 17914, 17923, 18401, 18852
\l__regex_balance_int
 ... 766, 828, 17065, 18925, 19020,
 19024, 19025, 19191, 19220, 19382,
 19396, 19678, 19703, 19726, 19730,
 19731, 19738, 19739, 19743, 19744
\g__regex_balance_intarray
 . 763, 766, 17062, 19019, 19174, 19182
\l__regex_balance_t1
 . 206, 828, 19134, 19192, 19221, 19271

__regex_branch:n
 777, 794, 812, 17059, 17588,
 18060, 18103, 18260, 18383, 18733
 __regex_break_point:TF
 767, 767, 789, 809, 17071,
 17077, 18636, 18637, 18858, 18881
 __regex_break_true:w
 ... 767, 767, 17071, 17077, 17082,
 17089, 17096, 17102, 17109, 17117,
 17162, 17173, 17189, 17864, 18873
 __regex_build:N
 837, 18542, 19468, 19474, 19478, 19482
 __regex_build:n
 837, 18542, 19463, 19472, 19477, 19480
 __regex_build_for_cs:n 17185, 18563
 __regex_build_new_state:
 18553, 18554,
 18567, 18568, 18599, 18618, 18650,
 18685, 18690, 18736, 18752, 18757,
 18796, 18815, 18850, 18854, 18887
 __regex_build_transition_-
 left:NNN 18595, 18798, 18812, 18829
 __regex_build_transition_-
 right:nNn 18595,
 18651, 18687, 18739, 18743, 18769,
 18794, 18801, 18809, 18837, 18847
 __regex_build_transitions_-
 lazyness:NNNNN 18616, 18658, 18664, 18676
 \l__regex_capturing_group_int ...
 764, 806, 843, 18541, 18551, 18704,
 18706, 18717, 18718, 18726, 18727,
 18730, 19268, 19640, 19708, 19716
 \l__regex_case_changed_char_int .
 768,
 17098, 17101, 17112, 17115, 17116,
 17123, 17127, 17132, 18902, 18988
 \c__regex_catcode_A_int 17477
 \c__regex_catcode_B_int 17477
 \c__regex_catcode_C_int 17477
 \c__regex_catcode_D_int 17477
 \c__regex_catcode_E_int 17477
 \c__regex_catcode_in_class_mode_-
 int 17467, 17575, 17930, 18159, 18175
 \g__regex_catcode_intarray
 763, 766, 17062, 19017, 19034
 \c__regex_catcode_L_int 17477
 \c__regex_catcode_M_int 17477
 \c__regex_catcode_mode_int 17467,
 17572, 17635, 17962, 18157, 18173
 \c__regex_catcode_0_int 17477
 \c__regex_catcode_P_int 17477
 \c__regex_catcode_S_int 17477
 \c__regex_catcode_T_int 17477
 \c__regex_catcode_U_int 17477
 \l__regex_catcodes_bool
 17474, 18180, 18184, 18219
 \l__regex_catcodes_int .. 778, 778,
 17474, 17585, 17666, 17667, 17673,
 17949, 17966, 18058, 18067, 18154,
 18177, 18212, 18214, 18220, 18221
 __regex_char_if_alphanumeric:N .
 17447
 __regex_char_if_alphanumeric:NTF ..
 17425, 17628
 __regex_char_if_special:N ... 17425
 __regex_char_if_special:NTF ...
 17425, 17624
 \g__regex_charcode_intarray
 763, 766, 17062, 19015, 19031
 __regex_chk_c_allowed:TF
 17557, 18147
 __regex_class:NnnnN
 777, 784, 785, 792,
 17060, 17662, 17957, 17958, 17964,
 18276, 18352, 18362, 18396, 18630
 \c__regex_class_mode_int
 17467, 17562, 17576
 __regex_class_repeat:n
 810, 18640, 18646, 18662, 18671
 __regex_class_repeat:nN 18641, 18655
 __regex_class_repeat:nnN
 18642, 18669
 __regex_command_K:
 777, 18377, 18397, 18885
 __regex_compile:n 17617,
 18544, 19440, 19445, 19450, 19455
 __regex_compile:w
 783, 17580, 17619, 18226
 __regex_compile_\$: 17884
 __regex_compile_(: 18075
 __regex_compile_): 18106
 __regex_compile_.: 17855
 __regex_compile/_A: 17884
 __regex_compile/_B: 17908
 __regex_compile/_b: 17908
 __regex_compile/_c: 18146
 __regex_compile/_D: 17867
 __regex_compile/_d: 17867
 __regex_compile/_G: 17884
 __regex_compile/_H: 17867
 __regex_compile/_h: 17867
 __regex_compile/_K: 18374
 __regex_compile/_N: 17867
 __regex_compile/_S: 17867
 __regex_compile/_s: 17867
 __regex_compile/_u: 18296
 __regex_compile/_V: 17867

__regex_compile_v: 17867
 __regex_compile_W: 17867
 __regex_compile_w: 17867
 __regex_compile_Z: 17884
 __regex_compile_z: 17884
 __regex_compile\[\: 17941
 __regex_compile\]\: 17926
 __regex_compile\^\: 17884
 __regex_compile_abort_tokens:n .
 17676, 17700, 18039, 18049
 __regex_compile_anchor:NTF .. 17884
 __regex_compile_c\[:w 18169
 __regex_compile_c_lbrack_add:N ..
 18169
 __regex_compile_c_lbrack_end: 18169
 __regex_compile_c_lbrack_ -
 loop:NN 18169
 __regex_compile_c_test:NN ... 18146
 __regex_compile_class:NN 17971
 __regex_compile_class:TFNN
 792, 17956, 17967, 17971
 __regex_compile_class_catcode:w ..
 17948, 17960
 __regex_compile_class_normal:w ..
 17951, 17954
 __regex_compile_class_posix:NNNNw ..
 17990
 __regex_compile_class_posix_ -
 end:w 17990
 __regex_compile_class_posix_ -
 loop:w 17990
 __regex_compile_class_posix_ -
 test:w 17944, 17990
 __regex_compile_cs_aux:Nn ... 18235
 __regex_compile_cs_aux:NNnnN 18235
 __regex_compile_end:
 783, 783, 17580, 17644, 18244
 __regex_compile_end_cs: 17640, 18235
 __regex_compile_escaped:N ..
 17629, 17646
 __regex_compile_group_begin:N ..
 18053, 18089, 18094, 18112, 18114
 __regex_compile_group_end: ..
 18053, 18109
 __regex_compile_lparen:w ..
 18078, 18080
 __regex_compile_one:n ..
 17656, 17799, 17805, 17859,
 17870, 17873, 17883, 18030, 18251
 __regex_compile_quantifier:w ..
 17674, 17685, 17936, 18069
 __regex_compile_quantifier_*:w ..
 17712
 __regex_compile_quantifier\+_w ..
 17712
 __regex_compile_quantifier\?w ..
 17712
 __regex_compile_quantifier_ -
 abort:nNN ..
 ... 17694, 17722, 17741, 17755, 17778
 __regex_compile_quantifier_ -
 braced_auxi:w 17718
 __regex_compile_quantifier_ -
 braced_auxii:w 17718
 __regex_compile_quantifier_ -
 braced_auxiii:w 17718
 __regex_compile_quantifier_ -
 lazyness:nnNN 786, 17703, 17713,
 17715, 17717, 17730, 17751, 17773
 __regex_compile_quantifier_ -
 none: 17690, 17692, 17694
 __regex_compile_range:Nw ..
 17797, 17810
 __regex_compile_raw:N ..
 17507, 17625, 17629,
 17631, 17649, 17654, 17681, 17790,
 17792, 17812, 17858, 17904, 17939,
 17987, 18001, 18019, 18072, 18077,
 18090, 18100, 18108, 18124, 18125,
 18126, 18133, 18140, 18141, 18142,
 18150, 18191, 18240, 18308, 18314
 __regex_compile_raw_error:N ...
 17785, 17911, 17920, 18299, 18378
 __regex_compile_special:N ..
 17625, 17646, 17687, 17705, 17728,
 17733, 17749, 17763, 17796, 17815,
 17974, 17992, 18005, 18026, 18082,
 18117, 18133, 18178, 18301, 18317
 __regex_compile_special_group_ -
 -w 18115
 __regex_compile_special_group_ -
 ::w 18111
 __regex_compile_special_group_ -
 i:w 18115
 __regex_compile_special_group_ -
 !w 18111
 __regex_compile_u_end: ..
 18320, 18326, 18331
 __regex_compile_u_in_cs: ..
 18337, 18340
 __regex_compile_u_in_cs_aux:n ..
 18347, 18350
 __regex_compile_u_loop:NN ... 18296
 __regex_compile_u_not_cs: ..
 18335, 18356
 __regex_compile_|: 18098

__regex_compute_case_changed_-
 char: 17099, 17113, 17121
 __regex_count:nnN 19472, 19474, 19519
 \c__regex_cs_in_class_mode_int ..
 17467, 18232
 \c__regex_cs_mode_int . 17467, 18230
 \l__regex_cs_name_tl
 17066, 17182, 17187
 \l__regex_current_catcode_int ...
 17141,
 17160, 17168, 17179, 18902, 19033
 \l__regex_current_char_int
 17081, 17087,
 17088, 17095, 17107, 17108, 17123,
 17124, 17125, 17126, 17131, 17161,
 17863, 18879, 18902, 18987, 19030
 __regex_current_cs_to_str:
 17043, 17171, 17182
 \l__regex_current_pos_int
 765, 17046,
 18574, 18872, 18897, 18926, 18928,
 18931, 18952, 18961, 18986, 19002,
 19016, 19018, 19020, 19021, 19022,
 19032, 19035, 19116, 19125, 19570
 \l__regex_current_state_int
 818, 824, 18906, 19040,
 19043, 19044, 19046, 19050, 19053,
 19075, 19080, 19085, 19086, 19094
 \l__regex_current_submatches_-
 prop 18907, 18958, 19055,
 19087, 19088, 19106, 19115, 19127
 \l__regex_default_catcodes_int ..
 778, 17474, 17584,
 17585, 17673, 17966, 18058, 18067
 __regex_disable_submatches: ...
 .. 17184, 18227, 19108, 19513, 19522
 \l__regex_empty_success_bool ...
 .. 18915, 18944, 18950, 19123, 19580
 __regex_escape_u:w 17305
 __regex_escape_a:w 17305
 __regex_escape_break:w 17305
 __regex_escape_e:w 17305
 __regex_escape_f:w 17305
 __regex_escape_n:w 17305
 __regex_escape_r:w 17305
 __regex_escape_t:w 17305
 __regex_escape_x:w 17325
 __regex_escape_\:w 17289
 __regex_escape_break:w 17305
 __regex_escape_escaped:N
 17275, 17299, 17302
 __regex_escape_loop:N
 773, 17281, 17289, 17325,
 17364, 17372, 17373, 17390, 17402
 __regex_escape_raw:N
 .. 774, 17276, 17302, 17314, 17316,
 17318, 17320, 17322, 17324, 17341
 __regex_escape_unescape:N
 17274, 17292, 17302
 __regex_escape_use:nnnn
 772, 783, 17270, 17622, 19193
 __regex_escape_x:N 774, 17363, 17367
 __regex_escape_x_end:w .. 774, 17325
 __regex_escape_x_large:n 17325
 __regex_escape_x_loop:N
 774, 17360, 17376
 __regex_escape_x_loop_error: .. 17376
 __regex_escape_x_loop_error:n ..
 17379, 17391, 17396
 __regex_escape_x_test:N
 774, 17328, 17345
 __regex_escape_x_testii:N ... 17345
 \l__regex_every_match_tl
 18914, 18965, 18969, 18974
 __regex_extract: 839, 19537,
 19543, 19554, 19636, 19677, 19699
 __regex_extract_all:nnN 19486, 19531
 __regex_extract_b:wn 19636
 __regex_extract_e:wn 19636
 __regex_extract_once:nnN
 19484, 19531
 __regex_extract_seq_aux:n
 19599, 19616
 __regex_extract_seq_aux:ww .. 19616
 \l__regex_fresh_thread_bool ...
 819, 824, 18891, 18894,
 18915, 19000, 19062, 19064, 19124
 __regex_get_digits:NTFw
 17493, 17720, 17735
 __regex_get_digits_loop:nw ...
 17496, 17499, 17502
 __regex_get_digits_loop:w ... 17493
 __regex_group:nnN 777,
 794, 18089, 18094, 18390, 18557, 18701
 __regex_group_aux:nnnnN
 812, 18681, 18703, 18711, 18714
 __regex_group_aux:nnnnnN 812
 __regex_group_end_extract_seq:N ..
 19538, 19546, 19585, 19587
 __regex_group_end_replace:N ...
 19693, 19722, 19724
 \l__regex_group_level_int
 17466, 17583,
 17599, 17601, 17603, 18059, 18064
 __regex_group_no_capture:nnN ..
 777, 18112, 18392, 18701
 __regex_group_repeat:nn 18695, 18747

__regex_group_repeat:nnN
 18696, [18787](#)
 __regex_group_repeat:nnnN
 18697, [18818](#)
 __regex_group_repeat_aux:n
 . 813, 814, [18754](#), [18767](#), 18805, 18822
 __regex_group_resetting:nnnN ...
 777, 18114, 18394, [18712](#)
 __regex_group_resetting_-
 loop:nnN [18712](#)
 __regex_group_submatches:nNN ...
 .. 18755, [18760](#), 18790, 18806, 18820
 __regex_hexadecimal_use:N ... 17404
 __regex_hexadecimal_use:NTF ...
 17362, 17371, 17381, [17404](#)
 __regex_if_end_range>NN 17810
 __regex_if_end_range:NNTF ... [17810](#)
 __regex_if_in_class:TF
 17517, 17592, 17659,
 17674, 17794, 17857, 17928, 17943,
 18077, 18100, 18108, 1805, 18181
 __regex_if_in_class_or_catcode:TF
 .. 17537, 17886, 17910, 17919, 18298
 __regex_if_in_cs:TF
 17525, 18238, 19803, 19812
 __regex_if_match:nn
 19463, 19468, [19510](#)
 __regex_if_raw_digit>NN 17505
 __regex_if_raw_digit:NNTF ...
 17495, 17501, [17505](#)
 __regex_if_two_empty_matches:TF
 819, [18915](#), 18951, 18954, 19120
 __regex_if_within_catcode:TF ...
 17549, 17946
 __regex_internal_a_int
 786, 831, [17049](#),
 17720, 17731, 17742, 17752, 17756,
 17765, 17767, 17771, 17774, 17781,
 18663, 18666, 18672, 18677, 18756,
 18771, 18777, 18783, 18792, 18795,
 18799, 18802, 18807, 18810, 18813,
 18828, 18836, 18845, 19279, 19300
 __regex_internal_a_tl ... 772,
 801, 801, 841, 844, [17049](#), 17170,
 17172, 17273, 17287, 18008, 18013,
 18028, 18033, 18038, 18042, 18048,
 18049, 18246, 18255, 18303, 18333,
 18343, 18358, 18382, 18385, 18387,
 18431, 18446, 18465, 18488, 18499,
 18590, 18591, 18592, 18593, 18737,
 18738, 18742, 18744, 19190, 19225,
 19592, 19613, 19683, 19712, 19742
 __regex_internal_b_int
 [17049](#), 17735, 17765, 17768,
 17769, 17771, 17775, 17782, 18772,
 18777, 18782, 18828, 18836, 18845
 __regex_internal_b_tl
 17049, 17279,
 17284, 17335, 17337, 17399, 17401
 __regex_internal_bool
 .. [17049](#), 18007, 18012, 18032, 18041
 __regex_internal_c_int
 .. [17049](#), 18774, 18779, 18780, 18784
 __regex_internal_regex
 782, 794, [17490](#), 17582,
 17611, 18057, 18068, 18248, 18254,
 18545, 19441, 19446, 19451, 19456
 __regex_internal_seq
 [17049](#), 18515, 18516,
 18521, 18528, 18529, 18530, 18532
 __regex_internal_tl
 .. [17049](#), 17278, 17281, 18342, 18346
 __regex_item_caseful_equal:n ...
 777, [17079](#), 17199,
 17200, 17204, 17205, 17206, 17207,
 17208, 17217, 17222, 17240, 17258,
 17586, 18135, 18278, 18353, 18403
 __regex_item_caseful_range:nn ...
 777, [17079](#), 17196,
 17211, 17214, 17215, 17216, 17230,
 17237, 17244, 17246, 17248, 17251,
 17252, 17253, 17254, 17259, 17262,
 17267, 17268, 17587, 18136, 18405
 __regex_item_caseless_equal:n ..
 777, [17093](#), 18119, 18407
 __regex_item_caseless_range:nn ..
 777, [17093](#), 18120, 18409
 __regex_item_catcode: [17138](#)
 __regex_item_catcode:NTF
 . 777, 792, [17138](#), 17667, 17968, 18414
 __regex_item_catcode_reverse:NTF
 778, [17138](#), 17969, 18416
 __regex_item_cs:n
 766, 778, [17177](#), 18254, 18423
 __regex_item_equal:n
 17136, 17586, 17800, 17806,
 17836, 17849, 17850, 18119, 18135
 __regex_item_exact:nn
 778, 801, [17158](#), 18367, 18420
 __regex_item_exact_cs:n
 . 778, 798, [17158](#), 18255, 18365, 18422
 __regex_item_range:nn
 .. 17136, 17587, 17838, 18120, 18136
 __regex_item_reverse:n
 206, 778, 793, [17074](#), 17157,
 17221, 17874, 18032, 18418, 18882
 __regex_last_char_int
 18879, [18902](#), 18987

```

\l__regex_left_state_int . . . . .
    18537, 18555, 18584, 18591, 18605,
    18612, 18619, 18622, 18623, 18625,
    18626, 18652, 18660, 18663, 18688,
    18738, 18740, 18751, 18771, 18791,
    18793, 18821, 18824, 18827, 18830,
    18842, 18855, 18864, 18888, 18893
\l__regex_left_state_seq . . . . .
    . . . . . 18537, 18583, 18590, 18737
\__regex_match:n . . . . . .
    . . . . . 17187, 18921, 19516, 19526,
    19536, 19545, 19569, 19673, 19702
\l__regex_match_count_int . . . .
    . . . . . 838, 839, 19493, 19523, 19524, 19529
\__regex_match_loop: . . . . .
    . . . . . 820, 824, 18964, 18983
\__regex_match_once: . . . . .
    . . . . . 820, 821, 18945, 18948, 18979
\__regex_match_one_active:n . . 18983
\l__regex_match_success_bool . .
    819, 18918, 18957, 18970, 18976, 19122
\l__regex_max_active_int . . . .
    . . . . . 18910, 18959, 18992,
    18996, 19001, 19100, 19101, 19105
\l__regex_max_pos_int . . . .
    . . . . . 827, 17899, 17900, 17907,
    18509, 18574, 18897, 18931, 19002,
    19570, 19575, 19581, 19691, 19720
\l__regex_max_state_int . . . .
    806, 806, 850, 18534, 18552, 18566,
    18607, 18608, 18611, 18613, 18614,
    18673, 18686, 18750, 18770, 18772,
    18780, 18824, 18830, 18838, 18848,
    18926, 18936, 18938, 18942, 20015
\l__regex_min_active_int . 18910,
    18938, 18959, 18992, 18994, 19001
\l__regex_min_pos_int 827, 17897,
    17906, 18507, 18897, 18928, 18940
\l__regex_min_state_int .. 18534,
    18552, 18566, 18936, 18960, 20013
\l__regex_min_submatch_int . . .
    . . . . . 838, 841, 843, 18941,
    18943, 19496, 19596, 19707, 19715
\l__regex_mode_int . . . . .
    17467, 17519, 17527, 17530, 17539,
    17542, 17551, 17559, 17562, 17572,
    17573, 17575, 17576, 17621, 17635,
    17637, 17930, 17933, 17934, 17935,
    17962, 17973, 18155, 18156, 18171,
    18172, 18228, 18229, 18334, 18376
\__regex_mode_quit_c: . . . .
    . . . . . 17570, 17658, 18056
\__regex_msg_repeated:nnN . . .
    . . . . . 18461, 18483, 18487, 18497, 19993
\__regex_multi_match:n . . . . .
    819, 18967, 19524, 19543, 19551, 19699
\c__regex_no_match_regex . . . .
    . . . . . 17057, 17490, 19437
\c__regex_outer_mode_int . . . .
    17467, 17530, 17542, 17551, 17559,
    17573, 17621, 17637, 18334, 18376
\__regex_pop_lr_states: . . . .
    . . . . . 18571, 18581, 18693
\__regex_posix_alnum: . . . . .
    . . . . . 17224
\__regex_posix_alpha: . . . . .
    . . . . . 17224
\__regex_posix_ascii: . . . . .
    . . . . . 17224
\__regex_posix_blank: . . . . .
    . . . . . 17224
\__regex_posix_cntrl: . . . . .
    . . . . . 17224
\__regex_posix_digit: . . . . .
    . . . . . 17224
\__regex_posix_graph: . . . . .
    . . . . . 17224
\__regex_posix_lower: . . . . .
    . . . . . 17224
\__regex_posix_print: . . . . .
    . . . . . 17224
\__regex_posix_punct: . . . . .
    . . . . . 17224
\__regex_posix_space: . . . . .
    . . . . . 17224
\__regex_posix_upper: . . . . .
    . . . . . 17224
\__regex_posix_word: . . . . .
    . . . . . 17224
\__regex_posix_xdigit: . . . . .
    . . . . . 17224
\__regex_prop_-: . . . . .
    . . . . . 789, 17855
\__regex_prop_d: . . . . .
    . . . . . 789, 17195, 17242
\__regex_prop_h: . . . . .
    . . . . . 17195, 17234
\__regex_prop_N: . . . . .
    . . . . . 17195, 17883
\__regex_prop_s: . . . . .
    . . . . . 17195
\__regex_prop_v: . . . . .
    . . . . . 17195
\__regex_prop_w: . . . . .
    . . . . . 17195, 17263, 18880, 18882, 18883
\__regex_push_lr_states: . . . .
    . . . . . 18569, 18581, 18691
\__regex_query_get: . . . .
    . . . . . 18963, 18989, 19028
\__regex_query_range:nn 827, 19139,
    19144, 19163, 19232, 19686, 19719
\__regex_query_range_loop:ww . 19144
\__regex_query_set:nnn . . .
    . . . . . 820, 18927, 18930, 18932, 19013
\__regex_query_submatch:n . . .
    . . . . . 19161, 19269, 19630
\__regex_replace_all:nnN 19490, 19696
\__regex_replace_once:nnN . . .
    . . . . . 19488, 19667
\__regex_replacement:n . . .
    . . . . . 19187, 19672, 19701
\__regex_replacement_aux:n . . 19187
\__regex_replacement_balance_-
    one_match:n . . . . .
        . . . . . 206,
        826, 827, 19135, 19218, 19680, 19710
\__regex_replacement_c:w . . . .
    . . . . . 19309
\__regex_replacement_c_A:w 829, 19375
\__regex_replacement_c_B:w . . 19378

```

__regex_replacement_c_C:w . . . 19387
 __regex_replacement_c_D:w . . . 19389
 __regex_replacement_c_E:w . . . 19392
 __regex_replacement_c_L:w . . . 19401
 __regex_replacement_c_M:w . . . 19404
 __regex_replacement_c_O:w . . . 19407
 __regex_replacement_c_P:w . . . 19410
 __regex_replacement_c_S:w . . . 19416
 __regex_replacement_c_T:w . . . 19424
 __regex_replacement_c_U:w . . . 19427
 __regex_replacement_cat>NNN . . .
 19317, 19347
 \l__regex_replacement_category_-
 seq 19132,
 19212, 19215, 19216, 19245, 19361
 \l__regex_replacement_category_-
 tl 829, 19132,
 19240, 19246, 19249, 19362, 19363
 __regex_replacement_char:nNN . . .
 836, 19370,
 19377, 19384, 19391, 19398, 19403,
 19406, 19409, 19413, 19426, 19429
 \l__regex_replacement_csnames_-
 int 826, 19131, 19206,
 19208, 19210, 19270, 19325, 19330,
 19340, 19342, 19352, 19381, 19395
 __regex_replacement_cu_aux:Nw . . .
 19314, 19323, 19335
 __regex_replacement_do_one_-
 match:n . . . 19137, 19230, 19685, 19718
 __regex_replacement_error>NNN . . .
 19280, 19292,
 19303, 19318, 19321, 19336, 19431
 __regex_replacement_escaped:N . . .
 19202, 19254
 __regex_replacement_exp_not:N . . .
 832, 19143, 19314
 __regex_replacement_g:w . . . 19276
 __regex_replacement_g_digits>NNN . . .
 19276
 __regex_replacement_normal:n . . .
 19198, 19203, 19238, 19261, 19278,
 19284, 19311, 19334, 19344, 19359
 __regex_replacement_put_-
 submatch:n . . . 19259, 19266, 19299
 __regex_replacement_rbrace:N . . .
 19196, 19298, 19338
 __regex_replacement_u:w . . . 19332
 __regex_return:
 837, 19464, 19469, 19480, 19482, 19502
 \l__regex_right_state_int
 18537, 18558, 18572, 18586,
 18593, 18606, 18612, 18613, 18652,
 18659, 18665, 18678, 18686, 18688,
 18740, 18744, 18756, 18770, 18779,
 18791, 18795, 18799, 18802, 18807,
 18810, 18813, 18821, 18835, 18838,
 18841, 18844, 18848, 18864, 18893
 \l__regex_right_state_seq
 18537, 18558, 18572, 18586,
 18593, 18606, 18612, 18613, 18652,
 18659, 18665, 18678, 18686, 18688,
 18740, 18744, 18756, 18770, 18779,
 18791, 18795, 18799, 18802, 18807,
 18810, 18813, 18821, 18835, 18838,
 18841, 18844, 18848, 18864, 18893
 \l__regex_saved_success_bool
 819, 17186, 17191, 18918
 __regex_show:Nn
 837, 18380, 19456, 19460
 __regex_show_anchor_to_str:N
 18402, 18502
 __regex_show_class:NnnnN
 18396, 18463
 __regex_show_group_aux:nnnnN
 18391, 18393, 18395, 18454
 __regex_show_item_catcode:NnTF
 18415, 18417, 18513
 __regex_show_item_exact_cs:n
 18422, 18526
 \l__regex_show_lines_int
 17492, 18435, 18466, 18470
 __regex_show_one:n
 18386, 18398,
 18400, 18404, 18406, 18408, 18411,
 18421, 18425, 18433, 18449, 18456,
 18460, 18475, 18486, 18494, 18531
 __regex_show_pop: 18443, 18459
 \l__regex_show_prefix_seq
 17491, 18385,
 18387, 18426, 18439, 18444, 18446
 __regex_show_push:n
 18427, 18443, 18457, 18467
 __regex_show_scope:nn
 18419, 18424, 18443, 18518
 __regex_single_match:
 819, 17183, 18967, 19514, 19534, 19670
 __regex_split:nnN 19492, 19548
 __regex_standard_escapechar:
 17012, 17277, 17620, 18550
 \l__regex_start_pos_int
 17898, 18508,
 18897, 18952, 18956, 18962, 19553,
 19565, 19578, 19581, 19660, 19720
 \g__regex_state_active_intarray
 763, 818, 818, 819, 820, 18912,
 18937, 19042, 19045, 19052, 19079
 \l__regex_step_int
 763, 18909, 18939, 18985,
 19043, 19046, 19054, 19068, 19070
 __regex_store_state:n
 18960, 19093, 19096
 __regex_store_submatches:
 19096, 19110

__regex_submatch_balance:n
 .. 19136, 19167, 19222, 19272, 19621
 \g__regex_submatch_begin_-_intarray
 764, 826, 19141, 19164, 19179, 19183, 19234, 19499,
 19560, 19563, 19576, 19642, 19664
 \g__regex_submatch_end_intarray
 764, 19165, 19171, 19175, 19499,
 19557, 19573, 19644, 19666, 19688
 \l__regex_submatch_int
 764, 838, 840, 841, 843,
 18943, 19496, 19572, 19574, 19577,
 19579, 19582, 19598, 19639, 19643,
 19645, 19647, 19648, 19709, 19717
 \g__regex_submatch_prev_intarray
 764, 838, 842, 19140, 19233,
 19499, 19555, 19571, 19646, 19659
 \g__regex_success_bool 819,
 17186, 17188, 17191, 18918, 18934,
 18970, 18977, 19504, 19638, 19674
 \l__regex_success_pos_int
 .. 18897, 18940, 18956, 19125, 19553
 \l__regex_success_submatches_-_prop .. 818, 842, 18907, 19126, 19650
 __regex_tests_action_cost:n
 18630, 18651, 18660, 18678
 \g__regex_thread_state_intarray
 763,
 817, 819, 819, 825, 18912, 19010, 19099
 __regex_tmp:w
 17031, 17033, 17037, 17039, 17048,
 17867, 17877, 17878, 17879, 17880,
 17881, 17892, 17897, 17898, 17899,
 17900, 17901, 17906, 17907, 19475,
 19484, 19486, 19488, 19490, 19492
 __regex_toks_clear:N . 17015, 18611
 __regex_toks_memcpy:NNn 17020, 18781
 __regex_toks_put_left:Nn
 17029, 18596, 18763, 18764
 __regex_toks_put_right:Nn
 765, 17029, 18555, 18558,
 18572, 18598, 18619, 18855, 18888
 __regex_toks_set:Nn
 17015, 19021, 19105
 __regex_toks_use:w
 17014, 19011, 19044, 19157, 20018
 __regex_trace_states:n
 18560, 18578, 20009
 __regex_use_state:
 19037, 19056, 19082
 __regex_use_state_and_submatches:nn
 821, 19009, 19048
 \l__regex_zeroth_submatch_int
 838, 842, 19496,
 19556, 19558, 19561, 19564, 19639,
 19657, 19660, 19681, 19685, 19689
 \regular_expression 205
 \relax 14, 21, 39, 43, 49,
 90, 92, 98, 123, 146, 167, 181, 212,
 213, 214, 215, 216, 217, 218, 219,
 220, 221, 222, 225, 226, 227, 228,
 229, 230, 231, 232, 233, 234, 235, 520
 \relpenalty 521
 \RequirePackage 149
 reverse commands:
 \reverse_if:N
 21, 21, 353, 381, 381, 382,
 586, 1297, 3853, 3993, 4834, 4836,
 4838, 4840, 9340, 9345, 9349, 9351,
 11441, 14648, 15316, 15339, 17087,
 17088, 17107, 17108, 17115, 17116
 \right 522
 right commands:
 \c_right_brace_str
 .. 55, 3927, 17389, 17728, 17749,
 17763, 18236, 18240, 18319, 19195
 \rightghost 948
 \righthyphenmin 523
 \rightmarginkern 785
 \rightskip 524
 \romannumeral 525
 round 187
 round+ 11100
 round- 11100
 round0 11100
 \rpcode 786
 \rule 21194, 21249

S

\saveboxresource 973
 \savecatcodetable 925
 \saveimageresource 974
 \savepos 972
 \savinghyphcodes 656
 \savingvdiscards 657
 scan commands:
 \scan_align_safe_stop: 21537
 \scan_stop: 9,
 9, 67, 87, 88, 118, 118, 118, 119,
 252, 266, 286, 286, 286, 289, 301,
 301, 319, 319, 332, 353, 353, 381,
 386, 405, 445, 445, 451, 454, 455,
 463, 463, 482, 583, 586, 586, 587,
 588, 591, 778, 778, 798, 937, 1324,
 1493, 1625, 1643, 1653, 1671, 2101,
 2334, 2375, 2417, 2713, 2737, 3854,
 3949, 3962, 3967, 3968, 3969, 4032,
 4383, 5009, 5426, 5427, 5429, 5431,

5435, 5667, 6876, 6950, 7162, 7276,
 7285, 7294, 8634, 8636, 8694, 8696,
 8810, 8812, 8848, 8850, 9514, 9525,
 9530, 9556, 9562, 9565, 9612, 9623,
 9628, 9633, 9686, 9710, 9747, 9751,
 9752, 9769, 9770, 11439, 11443,
 11606, 11623, 11923, 11970, 11971,
 12183, 12213, 12817, 16136, 16568,
 16582, 16655, 16681, 16685, 16691,
 16693, 16704, 16735, 16737, 17171,
 17172, 17503, 18262, 18528, 19032,
 19035, 19058, 19372, 19421, 19589,
 20157, 21190, 21245, 21558, 22119,
 22120, 22282, 22285, 22306, 23400,
 23402, 23404, 23512, 23518, 23606

scan internal commands:

- \g__scan_marks_t1 5657, 5660, 5666
- _scan_new:N 88, 88, 5658, 5670, 5672, 7299, 10630, 10636, 10637, 10638, 10639, 10640, 10641, 10642, 16521
- \scantextokens 926
- \scantokens 658
- \scriptfont 526
- \scriptscriptfont 527
- \scriptscriptstyle 528
- \scriptspace 529
- \scriptstyle 530
- \scrollmode 531
- sec 188
- secd 188

seq commands:

- \c_empty_seq 66, 360, 4090, 4094, 4098, 4101, 4283, 4320, 4328
- \l_foo_seq 201, 201
- \seq_clear:N 57, 57, 66, 4097, 4104, 4227, 7920, 7983, 18426, 19216
- \seq_clear:N 57
- \seq_clear_new:N 57, 57, 4103
- \seq_concat:NNN 58, 58, 66, 66, 4180, 8516
- \seq_count:N 59, 63, 63, 65, 81, 4446, 4518, 4532, 19215, 22111
- \seq_elt:w 360, 360
- \seq_elt_end: 360, 360
- \seq_gclear:N 57, 4097, 4107
- \seq_gclear_new:N 57, 4103
- \seq_gconcat:NNN 58, 4180
- \seq_get:NN 64, 64, 4585, 18737, 18742
- \seq_get:NNTF 64, 64, 4591
- \seq_get_left:NN 58, 58, 4336, 4585, 4586, 4591, 4592
- \seq_get_left:NNTF 60, 60, 4403, 4407, 4408
- \seq_get_right:NN 59, 59, 4361
- \seq_get_right:NNTF 60, 60, 4403, 4410, 4411
- \seq_gpop:NN 64, 64, 4585, 8578
- \seq_gpop:NNTF 65, 65, 4591, 8678, 8832
- \seq_gpop_left:NN 59, 59, 4347, 4589, 4590, 4595, 4596
- \seq_gpop_left:NNTF 60, 60, 4413, 4424, 4425
- \seq_gpop_right:NN 59, 59, 4376
- \seq_gpop_right:NNTF 60, 60, 4413, 4430, 4431
- \seq_gpush:Nn 24, 65, 4565, 8575, 8705, 8859
- \seq_gput_left:Nn 58, 4190, 4575, 4576, 4577, 4578, 4579, 4580, 4581, 4582, 4583, 4584
- \seq_gput_right:Nn 58, 4211, 8443, 8568, 8573, 8616
- \seq_gremove_all:Nn 61, 4237
- \seq_gremove_duplicates:N 61, 4221
- \seq_greverse:N 61, 4263
- \seq_gset_eq:NN 57, 4101, 4109, 4224
- \seq_gset_filter:NNn 230, 22088
- \seq_gset_from_clist:NN 57, 4117
- \seq_gset_from_clist:Nn 57, 4117
- \seq_gset_map:NNn 230, 22098
- \seq_gset_split:Nnn 58, 4143, 8623, 8795
- \seq_gsort:Nn 61, 4281, 16211
- \seq_if_empty:NTF 61, 61, 4281, 4290, 4291, 4600, 6038, 19212, 22110
- \seq_if_empty_p:N 61, 61, 4281
- \seq_if_exist:NTF 58, 58, 4104, 4107, 4186, 4530, 4600
- \seq_if_exist_p:N 58, 58, 4186
- \seq_if_in:NNTF 61, 61, 65, 66, 66, 4230, 4293, 4312, 4313, 4314, 4315, 8586, 8704, 8858
- \seq_item:Nn 59, 59, 907, 4433, 8001, 8002, 8007, 22111
- \seq_log:N 67, 67, 4604
- \seq_map_break: 62, 62, 230, 230, 4457, 4467, 4468, 4503, 4514, 8526, 10388
- \seq_map_break:n 63, 371, 4457, 7940, 7954, 16214, 16221
- \seq_map_function:NN 4, 61, 61, 484, 485, 4461, 4523, 4601, 6044, 8005, 18439, 18521
- \seq_map_inline:Nn 61, 62, 62, 66, 66, 66, 66, 907, 4228, 4499, 7935, 8458, 8520, 8609, 10381, 16214, 16221
- \seq_map_variable:NNn 62, 62, 4506

```

\seq_mapthread_function:NNN . . .
    ..... 230, 230, 22065
\seq_new:N 4, 57, 57, 57, 4091, 4104,
    4107, 4220, 4607, 4608, 4609, 4610,
    6663, 6666, 7892, 7893, 8437, 8438,
    8448, 8450, 8453, 8621, 8791, 9808,
    17055, 17491, 18539, 18540, 19133
\seq_pop:NN . . .
    ... 64, 64, 4585, 18590, 18592, 19245
\seq_pop:NNTF . . . 65, 65, 4591
\seq_pop_left:NN . . .
    . 59, 59, 4347, 4587, 4588, 4593, 4594
\seq_pop_left:NNTF . . .
    ..... 60, 60, 4413, 4421, 4422
\seq_pop_right:NN . . .
    ..... 59, 59, 4376, 18385, 18446
\seq_pop_right:NNTF . . .
    ..... 60, 60, 4413, 4427, 4428
\seq_push:Nn . . . 65,
    65, 4565, 4572, 18583, 18585, 19361
\seq_put_left:Nn . . . 58, 58,
    4190, 4565, 4566, 4567, 4568, 4569,
    4570, 4571, 4572, 4573, 4574, 7930
\seq_put_right:Nn . . .
    ..... 58, 58, 65, 66, 66, 4211,
    4231, 7991, 8587, 8602, 18387, 18444
\seq_rand_item:N . . . 230, 230, 22108
\seq_remove_all:Nn . . . 58,
    61, 61, 65, 65, 66, 66, 66, 4237, 8592
\seq_remove_duplicates:N . . .
    ..... 61, 61, 65, 66, 4221, 8607
\seq_reverse:N . . . 61, 61, 365, 4263
\seq_set_eq:NN . 57, 57, 66, 66, 66,
    66, 4098, 4109, 4222, 8513, 8532, 8596
\seq_set_filter:NNn . . .
    ..... 230, 230, 907, 18516, 22088
\seq_set_from_clist:NN . 57, 57, 4117
\seq_set_from_clist:Nn 57, 4117, 10310
\seq_set_map:NNn 230, 230, 18529, 22098
\seq_set_split:Nnn . . . 58, 58, 58,
    4143, 6664, 6667, 8515, 18515, 18528
\seq_show:N . . . 67, 67, 375, 4597, 4605
\seq_sort:Nn . . . 61, 61, 4281, 16211
\seq_use:Nn . . . 64, 64, 4528, 18532
\seq_use:Nnnn . . . 63, 63, 4528
\g_tmpa_seq . . . 67, 4607
\l_tmpa_seq . . . 67, 4607
\g_tmpb_seq . . . 67, 4607
\l_tmpb_seq . . . 67, 4607
seq internal commands:
    \seq_count:n . . . 4518
    \seq_get_left:wnw . . . 4336
    \seq_get_right_loop:nn . . . 368, 4361
    \seq_if_in: . . . 4293
\l_seq_internal_a_tl . . .
    ..... 362, 4087, 4151, 4155, 4161,
    4166, 4168, 4252, 4257, 4297, 4301
\l_seq_internal_b_tl . . .
    ..... 4087, 4248, 4252, 4300, 4301
\__seq_item:n . . . 67, 67, 67, 67, 359,
    359, 363, 366, 366, 367, 368, 369,
    371, 371, 371, 372, 372, 373, 373,
    733, 734, 734, 841, 906, 907, 4082,
    4194, 4202, 4212, 4214, 4219, 4269,
    4270, 4272, 4277, 4298, 4341, 4344,
    4354, 4382, 4383, 4394, 4480, 4485,
    4491, 4495, 4539, 4554, 4557, 4560,
    16215, 16222, 19589, 19618, 22104
\__seq_item:nn . . . 4433
\__seq_item:nnn . . . 4433
\__seq_item:wNn . . . 4433
\__seq_map_function:NNn . . . 4461
\__seq_mapthread_function:Nnnwnm
    ..... 22065
\__seq_mapthread_function:wNN . . . 22065
\__seq_mapthread_function:wNw . . . 22065
\__seq_pop:NNNN . . .
    ..... 4318, 4348, 4350, 4377, 4379
\__seq_pop_item_def: . . . 67, 67, 67,
    4259, 4477, 4503, 4514, 22096, 22106
\__seq_pop_left:NNNN . . . 4347, 4414, 4416
\__seq_pop_left:wnwNNN . . . 4347
\__seq_pop_right:NNN . . .
    ..... 364, 4376, 4418, 4420
\__seq_pop_right_loop:nn . . . 4376
\__seq_pop_TF:NNNN . . . 369, 4318,
    4404, 4406, 4414, 4416, 4418, 4420
\__seq_push_item_def: . . . 4477
\__seq_push_item_def:n . . .
    ..... 67, 67, 67, 67,
    4243, 4477, 4501, 4508, 22094, 22104
\__seq_put_left_aux:w . . . 363, 4190
\__seq_remove_all_aux:NNn . . . 4237
\__seq_remove_duplicates:NN . . . 4221
\l_seq_remove_seq . . .
    ..... 4220, 4227, 4230, 4231, 4233
\__seq_reverse:NN . . . 4263
\__seq_reverse_item:nw . . . 365, 365
\__seq_reverse_item:nwn . . . 4263
\__seq_set_filter:NNNN . . . 22088
\__seq_set_map:NNNN . . . 22098
\__seq_set_split:NNnn . . . 4143
\__seq_set_split_auxi:w 362, 362, 4143
\__seq_set_split_auxii:w . . . 362, 4143
\__seq_set_split_end: 362, 362, 4143
\__seq_tmp:w . . .
    ..... 4089, 4269, 4272, 4382, 4394
\__seq_use:NNnNnn . . . 4528

```

__seq_use:nwwn 4528
 __seq_use:nwwwnwn 4528
 __seq_use_setup:w 4528
 __seq_wrap_item:n
 362, 907, 4120, 4125, 4130,
 4135, 4152, 4177, 4219, 4255, 22094
 \setbox 532
 \setfontid 927
 \setlanguage 533
 \setrandomseed 975
 \sfcode 183, 534
 \sffamily 21183
 \shapemode 928
 \shellescape 858
 \shipout 535
 \ShortText 75, 116, 133
 \show 536
 \showbox 537
 \showboxbreadth 538
 \showboxdepth 539
 \showgroups 659
 \showifs 660
 \showlists 540
 \showmode 1149
 \showthe 541
 \ShowTokens 195
 \showtokens 661
 sign 187
 sin 188
 sind 188
 \sjis 1150
 \skewchar 542
 \skip 543, 7026
 skip commands:
 \c_max_skip 157, 9576
 \skip_add:Nn 155, 155, 9524
 \skip_const:Nn
 155, 155, 9493, 9576, 9577
 \skip_eval:n 156,
 156, 156, 156, 9537, 9555, 9571
 \skip_gadd:Nn 155, 9524
 \skip_gset:N 166, 10202
 \skip_gset:Nn 155, 9496, 9513
 \skip_gset_eq:NN 155, 9518
 \skip_gsub:Nn 156, 9524
 \skip_gzero:N 155, 9499, 9506
 \skip_gzero_new:N 155, 9503
 \skip_horizontal:N 157, 157, 157, 9560
 \skip_horizontal:n 157,
 157, 9560, 23541, 23692, 24097, 24490
 \skip_if_eq:nnTF 156, 9534
 \skip_if_eq_p:nn 156, 156, 9534
 \skip_if_exist:NTF
 155, 155, 9504, 9506, 9509
 \skip_if_exist_p:N 155, 155, 9509
 \skip_if_finite:nTF
 156, 156, 9544, 22117
 \skip_if_finite_p:n 156, 156, 9544
 \skip_log:N 157, 157, 9572
 \skip_log:n 157, 157, 9572
 \skip_new:N 155, 155, 155, 9485, 9495,
 9504, 9506, 9578, 9579, 9580, 9581
 .skip_set:N 166, 10202
 \skip_set:Nn 155, 155, 9513
 \skip_set_eq>NN 155, 155, 9518
 \skip_show:N 156, 156, 9568
 \skip_show:n 157, 157, 521, 9570, 9575
 \skip_split_finite_else_action:nnNN
 230, 230, 22115
 \skip_sub:Nn 156, 156, 9524
 \skip_use:N
 156, 156, 156, 9549, 9556, 9557
 \skip_vertical:N 158, 158, 158, 9560
 \skip_vertical:n 158, 158, 9560
 \skip_zero:N 155, 155, 158, 9499, 9504
 skip_zero:N 155
 \skip_zero_new:N 155, 155, 9503
 \g_tmpa_skip 157, 9578
 \l_tmpa_skip 157, 9578
 \g_tmpb_skip 157, 9578
 \l_tmpb_skip 157, 9578
 \c_zero_skip
 157, 9499, 9576, 22123, 22124
 skip internal commands:
 __skip_if_finite:wwNw 9544
 \skipdef 544
 sort commands:
 \sort_ordered: 16500, 21555
 \sort_return_same: 194, 194,
 736, 16317, 16505, 16506, 16507, 21555
 \sort_return_swapped: 194, 194,
 736, 16317, 16514, 16515, 16516, 21556
 \sort_reversed: 16500, 21556
 sort internal commands:
 __sort:nnNnn 738, 738, 738, 738, 739
 \l__sort_A_int
 736, 16100, 16105, 16112,
 16115, 16124, 16282, 16287, 16290,
 16310, 16325, 16344, 16346, 16347
 \l__sort_B_int
 736, 736, 16100, 16287, 16291,
 16299, 16301, 16302, 16334, 16335,
 16344, 16345, 16354, 16355, 16357
 \l__sort_begin_int 730,
 736, 736, 16098, 16279, 16347, 16357
 \l__sort_block_int
 729, 730, 735, 16097, 16107,

16112, 16116, 16119, 16124, 16125,
 16203, 16270, 16273, 16280, 16283
 $\backslash\text{--sort_C_int}$
 736, 736, 16100, 16288,
 16292, 16299, 16300, 16311, 16326,
 16334, 16336, 16337, 16354, 16356
 --sort_clist:NNn 16239
 --sort_compare:nn . 737, 16202, 16309
 $\text{--sort_compute_range:}$
 729, 730, 731, 16129, 16190
 $\text{--sort_copy_block:}$ 735, 16289, 16297
 $\text{--sort_disable_toksdef:}$ 16189, 16449
 $\text{--sort_disabled_toksdef:n}$... 16449
 --sort_end_int 730, 735, 736, 736,
 16098, 16271, 16279, 16280, 16281,
 16282, 16283, 16284, 16285, 16302
 --sort_error: ... 16443, 16456, 16474
 --sort_i:nnnnNn 739
 --sort_length_int
 729, 730, 16092, 16200, 16270
 --sort_level: 742, 16204, 16268, 16447
 --sort_loop:wNn 738, 739, 739, 739, 739
 $\text{--sort_main:NNNnNn}$... 733, 16186,
 16213, 16220, 16227, 16234, 16249
 --sort_max_int 729,
 730, 730, 16092, 16109, 16183, 16194
 $\text{c_sort_max_length_int}$ 16129
 $\text{--sort_merge_blocks:}$
 16272, 16277, 16446
 $\text{--sort_merge_blocks_aux:}$
 735, 16293, 16307, 16340, 16350, 16445
 $\text{--sort_merge_blocks_end:}$
 737, 16348, 16352
 --sort_min_int ... 729, 730, 732,
 734, 16092, 16106, 16114, 16132,
 16148, 16156, 16169, 16181, 16191,
 16201, 16259, 16271, 16472, 16473
 $\text{--sort_quick_cleanup:w}$ 16362
 $\text{--sort_quick_end:nnTFNn}$
 741, 741, 16382, 16417
 $\text{--sort_quick_only_i:NnnnnNn}$. 16387
 $\text{--sort_quick_only_i_end:nnnw}$..
 16396, 16417
 $\text{--sort_quick_only_ii:NnnnnNn}$. 16387
 $\text{--sort_quick_only_ii_end:nnnw}$..
 16402, 16417
 $\text{--sort_quick_prepare:Nnnn}$... 16362
 $\text{--sort_quick_prepare_end:NNNnw}$..
 16362
 $\text{--sort_quick_single_end:nnnw}$..
 16390, 16417
 $\text{--sort_quick_split:NNn}$
 ... 739, 739, 739, 739, 740, 16382,
 16387, 16422, 16429, 16435, 16437

 $\text{_--sort_quick_split_end:nnnw}$..
 16408, 16414, 16417
 $\text{_--sort_quick_split_i:NnnnnNn}$..
 739, 16387
 $\text{_--sort_quick_split_ii:NnnnnNn}$ 16387
 $\text{_--sort_redefine_compute_range:}$..
 16129
 $\text{_--sort_return_mark:N}$
 16313, 16314, 16317
 $\text{_--sort_return_none_error:}$..
 736, 16315, 16317
 $\text{_--sort_return_same:}$
 16318, 16327, 16332
 $\text{_--sort_return_swapped:}$ 16320, 16342
 $\text{_--sort_return_two_error:w}$... 16317
 $\text{_--sort_shrink_range:}$... 730, 731,
 731, 16103, 16134, 16150, 16158, 16171
 $\text{_--sort_shrink_range_loop:}$... 16103
 _--sort_toks:NN 734, 16215,
 16222, 16229, 16236, 16253, 16258
 _--sort_toks:NNw 16258
 $\text{_--sort_too_long_error:NNw}$..
 16195, 16467
 l--sort_top_int ... 729, 729, 732,
 734, 736, 736, 16092, 16191, 16194,
 16197, 16198, 16201, 16262, 16281,
 16284, 16285, 16288, 16337, 16473
 $\text{_l_--sort_true_max_int}$ 729,
 730, 16092, 16106, 16119, 16133,
 16149, 16157, 16170, 16182, 16472
sp 191
spac commands:
 \spac_directions_normal_body_dir 1286
 \spac_directions_normal_page_dir 1287

\space 55
\spacefactor 545
\spaceskip 546
\span 547
\special 548
\splitbotmark 549
\splitbotmarks 662
\splitdiscards 663
\splitfirstmark 550
\splitfirstmarks 664
\splitmaxdepth 551
\splittopskip 552
sqrt 189
sr commands:
 \sr_if_empty_p:N 49
\ss 23100
\ss 23100

str commands:

- \c_ampersand_str 55, 3927
- \c_atsign_str 55, 3927
- \c_backslash_str 55, 3927, 17295, 17846
- \c_circumflex_str 55, 3927
- \c_colon_str 55, 3927, 6939, 7044, 7050, 9903
- \c_dollar_str 55, 3927
- \c_hash_str 55, 3927, 3993, 24475, 24637, 24644, 24684
- \c_percent_str .. 55, 3927, 24786, 24787, 24788, 24799, 24800, 24801
- \str_case:nn 49, 49, 3535, 8282, 11124, 17994
- \str_case:nn(TF) 383
- \str_case:nnn 21538, 21539
- \str_case:nnTF 49, 49, 3535, 3540, 3545, 3558, 3559, 10053, 18505, 21538, 21539, 22708
- str_case:nnTF 514
- \str_case_x:nn 50, 3535
- \str_case_x:nnn 21540
- \str_case_x:nnTF 50, 50, 3535, 3572, 3577, 17726, 21540
- \str_clear:N 48, 48, 3428
- str_clear:N 48
- \str_clear_new:N 48, 48, 3428
- \str_const:Nn 47, 47, 3451, 3927, 3928, 3929, 3930, 3931, 3932, 3933, 3934, 3935, 3936, 3937, 3938, 21413, 21417, 21439, 21447, 21459, 21466, 21475, 21488, 21499
- \str_count:N 50, 50, 3795, 8909, 8977, 16925
- \str_count:n 50, 50, 50, 56, 3795, 16919
- \str_count_ignore_spaces:n 50, 50, 352, 3795, 16551
- \str_count_spaces:N 50, 3775
- \str_count_spaces:n 50, 50, 352, 3775, 3801
- \str_fold_case:n 53, 53, 54, 54, 54, 54, 232, 3863, 11592
- \str_gclear:N 48, 3428
- \str_gclear_new:N 3428
- \str_gput_left:Nn 48, 3451
- \str_gput_right:Nn 48, 3451
- \str_gset:Nn 48, 3451
- \str_gset_eq>NN 48, 3428
- \str_head:N 51, 51, 353, 3833
- \str_head:n 51, 51, 51, 321, 337, 353, 353, 2786, 3263, 3310, 3833
- \str_head_ignore_spaces:n 51, 51, 3833
- \str_if_empty:NTF 49, 49, 3472
- \str_if_empty_p:N 49, 3472
- \str_if_eq:NN 345
- \str_if_eq:nn 122, 127
- \str_if_eq:NNTF 49, 49, 3526, 3531, 3532
- \str_if_eq:nnTF 49, 49, 49, 50, 124, 229, 344, 364, 3506, 3515, 3516, 3517, 3518, 3563, 4245, 7775, 7951, 9857, 9875, 10385, 10613, 11488, 17705, 17796, 17974, 18005, 18025, 18082, 18117, 18132, 18178, 19359, 22525, 22543, 22553, 22566
- \str_if_eq_p:NN 49, 49, 3526
- \str_if_eq_p:nn 49, 49, 3506
- \str_if_eq_x:nn 461, 461
- \str_if_eq_x:nnTF 49, 49, 55, 1931, 3506, 3591, 7399, 7490, 7994, 9947, 17347, 17369, 17378, 17748, 17762, 18301, 19278, 19334, 19883, 19995
- \str_if_eq_x_p:nn 49, 49, 3506
- \str_if_exist:NTF 48, 48, 3472
- \str_if_exist_p:N 48, 48, 3472
- \str_item:Nn 51, 51, 3641
- \str_item:nn 51, 51, 51, 348, 352, 3641
- \str_item_ignore_spaces:nn 51, 51, 348, 3641
- \str_lower_case:n .. 53, 53, 232, 3863
- \str_new:N 47, 48, 3428, 3939, 3940, 3941, 3942
- \str_put_left:Nn 48, 48, 3451
- \str_put_right:Nn 48, 48, 3451
- \str_range:Nnn 52, 3699
- \str_range:nnn 52, 52, 56, 352, 3699, 16922
- \str_range_ignore_spaces:nnn 52, 3699
- \str_set:Nn 48, 48, 3451
- \str_set_eq:NN 48, 48, 3428
- \str_show:N 54, 54, 3943
- \str_show:n 54, 3943
- \str_tail:N 51, 51, 3848
- \str_tail:n 51, 51, 51, 750, 3848
- \str_tail_ignore_spaces:n 51, 51, 3848
- \str_upper_case:n .. 53, 53, 232, 3863
- \str_use:N 50, 50, 3428
- \c_tilde_str 55, 3927
- \g_tmtpa_str 55, 3939
- \l_tmtpa_str 55, 3939
- \g_tmtpb_str 55, 3939
- \l_tmtpb_str 55, 3939
- \c_underscore_str 55, 3927

str internal commands:

- __str_analysis_map_inline:nn .. 745
- __str_case:nnTF 3535
- __str_case:nw 3535
- __str_case_end:nw 3535
- __str_case_x:nnTF 3535

```

\__str_case_x:nw ..... 3535
\__str_change_case:nn ..... 3863
\__str_change_case_aux:nn ..... 3863
\__str_change_case_char:nN ... 3863
\__str_change_case_char_aux:nN ...
    ..... 3905, 3910, 3924
\__str_change_case_end:nw ... 3863
\__str_change_case_end:wn 3882, 3899
\__str_change_case_loop:nw ... 3863
\__str_change_case_output:nw . 3863
\__str_change_case_result:n .. 3863
\__str_change_case_space:n ... 3863
\__str_collect_delimit_by_q_-
    stop:w ..... 3727, 3750
\__str_collect_end:nnnnnnnnw ...
    ..... 351, 3750
\__str_collect_end:wn ..... 3750
\__str_collect_loop:wn ..... 3750
\__str_collect_loop:wnNNNNNNN . 3750
\__str_count:n .....
    ..... 56, 56, 352, 3657, 3714, 3795
\__str_count_aux:n ..... 3795
\__str_count_loop>NNNNNNNNN .. 3795
\__str_count_spaces_loop:w ... 3775
\__str_escape_x:n ..... 3476
\__str_head:w .... 353, 353, 353, 3833
\__str_if_eq_x:nn .....
    .. 55, 55, 3476, 3500, 3509, 3523,
    3528, 3996, 4020, 4031, 4039, 4051,
    6942, 9537, 11973, 11987, 12185, 14636
\__str_if_eq_x_return:nn .....
    ..... 55, 55, 446, 3498, 7000
\__str_item:nn ..... 348, 348, 3641
\__str_item:w ..... 348, 3641
\__str_lookup_fold:N ..... 3863
\__str_lookup_lower:N ..... 3863
\__str_lookup_upper:N ..... 3863
\__str_range:nnn ..... 56, 56, 3699
\__str_range:nnw ..... 3699
\__str_range:w ..... 3699
\__str_range_normalize:nn .....
    ..... 3722, 3723, 3731
\__str_skip_end>NNNNNNNN .. .
    ..... 349, 349, 3681
\__str_skip_end:w ..... 3681
\__str_skip_exp_end:w .....
    .. 349, 351, 3668, 3677, 3681, 3729
\__str_skip_loop:wNNNNNNNN .. 3681
\__str_tail_auxi:w ..... 3848
\__str_tail_auxii:w ..... 353, 3848
\__str_tmp:n .....
    .. 3429, 3435, 3438, 3452, 3459, 3462
\__str_to_other:n ..... 55, 55,
    56, 56, 56, 347, 352, 3596, 3648, 3706
\__str_to_other_end:w ..... 347, 3596
\__str_to_other_fast:n .....
    .. 56, 56, 3619, 8908, 8989, 17278, 18343
\__str_to_other_fast_end:w ... 3619
\__str_to_other_fast_loop:w ... 3619
\__str_to_other_loop:w ..... 347, 3596
\strcmp ..... 40
]string ..... 553
\suppressfontnotfounderror ..... 805
\suppressifcsnameerror ..... 929
\suppresslongerror ..... 930
\suppressmathparerror ..... 931
\suppressoutererror ..... 932
\synctex ..... 787
sys commands:
\c_sys_day_int ..... 223, 21419
\c_sys_engine_str ..... 223, 21426
\sys_gset_rand_seed:n 231, 231, 22139
\c_sys_hour_int ..... 223, 21419
\sys_if_engine_luatex:TF .....
    ..... 223, 238, 21426,
    21435, 21436, 21526, 21527, 21528,
    22143, 22178, 22180, 22199, 23415
\sys_if_engine_luatex_p: .....
    ..... 223, 21426, 21525
\sys_if_engine_pdftex:TF .....
    ..... 223, 223, 21426,
    21443, 21444, 21530, 21531, 21532
\sys_if_engine_pdftex_p: .....
    ..... 223, 21426, 21529, 22913
\sys_if_engine_ptex:TF .....
    ..... 223, 21426, 21462, 21463
\sys_if_engine_ptex_p: ..... 223, 21426
\sys_if_engine_uptex:TF .....
    ..... 223, 21426, 21455, 21456
\sys_if_engine_uptex_p: .....
    ..... 223, 21426, 22914
\sys_if_engine_xetex:TF .....
    ..... 4, 223, 21426,
    21471, 21472, 21544, 21545, 21546
\sys_if_engine_xetex_p: .....
    ..... 223, 21426, 21543
\sys_if_output_dvi:TF ..... 224,
    224, 21477, 21480, 21481, 21491, 21492
\sys_if_output_dvi_p: ... 224, 21477
\sys_if_output_pdf:TF .....
    ..... 224, 21477, 21484, 21485, 21495, 21496
\sys_if_output_pdf_p: ... 224, 21477
\sys_if_rand_exist:TF 231, 231, 22129
\sys_if_rand_exist_p: 231, 231, 22129
\sys_if_shell: ..... 231, 909
\sys_if_shell:TF ... 231, 231, 22154
\sys_if_shell_p: ... 231, 231, 22154
\sys_if_shell_restricted: ... 22170

```

\sys_if_shell_restricted:TF	231, 231	\box	209, 764
\sys_if_shell_restricted_p:	231, 231	\char	120
\sys_if_shell_unrestricted:TF	...	\chardef	378
.....	231, 231, 22162, 22170	\copy	209
\sys_if_shell_unrestricted_p:	...	\count	120, 731, 731, 731, 764
.....	231, 231, 22162, 22170	\cr	418
\c_sys_jobname_str	\csname	16
.....	138, 223, 21409, 21516	\csstring	285
\c_sys_minute_int	\current@color	...
.....	223, 21419	23587, 23725, 23816, 24136, 24531	
\c_sys_month_int	\currentgrouplevel	297, 484
.....	223, 21419	\currentgroupstype	297, 484
\c_sys_output_str	\def	120
.....	224, 21477	\detokenize	329
\sys_rand_seed:	\dimen	446, 446
....	231, 231, 22138	\dimendef	446
\c_sys_shell_escape_int	\dimexpr	161
....	231, 22141, 22156, 22164, 22172	\directlua	238
\sys_shell_now:n	\dp	209, 588, 589
....	232, 232, 22180	\e@alloc@top	731, 16149
\sys_shell_shipout:n	\edef	1, 314
....	232, 232, 22199	\endcsname	16
\c_sys_year_int	\endinput	130
sys internal commands:		\newlinechar	37, 37, 140,
\c__sys_shell_stream_int	140, 140, 320, 320, 320, 320, 321, 447	
.....	22178, 22195, 22214	\endtemplate	97, 418
syst commands:		\errhelp	467, 467
\c_syst_last_allocated_read	\errmessage	467, 467, 467, 467, 468
.....	3957, 3958	\errorcontextlines	146, 468, 485, 854
\c_syst_last_allocated_toks	..	\escapechar	41, 41, 41, 284, 297, 502, 764
		\everyeof	319, 321
T		\expandafter	30, 31
\t	\expanded	264
\tabskip	\fi	119
>tagcode	\firstmark	309, 447
\tan	\fontdimen	...
\tand 81, 81, 81, 397, 397, 398, 398	
\tate	\frozen@everydisplay	1171
\tbaselineshift	\frozen@everymath	1172
\ttemp	163, 169, 174, 177, 178, 185, 190, 193	\futurelet	...
TEX and L ^A T _E X 2 _ε commands:		... 418, 450, 451, 746, 748, 750, 750	
\@	3928	\global	253
\@end	270, 1169, 1170	\halign	97, 418, 439
\@hyph	1173	\hskip	157
\@input	1174	\ht	209, 588, 589
\@italiccorr	1175	\hyphen	447, 447
\@tracingfonts	271	\hyphenchar	397
\@underline	1176	\ifcase	79
\@addtofilelist	8572	\ifdim	160
\@currname	486, 8434, 8435	\ifeof	145
\@filelist	487,	\iffalse	91
490, 491, 8571, 8598, 8600, 8615	\ifhbox	217	
\@firstoftwo	277	\ifnum	79
\@ifpackageloaded	...	\ifodd	79, 937
23586, 23724, 23801, 24135, 24530			
\@secondoftwo	277		
\@tempa	143, 145		
\@unexpandable@protect	587		
\afterassignment	762		
\botmark	447		

\iftrue	91	\randomseed	190
\ifvbox	217	\read	140
\ifvoid	217	\readline	141
\ifx	21, 249	\relax	119, 249, 281, 286, 297, 399, 399, 550, 552, 552, 572, 601
\input@path	8511, 8515, 8530	\RequirePackage	6, 249
\italiccorr	447, 447	\reserveinserts	249, 249
\jobname	223	\robustify	233
\l@expl@check@declarations@bool	1768, 2631, 5533, 5707, 8202	\romannumeral	79
\l@expl@log@functions@bool	1705, 1753, 7592	\scantokens	319, 320, 321
\lastnamedcs	287	\setrandomseed	190
\lccode	249, 356, 396, 749, 753	\sfcode	250
\let	253	\show	16, 45, 297
\letcharcode	437	\showbox	854
\loctoks	731, 731, 731	\showthe	297, 395, 517, 521, 523
\long	120, 120	\showtokens	46, 136, 484, 485, 485, 485
\lower	893	\skip	753, 754
\lowercase	833, 834, 834	\space	447, 447
\luaescapestring	239	\splitbotmark	447
\m@ne	397, 5413	\splitfirstmark	447
\makeatletter	6	\strcmp	246, 263
\mathchar	120	\string	112, 748, 750, 751, 753
\mathchardef	378	\synctex	263
\meaning	15, 112, 120, 120, 445, 446, 446, 446, 446, 451, 748, 937	\tenrm	119
\newif	91	\tex_lowercase:D	440, 440
\newlinechar	37, 37, 146, 288, 320, 320, 320, 320, 468, 485, 500, 500	\the	70, 119, 152, 156, 159, 301, 301
\newread	493, 493, 493	\toks	xx, 194, 194, 194, 196, 196, 196, 196, 196, 729, 730, 730, 731, 731, 731, 731, 732, 732, 732, 733, 733, 734, 735, 736, 737, 737, 737, 737, 742, 742, 747, 747, 749, 751, 752, 754, 756, 760, 760, 760, 760, 760, 760, 761, 762, 762, 762, 763, 763, 763, 763, 763, 763, 764, 764, 764, 765, 766, 769, 806, 806, 813, 813, 817, 818, 818, 818, 818, 819, 820, 822, 825, 827, 828, 833, 850
\newtoks	194, 742, 763	\toksdef	742
\newwrite	498	\topmark	120, 447
\noexpand	30, 119	\tracingfonts	271
\nullfont	447, 448, 448	tracingfonts	271
\number	80, 633	\tracingonline	854
\numexpr	80	\uccode	356
\or	79	\Ucharcat	438, 439, 440
\outer	120, 120, 249, 493, 493, 498, 891, 937, 937	\ucharcat@table	52, 55
\par	888	\unexpanded	30, 42, 42, 42, 45, 59, 63, 64, 100, 103, 104, 105, 124, 227, 229, 230, 232, 236, 236, 314, 336, 337, 381, 439
\pdfmapfile	272	\unhbox	213
\pdfmapline	272	\unhcopy	213
\pdfrandomseed	190	\uniformdeviate	190, 190, 723
\pdfsetrandomseed	190	\unless	21
\pdfstrcmp	xiii, 246, 246, 247, 249, 249, 263, 939	\unvbox	214
\pdfuniformdeviate	190, 723		
\pgf@sys@...	241		
\protect	586, 587, 587, 587, 587, 919		
\protected	120, 120		
\protected@edef	504, 8983		
\ProvidesClass	6		
\ProvidesFile	6		
\ProvidesPackage	6		

\unvcopy	214	
\uppercase	833	
\valign	418	
\vskip	158	
\vsplit	214	
\vtop	870	
\wd	210, 588, 589	
\write	143, 500	
\zap@space	547	
tex commands:		
\tex_above:D	287	
\tex_abovedisplayshortskip:D . .	288	
\tex_abovedisplayskip:D	289	
\tex_abovewithdelims:D	290	
\tex_accent:D	291	
\tex_adjdemerits:D	292	
\tex_advance:D	293, 4748, 4750, 4760, 4762, 9269, 9274, 9525, 9530, 9623, 9628, 16273, 16280, 16283, 16676, 16705, 16708, 17933	
\tex_afterassignment:D	294, 7106, 16628, 16660	
\tex_aftergroup:D	295, 1329	
\tex_atop:D	296	
\tex_atopwithdelims:D	297	
\tex_badness:D	298	
\tex_baselineskip:D	299	
\tex_batchmode:D	300	
\tex_begingroup:D	301, 1325	
\tex_belowdisplayshortskip:D . .	302	
\tex_belowdisplayskip:D	303	
\tex_binoppenalty:D	304	
\tex_botmark:D	305	
\tex_box:D	306, 20052, 20076	
\tex_boxmaxdepth:D	307	
\tex_brokenpenalty:D	308	
\tex_catcode:D	309, 2320, 2324, 2737, 3967, 3968, 6556, 6560	
\tex_char:D	310	
\tex_chardef:D	276, 311, 1319, 1346, 1348, 1592, 1593, 3949, 3952, 4721, 5701, 5703, 5741, 5746, 7029, 8694, 8848	
\tex_cleaders:D	312	
\tex_closein:D	313, 3972, 8702	
\tex_closeout:D	314, 8856	
\tex_clubpenalty:D	315	
\tex_copy:D	316, 20046, 20077	
\tex_count:D	317, 3959, 8634, 8636, 8810, 8812, 16132, 16148, 16156, 16157	
\tex_countdef:D	318	
\tex_cr:D	319	
\tex_crcr:D	320	
\tex_csname:D	321, 1313	
\tex_day:D	322, 21423	
\tex_deadcycles:D	323	
\tex_def:D	324, 791, 792, 793, 1330, 1332, 1334, 1335, 1352, 1354, 1355, 1356, 1358, 1359, 1360, 1362, 1363, 1364	
\tex_defaulthyphenchar:D	325	
\tex_defaultskewchar:D	326	
\tex_delcode:D	327	
\tex_delimiter:D	328	
\tex_delimiterfactor:D	329	
\tex_delimiter shortfall:D	330	
\tex_dimen:D	331	
\tex_dimendef:D	332	
\tex_discretionary:D	333	
\tex_displayindent:D	334	
\tex_displaylimits:D	335	
\tex_displaystyle:D	336	
\tex_displaywidowpenalty:D	337	
\tex_displaywidth:D	338	
\tex_divide:D	339, 16125, 17934	
\tex_doublehyphendemerits:D	340	
\tex_dp:D	341, 20062	
\tex_dump:D	342	
\tex_eodef:D	343, 1353, 1357, 1361, 1365, 9031, 9088, 21506	
\tex_else:D	344, 1180, 1300, 1349	
\tex_emergencystretch:D	345	
\tex_end:D	346, 1170, 1269, 1736, 7826	
\tex_endcsname:D	347, 1314	
\tex_endgroup:D	348, 1167, 1326	
\tex_endinput:D	349, 7837	
\tex_endlinechar:D	251, 252, 266, 350, 2710, 2711, 2712, 2768, 8742, 8744	
\tex_eqno:D	351	
\tex_errhelp:D	352, 7687	
\tex_errmessage:D	353, 1728, 7718	
\tex_errorcontextlines:D	354, 7713, 7745, 8399, 20146	
\tex_errorstopmode:D	355	
\tex_escapechar:D	356, 2039, 8921, 8956, 8962, 16564, 16593, 16657, 16658, 17013	
\tex_everycr:D	357	
\tex_everydisplay:D	358, 1171	
\tex_everyhbox:D	359	
\tex_everyjob:D	360, 1270, 8427, 8429, 8440, 8442, 21410, 21412	
\tex_everymath:D	361, 1172	
\tex_everypar:D	362	
\tex_everyvbox:D	363	
\tex_exhyphenpenalty:D	364	
\tex_expandafter:D	365, 796, 1315	
\tex_fam:D	366	

\tex_fi:D 367, 797, 1184, 1185, 1186, 1239, 1241, 1242, 1246, 1260, 1265, 1280, 1288, 1293, 1301, 1351, 2689
\tex_finalhyphendemerits:D 368
\tex_firstmark:D 369
\tex_floatingpenalty:D 370
\tex_font:D 371, 5426
\tex_fontdimen:D 372, 5429, 5431, 5435, 5461
\tex_fontname:D 373
\tex_futurelet:D 374, 7100, 7102, 16601
\tex_gdef:D 375, 1366, 1369, 1373, 1377
\tex_global:D 272, 277, 279, 302, 376, 798, 800, 1835, 1842, 4708, 4728, 4740, 4752, 4754, 4764, 4766, 4773, 5426, 5703, 5746, 6835, 6837, 6847, 7102, 8694, 8848, 9244, 9259, 9265, 9270, 9275, 9500, 9515, 9521, 9526, 9531, 9598, 9613, 9619, 9624, 9629, 20048, 20054, 20112, 20160, 20169, 20179, 20222, 20231, 20240, 20250
\tex_globaldefs:D 377
\tex_halign:D 378
\tex_hangafter:D 379
\tex_hangindent:D 380
\tex_hbadness:D 381
\tex_hbox:D 382, 20157, 20159, 20165, 20174, 20190, 20194
\tex_hfil:D 383
\tex_hfill:D 384
\tex_hfilneg:D 385
\tex_hfuzz:D 386
\tex_hoffset:D 387, 1282
\tex_holdinginserts:D 388
\tex_hrule:D 389
\tex_hsize:D 390, 20667, 20669, 20670, 20713, 20715, 20716
\tex_hskip:D 391, 9560
\tex_hss:D 392, 20196, 20198, 20552, 20561
\tex_ht:D 393, 20061
\tex_hyphen:D 286, 1173
\tex_hyphenation:D 394
\tex_hyphenchar:D 395, 5427, 5428, 5429, 5433
\tex_hyphenpenalty:D 396
\tex_if:D 107, 397, 1303, 1304
\tex_ifcase:D 398, 4618
\tex_ifcat:D 399, 1305
\tex_ifdim:D 400, 9226
\tex_ifeof:D 401, 3977, 8719
\tex_iffalse:D 402, 1298
\tex_ifhbox:D 403, 20088
\tex_ifhmode:D 404, 1308
\tex_ifinner:D 405, 1310
\tex_ifmmode:D 406, 1307
\tex_ifnum:D 407, 1240, 1327
\tex_ifodd:D 408, 1705, 1753, 1768, 2631, 4617, 5533, 5675, 5676
\tex_iftrue:D 409, 1297
\tex_ifvbox:D 410, 20089
\tex_ifvmode:D 411, 1309
\tex_ifvoid:D 412, 20090
\tex_ifx:D 413, 1306
\tex_ignorespaces:D 414
\tex_immediate:D 415, 1696, 1698, 8850, 8856, 8886, 23647
\tex_indent:D 416
\tex_input:D 417, 1174, 1271, 8577, 22285, 22308
\tex_inputlineno:D 418, 1743, 7661
\tex_insert:D 419
\tex_insertpenalties:D 420
\tex_interlinepenalty:D 421
\tex_italiccorrection:D 285, 1175, 1283
\tex_jobname:D 422, 8430, 21413, 21417
\tex_kern:D 423, 20308, 20550, 20559, 20979, 20984, 21050, 21051, 21338, 21339, 21570, 21572, 21616, 21618, 21698, 24043
\tex_language:D 424, 1272
\tex_lastbox:D 425, 20110
\tex_lastkern:D 426
\tex_lastpenalty:D 427
\tex_lastskip:D 428
\tex_lccode:D 429, 3602, 3603, 3625, 3626, 3919, 4032, 6638, 6642, 16574, 16584, 16655, 16657, 16677, 19372, 19421, 22490
\tex_leaders:D 430
\tex_left:D 431, 1290
\tex_lefthyphenmin:D 432
\tex_leftskip:D 433
\tex_leqno:D 434
\tex_let:D 273, 277, 279, 435, 798, 800, 1170, 1171, 1172, 1173, 1174, 1175, 1176, 1177, 1179, 1183, 1188, 1189, 1190, 1191, 1192, 1193, 1194, 1195, 1196, 1197, 1198, 1199, 1200, 1201, 1202, 1203, 1204, 1205, 1206, 1207, 1208, 1209, 1210, 1211, 1212, 1213, 1214, 1215, 1216, 1217, 1218, 1219, 1220, 1221, 1222, 1223, 1224, 1225, 1226, 1227, 1228, 1229, 1230, 1231, 1232, 1233, 1234, 1235, 1236, 1237, 1238, 1244, 1245, 1250, 1251, 1252, 1253,

- 1254, 1255, 1256, 1257, 1258, 1259,
 1262, 1263, 1264, 1269, 1270, 1271,
 1272, 1273, 1274, 1275, 1276, 1277,
 1278, 1279, 1282, 1283, 1284, 1285,
 1286, 1287, 1290, 1291, 1292, 1297,
 1298, 1299, 1300, 1301, 1302, 1303,
 1304, 1305, 1306, 1307, 1308, 1309,
 1310, 1311, 1312, 1313, 1314, 1315,
 1316, 1317, 1318, 1320, 1321, 1322,
 1323, 1324, 1325, 1326, 1327, 1328,
 1329, 1345, 1352, 1353, 1366, 1367,
 1831, 6835, 6837, 6847, 16575, 16585
`\tex_limits:D` 436
`\tex_linepenalty:D` 437
`\tex_lineskip:D` 438
`\tex_lineskiplimit:D` 439
`\tex_long:D` 440, 791, 792,
 793, 1330, 1332, 1335, 1354, 1355,
 1356, 1357, 1358, 1360, 1362, 1363,
 1364, 1365, 1369, 1371, 1377, 1379
`\tex_looseness:D` 441
`\tex_lower:D` 442, 20087
`\tex_lowercase:D`
 443, 2744, 2761, 3413,
 3414, 3604, 3627, 6677, 6800, 7699,
 16575, 16585, 16656, 19373, 19422
`\tex_mag:D` 444
`\tex_mark:D` 445
`\tex_mathaccent:D` 446
`\tex_mathbin:D` 447
`\tex_mathchar:D` 448
`\tex_mathchardef:D`
 276, 449, 1350, 4724, 4725
`\tex_mathchoice:D` 450
`\tex_mathclose:D` 451
`\tex_mathcode:D` 452, 6629, 6633
`\tex_mathinner:D` 453
`\tex_mathop:D` 454, 1273
`\tex_mathopen:D` 455
`\tex_mathord:D` 456
`\tex_mathpunct:D` 457
`\tex_mathrel:D` 458
`\tex_mathsurround:D` 459
`\tex_maxdeadcycles:D` 460
`\tex_maxdepth:D` 461
`\tex_meaning:D` 462, 1320, 1321
`\tex_medmuskip:D` 463
`\tex_message:D` 464
`\tex_middle:D` 1291
`\tex_mkern:D` 465
`\tex_month:D` 466, 1274, 21424
`\tex_moveleft:D` 467, 20081
`\tex_moveright:D` 468, 20083
`\tex_msip:D` 469
`\tex_multiply:D` 470, 16678
`\tex_muskip:D` 471
`\tex_muskipdef:D` 472
`\tex_newlinechar:D` 473, 1727,
 2712, 2740, 2743, 7711, 8397, 8887
`\tex_noalign:D` 474
`\tex_noboundary:D` 475
`\tex_noexpand:D` 476, 1316
`\tex_noindent:D` 477
`\tex_nolimits:D` 478
`\tex_nonscript:D` 479
`\tex_nonstopmode:D` 480
`\tex_nulldelimiterspace:D` 481
`\tex_nullfont:D` 482, 7058
`\tex_number:D` 483, 4614
`\tex_omit:D` 484
`\tex_openin:D` 485, 3969, 8696
`\tex_openout:D` 486, 8850
`\tex_or:D` 487, 1299
`\tex_outer:D` 488, 1275, 21506
`\tex_output:D` 489
`\tex_outputpenalty:D` 490
`\tex_over:D` 491, 1276
`\tex_overfullrule:D` 492
`\tex_overline:D` 493
`\tex_overwithdelims:D` 494
`\tex_pagedepth:D` 495
`\tex_pagefilllstretch:D` 496
`\tex_pagefillstretch:D` 497
`\tex_pagefilstretch:D` 498
`\tex_pagegoal:D` 499
`\tex_pageshrink:D` 500
`\tex_pagestretch:D` 501
`\tex_pagetotal:D` 502
`\tex_par:D` 503
`\tex_parfillskip:D` 504
`\tex_parindent:D` 505
`\tex_parshape:D` 506
`\tex_parskip:D` 507
`\tex_patterns:D` 508
`\tex_pausing:D` 509
`\tex_penalty:D` 510
`\tex_postdisplaypenalty:D` 511
`\tex_predisplaypenalty:D` 512
`\tex_predisplaysize:D` 513
`\tex_pretolerance:D` 514
`\tex_prevdepth:D` 515
`\tex_prevgraf:D` 516
`\tex_radical:D` 517
`\tex_raise:D` 518, 20085
`\tex_read:D` 519, 3983, 8737
`\tex_relax:D`
 520, 552, 1324, 1727, 4616, 9228
`\tex_relp penalty:D` 521

\tex_right:D 522, 1292
 \tex_righthyphenmin:D 523
 \tex_rightskip:D 524
 \tex_roman numeral:D
 284, 284, 284, 284,
 284, 307, 525, 1318, 1328, 1596, 2318
 \tex_roman numeral1:D 307
 \tex_scriptfont:D 526
 \tex_scriptscriptfont:D 527
 \tex_scriptscriptstyle:D 528
 \tex_scriptspace:D 529
 \tex_scriptstyle:D 530
 \tex_scrollmode:D 531
 \tex_setbox:D 532, 20046,
 20052, 20110, 20159, 20165, 20174,
 20219, 20227, 20236, 20245, 20265
 \tex_setlanguage:D 533
 \tex_sfcode:D 534, 6656, 6660
 \tex_shipout:D 535
 \tex_show:D 536
 \tex_showbox:D 537, 20147
 \tex_showboxbreadth:D 538, 20143
 \tex_showboxdepth:D 539, 20144
 \tex_showlists:D 540
 \tex_showthe:D 541
 \tex_skewchar:D 542
 \tex_skip:D 543, 16680,
 16703, 16722, 16776, 16792, 16798
 \tex_skipdef:D 544
 \tex_space:D 284
 \tex_spacefactor:D 545
 \tex_spaceskip:D 546
 \tex_span:D 547
 \tex_special:D
 . 548, 23674, 23676, 23678, 23697,
 23710, 23731, 23735, 23753, 23775,
 23785, 23809, 23813, 24060, 24070,
 24072, 24142, 24146, 24150, 24153,
 24157, 24158, 24159, 24160, 24164,
 24165, 24166, 24417, 24419, 24420,
 24421, 24428, 24434, 24537, 24541
 \tex_splitbotmark:D 549
 \tex_splitfirstmark:D 550
 \tex_splitmaxdepth:D 551
 \tex_splittopskip:D 552
 \tex_string:D 553, 1323
 \tex_tabskip:D 554
 \tex_textfont:D 555
 \tex_textstyle:D 556
 \tex_the:D 252, 301,
 557, 583, 588, 588, 589, 734, 1743,
 2024, 2107, 2111, 4776, 4778, 5433,
 6560, 6633, 6642, 6651, 6660, 8429,
 8442, 9441, 9443, 9557, 9559, 9634,
 11520, 11555, 11987, 16263, 16310,
 16311, 16325, 16326, 16791, 16902,
 16961, 17014, 17033, 17039, 17042,
 17046, 20130, 21412, 22138, 23658
 \tex_thickmuskip:D 558
 \tex_thinmuskip:D 559
 \tex_time:D 560, 21420, 21422
 \tex_toks:D 561, 734, 16197,
 16263, 16299, 16310, 16311, 16325,
 16326, 16334, 16344, 16354, 16636,
 16656, 16791, 16961, 16991, 17000,
 17014, 17016, 17017, 17024, 17032,
 17033, 17038, 17039, 17042, 17046
 \tex_toksdef:D 562, 16457
 \tex_tolerance:D 563
 \tex_topmark:D 564
 \tex_topskip:D 565
 \tex_tracingcommands:D 566
 \tex_tracinglostchars:D 567
 \tex_tracingmacros:D 568
 \tex_tracingonline:D 569, 20145
 \tex_tracingoutput:D 570
 \tex_tracingpages:D 571
 \tex_tracingparagraphs:D 572
 \tex_tracingrestores:D 573
 \tex_tracingstats:D 574
 \tex_uccode:D
 . 575, 3920, 6647, 6651, 22491
 \tex_uchyph:D 576
 \tex_undefined:D
 . 272, 279, 447, 448, 448, 800, 1177,
 1244, 1245, 1262, 1263, 1264, 1848,
 1856, 5486, 10001, 10015, 10048,
 10069, 16135, 16575, 16585, 16746
 \tex_underline:D 577, 1176
 \tex_unhbox:D 578, 20200
 \tex_unhcopy:D 579, 20199
 \tex_unkern:D 580
 \tex_unpenalty:D 581
 \tex_unskip:D 582
 \tex_unvbox:D 583, 20261
 \tex_unvcopy:D 584, 20260
 \tex_uppercase:D 585, 3422, 3423
 \tex_vadjust:D 586
 \tex_valign:D 587
 \tex_vbadness:D 588
 \tex_vbox:D 589, 20204,
 20209, 20214, 20219, 20236, 20245
 \tex_vccenter:D 590, 1277
 \tex_vfil:D 591
 \tex_vfill:D 592
 \tex_vfilneg:D 593
 \tex_vfuzz:D 594
 \tex_voffset:D 595, 1284

\tex_vrule:D 596, 21190, 21245
 \tex_vsize:D 597
 \tex_vskip:D 598, 9563
 \tex_vspliD 599, 20265
 \tex_vss:D 600
 \tex_vtop:D 601, 20206, 20227
 \tex_wd:D 602, 20063
 \tex_widowpenalty:D 603
 \tex_write:D
 ... 604, 1696, 1698, 8868, 8871, 8888
 \tex_xdef:D 605, 1367, 1371, 1375, 1379
 \tex_xleaders:D 606
 \tex_xspaceskip:D 607
 \tex_year:D 608, 21425
 \textdir 949
 \textfont 555
 \textstyle 556
 \texttt 12751, 24115
 \TeXeTstate 665
 \tfont 1153
 \TH 23101
 \th 23101
 \the 67, 212, 213, 214,
 215, 216, 217, 218, 219, 220, 221, 557
 \thickmuskip 558
 \thimuskip 559
 thousand commands:
 \c_one_thousand 78, 5394
 \c_ten_thousand 78, 5394
 \time 560
 \tiny 21183
 tl commands:
 \c_empty_tl
 ... 46, 388, 2519, 2535, 2537, 2562,
 2888, 3984, 5104, 5110, 6008, 9738
 \l_my_tl 197, 202, 202
 \c_space_tl 46,
 2563, 3185, 3891, 6423, 6432, 7665,
 8577, 15858, 22367, 22427, 23148,
 23553, 23558, 23561, 23790, 24613
 \tl_act 334
 \tl_case:Nn 39, 39, 2980
 \tl_case:nn 345
 \tl_case:Nnn 21541, 21542
 \tl_case:NnTF 39, 39, 2980,
 2985, 2990, 3009, 3010, 21541, 21542
 \l_tl_case_change_accents_tl ...
 ... 234, 22534, 23107
 \l_tl_case_change_exclude_tl ...
 ... 234, 234, 234, 22556, 23277
 \l_tl_case_change_math_tl ...
 ... 233, 233, 22377, 23158, 23272
 \tl_clear:N 34,
 34, 2534, 2541, 6016, 6017, 8389,
 8997, 8998, 9001, 9010, 9119, 9122,
 9173, 9853, 10342, 10414, 16980, 19192
 \tl_clear:N 35
 \tl_clear_new:N 35, 35, 2540, 6020, 6021
 \tl_concat:Nnn 35, 35, 2554, 2675
 \tl_const:Nn 34, 34,
 2522, 2562, 2563, 2691, 4004, 4022,
 4034, 4069, 4070, 4071, 4090, 5566,
 6014, 6802, 6853, 7303, 7576, 7577,
 7629, 7634, 7636, 7638, 7640, 7642,
 7647, 7648, 7655, 8897, 8922, 8928,
 9790, 9791, 9792, 9793, 9794, 9795,
 9796, 10643, 10644, 10645, 10646,
 10647, 10655, 13707, 14150, 14151,
 14152, 14153, 14154, 14155, 14156,
 14157, 14158, 16040, 16932, 17057,
 19451, 22859, 22860, 22861, 22870,
 22871, 22874, 22875, 22876, 22877,
 22878, 22884, 22892, 22929, 22942,
 23063, 23086, 23087, 23104, 23105
 \tl_count:N 38, 41, 42, 42, 3074
 \tl_count:n 39,
 41, 41, 42, 293, 352, 352, 376, 562,
 1450, 1454, 1893, 1938, 3074, 3378,
 12526, 12547, 18279, 23294, 23304
 \tl_count_tokens:n .. 232, 232, 22253
 \tl_gclear:N 34,
 2534, 2543, 6018, 6019, 24651, 24687
 \tl_gclear_new:N . 35, 2540, 6022, 6023
 \tl_gconcat:Nnn 35, 2554, 2682
 \tl_gput_left:Nn . 35, 2582, 2652, 2653
 \tl_gput_right:Nn
 ... 35, 2606, 2656, 2657, 4214, 5666
 \tl_gremove_all:Nn 36, 2870
 \tl_gremove_once:Nn 36, 2864
 \tl_greplace_all:Nnn . 36, 2795, 2873
 \tl_greplace_once:Nnn . 36, 2795, 2867
 \tl_greverse:N 42, 3215
 .tl_gset:N 166, 10210
 \tl_gset:Nn 35, 58, 196, 196,
 322, 733, 733, 2557, 2564, 2649,
 2687, 2695, 2798, 2802, 3108, 3218,
 4129, 4134, 4146, 4183, 4200, 4240,
 4266, 4350, 4379, 4416, 4420, 6035,
 6063, 6109, 6152, 6190, 6242, 6273,
 7354, 7383, 7419, 7431, 7454, 8430,
 8576, 16038, 16220, 16234, 16242,
 16713, 16970, 16972, 17278, 18342,
 22091, 22101, 22272, 22297, 24610
 \tl_gset_eq:NN .. 35, 2537, 2546,
 2667, 3448, 4113, 4114, 4115, 4116,
 6028, 6029, 6030, 6031, 7326, 7327,
 7328, 7329, 8435, 8579, 16045, 19446
 \tl_gset_from_file:Nnn . 235, 22269

\tl_gset_from_file_x:Nnn 236, 22294
 \tl_gset_rescan:Nnn 37, 2692
 .tl_gset_x:N 166, 10210
 \tl_gsort:Nn 43, 3149, 16225
 \tl_gtrim_spaces:N 42, 3103
 \tl_head:N 43, 3221
 \tl_head:n 43, 43,
 43, 44, 336, 337, 935, 935, 3221, 23302
 \tl_head:w 44, 44,
 337, 337, 338, 3221, 3261, 3280, 3302
 \tl_if_blank:nTF 37, 37, 43, 44,
 44, 2876, 2879, 2880, 2883, 2884,
 3248, 3431, 3454, 4008, 4018, 6431,
 6510, 9851, 10338, 10353, 10492,
 16366, 22713, 22733, 22783, 23293
 \tl_if_blank_p:n 37, 37, 2876
 \tl_if_empty:N 3474, 3475, 6288, 6290
 \tl_if_empty:NTF 38, 38, 137,
 2886, 2895, 2896, 9014, 9103, 9178,
 9837, 9861, 10358, 16843, 19240, 24613
 \tl_if_empty:nTF 38, 38, 327,
 328, 329, 427, 1479, 1561, 2700,
 2809, 2898, 2909, 2910, 2911, 2959,
 3341, 3362, 4058, 4149, 5589, 5596,
 5613, 6051, 6095, 6301, 6318, 6356,
 7046, 7210, 7670, 7961, 7976, 8096,
 8100, 8141, 8260, 8261, 8268, 8275,
 8281, 8288, 8363, 9008, 9905, 9997,
 11270, 15842, 15936, 16852, 16936,
 16937, 22224, 23397, 23805, 24707
 \tl_if_empty_p:N 38, 38, 2886
 \tl_if_empty_p:n 38, 38, 2898, 2911
 \tl_if_eq:NN 345
 \tl_if_eq:nn(TF) 100, 100
 \tl_if_eq:NNTF 38, 38,
 39, 84, 364, 2922, 2932, 2933, 3004,
 4252, 7474, 7942, 7996, 21256, 21259
 \tl_if_eq:nnTF 38, 38, 61, 61, 364, 2934
 \tl_if_eq_p:NN 38, 38, 2922
 \tl_if_exist:N 3472, 3473
 \tl_if_exist:NTF 35,
 35, 2541, 2543, 2560, 3067, 3392, 16839
 \tl_if_exist_p:N 35, 35, 2560
 \tl_if_head_eq_catcode:nN 338
 \tl_if_head_eq_catcode:nTF
 44, 44, 3254, 22313, 22313
 \tl_if_head_eq_catcode_p:nN
 44, 44, 3254
 \tl_if_head_eq_charcode:nN 337
 \tl_if_head_eq_charcode:nTF
 44, 44, 3254, 3271, 3272
 \tl_if_head_eq_charcode_p:nN
 44, 44, 3254
 \tl_if_head_eq_meaning:nN 338

\tl_if_head_eq_meaning:nNTF
 44, 44, 3254
 \tl_if_head_eq_meaning_p:nN
 44, 44, 3254, 18278
 \tl_if_head_is_group:nTF
 45, 45, 3161,
 3283, 3318, 3344, 22340, 22404, 23130
 \tl_if_head_is_group_p:n 45, 45, 3344
 \tl_if_head_is_N_type:n 338
 \tl_if_head_is_N_type:nTF
 45, 45, 3158, 3258, 3277, 3294,
 3327, 22221, 22337, 22401, 22617,
 22649, 22742, 22792, 23127, 23245
 \tl_if_head_is_N_type_p:n 45, 45, 3327
 \tl_if_head_is_space:nTF
 45, 45, 3355, 3885
 \tl_if_head_is_space_p:n 45, 45, 3355
 \tl_if_in:Nn 427
 \tl_if_in:NnTF 38, 38, 2831, 2949,
 2949, 2950, 2952, 2953, 5660, 8478
 \tl_if_in:nnTF 38, 38, 328, 328, 2751,
 2815, 2817, 2949, 2950, 2951, 2955,
 2963, 2964, 8560, 17172, 21065, 23395
 \tl_if_single:n 328
 \tl_if_single:NTF
 38, 38, 2966, 2967, 2968
 \tl_if_single:nTF
 39, 39, 2967, 2968, 2969, 2970, 8388
 \tl_if_single_p:N 38, 38, 2966
 \tl_if_single_p:n 39, 39, 2966, 2970
 \tl_if_single_token:nTF
 232, 232, 22219
 \tl_if_single_token_p:n
 232, 232, 22219
 \tl_item:Nn 45, 3367
 \tl_item:nn 45, 45, 935, 3367, 23294
 \tl_log:N 46, 46, 3398
 \tl_log:n 46, 46, 3398
 \tl_lower_case:n 232, 22314
 \tl_lower_case:n 53
 \tl_lower_case:nn 232, 22314
 \tl_map_break:
 40, 40, 757, 3019, 3025,
 3037, 3046, 3053, 3058, 16834, 16835
 \tl_map_break:n
 40, 40, 40, 3058, 16228, 16235
 \tl_map_function:NN 39,
 39, 39, 3015, 3082, 8470, 18346
 \tl_map_function:nN 39, 39,
 39, 39, 1932, 3015, 3077, 4152, 17680
 \tl_map_inline:Nn
 39, 39, 40, 733, 3029, 16228, 16235
 \tl_map_inline:nn 39, 39,
 40, 86, 3029, 8925, 12111, 14821, 16163

\tl_map_variable:NNn 40, 40, 3042
 \tl_map_variable:nNn 40, 40, 331, 3042
 \tl_mixed_case:n 232, 22314
 \tl_mixed_case:n 53, 234
 \tl_mixed_case:nn 232, 913, 22314
 \l_tl_mixed_case_ignore_tl
 235, 23203, 23282
 \l_tl_mixed_change_ignore_tl 235
 \tl_new:N 34, 34, 35, 112, 316, 2516,
 2541, 2543, 2947, 2948, 3403, 3404,
 3405, 3406, 4087, 4088, 5657, 6009,
 6011, 6012, 6735, 7094, 7302, 7574,
 7889, 7890, 8210, 8425, 8446, 8447,
 8626, 8777, 8802, 8900, 8902, 8914,
 8916, 8917, 8919, 9653, 9654, 9655,
 9798, 9800, 9801, 9804, 9805, 9809,
 9810, 16522, 16529, 16946, 17049,
 17050, 17056, 17066, 18914, 19132,
 19134, 20572, 20595, 20596, 21178,
 23107, 23272, 23277, 23282, 23581,
 23608, 23719, 24130, 24525, 24617
 \tl_put_left:Nn 35, 35, 2582, 2650, 2651
 \tl_put_right:Nn 35, 35, 196, 196,
 196, 2606, 2654, 2655, 4212, 6767,
 6769, 6772, 6774, 6780, 6782, 6784,
 6786, 6787, 6789, 6791, 6793, 6795,
 9135, 9138, 9143, 10356, 16949, 19271
 \tl_rand_item:N 236, 236, 23291
 \tl_rand_item:n 236, 236, 23291
 \tl_range:Nnn 236, 236, 23298
 \tl_range:nnn 236, 236, 23298
 \tl_remove_all:Nn
 36, 36, 36, 36, 2870, 8477
 \tl_remove_once:Nn 36, 36, 2864
 \tl_replace_all:Nnn
 36, 36, 362, 425, 2795, 2871, 4161
 \tl_replace_once:Nnn
 36, 36, 2795, 2865, 6810
 \tl_rescan:nn 37, 37, 37, 37, 37, 2692
 \tl_reverse:N 42, 42, 42, 3215
 \tl_reverse:n
 42, 42, 42, 42, 911, 3195, 3216, 3218
 \tl_reverse_items:n 42, 42, 42, 42, 3087
 \tl_reverse_tokens:n
 232, 232, 232, 22229
 .tl_set:N 166, 10210
 \tl_set:Nn 35,
 35, 36, 37, 58, 166, 196, 196, 308,
 316, 322, 473, 733, 733, 772, 2555,
 2564, 2648, 2680, 2693, 2722, 2781,
 2796, 2800, 2937, 2938, 3052, 3106,
 3216, 4119, 4124, 4144, 4151, 4155,
 4166, 4181, 4192, 4238, 4248, 4257,
 4264, 4297, 4300, 4321, 4329, 4338,
 4348, 4357, 4367, 4377, 4391, 4414,
 4418, 4510, 5001, 6033, 6061, 6107,
 6141, 6147, 6150, 6156, 6163, 6188,
 6240, 6271, 6399, 6765, 6770, 7112,
 7133, 7348, 7364, 7365, 7373, 7374,
 7376, 7382, 7385, 7408, 7409, 7418,
 7430, 7434, 7452, 7457, 7515, 7900,
 7982, 8208, 8390, 8460, 8463, 8464,
 8480, 8504, 8509, 8525, 8686, 8840,
 8907, 8968, 8974, 8975, 8987, 8999,
 9146, 9165, 9167, 9662, 9681, 9688,
 9705, 9712, 9723, 9779, 9816, 9818,
 9846, 9859, 9865, 9868, 9877, 9878,
 9881, 9981, 10236, 10238, 10249,
 10250, 10274, 10275, 10315, 10336,
 10348, 10354, 10416, 12809, 16036,
 16213, 16227, 16240, 16966, 16968,
 17170, 17182, 17279, 17337, 17401,
 18008, 18013, 18246, 18303, 18333,
 18969, 18974, 19055, 19087, 19363,
 19592, 19613, 19683, 19712, 19736,
 20579, 20583, 20769, 21066, 21067,
 21180, 21183, 22089, 22099, 22270,
 22293, 22295, 22307, 22900, 22919,
 23072, 23108, 23274, 23279, 23283,
 23582, 23587, 23612, 23623, 23720,
 23725, 24131, 24136, 24526, 24531
 \tl_set_eq:NN
 35, 35, 2535, 2546, 2661,
 3447, 4109, 4110, 4111, 4112, 6024,
 6025, 6026, 6027, 7322, 7323, 7324,
 7325, 7945, 7953, 10410, 16044, 19441
 \tl_set_from_file:Nnn 235, 235, 22269
 \tl_set_from_file_x:Nnn
 236, 236, 22294
 \tl_set_rescan:Nnn 37, 37, 37, 37, 2692
 .tl_set_x:N 166, 10210
 \tl_show:N 45, 45,
 46, 342, 757, 3390, 3399, 3944, 16846
 \tl_show:n
 46, 46, 46, 342, 3396, 3402, 3943
 \tl_show_analysis:N 195, 746, 16837
 \tl_show_analysis:n
 195, 195, 746, 16837
 \tl_sort:Nn 43, 43, 3149, 16225
 \tl_sort:nN
 43, 43, 738, 738, 739, 3149, 16362
 \tl_tail:N 44, 3221, 18255
 \tl_tail:n 44, 44, 44, 3221
 \tl_to_lowercase:n 109, 3407, 21549
 \tl_to_str:N
 41, 41, 47, 144, 503, 832, 3063,
 3393, 3528, 3903, 8465, 8975, 8980

```

\__tl_act_reverse:n ..... 3149, 3210, 3212, 3214, 22245
\__tl_act_space:wwNNN .... 334, 3149
\__tl_analysis:n ..... 748, 757, 16552, 16819, 16841, 16850
\__tl_analysis_a:n .... 16557, 16591
\__tl_analysis_a_bgroup:w ..... 16621, 16643
\__tl_analysis_a_cs:ww ..... 16688
\__tl_analysis_a_egrpoup:w ..... 16623, 16643
\__tl_analysis_a_group:nw .... 16643
\__tl_analysis_a_group_test:w .. 16643
\__tl_analysis_a_loop:w .. 16597, 16600, 16641, 16672, 16685, 16698
\__tl_analysis_a_safe:N 16622, 16688
\__tl_analysis_a_space:w 16620, 16626
\__tl_analysis_a_space_test:w ... 751, 16626
\__tl_analysis_a_store: ..... 751, 16637, 16668, 16674
\__tl_analysis_a_type:w 16601, 16602
\__tl_analysis_b:n .... 16558, 16711
\__tl_analysis_b_char:Nww ..... 16738, 16744
\__tl_analysis_b_cs:Nww 16740, 16764
\__tl_analysis_b_cs_test:ww .. 16764
\__tl_analysis_b_loop:w ..... 756, 16711, 16810, 16815
\__tl_analysis_b_normal:wwN ..... 16724, 16785
\__tl_analysis_b_normals:ww ... 755, 755, 16721, 16724, 16761, 16771
\__tl_analysis_b_special:w ..... 16727, 16782
\__tl_analysis_b_special_char:wN ..... 16782
\__tl_analysis_b_special_space:w ..... 16782
\__tl_analysis_char_token ..... 746, 751, 752, 16523, 16630, 16635, 16662, 16667
\__tl_analysis_cs_space_count:NN ..... 16536, 16696, 16767
\__tl_analysis_cs_space_count:w .. 16536
\__tl_analysis_cs_space_count_- end:w ..... 16536
\__tl_analysis_disable_loop:N . 16562
\__tl_analysis_extract_charcode: ..... 16530, 16655
\__tl_analysis_extract_charcode_- aux:w ..... 16530

```

```

\__tl_analysis_from_str_map_-
    inline:nn ..... 745
\l__tl_analysis_index_int 752, 753,
    16526, 16595, 16598, 16636, 16656,
    16680, 16682, 16703, 16705, 16788
\l__tl_analysis_internal_tl .. 16522
\__tl_analysis_map_inline:nn ...
    .... 744, 744, 16817, 18358, 18929
\__tl_analysis_map_inline_aux:Nn
    ..... 16817
\l__tl_analysis_nesting_int ...
    .... 750, 16527, 16596, 16676, 16684
\l__tl_analysis_normal_int ...
    16525, 16594, 16639, 16670, 16681,
    16683, 16694, 16704, 16706, 16708
\g__tl_analysis_result_tl ...
    .... 757, 16529, 16713, 16833, 16862
\__tl_analysis_setup:n ...
    ..... 747, 16556, 16562
\__tl_analysis_show: ...
    ..... 16837
\__tl_analysis_show_active:n ...
    ..... 16876, 16905
\__tl_analysis_show_cs:n 16872, 16905
\c__tl_analysis_show_etc_str ...
    ..... 759, 16925, 16927, 16932
\__tl_analysis_show_long:nn .. 16905
\__tl_analysis_show_long_-
    aux:nnnn ..... 16905, 16911
\__tl_analysis_show_loop:wNw ...
    ..... 16862, 16867
\__tl_analysis_show_normal:n ...
    ..... 16879, 16885
\__tl_analysis_show_value:N ...
    ..... 16890, 16914
\l__tl_analysis_token ...
    746, 747, 750, 750, 752, 16523, 16533,
    16601, 16605, 16608, 16611, 16667
\l__tl_analysis_type_int ...
    ..... 750, 752, 753, 16528,
    16604, 16619, 16676, 16678, 16681
\__tl_build:Nw ...
    ..... 196, 196, 196, 196, 772, 802, 16965,
    17273, 18057, 18382, 18465, 19190
\__tl_build_aux:NNw ...
    ..... 761, 16965
\__tl_build_end: ...
    ..... 196, 196, 196, 761, 802,
    16982, 17285, 17610, 17615, 18066,
    18429, 18474, 18479, 18493, 19224
\__tl_build_end_assignment:n ...
    ..... 761, 16976, 16982
\__tl_build_end_aux:n ...
    ..... 761
\l__tl_build_index_int ... 16944,
    16958, 16979, 16991, 16992, 16993,
    16995, 17000, 17001, 17002, 17004
\__tl_build_one:n ...
    ..... 196, 196, 196, 196, 760,
    772, 772, 829, 16989, 17284, 17335,
    17399, 17588, 17605, 17611, 17614,
    17661, 17664, 17695, 17706, 17708,
    17833, 17847, 17888, 17913, 17922,
    17931, 17963, 17976, 17980, 18055,
    18060, 18065, 18068, 18102, 18344,
    18360, 18377, 18436, 18483, 18488,
    18499, 19209, 19241, 19269, 19326,
    19328, 19341, 19373, 19388, 19422
\l__tl_build_result_tl ...
    ..... 760, 761, 16946, 16949, 16980, 16986
\l__tl_build_start_index_int ...
    ..... 16944, 16952, 16978, 16995, 17004
\__tl_build_unpack: ...
    ..... 16947, 16984, 16994, 17003
\__tl_build_unpack_loop:w ...
    ..... 16947
\__tl_build_x:Nw ...
    ..... 196, 16965, 17582
\__tl_case:NnTF ...
    ..... 2983, 2988, 2993, 2998, 3000
\__tl_case:nnTF ...
    ..... 2980
\__tl_case:Nw ...
    ..... 2980
\__tl_case_end:nw ...
    ..... 2980
\__tl_change_case:nnn ...
    ..... 22314, 22315, 22317, 22318, 22320
\__tl_change_case_aux:nnn ...
    ..... 22320
\__tl_change_case_char:nN ...
    ..... 22320, 23231
\__tl_change_case_char:Nnn ...
    ..... 22320
\__tl_change_case_char_auxi:nN 22320
\__tl_change_case_char_auxii:nN ...
    ..... 22320
\__tl_change_case_char_UTFviii:nn ...
    ..... 22320
\__tl_change_case_char_UTFviii:nNN ...
    ..... 22320
\__tl_change_case_char_UTFviii:nnN ...
    ..... 22320
\__tl_change_case_char_UTFviii:nNNNN ...
    ..... 22497, 22499, 22501, 22502
\__tl_change_case_char_UTFviii:nNNNN ...
    ..... 22320
\__tl_change_case_char_UTFviii:nNNNNN ...
    ..... 22464, 22500
\__tl_change_case_cs:N ...
    ..... 22320
\__tl_change_case_cs:NN ...
    ..... 22320
\__tl_change_case_cs:NNn ...
    ..... 22320
\__tl_change_case_cs_accents:NN ...
    ..... 22320
\__tl_change_case_cs_expand:NN 22320
\__tl_change_case_cs_expand:Nnw ...
    ..... 22320

```

```

\__tl_change_case_cs_letterlike:Nnn
    ..... 22320, 23179
\__tl_change_case_end:wn 22320, 23156
\__tl_change_case_group:nwnn . 22320
\__tl_change_case_if_expandable:Ntf
    ..... 22320,
    22623, 22656, 22748, 22798, 23251
\__tl_change_case_loop:wn .... 921
\__tl_change_case_loop:wnn ....
    . 914, 916, 22320, 23146, 23185, 23189
\__tl_change_case_lower_az:Nnw 22635
\__tl_change_case_lower_lt:nNnw .
    ..... 22705
\__tl_change_case_lower_lt:NNw 22705
\__tl_change_case_lower_lt:Nnw 22705
\__tl_change_case_lower_lt:nnw 22705
\__tl_change_case_lower_lt:Nw . 22705
\__tl_change_case_lower_sigma:Nnw
    ..... 22605
\__tl_change_case_lower_sigma:Nw
    ..... 22605
\__tl_change_case_lower_sigma:w .
    ..... 22605
\__tl_change_case_lower_tr:Nnw 22635
\__tl_change_case_lower_tr_-_
    auxi:Nw ..... 22635
\__tl_change_case_lower_tr_-_
    auxii:Nw ..... 22635
\__tl_change_case_math:NNNnnn ...
    ..... 915, 22320, 23169
\__tl_change_case_math:NwNNnn . 22320
\__tl_change_case_math_group:nwNNnn
    ..... 22320
\__tl_change_case_math_loop:wNNnn
    ..... 22320
\__tl_change_case_math_space:wNNnn
    ..... 22320
\__tl_change_case_mixed_nl:NNw 23233
\__tl_change_case_mixed_nl:Nnw 23233
\__tl_change_case_mixed_nl:Nw . 23233
\__tl_change_case_N_type:Nnnn . 22320
\__tl_change_case_N_type:NNNnnn .
    ..... 22320
\__tl_change_case_N_type:Nwnn . 22320
\__tl_change_case_output:nwn ...
    22320, 22609, 22643, 22651, 22665,
    22667, 22677, 22688, 22700, 22727,
    22736, 22764, 22786, 22815, 23138,
    23150, 23214, 23228, 23239, 23265
\__tl_change_case_protect:wNN . 22320
\__tl_change_case_result:n ....
    .. 22333, 22346, 22347, 22349, 23123
\__tl_change_case_setup>NN ....
    ..... 23083, 23088, 23090
\__tl_change_case_space:wnn .. 22320
\__tl_change_case_upper_az:Nnw 22635
\__tl_change_case_upper_de-alt:Nnw
    ..... 22812
\__tl_change_case_upper_lt:NNw 22705
\__tl_change_case_upper_lt:Nnw 22705
\__tl_change_case_upper_lt:nnw 22705
\__tl_change_case_upper_lt:Nw . 22705
\__tl_change_case_upper_sigma:Nnw
    ..... 22605
\__tl_change_case_upper_tr:Nnw 22635
\__tl_count:n .... 332, 3074
\__tl_from_file_do:w ..... 22269
\__tl_gbuild:Nw ..... 196, 16965
\__tl_gbuild_x:Nw ..... 196, 16965
\__tl_head_auxi:nw ..... 3221
\__tl_head_auxii:n ..... 3221
\__tl_if_blank_p:NNw ..... 2876
\__tl_if_empty_return:n . 326, 327,
    2877, 2911, 5640, 5646, 22222, 22226
\__tl_if_head_eq_meaning_-
    normal:nN ..... 3295, 3299
\__tl_if_head_eq_meaning_-
    special:nN ..... 3296, 3308
\__tl_if_head_is_N_type:w . 339, 3327
\__tl_if_head_is_space:w .... 3355
\__tl_if_single:nw .. 329, 2972, 2979
\__tl_if_single:nTF ..... 2970
\__tl_if_single_p:n ..... 2970
\__tl_internal_a_tl .....
    .... 319, 2717, 2722, 2781, 2934,
    22287, 22293, 22307, 22310, 22900,
    22902, 22919, 22924, 23072, 23074
\__tl_internal_b_tl ..... 2934
\__tl_item:nn ..... 3367
\__tl_item_aux:nn ..... 3367
\__tl_lookup_lower:N ..... 22320
\__tl_lookup_title:N ..... 22320
\__tl_lookup_upper:N ..... 22320
\__tl_loop:nn ... 22916, 22925, 22956
\__tl_map_function:Nn 330, 3015, 3034
\__tl_map_variable:Nnn ..... 3042
\__tl_mixed_case:nn .....
    ..... 22316, 22319, 23110
\__tl_mixed_case_aux:nn ..... 23110
\__tl_mixed_case_char:N ..... 23110
\__tl_mixed_case_char:Nn 23184, 23190
\__tl_mixed_case_char:nN ..... 23110
\__tl_mixed_case_group:nwn .. 23110
\__tl_mixed_case_letterlike:Nw 23110
\__tl_mixed_case_loop:wn .... 23110
\__tl_mixed_case_N_type:Nnn .. 23110
\__tl_mixed_case_N_type:NNNnn . 23110
\__tl_mixed_case_N_type:Nwn .. 23110

```

_tl_mixed_case_skip:N 23110
_tl_mixed_case_skip>NN 23110
_tl_mixed_case_skip_tidy:Nwn 23110
_tl_mixed_case_space:wn 23110
_tl_range:nnnw 23298
_tl_range:nnw 935, 23298
_tl_range_collect:w ... 935, 23298
_tl_range_normalize:nn 23298, 23347
_tl_range_skip:w 935, 23298
_tl_replace:NnNNNnn
 322, 323, 2796, 2798, 2800, 2802, 2807
_tl_replace_auxi:NnnNNNnn 323, 2807
_tl_replace_auxii:nNNNnn
 323, 323, 324, 2807
_tl_replace_next:w 322,
 324, 324, 324, 2800, 2802, 2807
_tl_replace_wrap:w ... 322, 324,
 324, 324, 324, 2796, 2798, 2807
_tl_rescan:w
 321, 321, 2692, 2773, 2774, 2789
\c_tl_rescan_marker_tl 321,
 2691, 2709, 2730, 2778, 22281, 22292
_tl_reverse_group:nn 22229
_tl_reverse_group_preserve:nn 3195
_tl_reverse_items:nwNwn 3087
_tl_reverse_items:wn 3087
_tl_reverse_normal:nN .. 3195, 22235
_tl_reverse_space:n .. 3195, 22237
_tl_set_from_file:NNnn 22269
_tl_set_from_file_x:NNnn ... 22294
_tl_set_rescan:n 319, 319, 2714, 2736
_tl_set_rescan:NNnn 2692
_tl_set_rescan:NnTF 2736
_tl_set_rescan_multi:n
 319, 321, 2692, 2746
_tl_set_rescan_multiple:n ... 320
_tl_set_rescan_single:nn 320, 2736
_tl_set_rescan_single_aux:nn 2736
_tl_tmp:w 328,
 333, 2958, 2959, 3111, 3148, 22883,
 22887, 22890, 22902, 22906, 22907,
 22908, 22909, 22924, 22927, 23058,
 23061, 23074, 23077, 23078, 23079
_tl_trim_spaces:nn 46,
 46, 525, 3104, 3111, 6081, 6524, 9779
_tl_trim_spaces_auxi:w .. 333, 3111
_tl_trim_spaces_auxii:w .. 333, 3111
_tl_trim_spaces_auxiii:w 333, 3111
_tl_trim_spaces_auxiv:w .. 333, 3111
token commands:
 \c_alignment_token
 111, 442, 6833, 6872, 16753
 \c_parameter_token
 111, 442, 442, 827, 6833, 6876, 6879

\g_peek_token ... 116, 116, 7091, 7102
\l_peek_token ... 115, 116, 451, 452,
 937, 937, 7091, 7100, 7150, 7162,
 7182, 7191, 23373, 23374, 23375, 23378
\c_space_token 45, 46, 111,
 237, 338, 443, 3285, 3320, 6833,
 6896, 7191, 9068, 16605, 16635,
 16756, 17349, 17384, 22308, 23375
\token_get_arg_spec:N 118, 118, 7265
\token_get_prefix_spec:N
 119, 119, 7265
\token_get_replacement_spec:N ...
 118, 118, 7265, 10542
\token_if_active:NTF .. 113, 113, 6909
\token_if_active_p:N .. 113, 113, 6909
\token_if_alignment:NTF
 112, 112, 113, 6870
\token_if_alignment_p:N 112, 112, 6870
\token_if_chardef:NTF
 114, 114, 6983, 16894
\token_if_chardef_p:N 114, 114, 6983
\token_if_cs:NTF
 114, 114, 6948, 22434, 23177
\token_if_cs_p:N 114,
 114, 6948, 22663, 22755, 22805, 23258
\token_if_dim_register:NTF
 114, 114, 6983, 16896
\token_if_dim_register_p:N
 114, 114, 6983
\token_if_eq_catcode:NNTF
 113, 113, 116, 116, 116, 6919
\token_if_eq_catcode_p:NN
 113, 113, 6919
\token_if_eq_charcode:NNTF
 113, 113, 116, 117, 117, 6924,
 8496, 9068, 14412, 17384, 17389,
 17986, 18201, 18319, 19243, 19313
\token_if_eq_charcode_p:NN
 113, 113, 6924
\token_if_eq_meaning:NNTF
 113, 113, 117, 117,
 117, 118, 452, 6914, 9120, 11030,
 12051, 12760, 12762, 12767, 12999,
 14941, 17687, 17992, 18019, 18150,
 18191, 18314, 18317, 19284, 19311,
 19349, 22384, 22412, 22416, 23165
\token_if_eq_meaning_p:NN
 113, 113, 6914, 22588
\token_if_expandable:NTF
 114, 114, 6953, 16892, 22584
\token_if_expandable_p:N
 114, 114, 6953
\token_if_group_begin:NTF
 112, 112, 6855

```

\token_if_group_begin_p:N . . . .
    ..... 112, 112, 6855
\token_if_group_end:NTF 112, 112, 6860
\token_if_group_end_p:N 112, 112, 6860
\token_if_int_register:NTF . . .
    ..... 115, 115, 6983, 16897
\token_if_int_register_p:N . . .
    ..... 115, 115, 6983
\token_if_letter:N . . . .
445
\token_if_letter:NTF . . .
    ..... 113, 113, 6899, 22629
\token_if_letter_p:N . 113, 113, 6899
\token_if_long_macro:NTF . . .
    ..... 114, 114, 6983
\token_if_long_macro_p:N . . .
    ..... 114, 114, 6983
\token_if_macro:NTF . . .
    ..... 114, 114, 6929, 7035, 7271, 7280, 7289
\token_if_macro_p:N . 114, 114, 6929
\token_if_math_subscript:NTF . .
    ..... 113, 113, 6889
\token_if_math_subscript_p:N . .
    ..... 113, 113, 6889
\token_if_math_superscript:NTF . .
    ..... 113, 113, 6883
\token_if_math_superscript_p:N . .
    ..... 113, 113, 6883
\token_if_math_toggle:NTF . . .
    ..... 112, 112, 6865
\token_if_math_toggle_p:N . . .
    ..... 112, 112, 6865
\token_if_mathchardef:NTF . . .
    ..... 114, 114, 6983, 16895
\token_if_mathchardef_p:N . . .
    ..... 114, 114, 6983
\token_if_muskip_register:NTF . .
    ..... 115, 115, 6983
\token_if_muskip_register_p:N . .
    ..... 115, 115, 6983
\token_if_other:NTF . 113, 113, 6904
\token_if_other_p:N . 113, 113, 6904
\token_if_parameter:NTF . 113, 6875
\token_if_parameter_p:N 113, 113, 6875
\token_if_primitive:NTF 115, 115, 7029
\token_if_primitive_p:N 115, 115, 7029
\token_if_protected_long_-
macro:NTF . . .
    ..... 114, 114, 6983
\token_if_protected_long_macro_-
p:N . . .
    ..... 114, 114, 6983, 22590
\token_if_protected_macro:NTF . .
    ..... 114, 114, 6983
\token_if_protected_macro_p:N . .
    ..... 114, 114, 6983, 22589
\token_if_skip_register:NTF . . .
    ..... 115, 115, 6983, 16898
\token_if_skip_register_p:N . . .
    ..... 115, 115, 6983
\token_if_space:NTF . 113, 113, 6894
\token_if_space_p:N . 113, 113, 6894
\token_if_toks_register:NTF . . .
    ..... 115, 115, 6983, 16899
\token_if_toks_register_p:N . . .
    ..... 115, 115, 6983
\token_new:Nn . . . .
111, 111, 6832
\token_to_meaning:N . . .
112, 112,
444, 447, 1320, 1334, 1749, 1759,
2343, 6832, 6935, 7004, 7039, 7274,
7283, 7292, 16533, 16888, 16913, 23378
\token_to_str:N 4, 17, 47, 112, 112,
112, 137, 144, 284, 340, 381, 446,
446, 583, 584, 585, 830, 1322, 1334,
1334, 1453, 1462, 1483, 1504, 1545,
1550, 1565, 1597, 1598, 1608, 1614,
1749, 1759, 1761, 1774, 1784, 1905,
1935, 1942, 2028, 2042, 2368, 2429,
2444, 2477, 2488, 2501, 2691, 3332,
3348, 3393, 3411, 3420, 4815, 5449,
5457, 5470, 5663, 5765, 6832, 7016,
7017, 7022, 7023, 7024, 7025, 7026,
7027, 8366, 8372, 8957, 8958, 8959,
8960, 8961, 9218, 9219, 9691, 9715,
10917, 10918, 11408, 11421, 11422,
11451, 11581, 11631, 11663, 11683,
11698, 11710, 11711, 11724, 11725,
11750, 11759, 11761, 11786, 11789,
11814, 11816, 11830, 11846, 11864,
11933, 11943, 11944, 11959, 11960,
12203, 12443, 15009, 16071, 16455,
16471, 16504, 16505, 16513, 16514,
16541, 16631, 16663, 16692, 16736,
16747, 16749, 16751, 16757, 16804,
16843, 16887, 16912, 16933, 17291,
17298, 17406, 17410, 18085, 19262,
19460, 20152, 20621, 20768, 21355,
21510, 21513, 22474, 22515, 22518,
22526, 23086, 23087, 23104, 23105
token internal commands:
\c__token_A_int . . . .
7029, 7066
\__token_delimit_by_char:w . .
6965
\__token_delimit_by_count:w . .
6965
\__token_delimit_by_dimen:w . .
6965
\__token_delimit_by_macro:w . .
6965
\__token_delimit_by_muskip:w . .
6965
\__token_delimit_by_skip:w . .
6965
\__token_delimit_by_toks:w . .
6965
\__token_if_macro_p:w . .
6929
\__token_if_primitive>NNw . .
7029

```

__token_if_primitive:Nw	7029	\uchyph	576
__token_if_primitive_loop:N	7029	\ucs	1164
__token_if_primitive_nullfont:N	7029	\Udelcode	980
__token_if_primitive_space:w	7029	\Udelcodenum	981
__token_if_primitive_undefined:N	7029	\Udelimiter	982
__token_if_primitive_undefined:N	7029	\Udelimiterover	983
__token_if_primitive_undefined:N	7029	\Udelimiterunder	984
__token_tmp:w	446, 6966, 6975, 6976, 6977, 6978, 6979, 6980, 6981, 6984, 7016, 7017, 7018, 7019, 7021, 7023, 7024, 7025, 7026, 7027	\Uhextensible	985
\toks	561, 7027	\Umathaccent	986
\toksapp	933	\Umathaxis	987
\toksdef	562, 16451	\Umathbinbinspacing	988
\tokspre	934	\Umathbinclosespacing	989
\tolerance	563	\Umathbininnerspacing	990
\topmark	564	\Umathbinopenspacing	991
\topmarks	666	\Umathbinopsspacing	992
\topskip	565	\Umathbinordspacing	993
\tpack	935	\Umathbinpunctspacing	994
trace commands:		\Umathbinrelspacing	995
\trace:nnn		\Umathchar	996
... 18602, 18924, 18933, 19040, 20017		\Umathcharclass	997
\trace_pop:nnn	17286, 18561, 18579, 18699, 18745, 18946, 19226	\Umathchardef	998
\trace_push:nnn	17272, 18549, 18565, 18683, 18735, 18923, 19189	\Umathcharfam	999
\tracingassigns	667	\Umathcharnum	1000
\tracingcommands	566	\Umathcharnumdef	1001
\tracingfonts	976	\Umathcharslot	1002
\tracinggroups	668	\Umathclosebinspacing	1003
\tracingifs	669	\Umathcloseclosespacing	1004
\tracinglostchars	567	\Umathcloseinnerspacing	1005
\tracingmacros	568	\Umathcloseopenspacing	1006
\tracingnesting	670	\Umathcloseopspacing	1007
\tracingonline	569	\Umathcloseordspacing	1008
\tracingoutput	570	\Umathclosepunctspacing	1009
\tracingpages	571	\Umathcloserelspacing	1010
\tracingparagraphs	572	\Umathcode	158, 1011
\tracingrestores	573	\Umathcodenum	1012
\tracingscantokens	671	\Umathconnectoroverlapmin	1013
\tracingstats	574	\Umathfractiondelsize	1014
true	191	\Umathfractiondenomdown	1015
trunc	187	\Umathfractiondenomvgap	1016
two commands:		\Umathfractionnumup	1017
\c_thirty_two	78, 5391	\Umathfractionnumvgap	1018
\c_two_hundred_fifty_five	78, 5392	\Umathfractionrule	1019
\c_two_hundred_fifty_six	78, 5392	\Umathinnerbinspacing	1020
		\Umathinnerclosespacing	1021
		\Umathinnerinnerspacing	1022
		\Umathinneropenspacing	1023
		\Umathinneropspacing	1024
		\Umathinnerordspacing	1025
		\Umathinnerpunctspacing	1026
		\Umathinnerrelspacing	1027
U		\Umathlimitabovebgap	1028
\u	xxi, 800, 23109	\Umathlimitabovekern	1029
\uccode	167, 182, 195, 197, 199, 201, 575	\Umathlimitabovevgap	1030
\Uchar	978	\Umathlimitbelowbgap	1031

\Umathlimitbelowkern	1032	\Umathrelordspacing	1086
\Umathlimitbelowvgap	1033	\Umathrelpunctspacing	1087
\Umathnolimitssubfactor	1034	\Umathrelrelspacing	1088
\Umathnolimitsupfactor	1035	\Umathskewedfractionhgap	1089
\Umathopbinspacing	1036	\Umathskewedfractionvgap	1090
\Umathpclosespacing	1037	\Umathspaceafterscript	1091
\Umathopenbinspacing	1038	\Umathstackdenomdown	1092
\Umathopenclosespacing	1039	\Umathstacknumup	1093
\Umathopeninnerspacing	1040	\Umathstackvgap	1094
\Umathopenopenspacing	1041	\Umathsubshiftdown	1095
\Umathopenopspacing	1042	\Umathsubshiftdrop	1096
\Umathopenordspacing	1043	\Umathsubsupshiftdown	1097
\Umathopenpunctspacing	1044	\Umathsubsupvgap	1098
\Umathopenrelspacing	1045	\Umathsubtopmax	1099
\Umathoperatorsize	1046	\Umathsupbottommin	1100
\Umathopinnerspacing	1047	\Umathsupshiftdrop	1101
\Umathopopenspacing	1048	\Umathsupshiftup	1102
\Umathopopspace	1049	\Umathsupsubbottommax	1103
\Umathopordspacing	1050	\Umathunderbarkern	1104
\Umathoppunctspacing	1051	\Umathunderbarrule	1105
\Umathoprelspacing	1052	\Umathunderbarvgap	1106
\Umathordbinspacing	1053	\Umathunderdelimiterbgap	1107
\Umathordclosespacing	1054	\Umathunderdelimtervgap	1108
\Umathordinnerspacing	1055	define commands:	
\Umathordopenspacing	1056	.undefined:	167, 10226
\Umathordopspacing	1057	\underline	577
\Umathordordspacing	1058	\unexpanded	672
\Umathordpunctspacing	1059	\unhbox	578
\Umathordrelspacing	1060	\unhcopy	579
\Umathoverbarkern	1061	unicode internal commands:	
\Umathoverbarrule	1062	\c__unicode_accents_lt_t1	
\Umathoverbarvgap	1063 22708, 22857	
\Umathoverdelimiterbgap	1064	__unicode_codepoint_to_UTFviii:n	
\Umathoverdelimitervgap	1065	22818, 22901, 22921, 22922, 23073	
\Umathpunctbinspacing	1066	__unicode_codepoint_to_UTFviii_-auxi:n	
\Umathpunctclosespacing	1067	22818	
\Umathpunctinnerspacing	1068	__unicode_codepoint_to_UTFviii_-auxii:Nnn	
\Umathpunctopenspacing	1069	22818	
\Umathpuncttopspacing	1070	__unicode_codepoint_to_UTFviii_-auxiii:n	
\Umathpunctordspacing	1071	22818	
\Umathpunctpunctspacing	1072	\g__unicode_data_ior	
\Umathpunctrelspacing	1073	3949, 3952, 3969, 3972, 3977, 3983	
\Umathquad	1074	\c__unicode_dot_above_t1 22764, 22857	
\Umathradicaldegreeafter	1075	\c__unicode_dotless_i_t1	
\Umathradicaldegreebefore	1076	22651, 22665, 22677, 22880	
\Umathradicaldegreeraise	1077	\c__unicode_dotted_I_t1 22700, 22880	
\Umathradicalkern	1078	\c__unicode_final_sigma_t1	
\Umathradicalrule	1079	22619, 22631, 22857	
\Umathradicalvgap	1080	\c__unicode_I_ogonek_t1 22776, 22880	
\Umathrelbinspacing	1081	\c__unicode_i_ogonek_t1 22721, 22880	
\Umathrelclosespacing	1082	__unicode_map_inline:n	
\Umathrelinnerspacing	1083	3964, 4028, 4046, 4061	
\Umathrelopspacing	1084	__unicode_map_loop: 3971, 3975, 3988	
\Umathrelopspacing	1085	__unicode_parse:w .. 3986, 3991, 3998	

__unicode_parse_auxi:w 3994, 4012, 4029, 4047
 __unicode_parse_auxii:w 4014, 4015,
 4040, 4044, 4049, 4050, 4053, 4056
 \c__unicode_std_sigma_tl 22630, 22857
 __unicode_store:nnnn 4002, 4045, 4059
 __unicode_tmp:NN 4064, 4072, 4074
 \l__unicode_tmp_tl .. 3983, 3984, 3986
 \c__unicode_upper_Eszett_tl 22815, 22857
 \uniformdeviate 977
 \unkern 580
 \unless 673
 \unpenalty 581
 \unskip 582
 \unvbox 583
 \unvcopy 584
 \Uoverdelimiter 1109
 \uppercase 585
 uptex commands:
 \uptex_disablecjktoken:D 1158, 4716, 4719, 21452
 \uptex_enablecjktoken:D 1159
 \uptex_forcecjktoken:D 1160
 \uptex_kchar:D 1161
 \uptex_kchardef:D 1162, 4720
 \uptex_kuten:D 1163
 \uptex_UCS:D 1164
 \Uradical 1110
 \Uroot 1111
 use commands:
 \use:N 16, 16, 16, 16, 82, 281, 1381,
 1478, 1560, 1688, 1690, 1692, 1694,
 3912, 3916, 4813, 5233, 5243, 5348,
 5352, 5354, 5356, 5357, 5361, 5815,
 5834, 7822, 7833, 7848, 7857, 7865,
 7873, 7879, 7905, 9026, 9334, 9891,
 9898, 10099, 17595, 19249, 19365,
 21997, 22445, 22482, 22486, 22487
 \use:n 17, 17,
 19, 19, 34, 119, 234, 277, 321, 482,
 482, 562, 584, 740, 740, 740, 740,
 740, 819, 854, 1382, 1387, 1465,
 1511, 1531, 1611, 1852, 2351, 2357,
 2378, 2498, 2637, 2697, 2769, 2771,
 3054, 3311, 3890, 3980, 4049, 4050,
 4053, 6401, 6789, 6847, 6929, 6968,
 6986, 7030, 7786, 7803, 8031, 8048,
 8300, 8353, 8489, 8740, 8793, 8876,
 9449, 9900, 10537, 10953, 10961,
 10970, 10987, 10995, 11023, 11411,
 11490, 11526, 11555, 16205, 16447,
 16902, 17008, 17151, 17560, 17563,
 17678, 18388, 18439, 18521, 18917,
 18954, 18990, 19073, 19610, 19733,
 20147, 21429, 21435, 21443, 21455,
 21462, 21471, 21481, 21484, 21491,
 21496, 22017, 22041, 23959, 23993,
 24006, 24692, 24732, 24749, 24767
 \use:nn ... 17, 17, 1387, 2080, 2729,
 6385, 7265, 9328, 11517, 14597, 22290
 \use:nnn 17, 17, 1387, 1902
 \use:nnnn 17, 17, 1387
 \use_i:nn
 ... 18, 18, 18, 275, 279, 410, 411,
 457, 692, 695, 707, 711, 712, 723,
 921, 1338, 1391, 1417, 1494, 1638,
 1666, 1884, 2495, 3235, 4181, 4183,
 5811, 5824, 5828, 5845, 5846, 7341,
 11490, 13193, 13677, 13906, 14421,
 14592, 14831, 14841, 14845, 15273,
 15488, 15914, 16303, 16338, 16348,
 16358, 17520, 17531, 17540, 17543,
 17552, 21437, 21445, 21457, 21464,
 21473, 21486, 21493, 22593, 22668
 \use_i:nnn 18, 18, 18, 723, 1393, 1620,
 4386, 5490, 7274, 13150, 15908, 19288
 \use_i:nnnn
 18, 18, 18, 1393, 13168, 13175, 13366
 \use_i_delimit_by_q_nil:nw
 ... 19, 19, 1404
 \use_i_delimit_by_q_recursion_-
 stop:nw 19, 19,
 1404, 5582, 5598, 5874, 5889, 18291,
 22386, 22545, 22568, 23167, 23212
 \use_i_delimit_by_q_stop:nw
 ... 19, 19,
 353, 353, 1404, 3667, 3676, 3791,
 3842, 3845, 6494, 15893, 15965, 22059
 \use_i_ii:nnn
 ... 18, 18, 1393, 2105, 4363, 4463
 \use_ii:nn 18, 18, 97, 275,
 279, 410, 411, 411, 457, 692, 695,
 707, 711, 712, 722, 791, 796, 804,
 1340, 1391, 1419, 1496, 1640, 1668,
 1882, 2049, 2741, 3237, 5824, 7342,
 10865, 10888, 13195, 14423, 14837,
 14843, 14847, 15275, 15490, 15846,
 17522, 17528, 17533, 17545, 17554,
 18284, 18458, 21430, 21482, 21497,
 22442, 22592, 22595, 22634, 23195
 \use_ii:nnn 18, 18, 484, 1393, 1622, 7283
 \use_ii:nnnn 18, 18, 1393
 \use_iii:nnn .. 18, 18, 1393, 2054, 7292
 \use_iii:nnnn 18, 18, 1393
 \use_iv:nnnn 18, 18, 1393

\use_none:n	18, 18, 22, 46, 326, 333, 333, 371, 423, 426, 467, 504, 580, 581, 581, 581, 757, 792, 798, 1407, 1511, 1700, 1716, 1720, 1854, 1906, 2856, 2877, 3146, 3234, 3250, 3314, 3331, 3341, 3342, 3347, 3362, 3365, 3978, 4085, 4451, 4472, 5103, 5109, 5480, 5584, 5599, 6043, 6168, 6261, 6287, 7079, 7721, 8096, 8100, 8999, 9054, 9109, 9728, 9751, 9769, 10493, 10808, 10812, 10816, 10820, 10899, 12065, 12230, 12242, 12254, 12268, 12304, 12335, 12395, 12436, 13169, 13172, 14093, 15009, 15854, 15898, 15905, 16252, 16582, 16692, 16736, 16828, 16869, 16988, 17153, 17310, 17420, 17566, 21428, 21436, 21444, 21456, 21463, 21472, 21480, 21485, 21492, 21495, 22081, 22082, 22222, 22728, 24707, 24709	971
\use_imageresource	971	
\Uskewed	1112	
\Uskewedwithdelims	1113	
\Ustack	1114	
\Ustartdisplaymath	1115	
\Ustartmath	1116	
\Ustopdisplaymath	1117	
\Ustopmath	1118	
\Usubscript	1119	
\Usuperscript	1120	
utex commands:		
\utex_binbinspacing:D	988	
\utex_binclosespacing:D	989	
\utex_bininnerspacing:D	990	
\utex_binopenspacing:D	991	
\utex_binopspacing:D	992	
\utex_binordspacing:D	993	
\utex_binpunctspacing:D	994	
\utex_binrelspacing:D	995	
\utex_char:D	355, 978, 1221, 3908, 3916, 3970, 4004, 4006, 4007, 4009, 4023, 4024, 4035, 4036, 4062, 22449, 22478, 22635, 22857, 22859, 22860, 22863, 22864, 22865, 22866, 22867, 22868, 22870, 22871, 22881, 22884	
\utex_charcat:D	979, 6744, 6760	
\utex_closebinspacing:D	1003	
\utex_closelowerspacing:D	1004	
\utex_closeinnerspacing:D	1005	
\utex_closeopenspacing:D	1006	
\utex_closeopsspacing:D	1007	
\utex_closeordspacing:D	1008	
\utex_closepunctspacing:D	1009	
\utex_closerelspace:D	1010	
\utex_connectoroverlapmin:D	1013	
\utex_delcode:D	980, 1250	
\utex_delcodenum:D	981, 1251	
\utex_delimiter:D	982, 1252	
\utex_delimiterover:D	983	
\utex_delimiterunder:D	984	
\utex_fractiondelsize:D	1014	
\utex_fractiondenomdown:D	1015	
\utex_fractiondenomvgap:D	1016	
\utex_fractionnumup:D	1017	
\utex_fractionnumvgap:D	1018	
\utex_fractionrule:D	1019	
\utex_hextensible:D	985	
\utex_innerbinspacing:D	1020	
\utex_innerclosespacing:D	1021	
\utex_innerinnerspacing:D	1022	
\utex_inneropenspacing:D	1023	
\utex_inneropsspacing:D	1024	
\utex_innerordspacing:D	1025	
\utex_innerpunctspacing:D	1026	
use internal commands:		
__use_none_delimit_by_s_stop:w	88, 88, 88, 5671	
\useboxresource	970	

\utex_innerrelspacing:D	1027	\utex_punctopenspacing:D	1069
\utex_limitabovebgap:D	1028	\utex_puncttopspacing:D	1070
\utex_limitabovekern:D	1029	\utex_punctordspacing:D	1071
\utex_limitabovevgap:D	1030	\utex_punctpunctspacing:D	1072
\utex_limitbelowbgap:D	1031	\utex_punctrelspacing:D	1073
\utex_limitbelowkern:D	1032	\utex_quad:D	1074
\utex_limitbelowvgap:D	1033	\utex_radical:D	1110
\utex_mathaccent:D	986, 1253	\utex_radicaldegreeafter:D ...	1075
\utex_mathaxis:D	987	\utex_radicaldegreebefore:D ..	1076
\utex_mathchar:D	996, 1254	\utex_radicaldegreeraise:D ...	1077
\utex_mathcharclass:D	997	\utex_radicalkern:D	1078
\utex_mathchardef:D	998, 1255	\utex_radicalrule:D	1079
\utex_mathcharfam:D	999	\utex_radicalvgap:D	1080
\utex_mathcharnum:D	1000, 1256	\utex_relbinspacing:D	1081
\utex_mathcharnumdef:D ...	1001, 1257	\utex_relclosespacing:D	1082
\utex_mathcharslot:D	1002	\utex_relinnerspacing:D	1083
\utex_mathcode:D	1011, 1258	\utex_relopspacing:D	1084
\utex_mathcodenum:D	1012, 1259	\utex_relopspacing:D	1085
\utex_nolimitsupfactor:D	1034	\utex_relordspacing:D	1086
\utex_nolimitsupfactor:D	1035	\utex_relpunctspacing:D	1087
\utex_opbinspacing:D	1036	\utex_relrelspacing:D	1088
\utex_opclosespacing:D	1037	\utex_root:D	1111
\utex_openbinspacing:D	1038	\utex_skewed:D	1112
\utex_openclosespacing:D	1039	\utex_skewedfractionhgap:D ...	1089
\utex_openinnerspacing:D	1040	\utex_skewedfractionvgap:D ...	1090
\utex_openopenspacing:D	1041	\utex_skewedwithdelims:D	1113
\utex_openopspacing:D	1042	\utex_spaceafterscript:D	1091
\utex_openordspacing:D	1043	\utex_stack:D	1114
\utex_openpunctspacing:D	1044	\utex_stackdenomdown:D	1092
\utex_openrelspacing:D	1045	\utex_stacknumup:D	1093
\utex_operatorsize:D	1046	\utex_stackvgap:D	1094
\utex opinnerspacing:D	1047	\utex_startdisplaymath:D	1115
\utex_opopenspacing:D	1048	\utex_startmath:D	1116
\utex_opopspace:D	1049	\utex_stopdisplaymath:D	1117
\utex_opordspacing:D	1050	\utex_stopmath:D	1118
\utex_oppunctspacing:D	1051	\utex_subscript:D	1119
\utex_oprelspacing:D	1052	\utex_subshiftdown:D	1095
\utex_ordbinspacing:D	1053	\utex_subshiftdrop:D	1096
\utex_ordclosespacing:D	1054	\utex_subsupshiftdown:D	1097
\utex_ordinnerspacing:D	1055	\utex_subsupvgap:D	1098
\utex_ordopenspacing:D	1056	\utex_subtopmax:D	1099
\utex_ordopspacing:D	1057	\utex_supbottommin:D	1100
\utex_ordordspacing:D	1058	\utex_superscript:D	1120
\utex_ordpunctspacing:D	1059	\utex_supshiftdrop:D	1101
\utex_ordrelspacing:D	1060	\utex_supshiftup:D	1102
\utex_overbarkern:D	1061	\utex_supsubbottommax:D	1103
\utex_overbarrule:D	1062	\utex_underbarkern:D	1104
\utex_overbarvgap:D	1063	\utex_underbarrule:D	1105
\utex_overdelimiter:D	1109	\utex_underbarvgap:D	1106
\utex_overdelimiterbgap:D ...	1064	\utex_underdelimiter:D	1121
\utex_overdelimitervgap:D ...	1065	\utex_underdelimiterbgap:D ...	1107
\utex_punctbinspacing:D	1066	\utex_underdelimitervgap:D ...	1108
\utex_punctclosespacing:D ...	1067	\utex_vextensible:D	1122
\utex_punctinnerspacing:D ...	1068	\Uunderdelimiter	1121

\Uvextensible	1122	\write	604	
V			X	
\v	23109	\xdef	605	
\vadjust	586	xetex commands:		
\valign	587	\xetex_charclass:D	806	
value commands:		\xetex_charglyph:D	807	
.value_forbidden:n	167, 10228	\xetex_countfeatures:D	808	
.value_required:n	167, 10228	\xetex_countglyphs:D	809	
\vbadness	588	\xetex_countselectors:D	810	
\vbox	589	\xetex_countvariations:D	811	
vbox commands:		\xetex_dashbreakstate:D	813	
\vbox:n	213, 213, 20203	\xetex_defaultencoding:D	812	
\vbox_gset:Nn	213, 20217	\xetex_featurecode:D	814	
\vbox_gset:Nw	214, 20243	\xetex_featurename:D	815	
\vbox_gset_end:	214, 20243	\xetex_findfeaturebyname:D	816	
\vbox_gset_to_ht:Nnn	214, 20234	\xetex_findselectorbyname:D	817	
\vbox_gset_top:Nn	214, 20225	\xetex_findvariationbyname:D	818	
\vbox_set:Nn 213, 213, 214, 20217, 20665		\xetex_firstfontchar:D	819	
\vbox_set:Nw	214, 214, 20243, 20712	\xetex_fonttype:D	820	
\vbox_set_end:	214, 214, 20243, 20720	\xetex_generateactualtext:D	821	
\vbox_set_split_to_ht>NNn	214, 214, 20264	\xetex_glyph:D	822	
\vbox_set_to_ht:Nnn	214, 214, 20234	\xetex_glyphbounds:D	823	
\vbox_set_top:Nn	214, 214, 20225, 20677, 20724	\xetex_glyphindex:D	824	
\vbox_to_ht:nn	213, 213, 20207	\xetex_glyphname:D	825	
\vbox_to_zero:n	213, 213, 20207	\xetex_if_engine:TF		
\vbox_top:n	213, 213, 20203 21544, 21545, 21546		
\vbox_unpack:N	214, 214, 214, 20260, 20677, 20724	\xetex_if_engine_p:	21543	
\vbox_unpack_clear:N	214, 20260	\xetex_inputencoding:D	826	
\vcenter	590	\xetex_inputnormalization:D	827	
vcoffin commands:		\xetex_interchartokenstate:D	828	
\vcoffin_set:Nnn	219, 219, 20661	\xetex_interchartokts:D	829	
\vcoffin_set:Nnw	219, 219, 20708	\xetex_isdefaultselector:D	830	
\vcoffin_set_end:	219, 219, 20708	\xetex_isexclusivefeature:D	831	
\vfil	591	\xetex_lastfontchar:D	832	
\vfill	592	\xetex_linebreaklocale:D	834	
\vfilneg	593	\xetex_linebreakpenalty:D	835	
\vfuzz	594	\xetex_linebreakskip:D	833	
\voffset	595	\xetex_OTcountfeatures:D	836	
\vpack	936	\xetex_OTcountlanguages:D	837	
\vrule	596	\xetex_OTcountsscripts:D	838	
\vsize	597	\xetex_OTfeaturetag:D	839	
\vskip	598	\xetex_OTlanguagetag:D	840	
\vsplit	599	\xetex_OTscripttag:D	841	
\vss	600	\xetex_pdffile:D	842, 23828	
\vtop	601	\xetex_pdfpagecount:D	843	
W			\xetex_picfile:D	844, 23824
\wd	602	\xetex_selectorname:D	845	
\widowpenalties	674	\xetex_suppressfontnotfounderror:D	805, 1222	
\widowpenalty	603	\xetex_tracingfonts:D	846	
		\xetex_upwardsmode:D	847	
		\xetex_useglyphmetrics:D	848	
		\xetex_variation:D	849	

\xetex_variationdefault:D	850	\XeTeXlinebreakpenalty	835
\xetex_variationmax:D	851	\XeTeXlinebreakskip	833
\xetex_variationmin:D	852	\XeTeXmathaccent	1253
\xetex_variationname:D	853	\XeTeXmathchar	1254
\xetex_XeTeXrevision:D	854	\XeTeXmathchardef	1255
\xetex_XeTeXversion:D		\XeTeXmathcharnum	1256
	855, 4718, 5402, 21469	\XeTeXmathcharnumdef	1257
\XeTeXcharclass	806	\XeTeXmathcode	1258
\XeTeXcharglyph	807	\XeTeXmathcodenum	1259
\XeTeXcountfeatures	808	\XeTeXOTcountfeatures	836
\XeTeXcountglyphs	809	\XeTeXOTcountlanguages	837
\XeTeXcountselectors	810	\XeTeXOTcountsscripts	838
\XeTeXcountvariations	811	\XeTeXOTfeaturetag	839
\XeTeXdashbreakstate	813	\XeTeXOTlanguagetag	840
\XeTeXdefaultencoding	812	\XeTeXOTscripttag	841
\XeTeXdelcode	1249, 1250	\XeTeXpdffile	842
\XeTeXdelcodenum	1251	\XeTeXpdfpagecount	843
\XeTeXdelimiter	1252	\XeTeXpicfile	844
\XeTeXfeaturecode	814	\XeTeXrevision	854
\XeTeXfeaturename	815	\XeTeXselectorname	845
\XeTeXfindfeaturebyname	816	\XeTeXtracingfonts	846
\XeTeXfindselectorbyname	817	\XeTeXupwardsmode	847
\XeTeXfindvariationbyname	818	\XeTeXuseglyphmetrics	848
\XeTeXfirstfontchar	819	\XeTeXvariation	849
\XeTeXfonttype	820	\XeTeXvariationdefault	850
\XeTeXgenerateactualtext	821	\XeTeXvariationmax	851
\XeTeXglyph	822	\XeTeXvariationmin	852
\XeTeXglyphbounds	823	\XeTeXvariationname	853
\XeTeXglyphindex	824	\XeTeXversion	855
\XeTeXglyphname	825	\xkanjiskip	1154
\XeTeXinputencoding	826	\xleaders	606
\XeTeXinputnormalization	827	\xspaceskip	607
\XeTeXinterchartokenstate	828	\xspcode	1155
\XeTeXinterchartoks	829		Y
\XeTeXisdefaultselector	830		
\XeTeXisexclusivefeature	831	\ybaselineshift	1156
\XeTeXlastfontchar	832	\year	608
\XeTeXlinebreaklocale	834	\yoko	1157