

The L^AT_EX3 Sources

The L^AT_EX3 Project*

September 27, 2015

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>T_EX</code> concepts not supported by <code>L^AT_EX3</code>	6
II	The <code>l3bootstrap</code> package: Bootstrap code	7
1	Using the <code>L^AT_EX3</code> modules	7
1.1	Internal functions and variables	8
III	The <code>l3names</code> package: Namespace for primitives	9
1	Setting up the <code>L^AT_EX3</code> programming language	9
IV	The <code>l3basics</code> package: Basic definitions	10
1	No operation functions	10
2	Grouping material	10
3	Control sequences and functions	11
3.1	Defining functions	11
3.2	Defining new functions using parameter text	12
3.3	Defining new functions using the signature	14
3.4	Copying control sequences	16
3.5	Deleting control sequences	17
3.6	Showing control sequences	17
3.7	Converting to and from control sequences	18
4	Using or removing tokens and arguments	19
4.1	Selecting tokens from delimited arguments	21
5	Predicates and conditionals	21
5.1	Tests on control sequences	23
5.2	Primitive conditionals	23

6	Internal kernel functions	24
V	The l3expan package: Argument expansion	26
1	Defining new variants	26
2	Methods for defining variants	27
3	Introducing the variants	27
4	Manipulating the first argument	29
5	Manipulating two arguments	30
6	Manipulating three arguments	31
7	Unbraced expansion	32
8	Preventing expansion	33
9	Controlled expansion	34
10	Internal functions and variables	35
VI	The l3prg package: Control structures	37
1	Defining a set of conditional functions	37
2	The boolean data type	39
3	Boolean expressions	41
4	Logical loops	42
5	Producing multiple copies	43
6	Detecting TeX's mode	44
7	Primitive conditionals	44
8	Internal programming functions	44
VII	The l3quark package: Quarks	46
1	Introduction to quarks and scan marks	46
1.1	Quarks	46

2	Defining quarks	47
3	Quark tests	47
4	Recursion	48
5	An example of recursion with quarks	49
6	Internal quark functions	49
7	Scan marks	50
 VIII The l3token package: Token manipulation		51
1	All possible tokens	51
2	Character tokens	52
3	Generic tokens	55
4	Converting tokens	56
5	Token conditionals	56
6	Peeking ahead at the next token	60
7	Decomposing a macro definition	63
 IX The l3int package: Integers		65
1	Integer expressions	65
2	Creating and initialising integers	66
3	Setting and incrementing integers	67
4	Using integers	68
5	Integer expression conditionals	68
6	Integer expression loops	70
7	Integer step functions	72
8	Formatting integers	72
9	Converting from other formats to integers	74

10	Viewing integers	75
11	Constant integers	76
12	Scratch integers	76
13	Primitive conditionals	77
14	Internal functions	77
X	The <code>l3skip</code> package: Dimensions and skips	79
1	Creating and initialising <code>dim</code> variables	79
2	Setting <code>dim</code> variables	80
3	Utilities for dimension calculations	80
4	Dimension expression conditionals	81
5	Dimension expression loops	83
6	Using <code>dim</code> expressions and variables	84
7	Viewing <code>dim</code> variables	86
8	Constant dimensions	86
9	Scratch dimensions	87
10	Creating and initialising <code>skip</code> variables	87
11	Setting <code>skip</code> variables	88
12	Skip expression conditionals	88
13	Using <code>skip</code> expressions and variables	89
14	Viewing <code>skip</code> variables	89
15	Constant skips	89
16	Scratch skips	90
17	Inserting skips into the output	90
18	Creating and initialising <code>muskip</code> variables	90

19	Setting muskip variables	91
20	Using muskip expressions and variables	92
21	Viewing muskip variables	92
22	Constant muskips	92
23	Scratch muskips	93
24	Primitive conditional	93
25	Internal functions	93
XI	The l3tl package: Token lists	94
1	Creating and initialising token list variables	95
2	Adding data to token list variables	96
3	Modifying token list variables	96
4	Reassigning token list category codes	97
5	Reassigning token list character codes	98
6	Token list conditionals	99
7	Mapping to token lists	100
8	Using token lists	102
9	Working with the content of token lists	103
10	The first token from a token list	104
11	Using a single item	107
12	Viewing token lists	107
13	Constant token lists	108
14	Scratch token lists	108
15	Internal functions	108
XII	The l3str package: Strings	109

1	Building strings	109
2	Adding data to string variables	110
2.1	String conditionals	111
3	Working with the content of strings	112
4	String manipulation	115
5	Viewing strings	116
6	Constant token lists	117
7	Scratch strings	117
7.1	Internal string functions	117
XIII	The l3seq package: Sequences and stacks	119
1	Creating and initialising sequences	119
2	Appending data to sequences	120
3	Recovering items from sequences	120
4	Recovering values from sequences with branching	122
5	Modifying sequences	123
6	Sequence conditionals	124
7	Mapping to sequences	124
8	Using the content of sequences directly	126
9	Sequences as stacks	126
10	Sequences as sets	128
11	Constant and scratch sequences	129
12	Viewing sequences	130
13	Internal sequence functions	130
XIV	The l3clist package: Comma separated lists	131
1	Creating and initialising comma lists	131

2	Adding data to comma lists	132
3	Modifying comma lists	133
4	Comma list conditionals	134
5	Mapping to comma lists	135
6	Using the content of comma lists directly	137
7	Comma lists as stacks	137
8	Using a single item	139
9	Viewing comma lists	139
10	Constant and scratch comma lists	139
XV	The l3prop package: Property lists	141
1	Creating and initialising property lists	141
2	Adding entries to property lists	142
3	Recovering values from property lists	142
4	Modifying property lists	143
5	Property list conditionals	143
6	Recovering values from property lists with branching	144
7	Mapping to property lists	144
8	Viewing property lists	145
9	Scratch property lists	146
10	Constants	146
11	Internal property list functions	146
XVI	The l3box package: Boxes	147
1	Creating and initialising boxes	147
2	Using boxes	148

3	Measuring and setting box dimensions	148
4	Box conditionals	149
5	The last box inserted	150
6	Constant boxes	150
7	Scratch boxes	150
8	Viewing box contents	150
9	Horizontal mode boxes	151
10	Vertical mode boxes	152
11	Primitive box conditionals	154
 XVII The l3coffins package: Coffin code layer		155
1	Creating and initialising coffins	155
2	Setting coffin content and poles	155
3	Joining and using coffins	157
4	Measuring coffins	157
5	Coffin diagnostics	158
	5.1 Constants and variables	158
 XVIII The l3color package: Color support		159
1	Color in boxes	159
 XIX The l3msg package: Messages		160
1	Creating new messages	160
2	Contextual information for messages	161
3	Issuing messages	162
4	Redirecting messages	164
5	Low-level message functions	165

6	Kernel-specific functions	167
7	Expandable errors	168
8	Internal l3msg functions	169
 XX The l3keys package: Key-value interfaces		171
1	Creating keys	172
2	Sub-dividing keys	176
3	Choice and multiple choice keys	176
4	Setting keys	179
5	Handling of unknown keys	179
6	Selective key setting	180
7	Utility functions for keys	181
8	Low-level interface for parsing key-val lists	182
 XXI The l3file package: File and I/O operations		184
1	File operation functions	184
	1.1 Input-output stream management	185
	1.2 Reading from files	186
2	Writing to files	187
	2.1 Wrapping lines in output	189
	2.2 Constant input-output streams	190
	2.3 Primitive conditionals	190
	2.4 Internal file functions and variables	190
	2.5 Internal input-output functions	191
 XXII The l3fp package: floating points		192
1	Creating and initialising floating point variables	193
2	Setting floating point variables	194
3	Using floating point numbers	194
4	Floating point conditionals	196

5	Floating point expression loops	197
6	Some useful constants, and scratch variables	199
7	Floating point exceptions	199
8	Viewing floating points	201
9	Floating point expressions	201
9.1	Input of floating point numbers	201
9.2	Precedence of operators	202
9.3	Operations	203
10	Disclaimer and roadmap	209
XXIII	The l3candidates package: Experimental additions to l3kernel	212
1	Important notice	212
2	Additions to l3basics	212
3	Additions to l3box	213
3.1	Affine transformations	213
3.2	Viewing part of a box	215
3.3	Internal variables	215
4	Additions to l3clist	216
5	Additions to l3coffins	216
6	Additions to l3file	217
7	Additions to l3fp	218
8	Additions to l3int	218
9	Additions to l3keys	219
10	Additions to l3msg	219
11	Additions to l3prg	220
12	Additions to l3prop	220
13	Additions to l3seq	220
14	Additions to l3skip	221

15	Additions to <code>l3tl</code>	222
16	Additions to <code>l3tokens</code>	226
XXIV	The <code>l3sys</code> package: System/runtime functions	228
1	The name of the job	228
2	Date and time	228
	2.1 Engine	228
	2.2 Output format	229
XXV	The <code>l3luatex</code> package: LuaTeX-specific functions	230
1	Breaking out to Lua	230
XXVI	The <code>l3drivers</code> package: Drivers	232
1	Box clipping	232
2	Box rotation and scaling	233
3	Color support	233
XXVII	Implementation	233
1	<code>l3bootstrap</code> implementation	233
	1.1 Format-specific code	234
	1.2 The <code>\pdfstrcmp</code> primitive with XeTeX and LuaTeX	235
	1.3 Emulating <code>\uchar</code> in LuaTeX	236
	1.4 Engine requirements	237
	1.5 Extending allocators	238
	1.6 The L ^A T _E X3 code environment	239
2	<code>l3names</code> implementation	241

3	l3basics implementation	261
3.1	Renaming some TeX primitives (again)	262
3.2	Defining some constants	264
3.3	Defining functions	264
3.4	Selecting tokens	265
3.5	Gobbling tokens from input	267
3.6	Conditional processing and definitions	267
3.7	Dissecting a control sequence	273
3.8	Exist or free	275
3.9	Defining and checking (new) functions	277
3.10	More new definitions	280
3.11	Copying definitions	281
3.12	Undefining functions	282
3.13	Generating parameter text from argument count	282
3.14	Defining functions from a given number of arguments	283
3.15	Using the signature to define functions	284
3.16	Checking control sequence equality	286
3.17	Diagnostic functions	287
3.18	Doing nothing functions	287
3.19	Breaking out of mapping functions	287
4	l3expan implementation	288
4.1	General expansion	288
4.2	Hand-tuned definitions	292
4.3	Definitions with the automated technique	294
4.4	Last-unbraced versions	295
4.5	Preventing expansion	297
4.6	Controlled expansion	297
4.7	Defining function variants	298
5	l3prg implementation	305
5.1	Primitive conditionals	305
5.2	Defining a set of conditional functions	305
5.3	The boolean data type	305
5.4	Boolean expressions	308
5.5	Logical loops	314
5.6	Producing multiple copies	315
5.7	Detecting TeX's mode	317
5.8	Internal programming functions	317
5.9	Deprecated functions	318
6	l3quark implementation	318
6.1	Quarks	318
6.2	Scan marks	321

7	l3token implementation	322
7.1	Character tokens	322
7.2	Generic tokens	328
7.3	Token conditionals	329
7.4	Peeking ahead at the next token	338
7.5	Decomposing a macro definition	344
8	l3int implementation	345
8.1	Integer expressions	346
8.2	Creating and initialising integers	348
8.3	Setting and incrementing integers	350
8.4	Using integers	350
8.5	Integer expression conditionals	351
8.6	Integer expression loops	355
8.7	Integer step functions	356
8.8	Formatting integers	358
8.9	Converting from other formats to integers	364
8.10	Viewing integer	367
8.11	Constant integers	367
8.12	Scratch integers	368
8.13	Deprecated functions	369
9	l3skip implementation	369
9.1	Length primitives renamed	369
9.2	Creating and initialising <code>dim</code> variables	369
9.3	Setting <code>dim</code> variables	370
9.4	Utilities for dimension calculations	371
9.5	Dimension expression conditionals	372
9.6	Dimension expression loops	374
9.7	Using <code>dim</code> expressions and variables	375
9.8	Viewing <code>dim</code> variables	376
9.9	Constant dimensions	377
9.10	Scratch dimensions	377
9.11	Creating and initialising <code>skip</code> variables	377
9.12	Setting <code>skip</code> variables	378
9.13	Skip expression conditionals	379
9.14	Using <code>skip</code> expressions and variables	380
9.15	Inserting skips into the output	380
9.16	Viewing <code>skip</code> variables	380
9.17	Constant skips	380
9.18	Scratch skips	381
9.19	Creating and initialising <code>muskip</code> variables	381
9.20	Setting <code>muskip</code> variables	382
9.21	Using <code>muskip</code> expressions and variables	383
9.22	Viewing <code>muskip</code> variables	383
9.23	Constant muskips	383

9.24	Scratch muskips	383
9.25	Deprecated functions	384
10	l3tl implementation	384
10.1	Functions	384
10.2	Constant token lists	386
10.3	Adding to token list variables	386
10.4	Reassigning token list category codes	389
10.5	Reassigning token list character codes	392
10.6	Modifying token list variables	393
10.7	Token list conditionals	396
10.8	Mapping to token lists	401
10.9	Using token lists	402
10.10	Working with the contents of token lists	403
10.11	Token by token changes	405
10.12	The first token from a token list	407
10.13	Using a single item	412
10.14	Viewing token lists	413
10.15	Scratch token lists	413
11	l3str implementation	414
11.1	Creating and setting string variables	414
11.2	String comparisons	415
11.3	Accessing specific characters in a string	418
11.4	Counting characters	423
11.5	The first character in a string	424
11.6	String manipulation	425
11.7	Viewing strings	427
12	l3seq implementation	427
12.1	Allocation and initialisation	428
12.2	Appending data to either end	431
12.3	Modifying sequences	432
12.4	Sequence conditionals	434
12.5	Recovering data from sequences	435
12.6	Mapping to sequences	439
12.7	Using sequences	442
12.8	Sequence stacks	442
12.9	Viewing sequences	444
12.10	Scratch sequences	444

13	l3clist implementation	444
13.1	Allocation and initialisation	445
13.2	Removing spaces around items	447
13.3	Adding data to comma lists	448
13.4	Comma lists as stacks	449
13.5	Modifying comma lists	451
13.6	Comma list conditionals	453
13.7	Mapping to comma lists	454
13.8	Using comma lists	458
13.9	Using a single item	459
13.10	Viewing comma lists	460
13.11	Scratch comma lists	461
14	l3prop implementation	461
14.1	Allocation and initialisation	462
14.2	Accessing data in property lists	463
14.3	Property list conditionals	467
14.4	Recovering values from property lists with branching	469
14.5	Mapping to property lists	469
14.6	Viewing property lists	470
14.7	Deprecated functions	470
15	l3box implementation	471
15.1	Creating and initialising boxes	471
15.2	Measuring and setting box dimensions	472
15.3	Using boxes	472
15.4	Box conditionals	473
15.5	The last box inserted	473
15.6	Constant boxes	474
15.7	Scratch boxes	474
15.8	Viewing box contents	474
15.9	Horizontal mode boxes	475
15.10	Vertical mode boxes	476
16	l3coffins Implementation	478
16.1	Coffins: data structures and general variables	478
16.2	Basic coffin functions	480
16.3	Measuring coffins	484
16.4	Coffins: handle and pole management	485
16.5	Coffins: calculation of pole intersections	487
16.6	Aligning and typesetting of coffins	491
16.7	Coffin diagnostics	495
16.8	Messages	501
17	l3color Implementation	502

18	l3msg implementation	503
18.1	Creating messages	503
18.2	Messages: support functions and text	504
18.3	Showing messages: low level mechanism	505
18.4	Displaying messages	508
18.5	Kernel-specific functions	515
18.6	Expandable errors	521
18.7	Showing variables	522
19	l3keys Implementation	526
19.1	Low-level interface	526
19.2	Constants and variables	529
19.3	The key defining mechanism	531
19.4	Turning properties into actions	532
19.5	Creating key properties	538
19.6	Setting keys	542
19.7	Utilities	547
19.8	Messages	549
19.9	Deprecated functions	550
20	l3file implementation	550
20.1	File operations	551
20.2	Input operations	556
20.2.1	Variables and constants	556
20.2.2	Stream management	557
20.2.3	Reading input	560
20.3	Output operations	561
20.3.1	Variables and constants	561
20.4	Stream management	562
20.4.1	Deferred writing	563
20.4.2	Immediate writing	564
20.4.3	Special characters for writing	565
20.4.4	Hard-wrapping lines to a character count	565
20.5	Messages	571
21	l3fp implementation	572

22	l3fp-aux implementation	572
22.1	Internal representation	572
22.2	Internal storage of floating points numbers	573
22.3	Using arguments and semicolons	573
22.4	Constants, and structure of floating points	574
22.5	Overflow, underflow, and exact zero	576
22.6	Expanding after a floating point number	577
22.7	Packing digits	579
22.8	Decimate (dividing by a power of 10)	581
22.9	Functions for use within primitive conditional branches	583
22.10	Small integer floating points	584
22.11	Length of a floating point array	585
22.12	x-like expansion expandably	585
22.13	Messages	586
23	l3fp-traps Implementation	586
23.1	Flags	587
23.2	Traps	587
23.3	Errors	591
23.4	Messages	591
24	l3fp-round implementation	592
24.1	Rounding tools	592
24.2	The round function	596
25	l3fp-parse implementation	599
25.1	Work plan	599
25.1.1	Storing results	600
25.1.2	Precedence and infix operators	602
25.1.3	Prefix operators, parentheses, and functions	605
25.1.4	Numbers and reading tokens one by one	605
25.2	Main auxiliary functions	607
25.3	Helpers	608
25.4	Parsing one number	609
25.4.1	Numbers: trimming leading zeros	615
25.4.2	Number: small significand	616
25.4.3	Number: large significand	618
25.4.4	Number: beyond 16 digits, rounding	620
25.4.5	Number: finding the exponent	623
25.5	Constants, functions and prefix operators	626
25.5.1	Prefix operators	626
25.5.2	Constants	628
25.5.3	Functions	630
25.6	Main functions	632
25.7	Infix operators	633
25.7.1	Closing parentheses and commas	634

	25.7.2 Usual infix operators	635
	25.7.3 Juxtaposition	636
	25.7.4 Multi-character cases	637
	25.7.5 Ternary operator	638
	25.7.6 Comparisons	639
	25.8 Candidate: defining new l3fp functions	641
	25.9 Messages	643
26	l3fp-logic Implementation	644
	26.1 Syntax of internal functions	644
	26.2 Existence test	644
	26.3 Comparison	644
	26.4 Floating point expression loops	647
	26.5 Extrema	648
	26.6 Boolean operations	649
	26.7 Ternary operator	650
27	l3fp-basics Implementation	651
	27.1 Common to several operations	652
	27.2 Addition and subtraction	653
	27.2.1 Sign, exponent, and special numbers	653
	27.2.2 Absolute addition	655
	27.2.3 Absolute subtraction	658
	27.3 Multiplication	663
	27.3.1 Signs, and special numbers	663
	27.3.2 Absolute multiplication	664
	27.4 Division	666
	27.4.1 Signs, and special numbers	666
	27.4.2 Work plan	668
	27.4.3 Implementing the significand division	671
	27.5 Square root	676
	27.6 Setting the sign	683
28	l3fp-extended implementation	684
	28.1 Description of fixed point numbers	684
	28.2 Helpers for numbers with extended precision	685
	28.3 Multiplying a fixed point number by a short one	686
	28.4 Dividing a fixed point number by a small integer	686
	28.5 Adding and subtracting fixed points	688
	28.6 Multiplying fixed points	688
	28.7 Combining product and sum of fixed points	690
	28.8 Extended-precision floating point numbers	692
	28.9 Dividing extended-precision numbers	695
	28.10 Inverse square root of extended precision numbers	698
	28.11 Converting from fixed point to floating point	700

29	l3fp-expo implementation	702
29.1	Logarithm	703
29.1.1	Work plan	703
29.1.2	Some constants	703
29.1.3	Sign, exponent, and special numbers	703
29.1.4	Absolute ln	704
29.2	Exponential	711
29.2.1	Sign, exponent, and special numbers	711
29.3	Power	716
30	l3fp-trig Implementation	723
30.1	Direct trigonometric functions	723
30.1.1	Filtering special cases	724
30.1.2	Distinguishing small and large arguments	727
30.1.3	Small arguments	728
30.1.4	Argument reduction in degrees	728
30.1.5	Argument reduction in radians	730
30.1.6	Computing the power series	736
30.2	Inverse trigonometric functions	739
30.2.1	Arctangent and arccotangent	740
30.2.2	Arcsine and arccosine	745
30.2.3	Arccosecant and arcsecant	748
31	l3fp-convert implementation	749
31.1	Trimming trailing zeros	749
31.2	Scientific notation	749
31.3	Decimal representation	751
31.4	Token list representation	753
31.5	Formatting	754
31.6	Convert to dimension or integer	754
31.7	Convert from a dimension	755
31.8	Use and eval	756
31.9	Convert an array of floating points to a comma list	756
32	l3fp-assign implementation	757
32.1	Assigning values	757
32.2	Updating values	758
32.3	Showing values	759
32.4	Some useful constants and scratch variables	759

33	l3candidates Implementation	759
33.1	Additions to l3basics	759
33.2	Additions to l3box	760
33.3	Affine transformations	760
33.4	Viewing part of a box	768
33.5	Additions to l3clist	771
33.6	Additions to l3coffins	771
33.7	Rotating coffins	771
33.8	Resizing coffins	776
33.9	Coffin diagnostics	779
33.10	Additions to l3file	779
33.11	Additions to l3fp-assign	781
33.12	Additions to l3int	781
33.13	Additions to l3keys	781
33.14	Additions to l3msg	782
33.15	Additions to l3prg	782
33.16	Additions to l3prop	783
33.17	Additions to l3seq	783
33.18	Additions to l3skip	785
33.19	Additions to l3tl	786
33.19.1	Unicode case changing	788
33.20	Additions to l3tokens	811
34	l3sys implementation	816
34.1	The name of the job	816
34.2	Time and date	817
34.3	Detecting the engine	817
34.4	Detecting the output	818
34.5	Deprecated functions	819
35	l3luatex implementation	820
35.0.1	Breaking out to Lua	820
35.1	Messages	820
36	l3drivers Implementation	821
36.1	Settings for direct PDF output	821
36.2	Driver utility functions	822
36.3	Box clipping	824
36.4	Box rotation and scaling	825
36.5	Color support	827

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 `LATEX3` programming language is found in [expl3.pdf](#).

1 Naming functions and variables

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

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

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

- D** The **D** specifier means *do not use*. All of the `TEX` 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 `TEX` structure containing the data. A **V** argument will be a single token (similar to **N**), for example `\foo:V \MyVariable`; on the other hand, using **v** a *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 $\text{\TeX}\ \backslash\text{edef}$ 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_my_a_tl { A }
\tl_set:Nn \l_my_b_tl { B }
\tl_set:Nf \l_my_a_tl { \l_my_a_tl \l_my_b_tl }
```

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

- T and F** For logic tests, there are the branch specifiers **T** (*true*) and **F** (*false*). Both specifiers treat the input in the same way as **n** (no change), but make the logic much easier to see.
- p** The letter **p** indicates $\text{\TeX}\ \textit{parameters}$. 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.

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.

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;

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).

tl 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, **T_EX** 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 **T_EX**’s stomach” (if you are familiar with the **T_EX**book parlance).

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

2 Documentation conventions

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

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

`\ExplSyntaxOn`
`\ExplSyntaxOff`

`\ExplSyntaxOn ... \ExplSyntaxOff`

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

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

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

`\seq_new:N` $\langle sequence \rangle$

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

Fully expandable functions Some functions are fully expandable, which allows it to be used within an `x`-type argument (in plain T_EX 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` $\langle cs \rangle$

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

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

`\seq_map_function:NN` ☆

`\seq_map_function:NN` $\langle seq \rangle$ $\langle function \rangle$

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

<code>\xetex_if_engine:<i><u>TF</u></i> *</code>	<code>\xetex_if_engine:TF {\langle true code \rangle} {\langle false code \rangle}</code>
--	---

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

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

<code>\l_tmpa_tl</code>	A short piece of text will describe the variable: there is no syntax illustration in this case.
-------------------------	---

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

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

The normal description text.

\TeX hackers note: Detail for the experienced \TeX or $\text{\LaTeX} 2_\epsilon$ 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 $\text{\LaTeX} 3$ programming, the descriptions of functions are intended to be reasonably “complete”. However, there is also a need to avoid repetition. Formal ideas which apply to general classes of function are therefore summarised here.

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

4 \TeX concepts not supported by $\text{\LaTeX}3$

The \TeX concept of an “`\outer`” macro is *not supported* at all by $\text{\LaTeX}3$. As such, the functions provided here may break when used on top of $\text{\LaTeX}2_{\epsilon}$ 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`

`\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>*.

`\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 alike 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

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

Part III

The l3names 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_YT_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_YT_EX;

`\luatex_...` Introduced by LuaT_EX.

Part IV

The l3basics package

Basic definitions

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

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

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_begin:`**

`\group_end:`**`\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 *`<token>`* to the list of *`<tokens>`* to be inserted when the current group level ends. The list of *`<tokens>`* 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 *`<token>`* 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 the appropriate arguments absorbed by the function. In the following, $\langle code \rangle$ is therefore used as a shorthand for “replacement text”.

Functions which are not “protected” will be fully expanded inside an \mathbf{x} expansion. In contrast, “protected” functions are not expanded within \mathbf{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 \mathbf{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

<code>\cs_new:Npn</code>	<code>\cs_new:Npn <function> <parameters> {<code>}</code>
--------------------------	---

<code>\cs_new:cpn</code>

<code>\cs_new:Npx</code>

<code>\cs_new:cpx</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.

<code>\cs_new_nopar:Npn</code>

<code>\cs_new_nopar:cpn</code>

<code>\cs_new_nopar:Npx</code>

<code>\cs_new_nopar:cpx</code>

<code>\cs_new_nopar:Npn <function> <parameters> {<code>}</code>

Creates `<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.

<code>\cs_new_protected:Npn</code>

<code>\cs_new_protected:cpn</code>

<code>\cs_new_protected:Npx</code>

<code>\cs_new_protected:cpx</code>

<code>\cs_new_protected:Npn <function> <parameters> {<code>}</code>

Creates `<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.

<code>\cs_new_protected_nopar:Npn</code>
--

<code>\cs_new_protected_nopar:cpn</code>
--

<code>\cs_new_protected_nopar:Npx</code>
--

<code>\cs_new_protected_nopar:cpx</code>
--

<code>\cs_new_protected_nopar:Npn <function> <parameters> {<code>}</code>

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

<code>\cs_set:Npn</code>

<code>\cs_set:cpn</code>

<code>\cs_set:Npx</code>

<code>\cs_set:cpx</code>

<code>\cs_set:Npn <function> <parameters> {<code>}</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.

<hr/>	
<code>\cs_set_nopar:Npn</code>	<code>\cs_set_nopar:Npn <function> <parameters> {<code>}</code>
<code>\cs_set_nopar:cpn</code>	
<code>\cs_set_nopar:Npx</code>	Sets $\langle function \rangle$ to expand to $\langle code \rangle$ as replacement text. Within the $\langle code \rangle$, the
<code>\cs_set_nopar:cpx</code>	$\langle parameters \rangle$ ($\#1$, $\#2$, <i>etc.</i>) will be replaced by those absorbed by the function. When the
<hr/>	$\langle function \rangle$ is used the $\langle parameters \rangle$ absorbed cannot contain <code>\par</code> tokens. The assignment
	of a meaning to the $\langle function \rangle$ is restricted to the current \TeX group level.
<hr/>	
<code>\cs_set_protected:Npn</code>	<code>\cs_set_protected:Npn <function> <parameters> {<code>}</code>
<code>\cs_set_protected:cpn</code>	
<code>\cs_set_protected:Npx</code>	Sets $\langle function \rangle$ to expand to $\langle code \rangle$ as replacement text. Within the $\langle code \rangle$, the
<code>\cs_set_protected:cpx</code>	$\langle parameters \rangle$ ($\#1$, $\#2$, <i>etc.</i>) will be replaced by those absorbed by the function. The
<hr/>	assignment of a meaning to the $\langle function \rangle$ is restricted to the current \TeX group level.
	The $\langle function \rangle$ will not expand within an x -type argument.
<hr/>	
<code>\cs_set_protected_nopar:Npn</code>	<code>\cs_set_protected_nopar:Npn <function> <parameters> {<code>}</code>
<code>\cs_set_protected_nopar:cpn</code>	
<code>\cs_set_protected_nopar:Npx</code>	
<code>\cs_set_protected_nopar:cpx</code>	
<hr/>	
	Sets $\langle function \rangle$ to expand to $\langle code \rangle$ as replacement text. Within the $\langle code \rangle$, the
	$\langle parameters \rangle$ ($\#1$, $\#2$, <i>etc.</i>) will be replaced by those absorbed by the function. When
	the $\langle function \rangle$ is used the $\langle parameters \rangle$ absorbed cannot contain <code>\par</code> tokens. The as-
	signment of a meaning to the $\langle function \rangle$ is restricted to the current \TeX group level. The
	$\langle function \rangle$ will not expand within an x -type argument.
<hr/>	
<code>\cs_gset:Npn</code>	<code>\cs_gset:Npn <function> <parameters> {<code>}</code>
<code>\cs_gset:cpn</code>	
<code>\cs_gset:Npx</code>	Globally sets $\langle function \rangle$ to expand to $\langle code \rangle$ as replacement text. Within the $\langle code \rangle$,
<code>\cs_gset:cpx</code>	the $\langle parameters \rangle$ ($\#1$, $\#2$, <i>etc.</i>) will be replaced by those absorbed by the function. The
<hr/>	assignment of a meaning to the $\langle function \rangle$ is <i>not</i> restricted to the current \TeX group
	level: the assignment is global.
<hr/>	
<code>\cs_gset_nopar:Npn</code>	<code>\cs_gset_nopar:Npn <function> <parameters> {<code>}</code>
<code>\cs_gset_nopar:cpn</code>	
<code>\cs_gset_nopar:Npx</code>	Globally sets $\langle function \rangle$ to expand to $\langle code \rangle$ as replacement text. Within the $\langle code \rangle$,
<code>\cs_gset_nopar:cpx</code>	the $\langle parameters \rangle$ ($\#1$, $\#2$, <i>etc.</i>) will be replaced by those absorbed by the function.
<hr/>	When the $\langle function \rangle$ is used the $\langle parameters \rangle$ absorbed cannot contain <code>\par</code> tokens. The
	assignment of a meaning to the $\langle function \rangle$ is <i>not</i> restricted to the current \TeX group
	level: the assignment is global.
<hr/>	
<code>\cs_gset_protected:Npn</code>	<code>\cs_gset_protected:Npn <function> <parameters> {<code>}</code>
<code>\cs_gset_protected:cpn</code>	
<code>\cs_gset_protected:Npx</code>	Globally sets $\langle function \rangle$ to expand to $\langle code \rangle$ as replacement text. Within the $\langle code \rangle$,
<code>\cs_gset_protected:cpx</code>	the $\langle parameters \rangle$ ($\#1$, $\#2$, <i>etc.</i>) will be replaced by those absorbed by the function. The
<hr/>	assignment of a meaning to the $\langle function \rangle$ is <i>not</i> restricted to the current \TeX group level:
	the assignment is global. The $\langle function \rangle$ will not expand within an x -type argument.

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

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

3.3 Defining new functions using the signature

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

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

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

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

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

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

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

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

<hr/>	
<code>\cs_set:Nn</code>	<code>\cs_set:Nn <function> {<code>}</code>
<code>\cs_set:(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> (<i>#1, #2, etc.</i>) will be replaced by those absorbed by the function. The assignment of a meaning to the <i><function></i> is restricted to the current \TeX group level.
<hr/>	
<code>\cs_set_nopar:Nn</code>	<code>\cs_set_nopar:Nn <function> {<code>}</code>
<code>\cs_set_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> (<i>#1, #2, etc.</i>) will be replaced by those absorbed by the function. When the <i><function></i> is used the <i><parameters></i> absorbed cannot contain <code>\par</code> tokens. The assignment of a meaning to the <i><function></i> is restricted to the current \TeX group level.
<hr/>	
<code>\cs_set_protected:Nn</code>	<code>\cs_set_protected:Nn <function> {<code>}</code>
<code>\cs_set_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> (<i>#1, #2, etc.</i>) 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 restricted to the current \TeX group level.
<hr/>	
<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> (<i>#1, #2, etc.</i>) will be replaced by those absorbed by the function. When the <i><function></i> is used the <i><parameters></i> absorbed cannot contain <code>\par</code> 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.
<hr/>	
<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> (<i>#1, #2, etc.</i>) will be replaced by those absorbed by the function. The assignment of a meaning to the <i><function></i> is global.
<hr/>	
<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> (<i>#1, #2, etc.</i>) will be replaced by those absorbed by the function. When the <i><function></i> is used the <i><parameters></i> absorbed cannot contain <code>\par</code> 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 $\langle function \rangle$ to expand to $\langle code \rangle$ as replacement text. Within the $\langle code \rangle$, the number of $\langle parameters \rangle$ is detected automatically from the function signature. These $\langle parameters \rangle$ ($\#1$, $\#2$, *etc.*) will be replaced by those absorbed by the function. The $\langle function \rangle$ will not expand within an x-type argument. The assignment of a meaning to the $\langle function \rangle$ is global.

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

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

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

Updated: 2012-01-14

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

3.4 Copying control sequences

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

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

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

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

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

```
\cs_set_eq:NN <cs1> <cs2>
\cs_set_eq:NN <cs1> <token>
```

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)
```

```
\cs_gset_eq:NN <cs1> <cs2>
\cs_gset_eq:NN <cs1> <token>
```

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
```

```
\cs_undefine:N <control sequence>
```

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

Updated: 2011-09-15

3.6 Showing control sequences

```
\cs_meaning:N ★
\cs_meaning:c ★
```

```
\cs_meaning:N <control sequence>
```

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.

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

```
\cs_show:N
\cs_show:c
```

```
\cs_show:N <control sequence>
```

Displays the definition of the $\langle control\ sequence \rangle$ on the terminal.

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

Updated: 2015-08-03

3.7 Converting to and from control sequences

`\use:c` ★ `\use:c {⟨control sequence name⟩}`

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:N ⟨control sequence⟩`

`\cs_if_exist_use:c` ★

New: 2012-11-10

Tests whether the *⟨control sequence⟩* is currently defined (whether as a function or another control sequence type), and if it does inserts the *⟨control sequence⟩* into the input stream.

`\cs_if_exist_use:NTF` ★

`\cs_if_exist_use:cTF` ★

New: 2012-11-10

`\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 does inserts the *⟨control sequence⟩* into the input stream followed by the *⟨true code⟩*.

`\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.

<hr/>	
<code>\use_i:nn</code> ★	<code>\use_i:nn {\langle arg_1 \rangle} {\langle arg_2 \rangle}</code>
<code>\use_ii:nn</code> ★	
<hr/>	
	These functions absorb two arguments from the input stream. The function <code>\use_i:nn</code> discards the second argument, and leaves the content of the first argument in the input stream. <code>\use_ii:nn</code> 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.
<hr/>	
<code>\use_i:nnn</code> ★	<code>\use_i:nnn {\langle arg_1 \rangle} {\langle arg_2 \rangle} {\langle arg_3 \rangle}</code>
<code>\use_ii:nnn</code> ★	
<code>\use_iii:nnn</code> ★	
<hr/>	
	These functions absorb three arguments from the input stream. The function <code>\use_i:nnn</code> discards the second and third arguments, and leaves the content of the first argument in the input stream. <code>\use_ii:nnn</code> and <code>\use_iii:nnn</code> 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.
<hr/>	
<code>\use_i:nnnn</code> ★	<code>\use_i:nnnn {\langle arg_1 \rangle} {\langle arg_2 \rangle} {\langle arg_3 \rangle} {\langle arg_4 \rangle}</code>
<code>\use_ii:nnnn</code> ★	
<code>\use_iii:nnnn</code> ★	
<code>\use_iv:nnnn</code> ★	
<hr/>	
	These functions absorb four arguments from the input stream. The function <code>\use_i:nnnn</code> discards the second, third and fourth arguments, and leaves the content of the first argument in the input stream. <code>\use_ii:nnnn</code> , <code>\use_iii:nnnn</code> and <code>\use_iv:nnnn</code> 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.
<hr/>	
<code>\use_i_ii:nnn</code> ★	<code>\use_i_ii:nnn {\langle arg_1 \rangle} {\langle arg_2 \rangle} {\langle arg_3 \rangle}</code>
<hr/>	
	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:
	$\backslash use_i_ii:nnn \{ abc \} \{ \{ def \} \} \{ ghi \}$
	will result in the input stream containing
	$abc \{ def \}$
	<i>i.e.</i> the outer braces will be removed and the third group will be removed.

<code>\use_none:n</code>	★	<code>\use_none:n {⟨group₁⟩}</code>
<code>\use_none:nn</code>	★	
<code>\use_none:nnn</code>	★	These functions absorb between one and nine groups from the input stream, leaving nothing on the resulting input stream. These functions work after a single expansion.
<code>\use_none:nnnn</code>	★	One or more of the <code>n</code> arguments may be an unbraced single token (<i>i.e.</i> an <code>N</code> argument).
<code>\use_none:nnnnn</code>	★	
<code>\use_none:nnnnnn</code>	★	
<code>\use_none:nnnnnnn</code>	★	
<code>\use_none:nnnnnnnn</code>	★	

<code>\use:x</code>	<code>\use:x {⟨expandable tokens⟩}</code>
---------------------	---

Updated: 2011-12-31 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.

<code>\use_none_delimit_by_q_nil:w</code>	★	<code>\use_none_delimit_by_q_nil:w ⟨balanced text⟩ \q_nil</code>
<code>\use_none_delimit_by_q_stop:w</code>	★	<code>\use_none_delimit_by_q_stop:w ⟨balanced text⟩ \q_stop</code>
<code>\use_none_delimit_by_q_recursion_stop:w</code>	★	<code>\use_none_delimit_by_q_recursion_stop:w ⟨balanced text⟩ \q_recursion_stop</code>

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

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

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

L^AT_EX3 has three concepts for conditional flow processing:

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

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

a function that 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 $\langle true\ code \rangle$ and/or $\langle false\ code \rangle$ 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_tl || \cs_if_free_p:N \g_tmpz_tl
} {\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 `LATEX 2ε`. Their use is discouraged in `expl3` (although still used in low-level definitions) because they are more fragile and in many cases require more expansion control (hence more code) than the two types of conditionals described above.

```
\c_true_bool
\c_false_bool
```

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

5.1 Tests on control sequences

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

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

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

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

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

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

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

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

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

<code>\if:w</code>	★	<code>\if:w <token₁> <token₂> <true code> \else: <false code> \fi:</code>
<code>\if_charcode:w</code>	★	<code>\if_catcode:w <token₁> <token₂> <true code> \else: <false code> \fi:</code>
<code>\if_catcode:w</code>	★	These conditionals 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 `<cs>` appears in the hash table or if the control sequence that can be formed from `<tokens>` appears in the hash table. The latter function does not turn the control sequence in question into `\scan_stop:!` This can be useful when dealing with control sequences which cannot be entered as a single token.

<code>\if_mode_horizontal:</code>	★	<code>\if_mode_horizontal: <true code> \else: <false code> \fi:</code>
<code>\if_mode_vertical:</code>	★	
<code>\if_mode_math:</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_inner:</code>	★	

6 Internal kernel functions

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

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

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

<code>__chk_log:x</code>	<code>__chk_log:x {(message text)}</code>
	If the <code>log-functions</code> option is active, this function writes the <code><message text></code> to the log file using <code>\iow_log:x</code> . Otherwise, the <code><message text></code> is ignored using <code>\use_none:n</code> .

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

<hr/> <code>__cs_count_signature:N</code> ★	<code>__cs_count_signature:N</code> $\langle function \rangle$
<hr/> <code>__cs_count_signature:c</code> ★	Splits the $\langle function \rangle$ into the $\langle name \rangle$ (<i>i.e.</i> the part before the colon) and the $\langle signature \rangle$ (<i>i.e.</i> after the colon). The $\langle number \rangle$ of tokens in the $\langle signature \rangle$ is then left in the input stream. If there was no $\langle signature \rangle$ then the result is the marker value -1 .
<hr/> <code>__cs_split_function:NN</code> ★	<code>__cs_split_function:NN</code> $\langle function \rangle$ $\langle processor \rangle$
	Splits the $\langle function \rangle$ into the $\langle name \rangle$ (<i>i.e.</i> the part before the colon) and the $\langle signature \rangle$ (<i>i.e.</i> after the colon). This information is then placed in the input stream after the $\langle processor \rangle$ function in three parts: the $\langle name \rangle$, the $\langle signature \rangle$ and a logic token indicating if a colon was found (to differentiate variables from function names). The $\langle name \rangle$ will not include the escape character, and both the $\langle name \rangle$ and $\langle signature \rangle$ are made up of tokens with category code 12 (other). The $\langle processor \rangle$ should be a function with argument specification <code>:nnN</code> (plus any trailing arguments needed).
<hr/> <code>__cs_get_function_name:N</code> ★	<code>__cs_get_function_name:N</code> $\langle function \rangle$
	Splits the $\langle function \rangle$ into the $\langle name \rangle$ (<i>i.e.</i> the part before the colon) and the $\langle signature \rangle$ (<i>i.e.</i> after the colon). The $\langle name \rangle$ is then left in the input stream without the escape character present made up of tokens with category code 12 (other).
<hr/> <code>__cs_get_function_signature:N</code> ★	<code>__cs_get_function_signature:N</code> $\langle function \rangle$
	Splits the $\langle function \rangle$ into the $\langle name \rangle$ (<i>i.e.</i> the part before the colon) and the $\langle signature \rangle$ (<i>i.e.</i> after the colon). The $\langle signature \rangle$ is then left in the input stream made up of tokens with category code 12 (other).
<hr/> <code>__cs_tmp:w</code>	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).
<hr/> <code>__kernel_register_show:N</code>	<code>__kernel_register_show:N</code> $\langle register \rangle$
<hr/> <code>__kernel_register_show:c</code>	Used to show the contents of a T _E X register at the terminal, formatted such that internal parts of the mechanism are not visible.
<hr/> <code>__prg_case_end:nw</code> ★	<code>__prg_case_end:nw</code> $\{ \langle code \rangle \}$ $\langle tokens \rangle$ <code>\q_mark</code> $\{ \langle true code \rangle \}$ <code>\q_mark</code> $\{ \langle false code \rangle \}$ <code>\q_stop</code>
	Used to terminate case statements (<code>\int_case:nnTF</code> , <i>etc.</i>) by removing trailing $\langle tokens \rangle$ and the end marker <code>\q_stop</code> , inserting the $\langle code \rangle$ for the successful case (if one is found) and either the <code>true code</code> or <code>false code</code> for the over all outcome, as appropriate.

Part V

The l3expan package

Argument expansion

This module provides generic methods for expanding T_EX 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 L^AT_EX3 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:No` was not defined it could be coded in the following way:

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

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

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

Providing variants in this way in style files is uncritical as the `\cs_new_nopar:Npn` function will silently accept definitions whenever the new definition is identical to an already given one. 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` $\langle parent\ control\ sequence \rangle$ $\{ \langle variant\ argument\ specifiers \rangle \}$

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

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

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 $\langle parent\ control\ sequence \rangle$ is already defined. If the $\langle parent\ control\ sequence \rangle$ is protected then the new sequence will also be protected. The $\langle variant \rangle$ is created globally, as is any `\exp_args:N` $\langle variant \rangle$ 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 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 `tl`, `num`, `int`, `skip`, `dim`, `toks`, or built-in `TEX` registers. The `v` type is the same except it first creates a control sequence out of its argument before returning the value. This recent addition to the argument specifiers may shake things up a bit as most places where `o` is used will be replaced by `V`. The documentation you are currently reading will therefore require a fair bit of re-writing.

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!

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

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

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

```
\tl_set:N0 \l_tmpa_tl { { \l_tmpa_tl } }
```

and

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

leave `\l_tmpa_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.

<hr/> <hr/>	<code>\exp_args:No</code> ★	<code>\exp_args:No</code> $\langle function \rangle$ $\{\langle tokens \rangle\}$...	This function absorbs two arguments (the $\langle function \rangle$ name and the $\langle tokens \rangle$). The $\langle tokens \rangle$ are expanded once, and the result is inserted in braces into the input stream <i>after</i> reinsertion of the $\langle function \rangle$. Thus the $\langle function \rangle$ may take more than one argument: all others will be left unchanged.
<hr/> <hr/>	<code>\exp_args:Nc</code> ★ <code>\exp_args:cc</code> ★	<code>\exp_args:Nc</code> $\langle function \rangle$ $\{\langle tokens \rangle\}$	This function absorbs two arguments (the $\langle function \rangle$ name and the $\langle tokens \rangle$). The $\langle tokens \rangle$ are expanded until only characters remain, and are then turned into a control sequence. (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 $\langle function \rangle$. Thus the $\langle function \rangle$ may take more than one argument: all others will be left unchanged. The <code>:cc</code> variant constructs the $\langle function \rangle$ name in the same manner as described for the $\langle tokens \rangle$.
<hr/> <hr/>	<code>\exp_args:Nv</code> ★	<code>\exp_args:Nv</code> $\langle function \rangle$ $\langle variable \rangle$	This function absorbs two arguments (the names of the $\langle function \rangle$ and the $\langle variable \rangle$). The content of the $\langle variable \rangle$ are recovered and placed inside braces into the input stream <i>after</i> reinsertion of the $\langle function \rangle$. Thus the $\langle function \rangle$ may take more than one argument: all others will be left unchanged.
<hr/> <hr/>	<code>\exp_args:Nv</code> ★	<code>\exp_args:Nv</code> $\langle function \rangle$ $\{\langle tokens \rangle\}$	This function absorbs two arguments (the $\langle function \rangle$ name and the $\langle tokens \rangle$). The $\langle tokens \rangle$ are expanded until only characters remain, and are then turned into a control sequence. (An internal error will occur if such a conversion is not possible). This control sequence should be the name of a $\langle variable \rangle$. The content of the $\langle variable \rangle$ are recovered and placed inside braces into the input stream <i>after</i> reinsertion of the $\langle function \rangle$. Thus the $\langle function \rangle$ may take more than one argument: all others will be left unchanged.
<hr/> <hr/>	<code>\exp_args:Nf</code> ★	<code>\exp_args:Nf</code> $\langle function \rangle$ $\{\langle tokens \rangle\}$	This function absorbs two arguments (the $\langle function \rangle$ name and the $\langle tokens \rangle$). The $\langle tokens \rangle$ are fully expanded until the first non-expandable token or space is found, and the result is inserted in braces into the input stream <i>after</i> reinsertion of the $\langle function \rangle$. Thus the $\langle function \rangle$ may take more than one argument: all others will be left unchanged.

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

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

5 Manipulating two arguments

<code>\exp_args:NNo</code>	★	<code>\exp_args:NNc</code>	<code>\langle token_1 \rangle \langle token_2 \rangle \{\langle tokens \rangle\}</code>
<code>\exp_args:(NNv NNV NNf Nco Ncf)</code>	★		
<code>\exp_args:NNc</code>	★		
<code>\exp_args:Ncc</code>	★		
<code>\exp_args:NVV</code>	★		

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.

<code>\exp_args:Nno</code>	★	<code>\exp_args:Noo</code>	<code>\langle token \rangle \{\langle tokens_1 \rangle\} \{\langle tokens_2 \rangle\}</code>
<code>\exp_args:(NnV Nnf Noo Nof Nff Nfo)</code>	★		
<code>\exp_args:Noc</code>	★		
<code>\exp_args:Nnc</code>	★		

Updated: 2012-01-14

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

<code>\exp_args:NNx</code>	<code>\exp_args:NNx</code>	<code>\langle token_1 \rangle \langle token_2 \rangle \{\langle tokens \rangle\}</code>
<code>\exp_args:Ncx</code>		
<code>\exp_args:Nnx</code>		
<code>\exp_args:(Nox Nxo Nxx)</code>		

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

<code>\exp_args:NNNo</code>	★	<code>\exp_args:NNNo <token₁> <token₂> <token₃> {<tokens>}</code>
<code>\exp_args:(NNNV NcNo Ncco)</code>	★	
<code>\exp_args:Nccc</code>	★	
<code>\exp_args:NcNc</code>	★	

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

<code>\exp_args:NNoo</code>	★	<code>\exp_args:NNoo <token₁> <token₂> {<token₃>} {<tokens>}</code>
<code>\exp_args:NNno</code>	★	
<code>\exp_args:Nnno</code>	★	
<code>\exp_args:Nooo</code>	★	
<code>\exp_args:Nnnc</code>	★	

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.

<code>\exp_args:NNNx</code>		<code>\exp_args:NNNx <token₁> <token₂> {<tokens₁>} {<tokens₂>}</code>
<code>\exp_args:Nccx</code>		
<code>\exp_args:NNnx</code>		
<code>\exp_args:(NNox Ncnx)</code>		
<code>\exp_args:Nnnx</code>		
<code>\exp_args:(Nnox Noox)</code>		

New: 2015-08-12

7 Unbraced expansion

<code>\exp_last_unbraced:Nf</code>	★	<code>\exp_last_unbraced:Nno</code>	<code><token></code>	<code><tokens₁></code>	<code><tokens₂></code>
<code>\exp_last_unbraced:(NV No Nv)</code>	★				
<code>\exp_last_unbraced:Nco</code>	★				
<code>\exp_last_unbraced:(NcV NNV NNo)</code>	★				
<code>\exp_last_unbraced:Nno</code>	★				
<code>\exp_last_unbraced:(Noo Nfo)</code>	★				
<code>\exp_last_unbraced:NNNV</code>	★				
<code>\exp_last_unbraced:NNNo</code>	★				
<code>\exp_last_unbraced:NnNo</code>	★				

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.

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

<code>\exp_last_unbraced:Nx</code>	<code>\exp_last_unbraced:Nx</code>	<code><function></code>	<code>{<tokens>}</code>
------------------------------------	------------------------------------	-------------------------------	-------------------------------

This functions fully expands the `<tokens>` and leaves the result in the input stream after reinsertion of `<function>`. This function is not expandable.

<code>\exp_last_two_unbraced:Noo</code>	★	<code>\exp_last_two_unbraced:Noo</code>	<code><token></code>	<code><tokens₁></code>	<code>{<tokens₂>}</code>
---	---	---	----------------------------	---	---

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.

<code>\exp_after:wN</code>	★	<code>\exp_after:wN</code>	<code><token₁></code>	<code><token₂></code>
----------------------------	---	----------------------------	--	--

Carries out a single expansion of `<token2>` (which may consume arguments) prior to the expansion of `<token1>`. If `<token2>` is a T_EX 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 T_EX category codes). Unless specifically required, expansion should be carried out using an appropriate argument specifier variant or the appropriate `\exp_arg:N` function.

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

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.

<hr/> <hr/> <code>\exp_not:N</code> ★	<code>\exp_not:N</code> $\langle token \rangle$ Prevents expansion of the $\langle token \rangle$ in a context where it would otherwise be expanded, for example an x -type argument. T_EXhackers note: This is the T _E X <code>\noexpand</code> primitive.
<hr/> <hr/> <code>\exp_not:c</code> ★	<code>\exp_not:c</code> $\{\langle tokens \rangle\}$ Expands the $\langle tokens \rangle$ until only unexpandable content remains, and then converts this into a control sequence. Further expansion of this control sequence is then inhibited.
<hr/> <hr/> <code>\exp_not:n</code> ★	<code>\exp_not:n</code> $\{\langle tokens \rangle\}$ Prevents expansion of the $\langle tokens \rangle$ in a context where they would otherwise be expanded, for example an x -type argument. T_EXhackers note: This is the ε -T _E X <code>\unexpanded</code> primitive. Hence its argument <i>must</i> be surrounded by braces.
<hr/> <hr/> <code>\exp_not:V</code> ★	<code>\exp_not:V</code> $\langle variable \rangle$ Recovers the content of the $\langle variable \rangle$, then prevents expansion of this material in a context where it would otherwise be expanded, for example an x -type argument.
<hr/> <hr/> <code>\exp_not:v</code> ★	<code>\exp_not:v</code> $\{\langle tokens \rangle\}$ Expands the $\langle tokens \rangle$ until only unexpandable content remains, and then converts this into a control sequence (which should be a $\langle variable \rangle$ name). The content of the $\langle variable \rangle$ is recovered, and further expansion is prevented in a context where it would otherwise be expanded, for example an x -type argument.
<hr/> <hr/> <code>\exp_not:o</code> ★	<code>\exp_not:o</code> $\{\langle tokens \rangle\}$ Expands the $\langle tokens \rangle$ once, then prevents any further expansion in a context where they would otherwise be expanded, for example an x -type argument.
<hr/> <hr/> <code>\exp_not:f</code> ★	<code>\exp_not:f</code> $\{\langle tokens \rangle\}$ Expands $\langle tokens \rangle$ 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.

<code>\exp_stop_f:</code> ★	<code>\foo_bar:f { <tokens> \exp_stop_f: <more tokens> }</code>
-----------------------------	---

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 (`_`).

9 Controlled expansion

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

Of course, deep down `TEX` is using expansion as always and there are cases where a programmer needs to control that expansion directly; typical situations are basic data manipulation tools. This section documents the functions for that level. 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!

<code>\exp:w</code> ★	<code>\exp:w <expandable-tokens> \exp_end:</code>
-----------------------	---

`\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.

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

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

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.

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

New: 2015-08-23

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

The difference to `\exp_end_continue_f:w` is that we first we pick up an argument which is then returned to the input stream. If `<further-tokens>` starts with 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_tl`

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.

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

```

\::n \cs_set_nopar:Npn \exp_args:Ncof { \::c \::o \::f \::: }
\::N Internal forms for the base expansion types. These names do not conform to the general
\::p LATEX3 approach as this makes them more readily visible in the log and so forth.
\::c
\::o
\::f
\::x
\::v
\::V
\:::

```

Part VI

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 typical states returned are *⟨true⟩* and *⟨false⟩* but other states are possible, say an *⟨error⟩* state for erroneous input, *e.g.*, text as input in a function comparing integers.

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. Important to note here is that the arguments are executed after exiting the underlying `\if... \fi:` structure.

1 Defining a set of conditional functions

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

Updated: 2012-02-06

These functions create a family of protected conditionals using the same *{⟨code⟩}* to perform the test created. The *⟨code⟩* does not need to be expandable. The **new** version 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:

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

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

An example can easily clarify matters here:

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

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

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

These functions copies a family of conditionals. The `new` version will check for existing definitions (*cf.* `\cs_new:Npn`) whereas the `set` version will not (*cf.* `\cs_set:Npn`). 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`.

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

These ‘return’ functions define the logical state of a conditional statement. They appear within the code for a conditional function generated by `\prg_set_conditional:Npnn`, *etc.*, to indicate when a true or false branch has been 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).

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

<code>\bool_new:N</code>	<code>\bool_new:N \langle boolean \rangle</code>
<code>\bool_new:c</code>	Creates a new <code>\langle boolean \rangle</code> or raises an error if the name is already taken. The declaration is global. The <code>\langle boolean \rangle</code> will initially be <code>false</code> .

<hr/> \bool_set_false:N \bool_set_false:c \bool_gset_false:N \bool_gset_false:c <hr/>	\bool_set_false:N $\langle\text{boolean}\rangle$ Sets $\langle\text{boolean}\rangle$ logically false .
<hr/> \bool_set_true:N \bool_set_true:c \bool_gset_true:N \bool_gset_true:c <hr/>	\bool_set_true:N $\langle\text{boolean}\rangle$ Sets $\langle\text{boolean}\rangle$ logically true .
<hr/> \bool_set_eq:NN \bool_set_eq:(cN Nc cc) \bool_gset_eq:NN \bool_gset_eq:(cN Nc cc) <hr/>	\bool_set_eq:NN $\langle\text{boolean}_1\rangle$ $\langle\text{boolean}_2\rangle$ Sets the content of $\langle\text{boolean}_1\rangle$ equal to that of $\langle\text{boolean}_2\rangle$.
<hr/> \bool_set:Nn \bool_set:cn \bool_gset:Nn \bool_gset:cn <hr/> Updated: 2012-07-08 <hr/>	\bool_set:Nn $\langle\text{boolean}\rangle$ $\{\langle\text{boolexpr}\rangle\}$ Evaluates the $\langle\text{boolean expression}\rangle$ as described for \bool_if:n(TF), and sets the $\langle\text{boolean}\rangle$ variable to the logical truth of this evaluation.
<hr/> \bool_if_p:N ★ \bool_if_p:c ★ \bool_if:NTF ★ \bool_if:cTF ★ <hr/>	\bool_if_p:N $\langle\text{boolean}\rangle$ \bool_if:NTF $\langle\text{boolean}\rangle$ $\{\langle\text{true code}\rangle\}$ $\{\langle\text{false code}\rangle\}$ Tests the current truth of $\langle\text{boolean}\rangle$, and continues expansion based on this result.
<hr/> \bool_show:N \bool_show:c <hr/> New: 2012-02-09 Updated: 2015-08-01 <hr/>	\bool_show:N $\langle\text{boolean}\rangle$ Displays the logical truth of the $\langle\text{boolean}\rangle$ on the terminal.
<hr/> \bool_show:n <hr/> New: 2012-02-09 Updated: 2015-08-07 <hr/>	\bool_show:n $\{\langle\text{boolean expression}\rangle\}$ Displays the logical truth of the $\langle\text{boolean expression}\rangle$ on the terminal.
<hr/> \bool_if_exist_p:N ★ \bool_if_exist_p:c ★ \bool_if_exist:NTF ★ \bool_if_exist:cTF ★ <hr/> New: 2012-03-03 <hr/>	\bool_if_exist_p:N $\langle\text{boolean}\rangle$ \bool_if_exist:NTF $\langle\text{boolean}\rangle$ $\{\langle\text{true code}\rangle\}$ $\{\langle\text{false code}\rangle\}$ Tests whether the $\langle\text{boolean}\rangle$ is currently defined. This does not check that the $\langle\text{boolean}\rangle$ really is a boolean variable.
<hr/> \l_tmpa_bool \l_tmpb_bool <hr/>	A scratch boolean for local assignment. It is never used by the kernel code, and so is safe for use with any L ^A T _E X3-defined function. However, it may be overwritten by other non-kernel code and so should only be used for short-term storage.

<hr/> <code>\g_tmpa_bool</code> <hr/>	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 $\langle true \rangle$ or $\langle false \rangle$ values, it seems only fitting that we also provide a parser for $\langle boolean\ expressions \rangle$.

A boolean expression is an expression which given input in the form of predicate functions and boolean variables, return boolean $\langle true \rangle$ or $\langle false \rangle$. It supports the logical operations And, Or and Not as the well-known infix operators `&&`, `||` and `!` with their usual precedences. 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 } ||
  \int_compare_p:n { 1 = \error } % is skipped
) &&
! ( \int_compare_p:n { 2 = 4 } )
```

is a valid boolean expression. Note that minimal evaluation is carried out whenever possible so that whenever a truth value cannot be changed any more, the remaining tests within the current group are skipped.

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

Updated: 2012-07-08

Tests the current truth of $\langle\textit{boolean expression}\rangle$, and continues expansion based on this result. The $\langle\textit{boolean expression}\rangle$ should consist of a series of predicates or boolean variables with the logical relationship between these defined using `&&` (“And”), `||` (“Or”), `!` (“Not”) and parentheses. Minimal evaluation is used in the processing, so that once a result is defined there is not further expansion of the tests. For example

```
\bool_if_p:n
{
  \int_compare_p:nNn { 1 } = { 1 }
  &&
  (
    \int_compare_p:nNn { 2 } = { 3 } ||
    \int_compare_p:nNn { 4 } = { 4 } ||
    \int_compare_p:nNn { 1 } = { \error } % is skipped
  )
  &&
  ! \int_compare_p:nNn { 2 } = { 4 }
}
```

will be `true` and will not evaluate `\int_compare_p:nNn { 1 } = { \error }`. The logical Not applies to the next predicate or group.

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

Updated: 2012-07-08

Function version of `!(\langle\textit{boolean expression}\rangle)` within a boolean expression.

<code>\bool_xor_p:nn</code>	★	<code>\bool_xor_p:nn {<boolexpr₁>} {<boolexpr₂>}</code>
-----------------------------	---	---

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.

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

Places the $\langle\textit{code}\rangle$ in the input stream for \TeX to process, and then checks the logical value of the $\langle\textit{boolean}\rangle$. If it is `false` then the $\langle\textit{code}\rangle$ will be inserted into the input stream again and the process will loop until the $\langle\textit{boolean}\rangle$ is `true`.

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

Places the $\langle\textit{code}\rangle$ in the input stream for \TeX to process, and then checks the logical value of the $\langle\textit{boolean}\rangle$. If it is `true` then the $\langle\textit{code}\rangle$ will be inserted into the input stream again and the process will loop until the $\langle\textit{boolean}\rangle$ is `false`.

<code>\bool_until_do:Nn</code> ☆ <code>\bool_until_do:cn</code> ☆	<code>\bool_until_do:Nn <boolean> {<code>}</code> <p>This function firsts checks the logical value of the <i><boolean></i>. If it is false the <i><code></i> is placed in the input stream and expanded. After the completion of the <i><code></i> the truth of the <i><boolean></i> is re-evaluated. The process will then loop until the <i><boolean></i> is true.</p>
--	---

<code>\bool_while_do:Nn</code> ☆ <code>\bool_while_do:cn</code> ☆	<code>\bool_while_do:Nn <boolean> {<code>}</code> <p>This function firsts checks the logical value of the <i><boolean></i>. If it is true the <i><code></i> is placed in the input stream and expanded. After the completion of the <i><code></i> the truth of the <i><boolean></i> is re-evaluated. The process will then loop until the <i><boolean></i> is false.</p>
--	---

<code>\bool_do_until:nn</code> ☆ <div style="text-align: right; font-size: small;">Updated: 2012-07-08</div>	<code>\bool_do_until:nn {<boolean expression>} {<code>}</code> <p>Places the <i><code></i> in the input stream for T_EX to process, and then checks the logical value of the <i><boolean expression></i> as described for <code>\bool_if:nTF</code>. If it is false then the <i><code></i> will be inserted into the input stream again and the process will loop until the <i><boolean expression></i> evaluates to true.</p>
---	---

<code>\bool_do_while:nn</code> ☆ <div style="text-align: right; font-size: small;">Updated: 2012-07-08</div>	<code>\bool_do_while:nn {<boolean expression>} {<code>}</code> <p>Places the <i><code></i> in the input stream for T_EX to process, and then checks the logical value of the <i><boolean expression></i> as described for <code>\bool_if:nTF</code>. If it is true then the <i><code></i> will be inserted into the input stream again and the process will loop until the <i><boolean expression></i> evaluates to false.</p>
---	---

<code>\bool_until_do:nn</code> ☆ <div style="text-align: right; font-size: small;">Updated: 2012-07-08</div>	<code>\bool_until_do:nn {<boolean expression>} {<code>}</code> <p>This function firsts checks the logical value of the <i><boolean expression></i> (as described for <code>\bool_if:nTF</code>). If it is false the <i><code></i> is placed in the input stream and expanded. After the completion of the <i><code></i> the truth of the <i><boolean expression></i> is re-evaluated. The process will then loop until the <i><boolean expression></i> is true.</p>
---	--

<code>\bool_while_do:nn</code> ☆ <div style="text-align: right; font-size: small;">Updated: 2012-07-08</div>	<code>\bool_while_do:nn {<boolean expression>} {<code>}</code> <p>This function firsts checks the logical value of the <i><boolean expression></i> (as described for <code>\bool_if:nTF</code>). If it is true the <i><code></i> is placed in the input stream and expanded. After the completion of the <i><code></i> the truth of the <i><boolean expression></i> is re-evaluated. The process will then loop until the <i><boolean expression></i> is false.</p>
---	--

5 Producing multiple copies

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

6 Detecting T_EX's mode

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

Detects if T_EX is currently in horizontal mode.

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

Detects if T_EX is currently in inner mode.

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

Detects if T_EX is currently in maths mode.

Updated: 2011-09-05

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

Detects if T_EX is currently in vertical mode.

7 Primitive conditionals

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

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

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

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

8 Internal programming functions

<code>\group_align_safe_begin:</code>	★	<code>\group_align_safe_begin:</code>
<code>\group_align_safe_end:</code>	★	<code>...</code>
		<code>\group_align_safe_end:</code>

Updated: 2011-08-11

These functions are used to enclose material in a T_EX alignment environment within a specially-constructed group. This group is designed in such a way that it does not add brace groups to the output but does act as a group for the `&` token inside `\halign`. This is necessary to allow grabbing of tokens for testing purposes, as T_EX uses group level to determine the effect of alignment tokens. Without the special grouping, the use of a function such as `\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.

<u>_prg_break_point:Nn</u> ★	_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 <i>not</i> applied: _prg_break_point:Nn is functionally-equivalent in these cases to \\use_ii:nn.
<u>_prg_map_break:Nn</u> ★	_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>.
<u>\\g__prg_map_int</u>	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, <i>etc.</i> , labelled by \\g__prg_map_int hold functions to be mapped over various list datatypes in inline and variable mappings.
<u>_prg_break_point:</u> ★	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.
<u>_prg_break:</u> ★	_prg_break:n {<tokens>} ... _prg_break_point:
<u>_prg_break:n</u> ★	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 VII

The l3quark 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

<u><code>\quark_new:N</code></u>	<code>\quark_new:N <quark></code> Creates a new <code><quark></code> which expands only to <code><quark></code> . The <code><quark></code> will be defined globally, and an error message will be raised if the name was already taken.
<u><code>\q_stop</code></u>	Used as a marker for delimited arguments, such as $\cs_set:Npn \tmp:w \#1\2 \q_stop \{ \#1 \}$
<u><code>\q_mark</code></u>	Used as a marker for delimited arguments when <code>\q_stop</code> is already in use.
<u><code>\q_nil</code></u>	Quark to mark a null value in structured variables or functions. Used as an end delimiter when this may itself may need to be tested (in contrast to <code>\q_stop</code> , which is only ever used as a delimiter).
<u><code>\q_no_value</code></u>	A canonical value for a missing value, when one is requested from a data structure. This is therefore used as a “return” value by functions such as <code>\prop_get:NnN</code> if there is no data to return.

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 later should therefore only be used when the argument can definitely take more than a single token.

<u><code>\quark_if_nil_p:N</code> ★</u>	<code>\quark_if_nil_p:N <token></code>
<u><code>\quark_if_nil:NTF</code> ★</u>	<code>\quark_if_nil:NTF <token> {<true code>} {<false code>}</code>
	Tests if the <code><token></code> is equal to <code>\q_nil</code> .
<u><code>\quark_if_nil_p:n</code> ★</u>	<code>\quark_if_nil_p:n {<token list>}</code>
<u><code>\quark_if_nil_p:(o V)</code> ★</u>	<code>\quark_if_nil:nTF {<token list>} {<true code>} {<false code>}</code>
<u><code>\quark_if_nil:nTF</code> ★</u>	Tests if the <code><token list></code> contains only <code>\q_nil</code> (distinct from <code><token list></code> being empty or containing <code>\q_nil</code> plus one or more other tokens).
<u><code>\quark_if_nil:(o V)TF</code> ★</u>	
<u><code>\quark_if_no_value_p:N</code> ★</u>	<code>\quark_if_no_value_p:N <token></code>
<u><code>\quark_if_no_value_p:c</code> ★</u>	<code>\quark_if_no_value:NTF <token> {<true code>} {<false code>}</code>
<u><code>\quark_if_no_value:NTF</code> ★</u>	Tests if the <code><token></code> is equal to <code>\q_no_value</code> .
<u><code>\quark_if_no_value:cTF</code> ★</u>	
<u><code>\quark_if_no_value_p:n</code> ★</u>	<code>\quark_if_no_value_p:n {<token list>}</code>
<u><code>\quark_if_no_value:nTF</code> ★</u>	<code>\quark_if_no_value:NTF {<token list>} {<true code>} {<false code>}</code>
	Tests if the <code><token list></code> contains only <code>\q_no_value</code> (distinct from <code><token list></code> being empty or containing <code>\q_no_value</code> plus one or more other tokens).

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.

<code>\q_recursion_tail</code>	This quark is appended to the data structure in question and appears as a real element there. This means it gets any list separators around it.
--------------------------------	---

<code>\q_recursion_stop</code>	This quark is added <i>after</i> the data structure. Its purpose is to make it possible to terminate the recursion at any point easily.
--------------------------------	---

<code>\quark_if_recursion_tail_stop:N</code>	<code>\quark_if_recursion_tail_stop:N <token></code>
--	--

Tests if $\langle token \rangle$ contains only the marker `\q_recursion_tail`, and if so terminates the recursion this is part of using `\use_none_delimit_by_q_recursion_stop:w`. The recursion input must include the marker tokens `\q_recursion_tail` and `\q_recursion_stop` as the last two items.

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

Updated: 2011-09-06

Tests if the $\langle token list \rangle$ contains only `\q_recursion_tail`, and if so terminates the recursion this is part of using `\use_none_delimit_by_q_recursion_stop:w`. The recursion input must include the marker tokens `\q_recursion_tail` and `\q_recursion_stop` as the last two items.

<code>\quark_if_recursion_tail_stop_do:Nn</code>	<code>\quark_if_recursion_tail_stop_do:Nn <token> {<insertion>}</code>
--	--

Tests if $\langle token \rangle$ contains only the marker `\q_recursion_tail`, and if so terminates the recursion this is part of using `\use_none_delimit_by_q_recursion_stop:w`. The recursion input must include the marker tokens `\q_recursion_tail` and `\q_recursion_stop` as the last two items. The $\langle insertion \rangle$ code is then added to the input stream after the recursion has ended.

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

Updated: 2011-09-06

Tests if the $\langle token list \rangle$ contains only `\q_recursion_tail`, and if so terminates the recursion this is part of using `\use_none_delimit_by_q_recursion_stop:w`. The recursion input must include the marker tokens `\q_recursion_tail` and `\q_recursion_stop` as the last two items. The $\langle insertion \rangle$ code is then added to the input stream after the recursion has ended.

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

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

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 `TEX` 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.

<code>__scan_new:N</code>	<code>__scan_new:N</code> $\langle scan\ mark \rangle$
----------------------------	---

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

<code>\s__stop</code>	Used at the end of a set of instructions, as a marker that can be jumped to using <code>__use_none_delimit_by_s__stop:w</code> .
-----------------------	---

<code>__use_none_delimit_by_s__stop:w</code>	<code>__use_none_delimit_by_s__stop:w</code> $\langle tokens \rangle$ <code>\s__stop</code>
---	--

Removes the $\langle tokens \rangle$ and `\s__stop` from the input stream. This leads to a low-level `TEX` error if `\s__stop` is absent.

Part VIII

The l3token package

Token manipulation

This module deals with tokens. Now this is perhaps not the most precise description so let's try with a better description: When programming in T_EX, it is often desirable to know just what a certain token is: is it a control sequence or something else. Similarly one often needs to know if a control sequence is expandable or not, a macro or a primitive, how many arguments it takes etc. Another thing of great importance (especially when it comes to document commands) is looking ahead in the token stream to see if a certain character is present and maybe even remove it or disregard other tokens while scanning. This module provides functions for both and as such will have two primary function categories: `\token_` for anything that deals with tokens and `\peek_` for looking ahead in the token stream.

Most of the time we will be using the term “token” but most of the time the function we're describing can equally well be used on a control sequence as such one is one token as well.

We shall refer to list of tokens as `tlists` and such lists represented by a single control sequence is a “token list variable” `tl var`. Functions for these two types are found in the `l3tl` module.

1 All possible tokens

Let us start by reviewing every case that a given token can fall into. It is very important to distinguish two aspects of a token: its meaning, and what it looks like.

For instance, `\if:w`, `\if_charcode:w`, and `\tex_if:D` are three names for the same internal operation of T_EX, namely the primitive testing the next two characters for equality of their character code. They behave identically in many situations. However, T_EX distinguishes them when searching for a delimited argument. Namely, the example function `\show_until_if:w` defined below 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
```

2 Character tokens

<code>\char_set_catcode_escape:N</code>	<code>\char_set_catcode_letter:N</code> $\langle character \rangle$
<code>\char_set_catcode_group_begin:N</code>	
<code>\char_set_catcode_group_end:N</code>	
<code>\char_set_catcode_math_toggle:N</code>	
<code>\char_set_catcode_alignment:N</code>	
<code>\char_set_catcode_end_line:N</code>	
<code>\char_set_catcode_parameter:N</code>	
<code>\char_set_catcode_math_superscript:N</code>	
<code>\char_set_catcode_math_subscript:N</code>	
<code>\char_set_catcode_ignore:N</code>	
<code>\char_set_catcode_space:N</code>	
<code>\char_set_catcode_letter:N</code>	
<code>\char_set_catcode_other:N</code>	
<code>\char_set_catcode_active:N</code>	
<code>\char_set_catcode_comment:N</code>	
<code>\char_set_catcode_invalid:N</code>	

Updated: 2015-08-09

Sets the category code of the $\langle character \rangle$ to that indicated in the function name. Depending on the current category code of the $\langle token \rangle$ the escape token may also be needed:

`\char_set_catcode_other:N` $\backslash\%$

The assignment is local. These commands update `\l_char_special_seq` as appropriate by adding or removing the $\langle character \rangle$, and in package mode they also update `\dospecials` (plain $\mathrm{T}_{\mathrm{E}}\mathrm{X}$ and $\mathrm{L}^{\mathrm{A}}\mathrm{T}_{\mathrm{E}}\mathrm{X} 2_{\varepsilon}$) and `\@sanitize` ($\mathrm{L}^{\mathrm{A}}\mathrm{T}_{\mathrm{E}}\mathrm{X} 2_{\varepsilon}$ only). The `\char_set_catcode_active:N` command also adds the active character to `\l_char_active_seq`.

<code>\char_set_catcode_escape:n</code>	<code>\char_set_catcode_letter:n {⟨integer expression⟩}</code>
<code>\char_set_catcode_group_begin:n</code>	
<code>\char_set_catcode_group_end:n</code>	
<code>\char_set_catcode_math_toggle:n</code>	
<code>\char_set_catcode_alignment:n</code>	
<code>\char_set_catcode_end_line:n</code>	
<code>\char_set_catcode_parameter:n</code>	
<code>\char_set_catcode_math_superscript:n</code>	
<code>\char_set_catcode_math_subscript:n</code>	
<code>\char_set_catcode_ignore:n</code>	
<code>\char_set_catcode_space:n</code>	
<code>\char_set_catcode_letter:n</code>	
<code>\char_set_catcode_other:n</code>	
<code>\char_set_catcode_active:n</code>	
<code>\char_set_catcode_comment:n</code>	
<code>\char_set_catcode_invalid:n</code>	

Updated: 2015-08-09

Sets the category code of the $\langle character \rangle$ which has character code as given by the $\langle integer expression \rangle$. This version can be used to set up characters which cannot otherwise be given (*cf.* the N-type variants). The assignment is local. These commands update `\l_char_special_seq` as appropriate by adding or removing the $\langle character \rangle$, and in package mode they also update `\dospecials` (plain T_EX and L^AT_EX 2_ε) and `\@sanitize` (L^AT_EX 2_ε only). The `\char_set_catcode_active:n` command also adds the active character to `\l_char_active_seq`.

<code>\char_set_catcode:nn</code>	<code>\char_set_catcode:nn {⟨intexpr₁⟩} {⟨intexpr₂⟩}</code>
-----------------------------------	---

Updated: 2015-08-09

These functions set the category code of the $\langle character \rangle$ which has character code as given by the $\langle integer expression \rangle$. The first $\langle integer expression \rangle$ is the character code and the second is the category code to apply. The setting applies within the current T_EX 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. This command updates `\l_char_special_seq` as appropriate by adding or removing the $\langle character \rangle$, and in package mode it also updates `\dospecials` (plain T_EX and L^AT_EX 2_ε) and `\@sanitize` (L^AT_EX 2_ε only). If the category code is 13 (active), the active character is added to `\l_char_active_seq`.

<code>\char_value_catcode:n</code> ★	<code>\char_value_catcode:n {⟨integer expression⟩}</code>
--------------------------------------	---

Expands to the current category code of the $\langle character \rangle$ with character code given by the $\langle integer expression \rangle$.

<code>\char_show_value_catcode:n</code>	<code>\char_show_value_catcode:n {⟨integer expression⟩}</code>
---	--

Displays the current category code of the $\langle character \rangle$ with character code given by the $\langle integer expression \rangle$ on the terminal.

<hr/> <code>\char_set_lccode:nn</code> <hr/>	<code>\char_set_lccode:nn {⟨intexpr₁⟩} {⟨intexpr₂⟩}</code>
Updated: 2015-08-06	Sets up the behaviour of the $\langle character \rangle$ when found inside <code>\tl_to_lowercase:n</code> , such that $\langle character_1 \rangle$ will be converted into $\langle character_2 \rangle$. The two $\langle characters \rangle$ may be specified using an $\langle integer expression \rangle$ for the character code concerned. This may include the T _E X ‘ $\langle character \rangle$ ’ method for converting a single character into its character code:
	<pre> \char_set_lccode:nn { ‘\A } { ‘\a } % Standard behaviour \char_set_lccode:nn { ‘\A } { ‘\A + 32 } \char_set_lccode:nn { 50 } { 60 } </pre>
	The setting applies within the current T _E X group.
<hr/> <code>\char_value_lccode:n</code> ★ <hr/>	<code>\char_value_lccode:n {⟨integer expression⟩}</code>
	Expands to the current lower case code of the $\langle character \rangle$ with character code given by the $\langle integer expression \rangle$.
<hr/> <code>\char_show_value_lccode:n</code> <hr/>	<code>\char_show_value_lccode:n {⟨integer expression⟩}</code>
	Displays the current lower case code of the $\langle character \rangle$ with character code given by the $\langle integer expression \rangle$ on the terminal.
<hr/> <code>\char_set_uccode:nn</code> <hr/>	<code>\char_set_uccode:nn {⟨intexpr₁⟩} {⟨intexpr₂⟩}</code>
Updated: 2015-08-06	Sets up the behaviour of the $\langle character \rangle$ when found inside <code>\tl_to_uppercase:n</code> , such that $\langle character_1 \rangle$ will be converted into $\langle character_2 \rangle$. The two $\langle characters \rangle$ may be specified using an $\langle integer expression \rangle$ for the character code concerned. This may include the T _E X ‘ $\langle character \rangle$ ’ method for converting a single character into its character code:
	<pre> \char_set_uccode:nn { ‘\a } { ‘\A } % Standard behaviour \char_set_uccode:nn { ‘\A } { ‘\A - 32 } \char_set_uccode:nn { 60 } { 50 } </pre>
	The setting applies within the current T _E X group.
<hr/> <code>\char_value_uccode:n</code> ★ <hr/>	<code>\char_value_uccode:n {⟨integer expression⟩}</code>
	Expands to the current upper case code of the $\langle character \rangle$ with character code given by the $\langle integer expression \rangle$.
<hr/> <code>\char_show_value_uccode:n</code> <hr/>	<code>\char_show_value_uccode:n {⟨integer expression⟩}</code>
	Displays the current upper case code of the $\langle character \rangle$ with character code given by the $\langle integer expression \rangle$ on the terminal.
<hr/> <code>\char_set_mathcode:nn</code> <hr/>	<code>\char_set_mathcode:nn {⟨intexpr₁⟩} {⟨intexpr₂⟩}</code>
Updated: 2015-08-06	This function sets up the math code of $\langle character \rangle$. The $\langle character \rangle$ is specified as an $\langle integer expression \rangle$ which will be used as the character code of the relevant character. The setting applies within the current T _E X group.

<hr/> <hr/>	<hr/>
<code>\char_value_mathcode:n</code> ★	<code>\char_value_mathcode:n {\langle integer expression \rangle}</code>
	Expands to the current math code of the $\langle character \rangle$ with character code given by the $\langle integer expression \rangle$.
<hr/>	<hr/>
<code>\char_show_value_mathcode:n</code>	<code>\char_show_value_mathcode:n {\langle integer expression \rangle}</code>
	Displays the current math code of the $\langle character \rangle$ with character code given by the $\langle integer expression \rangle$ on the terminal.
<hr/>	<hr/>
<code>\char_set_sfcode:nn</code>	<code>\char_set_sfcode:nn {\langle intexpr_1 \rangle} {\langle intexpr_2 \rangle}</code>
Updated: 2015-08-06	This function sets up the space factor for the $\langle character \rangle$. The $\langle character \rangle$ is specified as an $\langle integer expression \rangle$ which will be used as the character code of the relevant character. The setting applies within the current T _E X group.
<hr/>	<hr/>
<code>\char_value_sfcode:n</code> ★	<code>\char_value_sfcode:n {\langle integer expression \rangle}</code>
	Expands to the current space factor for the $\langle character \rangle$ with character code given by the $\langle integer expression \rangle$.
<hr/>	<hr/>
<code>\char_show_value_sfcode:n</code>	<code>\char_show_value_sfcode:n {\langle integer expression \rangle}</code>
	Displays the current space factor for the $\langle character \rangle$ with character code given by the $\langle integer expression \rangle$ on the terminal.
<hr/>	<hr/>
<code>\l_char_active_seq</code>	Used to track which tokens may require special handling at the document level as they are (or have been at some point) of category $\langle active \rangle$ (catcode 13). Each entry in the sequence consists of a single active character. Active tokens should be added to the sequence when they are defined for general document use. This sequence is automatically updated by <code>\char_set_catcode:nn</code> and more specific functions.
New: 2012-01-23	
Updated: 2015-08-09	
<hr/>	<hr/>
<code>\l_char_special_seq</code>	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 $\langle letter \rangle$ (catcode 11) or $\langle other \rangle$ (catcode 12). Each entry in the sequence consists of a single escaped token, for example <code>\</code> for the backslash or <code>\{</code> for an opening brace. Escaped tokens should be added to the sequence when they are defined for general document use. This sequence is automatically updated by <code>\char_set_catcode:nn</code> and more specific functions.
New: 2012-01-23	
Updated: 2015-08-09	

3 Generic tokens

<hr/> <hr/>	<hr/>
<code>\token_new:Nn</code>	<code>\token_new:Nn \langle token_1 \rangle {\langle token_2 \rangle}</code>
	Defines $\langle token_1 \rangle$ to globally be a snapshot of $\langle token_2 \rangle$. This will be an implicit representation of $\langle token_2 \rangle$.

```

\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_tl

```

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:c ★

```

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

Inserts the current meaning of the $\langle token \rangle$ into the input stream as a series of characters of category code 12 (other). This will be the primitive \TeX description of the $\langle token \rangle$, thus for example both functions defined by `\cs_set_nopar:Npn` and token list variables defined using `\tl_new:N` will be described as macros.

\TeX hackers note: This is the \TeX primitive `\meaning`.

```

\token_to_str:N ★
\token_to_str:c ★

```

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

Converts the given $\langle token \rangle$ 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 $\langle token \rangle$). This function requires only a single expansion.

\TeX hackers 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 \langle token \rangle
\token_if_group_begin:NTF ★ \token_if_group_begin:NTF \langle token \rangle {\langle true code \rangle} {\langle false code \rangle}

```

Tests if $\langle token \rangle$ has the category code of a begin group token (`{` when normal \TeX category codes are in force). Note that an explicit begin group token cannot be tested in this way, as it is not a valid N-type argument.

<code>\token_if_group_end_p:N</code>	★	<code>\token_if_group_end_p:N</code>	$\langle token \rangle$
<code>\token_if_group_end:NTF</code>	★	<code>\token_if_group_end:NTF</code>	$\langle token \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Tests if $\langle token \rangle$ has the category code of an end group token (`}` when normal T_EX category codes are in force). Note that an explicit end group token cannot be tested in this way, as it is not a valid N-type argument.

<code>\token_if_math_toggle_p:N</code>	★	<code>\token_if_math_toggle_p:N</code>	$\langle token \rangle$
<code>\token_if_math_toggle:NTF</code>	★	<code>\token_if_math_toggle:NTF</code>	$\langle token \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Tests if $\langle token \rangle$ has the category code of a math shift token (`$` when normal T_EX category codes are in force).

<code>\token_if_alignment_p:N</code>	★	<code>\token_if_alignment_p:N</code>	$\langle token \rangle$
<code>\token_if_alignment:NTF</code>	★	<code>\token_if_alignment:NTF</code>	$\langle token \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Tests if $\langle token \rangle$ has the category code of an alignment token (`&` when normal T_EX category codes are in force).

<code>\token_if_parameter_p:N</code>	★	<code>\token_if_parameter_p:N</code>	$\langle token \rangle$
<code>\token_if_parameter:NTF</code>	★	<code>\token_if_parameter:NTF</code>	$\langle token \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Tests if $\langle token \rangle$ has the category code of a macro parameter token (`#` when normal T_EX category codes are in force).

<code>\token_if_math_superscript_p:N</code>	★	<code>\token_if_math_superscript_p:N</code>	$\langle token \rangle$
<code>\token_if_math_superscript:NTF</code>	★	<code>\token_if_math_superscript:NTF</code>	$\langle token \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Tests if $\langle token \rangle$ has the category code of a superscript token (`^` when normal T_EX category codes are in force).

<code>\token_if_math_subscript_p:N</code>	★	<code>\token_if_math_subscript_p:N</code>	$\langle token \rangle$
<code>\token_if_math_subscript:NTF</code>	★	<code>\token_if_math_subscript:NTF</code>	$\langle token \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Tests if $\langle token \rangle$ has the category code of a subscript token (`_` when normal T_EX category codes are in force).

<code>\token_if_space_p:N</code>	★	<code>\token_if_space_p:N</code>	$\langle token \rangle$
<code>\token_if_space:NTF</code>	★	<code>\token_if_space:NTF</code>	$\langle token \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Tests if $\langle token \rangle$ 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</code>	$\langle token \rangle$
<code>\token_if_letter:NTF</code>	★	<code>\token_if_letter:NTF</code>	$\langle token \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Tests if $\langle token \rangle$ has the category code of a letter token.

<code>\token_if_other_p:N</code>	★	<code>\token_if_other_p:N</code>	$\langle token \rangle$
<code>\token_if_other:NTF</code>	★	<code>\token_if_other:NTF</code>	$\langle token \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Tests if $\langle token \rangle$ has the category code of an “other” token.

<code>\token_if_active_p:N</code>	★	<code>\token_if_active_p:N</code>	$\langle token \rangle$
<code>\token_if_active:NTF</code>	★	<code>\token_if_active:NTF</code>	$\langle token \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Tests if $\langle token \rangle$ has the category code of an active character.

<code>\token_if_eq_catcode_p:NN</code>	★	<code>\token_if_eq_catcode_p:NN</code>	$\langle token_1 \rangle$ $\langle token_2 \rangle$
<code>\token_if_eq_catcode:NNTF</code>	★	<code>\token_if_eq_catcode:NNTF</code>	$\langle token_1 \rangle$ $\langle token_2 \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Tests if the two $\langle tokens \rangle$ have the same category code.

<code>\token_if_eq_charcode_p:NN</code>	★	<code>\token_if_eq_charcode_p:NN</code>	$\langle token_1 \rangle$ $\langle token_2 \rangle$
<code>\token_if_eq_charcode:NNTF</code>	★	<code>\token_if_eq_charcode:NNTF</code>	$\langle token_1 \rangle$ $\langle token_2 \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Tests if the two $\langle tokens \rangle$ have the same character code.

<code>\token_if_eq_meaning_p:NN</code>	★	<code>\token_if_eq_meaning_p:NN</code>	$\langle token_1 \rangle$ $\langle token_2 \rangle$
<code>\token_if_eq_meaning:NNTF</code>	★	<code>\token_if_eq_meaning:NNTF</code>	$\langle token_1 \rangle$ $\langle token_2 \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Tests if the two $\langle tokens \rangle$ have the same meaning when expanded.

<code>\token_if_macro_p:N</code>	★	<code>\token_if_macro_p:N</code>	$\langle token \rangle$
<code>\token_if_macro:NTF</code>	★	<code>\token_if_macro:NTF</code>	$\langle token \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Updated: 2011-05-23

Tests if the $\langle token \rangle$ is a T_EX macro.

<code>\token_if_cs_p:N</code>	★	<code>\token_if_cs_p:N</code>	$\langle token \rangle$
<code>\token_if_cs:NTF</code>	★	<code>\token_if_cs:NTF</code>	$\langle token \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Tests if the $\langle token \rangle$ is a control sequence.

<code>\token_if_expandable_p:N</code>	★	<code>\token_if_expandable_p:N</code>	$\langle token \rangle$
<code>\token_if_expandable:NTF</code>	★	<code>\token_if_expandable:NTF</code>	$\langle token \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Tests if the $\langle token \rangle$ is expandable. This test returns $\langle false \rangle$ for an undefined token.

<code>\token_if_long_macro_p:N</code>	★	<code>\token_if_long_macro_p:N</code>	$\langle token \rangle$
<code>\token_if_long_macro:NTF</code>	★	<code>\token_if_long_macro:NTF</code>	$\langle token \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Updated: 2012-01-20

Tests if the $\langle token \rangle$ is a long macro.

<code>\token_if_protected_macro_p:N</code>	★	<code>\token_if_protected_macro_p:N</code>	$\langle token \rangle$
<code>\token_if_protected_macro:NTF</code>	★	<code>\token_if_protected_macro:NTF</code>	$\langle token \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Updated: 2012-01-20

Tests if the $\langle token \rangle$ is a protected macro: a macro which is both protected and long will return logical **false**.

<code>\token_if_protected_long_macro_p:N</code>	★	<code>\token_if_protected_long_macro_p:N</code>	$\langle token \rangle$
<code>\token_if_protected_long_macro:NTF</code>	★	<code>\token_if_protected_long_macro:NTF</code>	$\langle token \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Updated: 2012-01-20

Tests if the $\langle token \rangle$ is a protected long macro.

<code>\token_if_chardef_p:N</code>	★	<code>\token_if_chardef_p:N</code>	$\langle token \rangle$
<code>\token_if_chardef:NTF</code>	★	<code>\token_if_chardef:NTF</code>	$\langle token \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Updated: 2012-01-20

Tests if the $\langle token \rangle$ is defined to be a chardef.

T_EXhackers note: Booleans, boxes and small integer constants are implemented as chardefs.

<code>\token_if_mathchardef_p:N</code>	★	<code>\token_if_mathchardef_p:N</code>	$\langle token \rangle$
<code>\token_if_mathchardef:NTF</code>	★	<code>\token_if_mathchardef:NTF</code>	$\langle token \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Updated: 2012-01-20

Tests if the $\langle token \rangle$ is defined to be a mathchardef.

<code>\token_if_dim_register_p:N</code>	★	<code>\token_if_dim_register_p:N</code>	$\langle token \rangle$
<code>\token_if_dim_register:NTF</code>	★	<code>\token_if_dim_register:NTF</code>	$\langle token \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Updated: 2012-01-20

Tests if the $\langle token \rangle$ is defined to be a dimension register.

<code>\token_if_int_register_p:N</code>	★	<code>\token_if_int_register_p:N</code>	$\langle token \rangle$
<code>\token_if_int_register:NTF</code>	★	<code>\token_if_int_register:NTF</code>	$\langle token \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Updated: 2012-01-20

Tests if the $\langle token \rangle$ is defined to be an integer register.

T_EXhackers note: Constant integers may be implemented as integer registers, chardefs, or mathchardefs depending on their value.

<code>\token_if_muskip_register_p:N</code>	★	<code>\token_if_muskip_register_p:N</code>	$\langle token \rangle$
<code>\token_if_muskip_register:NTF</code>	★	<code>\token_if_muskip_register:NTF</code>	$\langle token \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

New: 2012-02-15

Tests if the $\langle token \rangle$ is defined to be a muskip register.

<code>\token_if_skip_register_p:N</code>	★	<code>\token_if_skip_register_p:N</code>	$\langle token \rangle$
<code>\token_if_skip_register:NTF</code>	★	<code>\token_if_skip_register:NTF</code>	$\langle token \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Updated: 2012-01-20

Tests if the $\langle token \rangle$ is defined to be a skip register.

<code>\token_if_toks_register_p:N</code>	★	<code>\token_if_toks_register_p:N</code>	$\langle token \rangle$
<code>\token_if_toks_register:NTF</code>	★	<code>\token_if_toks_register:NTF</code>	$\langle token \rangle$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$

Updated: 2012-01-20

Tests if the $\langle token \rangle$ is defined to be a toks register (not used by L^AT_EX3).

<code>\token_if_primitive_p:N</code>	★	<code>\token_if_primitive_p:N</code>	$\langle token \rangle$
<code>\token_if_primitive:NTF</code>	★	<code>\token_if_primitive:NTF</code>	$\langle token \rangle$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$

Updated: 2011-05-23

Tests if the $\langle token \rangle$ 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.

<code>\peek_after:Nw</code>	<code>\peek_after:Nw</code>	$\langle function \rangle$ $\langle token \rangle$
-----------------------------	-----------------------------	--

Locally sets the test variable `\l_peek_token` equal to $\langle token \rangle$ (as an implicit token, *not* as a token list), and then expands the $\langle function \rangle$. The $\langle token \rangle$ will remain in the input stream as the next item after the $\langle function \rangle$. The $\langle token \rangle$ here may be \sqcup , $\{$ or $\}$ (assuming normal T_EX category codes), *i.e.* it is not necessarily the next argument which would be grabbed by a normal function.

<code>\peek_gafter:Nw</code>	<code>\peek_gafter:Nw</code>	$\langle function \rangle$ $\langle token \rangle$
------------------------------	------------------------------	--

Globally sets the test variable `\g_peek_token` equal to $\langle token \rangle$ (as an implicit token, *not* as a token list), and then expands the $\langle function \rangle$. The $\langle token \rangle$ will remain in the input stream as the next item after the $\langle function \rangle$. The $\langle token \rangle$ here may be \sqcup , $\{$ or $\}$ (assuming normal T_EX category codes), *i.e.* it is not necessarily the next argument which would be grabbed by a normal function.

<code>\l_peek_token</code>	Token set by <code>\peek_after:Nw</code> and available for testing as described above.
----------------------------	--

<code>\g_peek_token</code>	Token set by <code>\peek_gafter:Nw</code> and available for testing as described above.
----------------------------	---

<code>\peek_catcode:NTF</code>	<code>\peek_catcode:NTF</code>	$\langle test\ token \rangle$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$
--------------------------------	--------------------------------	--

Updated: 2012-12-20

Tests if the next $\langle token \rangle$ in the input stream has the same category code as the $\langle test\ token \rangle$ (as defined by the test `\token_if_eq_catcode:NNTF`). Spaces are respected by the test and 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).

<code>\peek_catcode_ignore_spaces:NTF</code>	<code>\peek_catcode_ignore_spaces:NTF <test token> {\<true code>} {\<>false code>}</code>
--	--

Updated: 2012-12-20

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).

<code>\peek_catcode_remove:NTF</code>	<code>\peek_catcode_remove:NTF <test token> {\<true code>} {\<>false code>}</code>
---------------------------------------	---

Updated: 2012-12-20

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).

<code>\peek_catcode_remove_ignore_spaces:NTF</code>	<code>\peek_catcode_remove_ignore_spaces:NTF <test token> {\<true code>} {\<>false code>}</code>
---	---

Updated: 2012-12-20

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).

<code>\peek_charcode:NTF</code>	<code>\peek_charcode:NTF <test token> {\<true code>} {\<>false code>}</code>
---------------------------------	---

Updated: 2012-12-20

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).

<code>\peek_charcode_ignore_spaces:NTF</code>	<code>\peek_charcode_ignore_spaces:NTF <test token> {\<true code>} {\<>false code>}</code>
---	---

Updated: 2012-12-20

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).

<u>\peek_charcode_remove:NTF</u>	\peek_charcode_remove:NTF <i><test token></i> { <i><true code></i> } { <i><false code></i> }
Updated: 2012-12-20	Tests if the next <i><token></i> in the input stream has the same character code as the <i><test token></i> (as defined by the test \token_if_eq_charcode:NTTF). Spaces are respected by the test and the <i><token></i> will be removed from the input stream if the test is true. The function will then place either the <i><true code></i> or <i><false code></i> in the input stream (as appropriate to the result of the test).
<u>\peek_charcode_remove_ignore_spaces:NTF</u>	\peek_charcode_remove_ignore_spaces:NTF <i><test token></i> { <i><true code></i> } { <i><false code></i> }
Updated: 2012-12-20	Tests if the next non-space <i><token></i> in the input stream has the same character code as the <i><test token></i> (as defined by the test \token_if_eq_charcode:NTTF). Explicit and implicit space tokens (with character code 32 and category code 10) are ignored and removed by the test and the <i><token></i> will be removed from the input stream if the test is true. The function will then place either the <i><true code></i> or <i><false code></i> in the input stream (as appropriate to the result of the test).
<u>\peek_meaning:NTF</u>	\peek_meaning:NTF <i><test token></i> { <i><true code></i> } { <i><false code></i> }
Updated: 2011-07-02	Tests if the next <i><token></i> in the input stream has the same meaning as the <i><test token></i> (as defined by the test \token_if_eq_meaning:NTTF). Spaces are respected by the test and the <i><token></i> will be left in the input stream after the <i><true code></i> or <i><false code></i> (as appropriate to the result of the test).
<u>\peek_meaning_ignore_spaces:NTF</u>	\peek_meaning_ignore_spaces:NTF <i><test token></i> { <i><true code></i> } { <i><false code></i> }
Updated: 2012-12-05	Tests if the next non-space <i><token></i> in the input stream has the same meaning as the <i><test token></i> (as defined by the test \token_if_eq_meaning:NTTF). Explicit and implicit space tokens (with character code 32 and category code 10) are ignored and removed by the test and the <i><token></i> will be left in the input stream after the <i><true code></i> or <i><false code></i> (as appropriate to the result of the test).
<u>\peek_meaning_remove:NTF</u>	\peek_meaning_remove:NTF <i><test token></i> { <i><true code></i> } { <i><false code></i> }
Updated: 2011-07-02	Tests if the next <i><token></i> in the input stream has the same meaning as the <i><test token></i> (as defined by the test \token_if_eq_meaning:NTTF). Spaces are respected by the test and the <i><token></i> will be removed from the input stream if the test is true. The function will then place either the <i><true code></i> or <i><false code></i> in the input stream (as appropriate to the result of the test).

<code>\peek_meaning_remove_ignore_spaces:NTF</code>	<code>\peek_meaning_remove_ignore_spaces:NTF</code> $\langle test\ token \rangle$ $\{\langle true\ code \rangle\} \{\langle false\ code \rangle\}$
---	---

Updated: 2012-12-05

Tests if the next non-space $\langle token \rangle$ in the input stream has the same meaning as the $\langle test\ token \rangle$ (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 $\langle token \rangle$ will be removed from the input stream if the test is true. The function will then place either the $\langle true\ code \rangle$ or $\langle false\ code \rangle$ 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.

<code>\token_get_arg_spec:N</code> ★	<code>\token_get_arg_spec:N</code> $\langle token \rangle$
--------------------------------------	--

If the $\langle token \rangle$ is a macro, this function will leave the primitive \TeX 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 #1#2 { 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.

\TeX hackers note: If the arg spec. contains the string `->`, then the `spec` function will produce incorrect results.

<code>\token_get_replacement_spec:N</code> ★	<code>\token_get_replacement_spec:N</code> $\langle token \rangle$
--	--

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 #1#2 { x #1~y #2 }`

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

\TeX hackers 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` $\langle token \rangle$

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

Part IX

The l3int package

Integers

Calculation and comparison of integer values can be carried out using literal numbers, `int` registers, constants and integers stored in token list variables. The standard operators `+`, `-`, `/` and `*` and parentheses can be used within such expressions to carry arithmetic operations. This module carries out these functions on *integer expressions* (“`intexpr`”).

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.

T_EXhackers 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 T_EX-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.

<hr/> <code>\int_div_round:nn</code> ★ <hr/>	<code>\int_div_round:nn {\langle integer \rangle_1} {\langle integer \rangle_2}</code>
Updated: 2012-09-26	Evaluates the two $\langle integer expressions \rangle$ as described earlier, then 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 <code>/</code> directly in an $\langle integer expression \rangle$. The result is left in the input stream as an $\langle integer denotation \rangle$ after two expansions.

<hr/> <code>\int_div_truncate:nn</code> ★ <hr/>	<code>\int_div_truncate:nn {\langle integer \rangle_1} {\langle integer \rangle_2}</code>
Updated: 2012-02-09	Evaluates the two $\langle integer expressions \rangle$ as described earlier, then divides the first value by the second, and rounds the result towards zero. Note that division using <code>/</code> rounds to the closest integer instead. The result is left in the input stream as an $\langle integer denotation \rangle$ after two expansions.

<hr/> <code>\int_max:nn</code> ★	<code>\int_max:nn {\langle integer \rangle_1} {\langle integer \rangle_2}</code>
<code>\int_min:nn</code> ★	<code>\int_min:nn {\langle integer \rangle_1} {\langle integer \rangle_2}</code>
Updated: 2012-09-26	Evaluates the $\langle integer expressions \rangle$ as described for <code>\int_eval:n</code> and leaves either the larger or smaller value in the input stream as an $\langle integer denotation \rangle$ after two expansions.

<hr/> <code>\int_mod:nn</code> ★ <hr/>	<code>\int_mod:nn {\langle integer \rangle_1} {\langle integer \rangle_2}</code>
Updated: 2012-09-26	Evaluates the two $\langle integer expressions \rangle$ as described earlier, then calculates the integer remainder of dividing the first expression by the second. This is obtained by subtracting <code>\int_div_truncate:nn {\langle integer \rangle_1} {\langle integer \rangle_2}</code> times $\langle integer \rangle_2$ from $\langle integer \rangle_1$. Thus, the result has the same sign as $\langle integer \rangle_1$ and its absolute value is strictly less than that of $\langle integer \rangle_2$. The result is left in the input stream as an $\langle integer denotation \rangle$ after two expansions.

2 Creating and initialising integers

<hr/> <code>\int_new:N</code>	<code>\int_new:N \langle integer \rangle</code>
<code>\int_new:c</code> <hr/>	Creates a new $\langle integer \rangle$ or raises an error if the name is already taken. The declaration is global. The $\langle integer \rangle$ will initially be equal to 0.

<hr/> <code>\int_const:Nn</code>	<code>\int_const:Nn \langle integer \rangle {\langle integer expression \rangle}</code>
<code>\int_const:cn</code> <hr/>	Creates a new constant $\langle integer \rangle$ or raises an error if the name is already taken. The value of the $\langle integer \rangle$ will be set globally to the $\langle integer expression \rangle$.
Updated: 2011-10-22	

<hr/> <code>\int_zero:N</code>	<code>\int_zero:N \langle integer \rangle</code>
<code>\int_zero:c</code>	
<code>\int_gzero:N</code>	Sets $\langle integer \rangle$ to 0.
<code>\int_gzero:c</code> <hr/>	

```
\int_zero_new:N
\int_zero_new:c
\int_gzero_new:N
\int_gzero_new:c
```

New: 2011-12-13

`\int_zero_new:N` $\langle integer \rangle$

Ensures that the $\langle integer \rangle$ exists globally by applying `\int_new:N` if necessary, then applies `\int_(g)zero:N` to leave the $\langle integer \rangle$ 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` $\langle integer_1 \rangle$ $\langle integer_2 \rangle$

Sets the content of $\langle integer_1 \rangle$ equal to that of $\langle integer_2 \rangle$.

```
\int_if_exist_p:N *
\int_if_exist_p:c *
\int_if_exist:NTF *
\int_if_exist:cTF *
```

New: 2012-03-03

`\int_if_exist_p:N` $\langle int \rangle$

`\int_if_exist:NTF` $\langle int \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Tests whether the $\langle int \rangle$ is currently defined. This does not check that the $\langle int \rangle$ really is an integer variable.

3 Setting and incrementing integers

```
\int_add:Nn
\int_add:cn
\int_gadd:Nn
\int_gadd:cn
```

Updated: 2011-10-22

`\int_add:Nn` $\langle integer \rangle$ $\{\langle integer expression \rangle\}$

Adds the result of the $\langle integer expression \rangle$ to the current content of the $\langle integer \rangle$.

```
\int_decr:N
\int_decr:c
\int_gdecr:N
\int_gdecr:c
```

`\int_decr:N` $\langle integer \rangle$

Decreases the value stored in $\langle integer \rangle$ by 1.

```
\int_incr:N
\int_incr:c
\int_gincr:N
\int_gincr:c
```

`\int_incr:N` $\langle integer \rangle$

Increases the value stored in $\langle integer \rangle$ by 1.

```
\int_set:Nn
\int_set:cn
\int_gset:Nn
\int_gset:cn
```

Updated: 2011-10-22

`\int_set:Nn` $\langle integer \rangle$ $\{\langle integer expression \rangle\}$

Sets $\langle integer \rangle$ to the value of $\langle integer expression \rangle$, which must evaluate to an integer (as described for `\int_eval:n`).

<code>\int_sub:Nn</code>	<code>\int_sub:Nn <integer> {\integer expression}</code>
<code>\int_sub:cn</code>	
<code>\int_gsub:Nn</code>	Subtracts the result of the <i><integer expression></i> from the current content of the <i><integer></i> .
<code>\int_gsub:cn</code>	

Updated: 2011-10-22

4 Using integers

<code>\int_use:N</code>	★	<code>\int_use:N <integer></code>
<code>\int_use:c</code>	★	

Updated: 2011-10-22

Recovers the content of an *<integer>* 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 an *<integer>* is required (such as in the first and third arguments of `\int_compare:nNnTF`).

T_EXhackers note: `\int_use:N` is the T_EX primitive `\the`: this is one of several L^AT_EX3 names for this primitive.

5 Integer expression conditionals

<code>\int_compare_p:nNn</code>	★	<code>\int_compare_p:nNn {\intexpr₁} <relation> {\intexpr₂}</code>
<code>\int_compare:nNnTF</code>	★	<code>\int_compare:nNnTF</code> <code>{\intexpr₁} <relation> {\intexpr₂}</code> <code>{\true code} {\false code}</code>

This function first evaluates each of the *<integer expressions>* as described for `\int_eval:n`. The two results are then compared using the *<relation>*:

Equal	=
Greater than	>
Less than	<

```

\int_compare_p:n ★ \int_compare_p:n
\int_compare:nTF ★ {
    <intexpr1> <relation1>
    ...
    <intexprN> <relationN>
    <intexprN+1>
}
\int_compare:nTF
{
    <intexpr1> <relation1>
    ...
    <intexprN> <relationN>
    <intexprN+1>
}
{<true code>} {<false code>}

```

Updated: 2013-01-13

This function evaluates the *<integer expressions>* as described for `\int_eval:n` and compares consecutive result using the corresponding *<relation>*, namely it compares *<intexpr₁>* and *<intexpr₂>* using the *<relation₁>*, then *<intexpr₂>* and *<intexpr₃>* using the *<relation₂>*, until finally comparing *<intexpr_N>* and *<intexpr_{N+1}>* using the *<relation_N>*. The test yields **true** if all comparisons are **true**. Each *<integer expression>* is evaluated only once, and the evaluation is lazy, in the sense that if one comparison is **false**, then no other *<integer expression>* is evaluated and no other comparison is performed. The *<relations>* can be any of the following:

Equal	= or ==
Greater than or equal to	>=
Greater than	>
Less than or equal to	<=
Less than	<
Not equal	!=

<code>\int_case:nnTF</code> ★	<code>\int_case:nnTF {<test integer expression>}</code>
New: 2013-07-24	<code>{</code> <code> {<intexpr case₁>} {<code case₁>}</code> <code> {<intexpr case₂>} {<code case₂>}</code> <code> ...</code> <code> {<intexpr case_n>} {<code case_n>}</code> <code>}</code> <code>{<true code>}</code> <code>{<false code>}</code>

This function evaluates the *<test integer expression>* and compares this in turn to each of the *<integer expression cases>*. If the two are equal then the associated *<code>* is left in the input stream. If any of the cases are matched, the *<true code>* is also inserted into the input stream (after the code for the appropriate case), while if none match then the *<false code>* is inserted. The function `\int_case:nn`, which does nothing if there is no match, is also available. For example

```
\int_case:nnF
{ 2 * 5 }
{
  { 5 }      { Small }
  { 4 + 6 }  { Medium }
  { -2 * 10 } { Negative }
}
{ No idea! }
```

will leave “Medium” in the input stream.

<code>\int_if_even_p:n</code> ★	<code>\int_if_odd_p:n {<integer expression>}</code>
<code>\int_if_even:nTF</code> ★	<code>\int_if_odd:nTF {<integer expression>}</code>
<code>\int_if_odd_p:n</code> ★	<code>{<true code>} {<false code>}</code>
<code>\int_if_odd:nTF</code> ★	

This function first evaluates the *<integer expression>* as described for `\int_eval:n`. It then evaluates if this is odd or even, as appropriate.

6 Integer expression loops

<code>\int_do_until:nNnn</code> ☆	<code>\int_do_until:nNnn {<intexpr₁>} <relation> {<intexpr₂>} {<code>}</code>
-----------------------------------	---

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

<hr/> <code>\int_do_while:nNnn</code> ☆ <hr/>	<code>\int_do_while:nNnn {<intexpr₁>} <relation> {<intexpr₂>} {<code>}</code>
	Places the <i><code></i> in the input stream for T _E X to process, and then evaluates the relationship between the two <i><integer expressions></i> as described for <code>\int_compare:nNnTF</code> . If the test is true then the <i><code></i> will be inserted into the input stream again and a loop will occur until the <i><relation></i> is false .
<hr/> <code>\int_until_do:nNnn</code> ☆ <hr/>	<code>\int_until_do:nNnn {<intexpr₁>} <relation> {<intexpr₂>} {<code>}</code>
	Evaluates the relationship between the two <i><integer expressions></i> as described for <code>\int_compare:nNnTF</code> , and then places the <i><code></i> in the input stream if the <i><relation></i> is false . After the <i><code></i> has been processed by T _E X the test will be repeated, and a loop will occur until the test is true .
<hr/> <code>\int_while_do:nNnn</code> ☆ <hr/>	<code>\int_while_do:nNnn {<intexpr₁>} <relation> {<intexpr₂>} {<code>}</code>
	Evaluates the relationship between the two <i><integer expressions></i> as described for <code>\int_compare:nNnTF</code> , and then places the <i><code></i> in the input stream if the <i><relation></i> is true . After the <i><code></i> has been processed by T _E X the test will be repeated, and a loop will occur until the test is false .
<hr/> <code>\int_do_until:nn</code> ☆ <hr/>	<code>\int_do_until:nn {<integer relation>} {<code>}</code>
Updated: 2013-01-13	Places the <i><code></i> in the input stream for T _E X to process, and then evaluates the <i><integer relation></i> as described for <code>\int_compare:nTF</code> . If the test is false then the <i><code></i> will be inserted into the input stream again and a loop will occur until the <i><relation></i> is true .
<hr/> <code>\int_do_while:nn</code> ☆ <hr/>	<code>\int_do_while:nn {<integer relation>} {<code>}</code>
Updated: 2013-01-13	Places the <i><code></i> in the input stream for T _E X to process, and then evaluates the <i><integer relation></i> as described for <code>\int_compare:nTF</code> . If the test is true then the <i><code></i> will be inserted into the input stream again and a loop will occur until the <i><relation></i> is false .
<hr/> <code>\int_until_do:nn</code> ☆ <hr/>	<code>\int_until_do:nn {<integer relation>} {<code>}</code>
Updated: 2013-01-13	Evaluates the <i><integer relation></i> as described for <code>\int_compare:nTF</code> , and then places the <i><code></i> in the input stream if the <i><relation></i> is false . After the <i><code></i> has been processed by T _E X the test will be repeated, and a loop will occur until the test is true .
<hr/> <code>\int_while_do:nn</code> ☆ <hr/>	<code>\int_while_do:nn {<integer relation>} {<code>}</code>
Updated: 2013-01-13	Evaluates the <i><integer relation></i> as described for <code>\int_compare:nTF</code> , and then places the <i><code></i> in the input stream if the <i><relation></i> is true . After the <i><code></i> has been processed by T _E X the test will be repeated, and a loop will occur until the test is false .

7 Integer step functions

`\int_step_function:nnnN` ☆

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

`\int_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 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:nnnN { 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:nnnn`

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

`\int_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 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:nnnNn`

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

`\int_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 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_alph:n` ★ `\int_to_alph:n {⟨integer expression⟩}`

`\int_to_Alph:n` ★

Updated: 2011-09-17

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_alph:n { 1 }`

places a in the input stream,

`\int_to_alph:n { 26 }`

is represented as z and

`\int_to_alph:n { 27 }`

is converted to aa. For conversions using other alphabets, use `\int_to_symbols:nnn` to define an alphabet-specific function. The basic `\int_to_alph:n` and `\int_to_Alph:n` functions should not be modified. The resulting tokens are digits with category code 12 (other) and letters with category code 11 (letter).

`\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_alph:n` function is defined as

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

`\int_to_bin:n` ★ `\int_to_bin:n {⟨integer expression⟩}`

New: 2014-02-11

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

<hr/>	
<code>\int_to_hex:n</code> ★	<code>\int_to_hex:n {⟨integer expression⟩}</code>
<code>\int_to_Hex:n</code> ★	
<hr/> New: 2014-02-11 <hr/>	Calculates the value of the <i>⟨integer expression⟩</i> and places the hexadecimal (base 16) representation of the result in the input stream. Letters are used for digits beyond 9: lower case letters for <code>\int_to_hex:n</code> and upper case ones for <code>\int_to_Hex:n</code> . The resulting tokens are digits with category code 12 (other) and letters with category code 11 (letter).
<hr/>	
<code>\int_to_oct:n</code> ★	<code>\int_to_oct:n {⟨integer expression⟩}</code>
<hr/> New: 2014-02-11 <hr/>	Calculates the value of the <i>⟨integer expression⟩</i> and places the octal (base 8) representation of the result in the input stream. The resulting tokens are digits with category code 12 (other) and letters with category code 11 (letter).
<hr/>	
<code>\int_to_base:nn</code> ★	<code>\int_to_base:nn {⟨integer expression⟩} {⟨base⟩}</code>
<code>\int_to_Base:nn</code> ★	
<hr/> Updated: 2014-02-11 <hr/>	Calculates the value of the <i>⟨integer expression⟩</i> and converts it into the appropriate representation in the <i>⟨base⟩</i> ; the later may be given as an integer expression. For bases greater than 10 the higher “digits” are represented by letters from the English alphabet: lower case letters for <code>\int_to_base:n</code> and upper case ones for <code>\int_to_Base:n</code> . The maximum <i>⟨base⟩</i> value is 36. The resulting tokens are digits with category code 12 (other) and letters with category code 11 (letter).
 TeXhackers note: This is a generic version of <code>\int_to_bin:n</code> , <i>etc.</i>	
<hr/>	
<code>\int_to_roman:n</code> ★	<code>\int_to_roman:n {⟨integer expression⟩}</code>
<code>\int_to_Roman:n</code> ★	
<hr/> Updated: 2011-10-22 <hr/>	Places the value of the <i>⟨integer expression⟩</i> in the input stream as Roman numerals, either lower case (<code>\int_to_roman:n</code>) or upper case (<code>\int_to_Roman:n</code>). The Roman numerals are letters with category code 11 (letter).

9 Converting from other formats to integers

<hr/>	
<code>\int_from_alph:n</code> ★	<code>\int_from_alph:n {⟨letters⟩}</code>
<hr/> Updated: 2014-08-25 <hr/>	Converts the <i>⟨letters⟩</i> into the integer (base 10) representation and leaves this in the input stream. The <i>⟨letters⟩</i> are first converted to a string, with no expansion. Lower and upper case letters from the English alphabet may be used, with “a” equal to 1 through to “z” equal to 26. The function also accepts a leading sign, made of + and -. This is the inverse function of <code>\int_to_alph:n</code> and <code>\int_to_Alph:n</code> .
<hr/>	
<code>\int_from_bin:n</code> ★	<code>\int_from_bin:n {⟨binary number⟩}</code>
<hr/> New: 2014-02-11 Updated: 2014-08-25 <hr/>	Converts the <i>⟨binary number⟩</i> into the integer (base 10) representation and leaves this in the input stream. The <i>⟨binary number⟩</i> is first converted to a string, with no expansion. The function accepts a leading sign, made of + and -, followed by binary digits. This is the inverse function of <code>\int_to_bin:n</code> .

<hr/> <code>\int_from_hex:n</code> ★ <hr/> <div> <div>New: 2014-02-11</div> <div>Updated: 2014-08-25</div> </div> <hr/>	<code>\int_from_hex:n {\langle hexadecimal number \rangle}</code> Converts the $\langle hexadecimal number \rangle$ into the integer (base 10) representation and leaves this in the input stream. Digits greater than 9 may be represented in the $\langle hexadecimal number \rangle$ by upper or lower case letters. The $\langle hexadecimal number \rangle$ is first converted to a string, with no expansion. The function also accepts a leading sign, made of + and -. This is the inverse function of <code>\int_to_hex:n</code> and <code>\int_to_Hex:n</code> .
<hr/> <code>\int_from_oct:n</code> ★ <hr/> <div> <div>New: 2014-02-11</div> <div>Updated: 2014-08-25</div> </div> <hr/>	<code>\int_from_oct:n {\langle octal number \rangle}</code> Converts the $\langle octal number \rangle$ into the integer (base 10) representation and leaves this in the input stream. The $\langle octal number \rangle$ is first converted to a string, with no expansion. The function accepts a leading sign, made of + and -, followed by octal digits. This is the inverse function of <code>\int_to_oct:n</code> .
<hr/> <code>\int_from_roman:n</code> ★ <hr/> <div> <div>Updated: 2014-08-25</div> </div> <hr/>	<code>\int_from_roman:n {\langle roman numeral \rangle}</code> Converts the $\langle roman numeral \rangle$ into the integer (base 10) representation and leaves this in the input stream. The $\langle roman numeral \rangle$ is first converted to a string, with no expansion. The $\langle roman numeral \rangle$ 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 <code>\int_to_roman:n</code> and <code>\int_to_Roman:n</code> .
<hr/> <code>\int_from_base:nn</code> ★ <hr/> <div> <div>Updated: 2014-08-25</div> </div> <hr/>	<code>\int_from_base:nn {\langle number \rangle} {\langle base \rangle}</code> Converts the $\langle number \rangle$ expressed in $\langle base \rangle$ into the appropriate value in base 10. The $\langle number \rangle$ is first converted to a string, with no expansion. The $\langle number \rangle$ should consist of digits and letters (either lower or upper case), plus optionally a leading sign. The maximum $\langle base \rangle$ value is 36. This is the inverse function of <code>\int_to_base:nn</code> and <code>\int_to_Base:nn</code> .

10 Viewing integers

<hr/> <code>\int_show:N</code> <code>\int_show:c</code> <hr/>	<code>\int_show:N \langle integer \rangle</code> Displays the value of the $\langle integer \rangle$ on the terminal.
<hr/> <code>\int_show:n</code> <hr/> <div> <div>New: 2011-11-22</div> <div>Updated: 2015-08-07</div> </div> <hr/>	<code>\int_show:n {\langle integer expression \rangle}</code> Displays the result of evaluating the $\langle integer expression \rangle$ on the terminal.

11 Constant integers

`\c_minus_one`
`\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.

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

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

T_EXhackers note: These are both names for the T_EX primitive `\ifnum`.

<code>\if_case:w</code>	\star	<code>\if_case:w</code>	$\langle integer \rangle$	$\langle case_0 \rangle$
<code>\or:</code>	\star	<code>\or:</code>	$\langle case_1 \rangle$	
<hr/>		<code>\or:</code>	\dots	
		<code>\else:</code>	$\langle default \rangle$	
		<code>\fi:</code>		

Selects a case to execute based on the value of the $\langle integer \rangle$. The first case ($\langle case_0 \rangle$) is executed if $\langle integer \rangle$ is 0, the second ($\langle case_1 \rangle$) if the $\langle integer \rangle$ is 1, *etc.* The $\langle integer \rangle$ may be a literal, a constant or an integer expression (*e.g.* using `\int eval:n`).

T_EXhackers note: These are the T_EX primitives `\ifcase` and `\or.`

```

\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.

T_EXhackers note: This is the T_EX primitive `\ifodd`.

14 Internal functions

`_int_to_roman:w` ★ `_int_to_roman:w` *⟨integer⟩* *⟨space⟩* or *⟨non-expandable token⟩*

Converts $\langle integer \rangle$ to it 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 $\langle integer \rangle$ 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.

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

<code>__int_value:w</code>	★	<code>__int_value:w</code> $\langle integer \rangle$
		<code>__int_value:w</code> $\langle tokens \rangle$ $\langle optional\ space \rangle$

Expands $\langle tokens \rangle$ until an $\langle integer \rangle$ is formed. One space may be gobbled in the process.

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

<code>__int_eval:w</code>	★	<code>__int_eval:w</code> $\langle intexpr \rangle$ <code>__int_eval_end:</code>
<code>__int_eval_end:</code>	★	

Evaluates $\langle integer\ expression \rangle$ 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.

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

<code>__prg_compare_error:</code>	<code>__prg_compare_error:</code>
<code>__prg_compare_error:Nw</code>	<code>__prg_compare_error:Nw</code> $\langle token \rangle$

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

Part X

The l3skip package

Dimensions and skips

L^AT_EX3 provides two general length variables: `dim` and `skip`. Lengths stored as `dim` variables have a fixed length, whereas `skip` lengths have a rubber (stretch/shrink) component. In addition, the `muskip` type is available for use in math mode: this is a special form of `skip` where the lengths involved are determined by the current math font (in μ). There are common features in the creation and setting of length variables, but for clarity the functions are grouped by variable type.

1 Creating and initialising `dim` variables

`\dim_new:N`
`\dim_new:c`

`\dim_new:N` $\langle dimension \rangle$

Creates a new $\langle dimension \rangle$ or raises an error if the name is already taken. The declaration is global. The $\langle dimension \rangle$ will initially be equal to 0 pt.

`\dim_const:Nn`
`\dim_const:cn`

`\dim_const:Nn` $\langle dimension \rangle$ $\{ \langle dimension expression \rangle \}$

Creates a new constant $\langle dimension \rangle$ or raises an error if the name is already taken. The value of the $\langle dimension \rangle$ will be set globally to the $\langle dimension expression \rangle$.

New: 2012-03-05

`\dim_zero:N`
`\dim_zero:c`
`\dim_gzero:N`
`\dim_gzero:c`

`\dim_zero:N` $\langle dimension \rangle$

Sets $\langle dimension \rangle$ to 0 pt.

`\dim_zero_new:N`
`\dim_zero_new:c`
`\dim_gzero_new:N`
`\dim_gzero_new:c`

`\dim_zero_new:N` $\langle dimension \rangle$

Ensures that the $\langle dimension \rangle$ exists globally by applying `\dim_new:N` if necessary, then applies `\dim_(g)zero:N` to leave the $\langle dimension \rangle$ set to zero.

New: 2012-01-07

`\dim_if_exist_p:N` ★
`\dim_if_exist_p:c` ★
`\dim_if_exist:NTF` ★
`\dim_if_exist:cTF` ★

`\dim_if_exist_p:N` $\langle dimension \rangle$

`\dim_if_exist:NTF` $\langle dimension \rangle$ $\{ \langle true code \rangle \} \{ \langle false code \rangle \}$

Tests whether the $\langle dimension \rangle$ is currently defined. This does not check that the $\langle dimension \rangle$ really is a dimension variable.

New: 2012-03-03

2 Setting dim variables

<code>\dim_add:Nn</code>	<code>\dim_add:Nn <dimension> {<dimension expression>}</code>
<code>\dim_add:cn</code>	
<code>\dim_gadd:Nn</code>	Adds the result of the <i><dimension expression></i> to the current content of the <i><dimension></i> .
<code>\dim_gadd:cn</code>	

Updated: 2011-10-22

<code>\dim_set:Nn</code>	<code>\dim_set:Nn <dimension> {<dimension expression>}</code>
<code>\dim_set:cn</code>	
<code>\dim_gset:Nn</code>	Sets <i><dimension></i> to the value of <i><dimension expression></i> , which must evaluate to a length with units.
<code>\dim_gset:cn</code>	

Updated: 2011-10-22

<code>\dim_set_eq:NN</code>	<code>\dim_set_eq:NN <dimension₁₂</code>
<code>\dim_set_eq:(cN Nc cc)</code>	
<code>\dim_gset_eq:NN</code>	Sets the content of <i><dimension_{1 equal to that of <i><dimension_{2.}</i>}</i>
<code>\dim_gset_eq:(cN Nc cc)</code>	

<code>\dim_sub:Nn</code>	<code>\dim_sub:Nn <dimension> {<dimension expression>}</code>
<code>\dim_sub:cn</code>	
<code>\dim_gsub:Nn</code>	Subtracts the result of the <i><dimension expression></i> from the current content of the <i><dimension></i> .
<code>\dim_gsub:cn</code>	

Updated: 2011-10-22

3 Utilities for dimension calculations

<code>\dim_abs:n</code> ★	<code>\dim_abs:n {<dimexpr>}</code>
Updated: 2012-09-26	Converts the <i><dimexpr></i> to its absolute value, leaving the result in the input stream as a <i><dimension denotation></i> .

<code>\dim_max:nn</code> ★	<code>\dim_max:nn {<dimexpr₁₂</code>
<code>\dim_min:nn</code> ★	<code>\dim_min:nn {<dimexpr₁₂</code>
New: 2012-09-09	
Updated: 2012-09-26	Evaluates the two <i><dimension expressions></i> and leaves either the maximum or minimum value in the input stream as appropriate, as a <i><dimension denotation></i> .

<code>\dim_ratio:nn</code> ☆	<code>\dim_ratio:nn {<dimexpr₁>} {<dimexpr₂>}</code>
------------------------------	--

Updated: 2011-10-22

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

<code>\dim_compare_p:nNn</code> ★	<code>\dim_compare_p:nNn {<dimexpr₁>} <relation> {<dimexpr₂>}</code>
<code>\dim_compare:nNnTF</code> ★	<code>\dim_compare:nNnTF</code> <code>{<dimexpr₁>} <relation> {<dimexpr₂>}</code> <code>{<true code>} {<false code>}</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_p:n
\dim_compare:nTF ★ {
    <dimexpr1> <relation1>
    ...
    <dimexprN> <relationN>
    <dimexprN+1>
}
\dim_compare:nTF
{
    <dimexpr1> <relation1>
    ...
    <dimexprN> <relationN>
    <dimexprN+1>
}
{<true code>} {<false code>}

```

Updated: 2013-01-13

This function evaluates the *<dimension expressions>* as described for `\dim_eval:n` and compares consecutive result using the corresponding *<relation>*, namely it compares *<dimexpr₁>* and *<dimexpr₂>* using the *<relation₁>*, then *<dimexpr₂>* and *<dimexpr₃>* using the *<relation₂>*, until finally comparing *<dimexpr_N>* and *<dimexpr_{N+1}>* using the *<relation_N>*. The test yields **true** if all comparisons are **true**. Each *<dimension expression>* is evaluated only once, and the evaluation is lazy, in the sense that if one comparison is **false**, then no other *<dimension expression>* is evaluated and no other comparison is performed. The *<relations>* can be any of the following:

Equal	= or ==
Greater than or equal to	>=
Greater than	>
Less than or equal to	<=
Less than	<
Not equal	!=

`\dim_case:nnTF` ☆
 New: 2013-07-24

```
\dim_case:nnTF {<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 T_EX to process, and then evaluates the relationship between the two *<dimension expressions>* as described for `\dim_compare:nNnTF`. If the test is **false** then the *<code>* 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 T_EX to process, and then evaluates the relationship between the two *<dimension expressions>* as described for `\dim_compare:nNnTF`. If the test is **true** then the *<code>* will be inserted into the input stream again and a loop will occur until the *<relation>* is **false**.

<hr/> <code>\dim_until_do:nNnn</code> ☆ <hr/>	<code>\dim_until_do:nNnn {<dimexpr₁>} <relation> {<dimexpr₂>} {<code>}</code>
	Evaluates the relationship between the two <i><dimension expressions></i> as described for <code>\dim_compare:nNnTF</code> , and then places the <i><code></i> in the input stream if the <i><relation></i> is false . After the <i><code></i> has been processed by T _E X the test will be repeated, and a loop will occur until the test is true .
<hr/> <code>\dim_while_do:nNnn</code> ☆ <hr/>	<code>\dim_while_do:nNnn {<dimexpr₁>} <relation> {<dimexpr₂>} {<code>}</code>
	Evaluates the relationship between the two <i><dimension expressions></i> as described for <code>\dim_compare:nNnTF</code> , and then places the <i><code></i> in the input stream if the <i><relation></i> is true . After the <i><code></i> has been processed by T _E X the test will be repeated, and a loop will occur until the test is false .
<hr/> <code>\dim_do_until:nn</code> ☆ <hr/>	<code>\dim_do_until:nn {<dimension relation>} {<code>}</code>
Updated: 2013-01-13	Places the <i><code></i> in the input stream for T _E X to process, and then evaluates the <i><dimension relation></i> as described for <code>\dim_compare:nTF</code> . If the test is false then the <i><code></i> will be inserted into the input stream again and a loop will occur until the <i><relation></i> is true .
<hr/> <code>\dim_do_while:nn</code> ☆ <hr/>	<code>\dim_do_while:nn {<dimension relation>} {<code>}</code>
Updated: 2013-01-13	Places the <i><code></i> in the input stream for T _E X to process, and then evaluates the <i><dimension relation></i> as described for <code>\dim_compare:nTF</code> . If the test is true then the <i><code></i> will be inserted into the input stream again and a loop will occur until the <i><relation></i> is false .
<hr/> <code>\dim_until_do:nn</code> ☆ <hr/>	<code>\dim_until_do:nn {<dimension relation>} {<code>}</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 false . After the <i><code></i> has been processed by T _E X the test will be repeated, and a loop will occur until the test is true .
<hr/> <code>\dim_while_do:nn</code> ☆ <hr/>	<code>\dim_while_do:nn {<dimension relation>} {<code>}</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 true . After the <i><code></i> has been processed by T _E X the test will be repeated, and a loop will occur until the test is false .

6 Using dim expressions and variables

<hr/> <code>\dim_eval:n</code> ★ <hr/>	<code>\dim_eval:n {<dimension expression>}</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/\tl_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 T _E X-style assignment as it is <i>not</i> an <i><internal dimension></i> .

`\dim_use:N` ★ `\dim_use:N` $\langle dimension \rangle$

`\dim_use:c` ★

Recovers the content of a $\langle dimension \rangle$ 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 $\langle dimension \rangle$ is required (such as in the argument of `\dim_eval:n`).

TeXhackers note: `\dim_use:N` is the TeX primitive `\the`: this is one of several L^AT_EX3 names for this primitive.

`\dim_to_decimal:n` ★ `\dim_to_decimal:n` $\{\langle dimexpr \rangle\}$

New: 2014-07-15

Evaluates the $\langle dimension expression \rangle$, and leaves the result, expressed in points (`pt`) in the input stream, with *no units*. The result is rounded by TeX to four or five decimal places. If the decimal part of the result is zero, it is omitted, together with the decimal marker.

For example

`\dim_to_decimal:n { 1bp }`

leaves 1.00374 in the input stream, *i.e.* the magnitude of one “big point” when converted to (TeX) points.

`\dim_to_decimal_in_bp:n` ★ `\dim_to_decimal_in_bp:n` $\{\langle dimexpr \rangle\}$

New: 2014-07-15

Evaluates the $\langle dimension expression \rangle$, 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` ★ `\dim_to_decimal_in_sp:n` $\{\langle dimexpr \rangle\}$

New: 2015-05-18

Evaluates the $\langle dimension expression \rangle$, and leaves the result, expressed in scaled points (`sp`) in the input stream, with *no units*. The result will necessarily be an integer.

<code>\dim_to_decimal_in_unit:nn</code>	★	<code>\dim_to_decimal_in_unit:nn {⟨dimexpr₁⟩} {⟨dimexpr₂⟩}</code>
---	---	---

New: 2014-07-15

Evaluates the *⟨dimension expressions⟩*, and leaves the value of *⟨dimexpr₁⟩*, expressed in a unit given by *⟨dimexpr₂⟩*, in the input stream. The result is a decimal number, rounded by T_EX to four or five decimal places. If the decimal part of the result is zero, it is omitted, together with the decimal marker.

For example

```
\dim_to_decimal_in_unit:nn { 1bp } { 1mm }
```

leaves 0.35277 in the input stream, *i.e.* the magnitude of one big point when converted to millimetres.

Note that this function is not optimised for any particular output and as such may give different results to `\dim_to_decimal_in_bp:n` or `\dim_to_decimal_in_sp:n`. In particular, the latter is able to take a wider range of input values as it is not limited by the ability to calculate a ratio using ε -T_EX primitives, which is required internally by `\dim_to_decimal_in_unit:nn`.

<code>\dim_to_fp:n</code>	★	<code>\dim_to_fp:n {⟨dimexpr⟩}</code>
---------------------------	---	---------------------------------------

New: 2012-05-08

Expands to an internal floating point number equal to the value of the *⟨dimexpr⟩* 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

<code>\dim_show:N</code>	<code>\dim_show:N ⟨dimension⟩</code>
--------------------------	--------------------------------------

`\dim_show:c`

Displays the value of the *⟨dimension⟩* on the terminal.

<code>\dim_show:n</code>	<code>\dim_show:n {⟨dimension expression⟩}</code>
--------------------------	---

New: 2011-11-22

Updated: 2015-08-07

Displays the result of evaluating the *⟨dimension expression⟩* on the terminal.

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` $\langle skip \rangle$
Creates a new $\langle skip \rangle$ or raises an error if the name is already taken. The declaration is global. The $\langle skip \rangle$ will initially be equal to 0 pt.

`\skip_const:Nn`
`\skip_const:cn`

`\skip_const:Nn` $\langle skip \rangle$ $\{ \langle skip \text{ expression} \rangle \}$
Creates a new constant $\langle skip \rangle$ or raises an error if the name is already taken. The value of the $\langle skip \rangle$ will be set globally to the $\langle skip \text{ expression} \rangle$.

New: 2012-03-05

`\skip_zero:N`
`\skip_zero:c`
`\skip_gzero:N`
`\skip_gzero:c`

`\skip_zero:N` $\langle skip \rangle$
Sets $\langle skip \rangle$ to 0 pt.

`\skip_zero_new:N`
`\skip_zero_new:c`
`\skip_gzero_new:N`
`\skip_gzero_new:c`

`\skip_zero_new:N` $\langle skip \rangle$
Ensures that the $\langle skip \rangle$ exists globally by applying `\skip_new:N` if necessary, then applies `\skip_(g)zero:N` to leave the $\langle skip \rangle$ set to zero.

New: 2012-01-07

`\skip_if_exist_p:N` ★
`\skip_if_exist_p:c` ★
`\skip_if_exist:NTF` ★
`\skip_if_exist:cTF` ★

`\skip_if_exist_p:N` $\langle skip \rangle$
`\skip_if_exist:NTF` $\langle skip \rangle$ $\{ \langle true \text{ code} \rangle \}$ $\{ \langle false \text{ code} \rangle \}$
Tests whether the $\langle skip \rangle$ is currently defined. This does not check that the $\langle skip \rangle$ really is a skip variable.

New: 2012-03-03

11 Setting skip variables

<code>\skip_add:Nn</code>	<code>\skip_add:Nn <skip> {<skip expression>}</code>
---------------------------	--

<code>\skip_add:cn</code>

<code>\skip_gadd:Nn</code>

<code>\skip_gadd:cn</code>

Updated: 2011-10-22

<code>\skip_set:Nn</code>	<code>\skip_set:Nn <skip> {<skip expression>}</code>
---------------------------	--

<code>\skip_set:cn</code>

<code>\skip_gset:Nn</code>

<code>\skip_gset:cn</code>

Updated: 2011-10-22

<code>\skip_set_eq:NN</code>

<code>\skip_set_eq:(cN Nc cc)</code>

<code>\skip_gset_eq:NN</code>

<code>\skip_gset_eq:(cN Nc cc)</code>

<code>\skip_set_eq:NN <skip₁₂</code>
--

Sets the content of *<skip₁>* equal to that of *<skip₂>*.

<code>\skip_sub:Nn</code>	<code>\skip_sub:Nn <skip> {<skip expression>}</code>
---------------------------	--

<code>\skip_sub:cn</code>

<code>\skip_gsub:Nn</code>

<code>\skip_gsub:cn</code>

Updated: 2011-10-22

Subtracts the result of the *<skip expression>* from the current content of the *<skip>*.

12 Skip expression conditionals

<code>\skip_if_eq_p:nn</code> ★	<code>\skip_if_eq_p:nn {<skipexpr₁>} {<skipexpr₂>}</code>
---------------------------------	---

<code>\skip_if_eq:nnTF</code> ★	<code>\dim_compare:nTF</code>
---------------------------------	-------------------------------

<code>{<skipexpr₁>} {<skipexpr₂>}</code>
--

<code>{<true code>} {<false code>}</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.

<code>\skip_if_finite_p:n</code> ★	<code>\skip_if_finite_p:n {<skipexpr>}</code>
------------------------------------	---

<code>\skip_if_finite:nTF</code> ★	<code>\skip_if_finite:nTF {<skipexpr>} {<true code>} {<false code>}</code>
------------------------------------	--

New: 2012-03-05

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

<hr/> <code>\skip_eval:n</code> ★ <hr/>	<code>\skip_eval:n {\langle skip expression \rangle}</code>
Updated: 2011-10-22 <hr/>	Evaluates the $\langle skip expression \rangle$, expanding any skips and token list variables within the $\langle expression \rangle$ to their content (without requiring <code>\skip_use:N/\tl_use:N</code>) and applying the standard mathematical rules. The result of the calculation is left in the input stream as a $\langle glue denotation \rangle$ after two expansions. This will be expressed in points (<code>\pt</code>), and will require suitable termination if used in a T _E X-style assignment as it is <i>not</i> an $\langle internal glue \rangle$.

<hr/> <code>\skip_use:N</code> ★ <hr/>	<code>\skip_use:N \langle skip \rangle</code>
<code>\skip_use:c</code> ★ <hr/>	Recovers the content of a $\langle skip \rangle$ 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 $\langle dimension \rangle$ is required (such as in the argument of <code>\skip_eval:n</code>).

T_EXhackers note: `\skip_use:N` is the T_EX primitive `\the`: this is one of several L^AT_EX3 names for this primitive.

14 Viewing skip variables

<hr/> <code>\skip_show:N</code> <hr/>	<code>\skip_show:N \langle skip \rangle</code>
<code>\skip_show:c</code> <hr/>	Displays the value of the $\langle skip \rangle$ on the terminal.

<hr/> <code>\skip_show:n</code> <hr/>	<code>\skip_show:n {\langle skip expression \rangle}</code>
New: 2011-11-22 Updated: 2015-08-07 <hr/>	Displays the result of evaluating the $\langle skip expression \rangle$ on the terminal.

15 Constant skips

<hr/> <code>\c_max_skip</code> <hr/>	The maximum value that can be stored as a skip (equal to <code>\c_max_dim</code> in length), with no stretch nor shrink component.
Updated: 2012-11-02 <hr/>	

<hr/> <code>\c_zero_skip</code> <hr/>	A zero length as a skip, with no stretch nor shrink component.
Updated: 2012-11-01 <hr/>	

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:c`
`\skip_horizontal:n`

Updated: 2011-10-22

`\skip_horizontal:N` $\langle skip \rangle$
`\skip_horizontal:n` $\{\langle skipexpr \rangle\}$

Inserts a horizontal $\langle skip \rangle$ into the current list.

T_EXhackers note: `\skip_horizontal:N` is the T_EX primitive `\hskip` renamed.

`\skip_vertical:N`
`\skip_vertical:c`
`\skip_vertical:n`

Updated: 2011-10-22

`\skip_vertical:N` $\langle skip \rangle$
`\skip_vertical:n` $\{\langle skipexpr \rangle\}$

Inserts a vertical $\langle skip \rangle$ into the current list.

T_EXhackers note: `\skip_vertical:N` is the T_EX primitive `\vskip` renamed.

18 Creating and initialising muskip variables

`\muskip_new:N`
`\muskip_new:c`

`\muskip_new:N` $\langle muskip \rangle$

Creates a new $\langle muskip \rangle$ or raises an error if the name is already taken. The declaration is global. The $\langle muskip \rangle$ will initially be equal to 0 mu.

`\muskip_const:Nn`
`\muskip_const:cn`

New: 2012-03-05

`\muskip_const:Nn` $\langle muskip \rangle$ $\{\langle muskip expression \rangle\}$

Creates a new constant $\langle muskip \rangle$ or raises an error if the name is already taken. The value of the $\langle muskip \rangle$ will be set globally to the $\langle muskip expression \rangle$.

`\muskip_zero:N`
`\muskip_zero:c`
`\muskip_gzero:N`
`\muskip_gzero:c`

`\muskip_zero:N` $\langle muskip \rangle$

Sets $\langle muskip \rangle$ to 0 mu.

```
\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 $\langle muskip \rangle$ exists globally by applying $\backslash muskip_new:N$ if necessary, then applies $\backslash muskip_(\mathbf{g})zero:N$ to leave the $\langle muskip \rangle$ 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 $\langle muskip \rangle$ is currently defined. This does not check that the $\langle muskip \rangle$ 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 $\langle muskip \text{ expression} \rangle$ to the current content of the $\langle muskip \rangle$.

```
\muskip_set:Nn
\muskip_set:cn
\muskip_gset:Nn
\muskip_gset:cn
```

Updated: 2011-10-22

```
\muskip_set:Nn <muskip> {\muskip expression}
```

Sets $\langle muskip \rangle$ to the value of $\langle muskip \text{ expression} \rangle$, which must evaluate to a math length with units and may include a rubber component (for example 1 mu plus 0.5 mu).

```
\muskip_set_eq:NN
\muskip_set_eq:(cN|Nc|cc)
\muskip_gset_eq:NN
\muskip_gset_eq:(cN|Nc|cc)
```

```
\muskip_set_eq:NN <muskip1> <muskip2>
```

Sets the content of $\langle muskip_1 \rangle$ equal to that of $\langle muskip_2 \rangle$.

```
\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 $\langle muskip \text{ expression} \rangle$ from the current content of the $\langle skip \rangle$.

20 Using muskip expressions and variables

<hr/> <code>\muskip_eval:n</code> ★ <hr/>	<code>\muskip_eval:n {⟨<i>muskip expression</i>⟩}</code>
Updated: 2011-10-22	Evaluates the $\langle muskip expression \rangle$, expanding any skips and token list variables within the $\langle expression \rangle$ to their content (without requiring <code>\muskip_use:N/\tl_use:N</code>) and applying the standard mathematical rules. The result of the calculation is left in the input stream as a $\langle muglue denotation \rangle$ after two expansions. This will be expressed in <code>mu</code> , and will require suitable termination if used in a T _E X-style assignment as it is <i>not</i> an $\langle internal muglue \rangle$.

<hr/> <code>\muskip_use:N</code> ★	<code>\muskip_use:N ⟨<i>muskip</i>⟩</code>
<hr/> <code>\muskip_use:c</code> ★	Recovers the content of a $\langle skip \rangle$ 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 $\langle dimension \rangle$ is required (such as in the argument of <code>\muskip_eval:n</code>).

T_EXhackers note: `\muskip_use:N` is the T_EX primitive `\the`: this is one of several L^AT_EX3 names for this primitive.

21 Viewing muskip variables

<hr/> <code>\muskip_show:N</code>	<code>\muskip_show:N ⟨<i>muskip</i>⟩</code>
<hr/> <code>\muskip_show:c</code>	Displays the value of the $\langle muskip \rangle$ on the terminal.

<hr/> <code>\muskip_show:n</code>	<code>\muskip_show:n {⟨<i>muskip expression</i>⟩}</code>
New: 2011-11-22 Updated: 2015-08-07	Displays the result of evaluating the $\langle muskip expression \rangle$ on the terminal.

22 Constant muskips

<hr/> <code>\c_max_muskip</code>	The maximum value that can be stored as a muskip, with no stretch nor shrink component.
<hr/> <code>\c_zero_muskip</code>	A zero length as a muskip, with no stretch nor shrink component.

23 Scratch muskips

`\l_tmpa_muskip`
`\l_tmpb_muskip`

Scratch muskip 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_muskip`
`\g_tmpb_muskip`

Scratch muskip 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.

24 Primitive conditional

`\if_dim:w` `\if_dim:w` $\langle dimen_1 \rangle$ $\langle relation \rangle$ $\langle dimen_2 \rangle$
 $\langle true\ code \rangle$
`\else:`
 $\langle false \rangle$
`\fi:`

Compare two dimensions. The $\langle relation \rangle$ is one of $<$, $=$ or $>$ with category code 12.

T_EXhackers note: This is the T_EX primitive `\ifdim`.

25 Internal functions

`_dim_eval:w` ★ `_dim_eval:w` $\langle dimexpr \rangle$ `_dim_eval_end:`
`_dim_eval_end:` ★

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

T_EXhackers note: This is the ε -T_EX primitive `\dimexpr`.

Part XI

The l3tl package

Token lists

T_EX works with tokens, and L^AT_EX3 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 T_EX 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.

T_EXhackers note: When T_EX fetches an undelimited argument from the input stream, explicit character tokens with character code 32 (space) and category code 10 (space), which we here call “explicit space characters”, are ignored. If the following token is an explicit character token with category code 1 (begin-group) and an arbitrary character code, then T_EX scans ahead to obtain an equal number of explicit character tokens with category code 1 (begin-group) and 2 (end-group), and the resulting list of tokens (with outer braces removed) becomes the argument. Otherwise, a single token is taken as the argument for the macro: we call such single tokens “N-type”, as they are suitable to be used as an argument for a function with the signature `:N`.

When T_EX reads a character of category code 10 for the first time, it is converted to an explicit space character, with character code 32, regardless of the initial character code. “Funny” spaces with a different category code, can be produced using `\tl_to_lowercase:n` or `\tl_to_uppercase:n`. Explicit space characters are also produced as a result of `\token_to_str:N`, `\tl_to_str:n`, etc.

1 Creating and initialising token list variables

<hr/> <u>\tl_new:N</u> <u>\tl_new:c</u> <hr/>	<u>\tl_new:N</u> $\langle tl\ var \rangle$ Creates a new $\langle tl\ var \rangle$ or raises an error if the name is already taken. The declaration is global. The $\langle tl\ var \rangle$ will initially be empty.
<hr/> <u>\tl_const:Nn</u> <u>\tl_const:(Nx cn cx)</u> <hr/>	<u>\tl_const:Nn</u> $\langle tl\ var \rangle$ $\{ \langle token\ list \rangle \}$ Creates a new constant $\langle tl\ var \rangle$ or raises an error if the name is already taken. The value of the $\langle tl\ var \rangle$ will be set globally to the $\langle token\ list \rangle$.
<hr/> <u>\tl_clear:N</u> <u>\tl_clear:c</u> <u>\tl_gclear:N</u> <u>\tl_gclear:c</u> <hr/>	<u>\tl_clear:N</u> $\langle tl\ var \rangle$ Clears all entries from the $\langle tl\ var \rangle$.
<hr/> <u>\tl_clear_new:N</u> <u>\tl_clear_new:c</u> <u>\tl_gclear_new:N</u> <u>\tl_gclear_new:c</u> <hr/>	<u>\tl_clear_new:N</u> $\langle tl\ var \rangle$ Ensures that the $\langle tl\ var \rangle$ exists globally by applying <u>\tl_new:N</u> if necessary, then applies <u>\tl_(g)clear:N</u> to leave the $\langle tl\ var \rangle$ empty.
<hr/> <u>\tl_set_eq:NN</u> <u>\tl_set_eq:(cN Nc cc)</u> <u>\tl_gset_eq:NN</u> <u>\tl_gset_eq:(cN Nc cc)</u> <hr/>	<u>\tl_set_eq:NN</u> $\langle tl\ var_1 \rangle$ $\langle tl\ var_2 \rangle$ Sets the content of $\langle tl\ var_1 \rangle$ equal to that of $\langle tl\ var_2 \rangle$.
<hr/> <u>\tl_concat:NNN</u> <u>\tl_concat:ccc</u> <u>\tl_gconcat:NNN</u> <u>\tl_gconcat:ccc</u> <hr/>	<u>\tl_concat:NNN</u> $\langle tl\ var_1 \rangle$ $\langle tl\ var_2 \rangle$ $\langle tl\ var_3 \rangle$ 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.
<hr/> New: 2012-05-18 <hr/>	
<hr/> <u>\tl_if_exist_p:N</u> ★ <u>\tl_if_exist_p:c</u> ★ <u>\tl_if_exist:NTF</u> ★ <u>\tl_if_exist:cTF</u> ★ <hr/>	<u>\tl_if_exist_p:N</u> $\langle tl\ var \rangle$ <u>\tl_if_exist:NTF</u> $\langle tl\ var \rangle$ $\{ \langle true\ code \rangle \}$ $\{ \langle false\ code \rangle \}$ 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.
<hr/> New: 2012-03-03 <hr/>	

2 Adding data to token list variables

<code>\tl_set:Nn</code>	<code>\tl_set:Nn <tl var> {<tokens>}</code>
<code>\tl_set:(NV Nv No Nf Nx cn cV cv co cf cx)</code>	
<code>\tl_gset:Nn</code>	
<code>\tl_gset:(NV Nv No Nf Nx cn cV cv co cf cx)</code>	

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

<code>\tl_put_left:Nn</code>	<code>\tl_put_left:Nn <tl var> {<tokens>}</code>
<code>\tl_put_left:(NV No Nx cn cV co cx)</code>	
<code>\tl_gput_left:Nn</code>	
<code>\tl_gput_left:(NV No Nx cn cV co cx)</code>	

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

<code>\tl_put_right:Nn</code>	<code>\tl_put_right:Nn <tl var> {<tokens>}</code>
<code>\tl_put_right:(NV No Nx cn cV co cx)</code>	
<code>\tl_gput_right:Nn</code>	
<code>\tl_gput_right:(NV No Nx cn cV co cx)</code>	

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

3 Modifying token list variables

<code>\tl_replace_once:Nnn</code>	<code>\tl_replace_once:Nnn <tl var> {<old tokens>} {<new tokens>}</code>
<code>\tl_replace_once:cnn</code>	
<code>\tl_greplace_once:Nnn</code>	
<code>\tl_greplace_once:cnn</code>	

Updated: 2011-08-11

Replaces the first (leftmost) occurrence of $\langle old\ tokens \rangle$ in the $\langle tl\ var \rangle$ with $\langle new\ tokens \rangle$. $\langle Old\ tokens \rangle$ cannot contain $\{$, $\}$ or $\#$ (more precisely, explicit character tokens with category code 1 (begin-group) or 2 (end-group), and tokens with category code 6).

<code>\tl_replace_all:Nnn</code>	<code>\tl_replace_all:Nnn <tl var> {<old tokens>} {<new tokens>}</code>
<code>\tl_replace_all:cnn</code>	
<code>\tl_greplace_all:Nnn</code>	
<code>\tl_greplace_all:cnn</code>	

Updated: 2011-08-11

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

<code>\tl_remove_once:Nn</code>	<code>\tl_remove_once:Nn <tl var> {<tokens>}</code>
<code>\tl_remove_once:cn</code>	
<code>\tl_gremove_once:Nn</code>	
<code>\tl_gremove_once:cn</code>	

Updated: 2011-08-11

Removes the first (leftmost) occurrence of $\langle tokens \rangle$ from the $\langle tl\ var \rangle$. $\langle Tokens \rangle$ 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 $\langle tokens \rangle$ from the $\langle tl var \rangle$. $\langle Tokens \rangle$ 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 $\langle tokens \rangle$ may remain after the removal, for instance,

```
\tl_set:Nn \l_tmpa_tl {abbccd} \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:(Nno|Nnx|cnn|cno|cnx)
\tl_gset_rescan:Nnn
\tl_gset_rescan:(Nno|Nnx|cnn|cno|cnx)

```

Updated: 2015-08-11

```
\tl_set_rescan:Nnn <tl var> {<setup>} {<tokens>}
```

Sets $\langle tl var \rangle$ 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 `\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 `\tl_rescan:nn`.

TeXhackers note: The $\langle tokens \rangle$ are first turned into a string (using `\tl_to_str:n`). If the string contains one or more characters with character code `\newlinechar` (set equal to `\endlinechar` unless that is equal to 32, before the user $\langle setup \rangle$), 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

Updated: 2015-08-11

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

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

T_EXhackers note: The <tokens> are first turned into a string (using **\tl_to_str:n**). If the string contains one or more characters with character code **\newlinechar** (set equal to **\endlinechar** unless that is equal to 32, before the user <setup>), then it is split into lines at these characters, then read as if reading multiple lines from a file, ignoring spaces (catcode 10) at the beginning and spaces and tabs (character code 32 or 9) at the end of every line. Otherwise, spaces (and tabs) are retained at both ends of the single-line string, as if it appeared in the middle of a line read from a file. Only the case of a single line is supported in LuaT_EX because of a bug in this engine.

5 Reassigning token list character codes

\tl_to_lowercase:n

Updated: 2012-09-08

\tl_to_lowercase:n {<tokens>}

Works through all of the <tokens>, replacing each character token with the lower case equivalent as defined by **\char_set_lccode:nn**. Characters with no defined lower case character code are left unchanged. This process does not alter the category code assigned to the <tokens>.

T_EXhackers note: This is a wrapper around the T_EX primitive **\lowercase**.

\tl_to_uppercase:n

Updated: 2012-09-08

\tl_to_uppercase:n {<tokens>}

Works through all of the <tokens>, replacing each character token with the upper case equivalent as defined by **\char_set_uccode:nn**. Characters with no defined upper case character code are left unchanged. This process does not alter the category code assigned to the <tokens>.

T_EXhackers note: This is a wrapper around the T_EX primitive **\uppercase**.

6 Token list conditionals

<code>\tl_if_blank_p:n</code>	★	<code>\tl_if_blank_p:n {⟨token list⟩}</code>
<code>\tl_if_blank_p:(V o)</code>	★	<code>\tl_if_blank:nTF {⟨token list⟩} {⟨true code⟩} {⟨false code⟩}</code>
<code>\tl_if_blank:nTF</code>	★	Tests if the <i>⟨token list⟩</i> consists only of blank spaces (<i>i.e.</i> contains no item). The test is
<code>\tl_if_blank:(V o)TF</code>	★	true if <i>⟨token list⟩</i> is zero or more explicit space characters (explicit tokens with character

code 32 and category code 10), and is **false** otherwise.

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

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

New: 2012-05-24

Updated: 2012-06-05

<code>\tl_if_eq_p:NN</code>	★	<code>\tl_if_eq_p:NN ⟨tl var₁⟩ ⟨tl var₂⟩</code>
<code>\tl_if_eq_p:(Nc cN cc)</code>	★	<code>\tl_if_eq:nNTF ⟨tl var₁⟩ ⟨tl var₂⟩ {⟨true code⟩} {⟨false code⟩}</code>
<code>\tl_if_eq:nNTF</code>	★	Compares the content of two <i>⟨token list variables⟩</i> and is logically true if the two contain
<code>\tl_if_eq:(Nc cN cc)TF</code>	★	the same list of tokens (<i>i.e.</i> 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**.

<code>\tl_if_eq:nnTF</code>	★	<code>\tl_if_eq:nnTF {⟨token list₁⟩} {⟨token list₂⟩} {⟨true code⟩} {⟨false code⟩}</code>
-----------------------------	---	--

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

<code>\tl_if_in:NnTF</code>	★	<code>\tl_if_in:NnTF ⟨tl var⟩ {⟨token list⟩} {⟨true code⟩} {⟨false code⟩}</code>
<code>\tl_if_in:cnTF</code>	★	

Tests if the *⟨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).

<code>\tl_if_in:nnTF</code>	★	<code>\tl_if_in:nnTF {⟨token list₁⟩} {⟨token list₂⟩} {⟨true code⟩} {⟨false code⟩}</code>
-----------------------------	---	--

<code>\tl_if_in:(Vn on no)TF</code>	★	Tests if <i>⟨token list₂⟩</i> is found inside <i>⟨token list₁⟩</i> . The <i>⟨token list₂⟩</i> cannot contain the
-------------------------------------	---	---

tokens `{`, `}` or `#` (more precisely, explicit character tokens with category code 1 (begin-group) or 2 (end-group), and tokens with category code 6).

<code>\tl_if_single_p:N</code> ★	<code>\tl_if_single_p:N <tl var></code>
<code>\tl_if_single_p:c</code> ★	<code>\tl_if_single:NTF <tl var> {\true code} {\false code}</code>
<code>\tl_if_single:NTF</code> ★	Tests if the content of the <code><tl var></code> consists of a single item, <i>i.e.</i> 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 <code>\tl_count:N</code> .
<code>\tl_if_single:cTF</code> ★	

Updated: 2011-08-13

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

<code>\tl_case:NnTF</code> ★	<code>\tl_case:NnTF <test token list variable></code>
<code>\tl_case:cnTF</code> ★	{
	<code><token list variable case₁> {\code case₁}</code>
	<code><token list variable case₂> {\code case₂}</code>
	...
	<code><token list variable case_n> {\code case_n}</code>
	}
	<code>{\true code}</code>
	<code>{\false code}</code>

New: 2013-07-24

This function compares the `<test token list variable>` in turn with each of the `<token list variable cases>`. If the two are equal (as described for `\tl_if_eq:NNTF`) then the associated `<code>` is left in the input stream. 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.

7 Mapping to token lists

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

Updated: 2012-06-29

<code>\tl_map_function:nN</code> ☆	<code>\tl_map_function:nN <token list> <function></code>
------------------------------------	--

Updated: 2012-06-29

Applies `<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`.

<hr/> <code>\tl_map_inline:Nn</code> <code>\tl_map_inline:cn</code> <hr/> Updated: 2012-06-29 <hr/>	<code>\tl_map_inline:Nn <tl var> {<inline function>}</code> <p>Applies the <i><inline function></i> to every <i><item></i> stored within the <i><tl var></i>. The <i><inline function></i> should consist of code which will receive the <i><item></i> as #1. One in line mapping can be nested inside another. See also <code>\tl_map_function:NN</code>.</p>
<hr/> <code>\tl_map_inline:nn</code> <hr/> Updated: 2012-06-29 <hr/>	<code>\tl_map_inline:nn <token list> {<inline function>}</code> <p>Applies the <i><inline function></i> to every <i><item></i> stored within the <i><token list></i>. The <i><inline function></i> should consist of code which will receive the <i><item></i> as #1. One in line mapping can be nested inside another. See also <code>\tl_map_function:nN</code>.</p>
<hr/> <code>\tl_map_variable:NNn</code> <code>\tl_map_variable:cNn</code> <hr/> Updated: 2012-06-29 <hr/>	<code>\tl_map_variable:NNn <tl var> <variable> {<function>}</code> <p>Applies the <i><function></i> to every <i><item></i> stored within the <i><tl var></i>. The <i><function></i> should consist of code which will receive the <i><item></i> stored in the <i><variable></i>. One variable mapping can be nested inside another. See also <code>\tl_map_inline:Nn</code>.</p>
<hr/> <code>\tl_map_variable:nNn</code> <hr/> Updated: 2012-06-29 <hr/>	<code>\tl_map_variable:nNn <token list> <variable> {<function>}</code> <p>Applies the <i><function></i> to every <i><item></i> stored within the <i><token list></i>. The <i><function></i> should consist of code which will receive the <i><item></i> stored in the <i><variable></i>. One variable mapping can be nested inside another. See also <code>\tl_map_inline:nn</code>.</p>
<hr/> <code>\tl_map_break: ☆</code> <hr/> Updated: 2012-06-29 <hr/>	<code>\tl_map_break:</code> <p>Used to terminate a <code>\tl_map...</code> function before all entries in the <i><token list variable></i> have been processed. This will normally take place within a conditional statement, for example</p> <pre> \tl_map_inline:Nn \l_my_tl { \str_if_eq:nnT { #1 } { bingo } { \tl_map_break: } % Do something useful } </pre> <p>See also <code>\tl_map_break:n</code>. Use outside of a <code>\tl_map...</code> scenario will lead to low level TeX errors.</p>

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` { $\langle tokens \rangle$ }

Used to terminate a `\tl_map_...` function before all entries in the $\langle token list variable \rangle$ have been processed, inserting the $\langle tokens \rangle$ 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 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 $\langle tokens \rangle$ are inserted into the input stream. This will depend on the design of the mapping function.

8 Using token lists

`\tl_to_str:n` ★

`\tl_to_str:n` { $\langle token list \rangle$ }

Converts the $\langle token list \rangle$ to a $\langle string \rangle$, leaving the resulting character tokens in the input stream. A $\langle string \rangle$ is a series of tokens with category code 12 (other) with the exception of spaces, which retain category code 10 (space).

T_EXhackers note: Converting a $\langle token list \rangle$ to a $\langle string \rangle$ yields a concatenation of the string representations of every token in the $\langle token list \rangle$. The string representation of a control sequence is

- an escape character, whose character code is given by the internal parameter `\escapechar`, absent if the `\escapechar` is negative or greater than the largest character code;
- the control sequence name, as defined by `\cs_to_str:N`;
- a space, unless the control sequence name is a single character whose category at the time of expansion of `\tl_to_str:n` is not “letter”.

The string representation of an explicit character token is that character, doubled in the case of (explicit) macro parameter characters (normally #). In particular, the string representation of a token list may depend on the category codes in effect when it is evaluated, and the value of the `\escapechar`: for instance `\tl_to_str:n {\a}` normally produces the three character “backslash”, “lower-case a”, “space”, but it may also produce a single “lower-case a” if the escape character is negative and `a` is currently not a letter.

<code>\tl_to_str:N</code>	★	<code>\tl_to_str:N <tl var></code>
<code>\tl_to_str:c</code>	★	

Converts the content of the $\langle tl\ var \rangle$ into a series of characters with category code 12 (other) with the exception of spaces, which retain category code 10 (space). This $\langle string \rangle$ is then left in the input stream. For low-level details, see the notes given for `\tl_to_str:n`.

<code>\tl_use:N</code>	★	<code>\tl_use:N <tl var></code>
<code>\tl_use:c</code>	★	

Recovers the content of a $\langle tl\ var \rangle$ 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 $\langle tl\ var \rangle$ directly without an accessor function.

9 Working with the content of token lists

<code>\tl_count:n</code>	★	<code>\tl_count:n {\tokens}</code>
<code>\tl_count:(V o)</code>	★	

New: 2012-05-13

Counts the number of $\langle items \rangle$ in $\langle tokens \rangle$ 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 $\langle tokens \rangle$. See also `\tl_count:N`. This function requires three expansions, giving an $\langle integer\ denotation \rangle$.

<code>\tl_count:N</code>	★	<code>\tl_count:N <tl var></code>
<code>\tl_count:c</code>	★	

New: 2012-05-13

Counts the number of token groups in the $\langle tl\ var \rangle$ 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 the $\langle tl\ var \rangle$. See also `\tl_count:n`. This function requires three expansions, giving an $\langle integer\ denotation \rangle$.

<code>\tl_reverse:n</code>	★	<code>\tl_reverse:n {\token list}</code>
<code>\tl_reverse:(V o)</code>	★	

Updated: 2012-01-08

Reverses the order of the $\langle items \rangle$ in the $\langle token\ list \rangle$, so that $\langle item_1 \rangle \langle item_2 \rangle \langle item_3 \rangle \dots \langle item_n \rangle$ becomes $\langle item_n \rangle \dots \langle item_3 \rangle \langle item_2 \rangle \langle item_1 \rangle$. This process will preserve unprotected space within the $\langle token\ list \rangle$. 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`.

T_EXhackers 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.

<code>\tl_reverse:N</code>		<code>\tl_reverse:N <tl var></code>
<code>\tl_reverse:c</code>		
<code>\tl_greverse:N</code>		
<code>\tl_greverse:c</code>		

Updated: 2012-01-08

Reverses the order of the $\langle items \rangle$ stored in $\langle tl\ var \rangle$, so that $\langle item_1 \rangle \langle item_2 \rangle \langle item_3 \rangle \dots \langle item_n \rangle$ becomes $\langle item_n \rangle \dots \langle item_3 \rangle \langle item_2 \rangle \langle item_1 \rangle$. This process will preserve unprotected spaces within the $\langle token\ list\ variable \rangle$. 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`.

<hr/> <code>\tl_reverse_items:n</code> ★ <hr/>	<code>\tl_reverse_items:n {\token list}</code>
<hr/> New: 2012-01-08 <hr/>	Reverses the order of the $\langle items \rangle$ stored in $\langle tl var \rangle$, so that $\{\langle item_1 \rangle\}\{\langle item_2 \rangle\}\{\langle item_3 \rangle\} \dots \{\langle item_n \rangle\}$ becomes $\{\langle item_n \rangle\} \dots \{\langle item_3 \rangle\}\{\langle item_2 \rangle\}\{\langle item_1 \rangle\}$. This process will remove any unprotected space within the $\langle token list \rangle$. Braced token groups are copied without reversing the order of tokens, and keep the outer set of braces. Items which are initially not braced are copied with braces in the result. In cases where preserving spaces is important, consider the slower function <code>\tl_reverse:n</code> .

T_EXhackers 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.

<hr/> <code>\tl_trim_spaces:n</code> ★ <hr/>	<code>\tl_trim_spaces:n {\token list}</code>
<hr/> New: 2011-07-09 Updated: 2012-06-25 <hr/>	Removes any leading and trailing explicit space characters (explicit tokens with character code 32 and category code 10) from the $\langle token list \rangle$ and leaves the result in the input stream.

T_EXhackers 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.

<hr/> <code>\tl_trim_spaces:N</code> <code>\tl_trim_spaces:c</code> <code>\tl_gtrim_spaces:N</code> <code>\tl_gtrim_spaces:c</code> <hr/>	<code>\tl_trim_spaces:N \langle tl var \rangle</code>
<hr/> New: 2011-07-09 <hr/>	Removes any leading and trailing explicit space characters (explicit tokens with character code 32 and category code 10) from the content of the $\langle tl var \rangle$. Note that this therefore <i>resets</i> the content of the variable.

10 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.

<hr/>	
<code>\tl_head:N</code>	★
<code>\tl_head:n</code>	★
<code>\tl_head:(V v f)</code>	★
<hr/>	
Updated: 2012-09-09	
<hr/>	

`\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.

<hr/>	
<code>\tl_head:w</code>	★
<hr/>	

`\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.

<code>\tl_tail:N</code>	★
<code>\tl_tail:n</code>	★
<code>\tl_tail:(V v f)</code>	★
Updated: 2012-09-01	

`\tl_tail:n` $\{ \langle token list \rangle \}$

Discards all leading explicit space characters (explicit tokens with character code 32 and category code 10) and the first $\langle item \rangle$ in the $\langle token list \rangle$, 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 $\langle token list \rangle$ (see `\tl_if_blank:nTF`) will result in `\tl_tail:n` leaving nothing in the input stream.

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

<code>\tl_if_head_eq_catcode_p:nN</code>	★	<code>\tl_if_head_eq_catcode_p:nN</code>	$\{ \langle token list \rangle \}$	$\langle test token \rangle$
<code>\tl_if_head_eq_catcode:nNTF</code>	★	<code>\tl_if_head_eq_catcode:nNTF</code>	$\{ \langle token list \rangle \}$	$\langle test token \rangle$
Updated: 2012-07-09		$\{ \langle true code \rangle \} \{ \langle false code \rangle \}$		

Tests if the first $\langle token \rangle$ in the $\langle token list \rangle$ has the same category code as the $\langle test token \rangle$. In the case where the $\langle token list \rangle$ is empty, the test will always be **false**.

<code>\tl_if_head_eq_charcode_p:nN</code>	★	<code>\tl_if_head_eq_charcode_p:nN</code>	$\{ \langle token list \rangle \}$	$\langle test token \rangle$
<code>\tl_if_head_eq_charcode_p:fN</code>	★	<code>\tl_if_head_eq_charcode:nNTF</code>	$\{ \langle token list \rangle \}$	$\langle test token \rangle$
<code>\tl_if_head_eq_charcode:nNTF</code>	★	$\{ \langle true code \rangle \} \{ \langle false code \rangle \}$		
<code>\tl_if_head_eq_charcode:fNTF</code>	★			

Updated: 2012-07-09

Tests if the first $\langle token \rangle$ in the $\langle token list \rangle$ has the same character code as the $\langle test token \rangle$. In the case where the $\langle token list \rangle$ is empty, the test will always be **false**.

<code>\tl_if_head_eq_meaning_p:nN</code>	★	<code>\tl_if_head_eq_meaning_p:nN</code>	$\{ \langle token list \rangle \}$	$\langle test token \rangle$
<code>\tl_if_head_eq_meaning:nNTF</code>	★	<code>\tl_if_head_eq_meaning:nNTF</code>	$\{ \langle token list \rangle \}$	$\langle test token \rangle$
Updated: 2012-07-09		$\{ \langle true code \rangle \} \{ \langle false code \rangle \}$		

Tests if the first $\langle token \rangle$ in the $\langle token list \rangle$ has the same meaning as the $\langle test token \rangle$. In the case where $\langle token list \rangle$ is empty, the test will always be **false**.

<code>\tl_if_head_is_group_p:n</code>	★	<code>\tl_if_head_is_group_p:n</code>	$\{ \langle token list \rangle \}$
<code>\tl_if_head_is_group:nTF</code>	★	<code>\tl_if_head_is_group:nTF</code>	$\{ \langle token list \rangle \} \{ \langle true code \rangle \} \{ \langle false code \rangle \}$
New: 2012-07-08			

Tests if the first $\langle token \rangle$ in the $\langle token list \rangle$ is an explicit begin-group character (with category code 1 and any character code), in other words, if the $\langle token list \rangle$ starts with a brace group. In particular, the test is **false** if the $\langle token list \rangle$ starts with an implicit token such as `\c_group_begin_token`, or if it is empty. This function is useful to implement actions on token lists on a token by token basis.

<code>\tl_if_head_is_N_type_p:n</code> ★	<code>\tl_if_head_is_N_type_p:n {⟨token list⟩}</code>
<code>\tl_if_head_is_N_type:nTF</code> ★	<code>\tl_if_head_is_N_type:nTF {⟨token list⟩} {⟨true code⟩} {⟨false code⟩}</code>

New: 2012-07-08

Tests if the first *⟨token⟩* in the *⟨token list⟩* is a normal N-type argument. In other words, it is neither an explicit space character (explicit token with character code 32 and category code 10) nor an explicit begin-group character (with category code 1 and any character code). An empty argument yields **false**, as it does not have a “normal” first token. This function is useful to implement actions on token lists on a token by token basis.

<code>\tl_if_head_is_space_p:n</code> ★	<code>\tl_if_head_is_space_p:n {⟨token list⟩}</code>
<code>\tl_if_head_is_space:nTF</code> ★	<code>\tl_if_head_is_space:nTF {⟨token list⟩} {⟨true code⟩} {⟨false code⟩}</code>

Updated: 2012-07-08

Tests if the first *⟨token⟩* in the *⟨token list⟩* is an explicit space character (explicit token with character code 12 and category code 10). In particular, the test is **false** if the *⟨token list⟩* starts with an implicit token such as `\c_space_token`, or if it is empty. This function is useful to implement actions on token lists on a token by token basis.

11 Using a single item

<code>\tl_item:nn</code> ★	<code>\tl_item:nn {⟨token list⟩} {⟨integer expression⟩}</code>
<code>\tl_item:Nn</code> ★	
<code>\tl_item:cn</code> ★	

New: 2014-07-17

Indexing items in the *⟨token list⟩* from 1 on the left, this function 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.

T_EXhackers 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.

12 Viewing token lists

<code>\tl_show:N</code>	<code>\tl_show:N ⟨tl var⟩</code>
<code>\tl_show:c</code>	

Updated: 2015-08-01

Displays the content of the *⟨tl var⟩* on the terminal.

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

<hr/> <code>\tl_show:n</code> <hr/>	<code>\tl_show:n</code> $\langle token list \rangle$
<code>Updated: 2015-08-07</code>	Displays the $\langle token list \rangle$ on the terminal.

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

13 Constant token lists

<hr/> <code>\c_empty_tl</code> <hr/>	Constant that is always empty.
<hr/> <code>\c_space_tl</code> <hr/>	An explicit space character contained in a token list (compare this with <code>\c_space_token</code>). For use where an explicit space is required.

14 Scratch token lists

<hr/> <code>\l_tmpa_tl</code> <hr/> <code>\l_tmpl_tl</code> <hr/>	Scratch token lists for local assignment. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
<hr/> <code>\g_tmpa_tl</code> <hr/> <code>\g_tmpl_tl</code> <hr/>	Scratch token lists for global assignment. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

15 Internal functions

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

Part XII

The l3str package

Strings

T_EX 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 T_EX sense.

A T_EX 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 T_EX 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 l3basics, l3tl and l3token, respectively.

Most expandable functions in this module come in three flavours:

- `\str_...:N`, which expect a token list or string variable as their argument;
- `\str_...:n`, taking any token list (or string) as an argument;
- `\str_..._ignore_spaces:n`, which ignores any space encountered during the operation: these functions are typically faster than those which take care of escaping spaces appropriately.

1 Building strings

`\str_new:N`
`\str_new:c`

New: 2015-09-18

`\str_new:N` $\langle str\ var \rangle$

Creates a new $\langle str\ var \rangle$ or raises an error if the name is already taken. The declaration is global. The $\langle str\ var \rangle$ will initially be empty.

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

New: 2015-09-18

`\str_const:Nn` $\langle str\ var \rangle$ $\{ \langle token\ list \rangle \}$

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

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

New: 2015-09-18

`\str_clear:N` $\langle str\ var \rangle$
Clears the content of the $\langle str\ var \rangle$.

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

New: 2015-09-18

`\str_clear_new:N` $\langle str\ var \rangle$
Ensures that the $\langle str\ var \rangle$ exists globally by applying `\str_new:N` if necessary, then applies `\str_(g)clear:N` to leave the $\langle str\ var \rangle$ 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` $\langle str\ var_1 \rangle$ $\langle str\ var_2 \rangle$
Sets the content of $\langle str\ var_1 \rangle$ equal to that of $\langle str\ var_2 \rangle$.

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` $\langle str\ var \rangle$ $\{ \langle token\ list \rangle \}$
Converts the $\langle token\ list \rangle$ to a $\langle string \rangle$, and stores the result in $\langle str\ var \rangle$.

```
\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` $\langle str\ var \rangle$ $\{ \langle token\ list \rangle \}$
Converts the $\langle token\ list \rangle$ to a $\langle string \rangle$, and prepends the result to $\langle str\ var \rangle$. The current contents of the $\langle str\ var \rangle$ 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` $\langle str\ var \rangle$ $\{ \langle token\ list \rangle \}$
Converts the $\langle token\ list \rangle$ to a $\langle string \rangle$, and appends the result to $\langle str\ var \rangle$. The current contents of the $\langle str\ var \rangle$ are not automatically converted to a string.

2.1 String conditionals

<code>\str_if_exist_p:N</code>	★	<code>\str_if_exist_p:N <str var></code>
<code>\str_if_exist_p:c</code>	★	<code>\str_if_exist:NTF <str var> {<true code>} {<false code>}</code>
<code>\str_if_exist:NTF</code>	★	Tests whether the <i><str var></i> is currently defined. This does not check that the <i><str var></i> really is a string.
<code>\str_if_exist:cTF</code>	★	

New: 2015-09-18

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

New: 2015-09-18

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

New: 2015-09-18

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

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**.

<code>\str_if_eq_x_p:nn</code>	★	<code>\str_if_eq_x_p:nn {<tl₁>} {<tl₂>}</code>
<code>\str_if_eq_x:nnTF</code>	★	<code>\str_if_eq_x:nnTF {<tl₁>} {<tl₂>} {<true code>} {<false code>}</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:nnTF` ★
`\str_case:(on|nV|nv)TF` ★

New: 2013-07-24
Updated: 2015-02-28

```
\str_case:nnTF {<test string>}
{
  {<string case1>} {<code case1>}
  {<string case2>} {<code case2>}
  ...
  {<string casen>} {<code casen>}
}
{<true code>}
{<false code>}
```

This function compares the $\langle test\ string \rangle$ in turn with each of the $\langle string\ cases \rangle$. If the two are equal (as described for `\str_if_eq:nnTF` 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 `\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:nnF {<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 $\langle test\ string \rangle$ in turn with the full expansion of the $\langle string\ cases \rangle$. If the two full expansions are equal (as described for `\str_if_eq:nnTF` 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 `\str_case_x:nn`, which does nothing if there is no match, is also available. The $\langle test\ string \rangle$ 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>
```

Recovers the content of a $\langle str\ var \rangle$ 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 $\langle str \rangle$ directly without an accessor function.

<code>\str_count:N</code>	★	<code>\str_count:n {⟨token list⟩}</code>
<code>\str_count:c</code>	★	
<code>\str_count:n</code>	★	
<code>\str_count_ignore_spaces:n</code>	★	

New: 2015-09-18

Leaves in the input stream the number of characters in the string representation of $\langle token list \rangle$, 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.

<code>\str_count_spaces:N</code>	★	<code>\str_count_spaces:n {⟨token list⟩}</code>
<code>\str_count_spaces:c</code>	★	
<code>\str_count_spaces:n</code>	★	

New: 2015-09-18

Leaves in the input stream the number of space characters in the string representation of $\langle token list \rangle$, as an integer denotation. Of course, this function has no `_ignore_spaces` variant.

<code>\str_head:N</code>	★	<code>\str_head:n {⟨token list⟩}</code>
<code>\str_head:c</code>	★	
<code>\str_head:n</code>	★	
<code>\str_head_ignore_spaces:n</code>	★	

New: 2015-09-18

Converts the $\langle token list \rangle$ into a $\langle string \rangle$. The first character in the $\langle string \rangle$ is then left in the input stream, with category code “other”. The functions differ if the first character is a space: `\str_head:N` and `\str_head:n` return a space token with category code 10 (blank space), while the `\str_head_ignore_spaces:n` function ignores this space character and leaves the first non-space character in the input stream. If the $\langle string \rangle$ is empty (or only contains spaces in the case of the `_ignore_spaces` function), then nothing is left on the input stream.

<code>\str_tail:N</code>	★	<code>\str_tail:n {⟨token list⟩}</code>
<code>\str_tail:c</code>	★	
<code>\str_tail:n</code>	★	
<code>\str_tail_ignore_spaces:n</code>	★	

New: 2015-09-18

Converts the $\langle token list \rangle$ to a $\langle string \rangle$, removes the first character, and leaves the remaining characters (if any) in the input stream, with category codes 12 and 10 (for spaces). The functions differ in the case where the first character is a space: `\str_tail:N` and `\str_tail:n` 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.

<code>\str_item:Nn</code>	★	<code>\str_item:nn {⟨token list⟩} {⟨integer expression⟩}</code>
<code>\str_item:nn</code>	★	
<code>\str_item_ignore_spaces:nn</code>	★	

New: 2015-09-18

Converts the $\langle token\ list \rangle$ to a $\langle string \rangle$, and leaves in the input stream the character in position $\langle integer\ expression \rangle$ of the $\langle string \rangle$, starting at 1 for the first (left-most) character. In the case of `\str_item:Nn` and `\str_item:nn`, all characters including spaces are taken into account. The `\str_item_ignore_spaces:nn` function skips spaces when counting characters. If the $\langle integer\ expression \rangle$ is negative, characters are counted from the end of the $\langle string \rangle$. Hence, -1 is the right-most character, *etc.*

<code>\str_range:Nnn</code>	★	<code>\str_range:nnn {⟨token list⟩} {⟨start index⟩} {⟨end index⟩}</code>
<code>\str_range:cnn</code>	★	
<code>\str_range:nnn</code>	★	
<code>\str_range_ignore_spaces:nnn</code>	★	

New: 2015-09-18

Converts the $\langle token\ list \rangle$ to a $\langle string \rangle$, and leaves in the input stream the characters from the $\langle start\ index \rangle$ to the $\langle end\ index \rangle$ inclusive. Positive $\langle indices \rangle$ are counted from the start of the string, 1 being the first character, and negative $\langle indices \rangle$ are counted from the end of the string, -1 being the last character. If either of $\langle start\ index \rangle$ or $\langle end\ index \rangle$ 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 bcd, cdef, ef, and an empty line to the terminal.

4 String manipulation

<code>\str_lower_case:n</code>	☆	<code>\str_lower_case:n {⟨tokens⟩}</code>
<code>\str_lower_case:f</code>	☆	<code>\str_upper_case:n {⟨tokens⟩}</code>
<code>\str_upper_case:n</code>	☆	
<code>\str_upper_case:f</code>	☆	

New: 2015-03-01

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`.

These functions are intended for case changing programmatic data in places where upper/lower case distinctions are meaningful. One example would be automatically generating a function name from user input where some case changing is needed. In this situation the input is programmatic, not textual, case does have meaning and a language-independent one-to-one mapping is appropriate. For example

```
\cs_new_protected:Npn \myfunc:nn #1#2
{
  \cs_set_protected:cpn
  {
    user
    \str_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.

T_EXhackers 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 pdfT_EX *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 X_ƎT_EX and LuaT_EX, subject only to the fact that X_ƎT_EX 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: 2015-09-04

`\str_fold_case:n` $\{ \langle tokens \rangle \}$

Converts the input $\langle tokens \rangle$ to their string representation, as described for `\tl_to_str:n`, and then folds the case of the resulting $\langle string \rangle$ to remove case information. The result of this process is left in the input stream.

String folding is a process used for material such as identifiers rather than for “text”. The folding provided by `\str_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` $\langle str\ var \rangle$

Displays the content of the $\langle str\ var \rangle$ 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: 2016-09-19

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 {\tl1} {\tl2}

```

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 {\tl1} {\tl2}

```

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:nn(TF)` coded for speed.

<hr/> <hr/>	<code>_str_to_other:n</code> ★	<code>_str_to_other:n {\token list}</code>	Converts the <i>⟨token list⟩</i> to a <i>⟨other string⟩</i> , 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.
<hr/> <hr/>	<code>_str_count:n</code> ★	<code>_str_count:n {\other string}</code>	This function expects an argument that is entirely made of characters with category “other”, as produced by <code>_str_to_other:n</code> . It leaves in the input stream the number of character tokens in the <i>⟨other string⟩</i> , faster than the analogous <code>\str_count:n</code> function.
<hr/> <hr/>	<code>_str_range:nnn</code> ★	<code>_str_range:nnn {\other string} {\start index} {\end index}</code>	Identical to <code>\str_range:nnn</code> except that the first argument is expected to be entirely made of characters with category “other”, as produced by <code>_str_to_other:n</code> , and the result is also an <i>⟨other string⟩</i> .

Part XIII

The l3seq package

Sequences and stacks

L^AT_EX3 implements a “sequence” data type, which contain an ordered list of entries which may contain any *⟨balanced text⟩*. It is possible to map functions to sequences such that the function is applied to every item in the sequence.

Sequences are also used to implement stack functions in L^AT_EX3. This is achieved using a number of dedicated stack functions.

1 Creating and initialising sequences

<code>\seq_new:N</code>	<code>\seq_new:N <sequence></code>
<code>\seq_new:c</code>	

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.

<code>\seq_clear:N</code>	<code>\seq_clear:N <sequence></code>
<code>\seq_clear:c</code>	
<code>\seq_gclear:N</code>	
<code>\seq_gclear:c</code>	

Clears all items from the *⟨sequence⟩*.

<code>\seq_clear_new:N</code>	<code>\seq_clear_new:N <sequence></code>
<code>\seq_clear_new:c</code>	
<code>\seq_gclear_new:N</code>	
<code>\seq_gclear_new:c</code>	

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

<code>\seq_set_eq:NN</code>	<code>\seq_set_eq:NN <sequence₁> <sequence₂></code>
<code>\seq_set_eq:(cN Nc cc)</code>	
<code>\seq_gset_eq:NN</code>	Sets the content of <i>⟨sequence₁⟩</i> equal to that of <i>⟨sequence₂⟩</i> .
<code>\seq_gset_eq:(cN Nc cc)</code>	

<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.

<hr/> <code>\seq_set_split:Nnn</code> <code>\seq_set_split:NnV</code> <code>\seq_gset_split:Nnn</code> <code>\seq_gset_split:NnV</code> <hr/>	<code>\seq_set_split:Nnn</code> $\langle sequence \rangle$ $\{\langle delimiter \rangle\}$ $\{\langle token list \rangle\}$ Splits the $\langle token list \rangle$ into $\langle items \rangle$ separated by $\langle delimiter \rangle$, and assigns the result to the $\langle sequence \rangle$. Spaces on both sides of each $\langle item \rangle$ are ignored, then one set of outer braces is removed (if any); this space trimming behaviour is identical to that of <code>l3clist</code> functions. Empty $\langle items \rangle$ are preserved by <code>\seq_set_split:Nnn</code> , and can be removed afterwards using <code>\seq_remove_all:Nn</code> $\langle sequence \rangle$ $\{\langle \rangle\}$. The $\langle delimiter \rangle$ may not contain <code>{</code> , <code>}</code> or <code>#</code> (assuming TeX's normal category code régime). If the $\langle delimiter \rangle$ is empty, the $\langle token list \rangle$ is split into $\langle items \rangle$ as a $\langle token list \rangle$.
<hr/> New: 2011-08-15 Updated: 2012-07-02 <hr/>	
<hr/> <code>\seq_concat:NNN</code> <code>\seq_concat:ccc</code> <code>\seq_gconcat:NNN</code> <code>\seq_gconcat:ccc</code> <hr/>	<code>\seq_concat:NNN</code> $\langle sequence_1 \rangle$ $\langle sequence_2 \rangle$ $\langle sequence_3 \rangle$ Concatenates the content of $\langle sequence_2 \rangle$ and $\langle sequence_3 \rangle$ together and saves the result in $\langle sequence_1 \rangle$. The items in $\langle sequence_2 \rangle$ will be placed at the left side of the new sequence.
<hr/> <code>\seq_if_exist_p:N</code> ★ <code>\seq_if_exist_p:c</code> ★ <code>\seq_if_exist:NTF</code> ★ <code>\seq_if_exist:cTF</code> ★ <hr/> New: 2012-03-03 <hr/>	<code>\seq_if_exist_p:N</code> $\langle sequence \rangle$ <code>\seq_if_exist:NTF</code> $\langle sequence \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$ Tests whether the $\langle sequence \rangle$ is currently defined. This does not check that the $\langle sequence \rangle$ really is a sequence variable.

2 Appending data to sequences

<hr/> <code>\seq_put_left:Nn</code> <code>\seq_put_left:(NV Nv No Nx cn cV cv co cx)</code> <code>\seq_gput_left:Nn</code> <code>\seq_gput_left:(NV Nv No Nx cn cV cv co cx)</code> <hr/>	<code>\seq_put_left:Nn</code> $\langle sequence \rangle$ $\{\langle item \rangle\}$ Appends the $\langle item \rangle$ to the left of the $\langle sequence \rangle$.
<hr/> <code>\seq_put_right:Nn</code> <code>\seq_put_right:(NV Nv No Nx cn cV cv co cx)</code> <code>\seq_gput_right:Nn</code> <code>\seq_gput_right:(NV Nv No Nx cn cV cv co cx)</code> <hr/>	<code>\seq_put_right:Nn</code> $\langle sequence \rangle$ $\{\langle item \rangle\}$ Appends the $\langle item \rangle$ to the right of the $\langle sequence \rangle$.

3 Recovering items from sequences

Items can be recovered from either the left or the right of sequences. For implementation reasons, the actions at the left of the sequence are faster than those acting on the right. These functions all assign the recovered material locally, *i.e.* setting the $\langle token list variable \rangle$ used with `\tl_set:Nn` and *never* `\tl_gset:Nn`.

<hr/> \seq_get_left:NN \seq_get_left:cN <hr/> Updated: 2012-05-14	\seq_get_left:NN $\langle sequence \rangle$ $\langle token\ list\ variable \rangle$ Stores the left-most item from a $\langle sequence \rangle$ in the $\langle token\ list\ variable \rangle$ without removing it from the $\langle sequence \rangle$. The $\langle token\ list\ variable \rangle$ is assigned locally. If $\langle sequence \rangle$ is empty the $\langle token\ list\ variable \rangle$ will contain the special marker $\backslash q_no_value$.
<hr/> \seq_get_right:NN \seq_get_right:cN <hr/> Updated: 2012-05-19	\seq_get_right:NN $\langle sequence \rangle$ $\langle token\ list\ variable \rangle$ Stores the right-most item from a $\langle sequence \rangle$ in the $\langle token\ list\ variable \rangle$ without removing it from the $\langle sequence \rangle$. The $\langle token\ list\ variable \rangle$ is assigned locally. If $\langle sequence \rangle$ is empty the $\langle token\ list\ variable \rangle$ will contain the special marker $\backslash q_no_value$.
<hr/> \seq_pop_left:NN \seq_pop_left:cN <hr/> Updated: 2012-05-14	\seq_pop_left:NN $\langle sequence \rangle$ $\langle token\ list\ variable \rangle$ Pops the left-most item from a $\langle sequence \rangle$ into the $\langle token\ list\ variable \rangle$, <i>i.e.</i> removes the item from the sequence and stores it in the $\langle token\ list\ variable \rangle$. Both of the variables are assigned locally. If $\langle sequence \rangle$ is empty the $\langle token\ list\ variable \rangle$ will contain the special marker $\backslash q_no_value$.
<hr/> \seq_gpop_left:NN \seq_gpop_left:cN <hr/> Updated: 2012-05-14	\seq_gpop_left:NN $\langle sequence \rangle$ $\langle token\ list\ variable \rangle$ Pops the left-most item from a $\langle sequence \rangle$ into the $\langle token\ list\ variable \rangle$, <i>i.e.</i> removes the item from the sequence and stores it in the $\langle token\ list\ variable \rangle$. The $\langle sequence \rangle$ is modified globally, while the assignment of the $\langle token\ list\ variable \rangle$ is local. If $\langle sequence \rangle$ is empty the $\langle token\ list\ variable \rangle$ will contain the special marker $\backslash q_no_value$.
<hr/> \seq_pop_right:NN \seq_pop_right:cN <hr/> Updated: 2012-05-19	\seq_pop_right:NN $\langle sequence \rangle$ $\langle token\ list\ variable \rangle$ Pops the right-most item from a $\langle sequence \rangle$ into the $\langle token\ list\ variable \rangle$, <i>i.e.</i> removes the item from the sequence and stores it in the $\langle token\ list\ variable \rangle$. Both of the variables are assigned locally. If $\langle sequence \rangle$ is empty the $\langle token\ list\ variable \rangle$ will contain the special marker $\backslash q_no_value$.
<hr/> \seq_gpop_right:NN \seq_gpop_right:cN <hr/> Updated: 2012-05-19	\seq_gpop_right:NN $\langle sequence \rangle$ $\langle token\ list\ variable \rangle$ Pops the right-most item from a $\langle sequence \rangle$ into the $\langle token\ list\ variable \rangle$, <i>i.e.</i> removes the item from the sequence and stores it in the $\langle token\ list\ variable \rangle$. The $\langle sequence \rangle$ is modified globally, while the assignment of the $\langle token\ list\ variable \rangle$ is local. If $\langle sequence \rangle$ is empty the $\langle token\ list\ variable \rangle$ will contain the special marker $\backslash q_no_value$.

`\seq_item:Nn` ★
`\seq_item:cn` ★

New: 2014-07-17

`\seq_item:Nn` $\langle sequence \rangle$ $\{\langle integer expression \rangle\}$

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

T_EXhackers note: The result is returned within the `\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 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` $\langle sequence \rangle$ $\langle token list variable \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

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

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

New: 2012-05-19

`\seq_get_right:NNTF` $\langle sequence \rangle$ $\langle token list variable \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

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

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

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

`\seq_pop_left:NNTF` $\langle sequence \rangle$ $\langle token list variable \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

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

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

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

`\seq_gpop_left:NNTF` $\langle sequence \rangle$ $\langle token list variable \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

If the $\langle sequence \rangle$ is empty, leaves the $\langle false code \rangle$ in the input stream. The value of the $\langle token list variable \rangle$ is not defined in this case and should not be relied upon. If the $\langle sequence \rangle$ is non-empty, pops the left-most item from a $\langle sequence \rangle$ in the $\langle token list variable \rangle$, *i.e.* removes the item from a $\langle sequence \rangle$. The $\langle sequence \rangle$ is modified globally, while the $\langle token list variable \rangle$ 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 $\langle sequence \rangle$ is empty, leaves the $\langle false code \rangle$ in the input stream. The value of the $\langle token list variable \rangle$ is not defined in this case and should not be relied upon. If the $\langle sequence \rangle$ is non-empty, pops the right-most item from a $\langle sequence \rangle$ in the $\langle token list variable \rangle$, *i.e.* removes the item from a $\langle sequence \rangle$. Both the $\langle sequence \rangle$ and the $\langle token list variable \rangle$ 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 $\langle sequence \rangle$ is empty, leaves the $\langle false code \rangle$ in the input stream. The value of the $\langle token list variable \rangle$ is not defined in this case and should not be relied upon. If the $\langle sequence \rangle$ is non-empty, pops the right-most item from a $\langle sequence \rangle$ in the $\langle token list variable \rangle$, *i.e.* removes the item from a $\langle sequence \rangle$. The $\langle sequence \rangle$ is modified globally, while the $\langle token list variable \rangle$ 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 $\langle sequence \rangle$, leaving the left most copy of each item in the $\langle sequence \rangle$. The $\langle item \rangle$ comparison takes place on a token basis, as for `\tl_if_eq:nn(TF)`.

T_EXhackers note: This function iterates through every item in the $\langle sequence \rangle$ and does a comparison with the $\langle items \rangle$ already checked. It is therefore relatively slow with large sequences.

```
\seq_remove_all:Nn
\seq_remove_all:cn
\seq_gremove_all:Nn
\seq_gremove_all:cn
```

```
\seq_remove_all:Nn <sequence> {(item)}
```

Removes every occurrence of $\langle item \rangle$ from the $\langle sequence \rangle$. The $\langle item \rangle$ comparison takes place on a token basis, as for `\tl_if_eq:nn(TF)`.

```
\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 $\langle sequence \rangle$.

6 Sequence conditionals

<code>\seq_if_empty_p:N</code> ★	<code>\seq_if_empty_p:N</code> $\langle sequence \rangle$
<code>\seq_if_empty_p:c</code> ★	<code>\seq_if_empty:N</code> $\langle sequence \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$
<code>\seq_if_empty:N</code> ★	Tests if the $\langle sequence \rangle$ is empty (containing no items).
<code>\seq_if_empty:c</code> ★	

<code>\seq_if_in:N</code> ★	<code>\seq_if_in:N</code> $\langle sequence \rangle$ $\{\langle item \rangle\}$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$
<code>\seq_if_in:(N Nv No Nx cn cV cv co cx)</code> ★	

Tests if the $\langle item \rangle$ is present in the $\langle sequence \rangle$.

7 Mapping to sequences

<code>\seq_map_function:N</code> ★	<code>\seq_map_function:N</code> $\langle sequence \rangle$ $\langle function \rangle$
<code>\seq_map_function:c</code> ★	Applies $\langle function \rangle$ to every $\langle item \rangle$ stored in the $\langle sequence \rangle$. The $\langle function \rangle$ will receive one argument for each iteration. The $\langle items \rangle$ are returned from left to right. The function <code>\seq_map_inline:N</code> is faster than <code>\seq_map_function:N</code> for sequences with more than about 10 items. One mapping may be nested inside another.
Updated: 2012-06-29	

<code>\seq_map_inline:N</code>	<code>\seq_map_inline:N</code> $\langle sequence \rangle$ $\{\langle inline function \rangle\}$
<code>\seq_map_inline:c</code>	Applies $\langle inline function \rangle$ to every $\langle item \rangle$ stored within the $\langle sequence \rangle$. The $\langle inline function \rangle$ should consist of code which will receive the $\langle item \rangle$ as #1. One in line mapping can be nested inside another. The $\langle items \rangle$ are returned from left to right.
Updated: 2012-06-29	

<code>\seq_map_variable:NN</code>	<code>\seq_map_variable:NN</code> $\langle sequence \rangle$ $\langle tl var. \rangle$ $\{\langle function using tl var. \rangle\}$
<code>\seq_map_variable:(Ncn cNn ccn)</code>	
Updated: 2012-06-29	

Stores each entry in the $\langle sequence \rangle$ in turn in the $\langle tl var. \rangle$ and applies the $\langle function using tl var. \rangle$. The $\langle function \rangle$ will usually consist of code making use of the $\langle tl var. \rangle$, but this is not enforced. One variable mapping can be nested inside another. The $\langle items \rangle$ 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 $\langle sequence \rangle$ 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 \TeX errors.

\TeX hackers 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 $\{\langle tokens \rangle\}$

Used to terminate a `\seq_map...` function before all entries in the $\langle sequence \rangle$ have been processed, inserting the $\langle tokens \rangle$ 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 \TeX errors.

\TeX hackers note: When the mapping is broken, additional tokens may be inserted by the internal macro `__prg_break_point:Nn` before the $\langle tokens \rangle$ 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 $\langle sequence \rangle$

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

8 Using the content of sequences directly

`\seq_use:Nnnn` ★
`\seq_use:cnnn` ★

New: 2013-05-26

`\seq_use:Nnnn` $\langle seq\ var \rangle$ $\{\langle separator\ between\ two \rangle\}$
`\seq_use:cnnn` $\{\langle separator\ between\ more\ than\ two \rangle\}$ $\{\langle separator\ between\ final\ two \rangle\}$

Places the contents of the $\langle seq\ var \rangle$ in the input stream, with the appropriate $\langle separator \rangle$ between the items. Namely, if the sequence has more than two items, the $\langle separator\ between\ more\ than\ two \rangle$ is placed between each pair of items except the last, for which the $\langle separator\ between\ final\ two \rangle$ is used. If the sequence has exactly two items, then they are placed in the input stream separated by the $\langle separator\ between\ two \rangle$. If the sequence has a single item, it is placed in the input stream, and an empty sequence produces no output. An error 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.

T_EXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the $\langle items \rangle$ 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` $\langle seq\ var \rangle$ $\{\langle separator \rangle\}$
`\seq_use:cn` $\langle seq\ var \rangle$ $\{\langle separator \rangle\}$

Places the contents of the $\langle seq\ var \rangle$ in the input stream, with the $\langle separator \rangle$ between the items. If the sequence has a single item, it is placed in the input stream with no $\langle separator \rangle$, and an empty sequence produces no output. An error 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.

T_EXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the $\langle items \rangle$ 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.

<code>\seq_get:NN</code> <code>\seq_get:cN</code> <hr/> Updated: 2012-05-14	<code>\seq_get:NN</code> $\langle sequence \rangle$ $\langle token\ list\ variable \rangle$ Reads the top item from a $\langle sequence \rangle$ into the $\langle token\ list\ variable \rangle$ without removing it from the $\langle sequence \rangle$. The $\langle token\ list\ variable \rangle$ is assigned locally. If $\langle sequence \rangle$ is empty the $\langle token\ list\ variable \rangle$ will contain the special marker <code>\q_no_value</code> .
---	---

<code>\seq_pop:NN</code> <code>\seq_pop:cN</code> <hr/> Updated: 2012-05-14	<code>\seq_pop:NN</code> $\langle sequence \rangle$ $\langle token\ list\ variable \rangle$ Pops the top item from a $\langle sequence \rangle$ into the $\langle token\ list\ variable \rangle$. Both of the variables are assigned locally. If $\langle sequence \rangle$ is empty the $\langle token\ list\ variable \rangle$ will contain the special marker <code>\q_no_value</code> .
---	---

<code>\seq_gpop:NN</code> <code>\seq_gpop:cN</code> <hr/> Updated: 2012-05-14	<code>\seq_gpop:NN</code> $\langle sequence \rangle$ $\langle token\ list\ variable \rangle$ Pops the top item from a $\langle sequence \rangle$ into the $\langle token\ list\ variable \rangle$. The $\langle sequence \rangle$ is modified globally, while the $\langle token\ list\ variable \rangle$ is assigned locally. If $\langle sequence \rangle$ is empty the $\langle token\ list\ variable \rangle$ will contain the special marker <code>\q_no_value</code> .
---	--

<code>\seq_get:NNTF</code> <code>\seq_get:cNTF</code> <hr/> New: 2012-05-14 Updated: 2012-05-19	<code>\seq_get:NNTF</code> $\langle sequence \rangle$ $\langle token\ list\ variable \rangle$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$ If the $\langle sequence \rangle$ is empty, leaves the $\langle false\ code \rangle$ in the input stream. The value of the $\langle token\ list\ variable \rangle$ is not defined in this case and should not be relied upon. If the $\langle sequence \rangle$ is non-empty, stores the top item from a $\langle sequence \rangle$ in the $\langle token\ list\ variable \rangle$ without removing it from the $\langle sequence \rangle$. The $\langle token\ list\ variable \rangle$ is assigned locally.
--	---

<code>\seq_pop:NNTF</code> <code>\seq_pop:cNTF</code> <hr/> New: 2012-05-14 Updated: 2012-05-19	<code>\seq_pop:NNTF</code> $\langle sequence \rangle$ $\langle token\ list\ variable \rangle$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$ If the $\langle sequence \rangle$ is empty, leaves the $\langle false\ code \rangle$ in the input stream. The value of the $\langle token\ list\ variable \rangle$ is not defined in this case and should not be relied upon. If the $\langle sequence \rangle$ is non-empty, pops the top item from the $\langle sequence \rangle$ in the $\langle token\ list\ variable \rangle$, <i>i.e.</i> removes the item from the $\langle sequence \rangle$. Both the $\langle sequence \rangle$ and the $\langle token\ list\ variable \rangle$ are assigned locally.
--	---

<code>\seq_gpop:NNTF</code> <code>\seq_gpop:cNTF</code> <hr/> New: 2012-05-14 Updated: 2012-05-19	<code>\seq_gpop:NNTF</code> $\langle sequence \rangle$ $\langle token\ list\ variable \rangle$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$ If the $\langle sequence \rangle$ is empty, leaves the $\langle false\ code \rangle$ in the input stream. The value of the $\langle token\ list\ variable \rangle$ is not defined in this case and should not be relied upon. If the $\langle sequence \rangle$ is non-empty, pops the top item from the $\langle sequence \rangle$ in the $\langle token\ list\ variable \rangle$, <i>i.e.</i> removes the item from the $\langle sequence \rangle$. The $\langle sequence \rangle$ is modified globally, while the $\langle token\ list\ variable \rangle$ is assigned locally.
--	--

<code>\seq_push:Nn</code> <code>\seq_push:(NV Nv No Nx cn cV cv co cx)</code> <code>\seq_gpush:Nn</code> <code>\seq_gpush:(NV Nv No Nx cn cV cv co cx)</code>	<code>\seq_push:Nn</code> $\langle sequence \rangle$ $\{\langle item \rangle\}$ Adds the $\{\langle item \rangle\}$ to the top of the $\langle sequence \rangle$.
--	---

10 Sequences as sets

Sequences can also be used as sets, such that all of their items are distinct. Usage of sequences as sets is not currently widespread, hence no specific set function is provided. Instead, it is explained here how common set operations can be performed by combining several functions described in earlier sections. When using sequences to implement sets, one should be careful not to rely on the order of items in the sequence representing the set.

Sets should not contain several occurrences of a given item. To make sure that a $\langle sequence\ variable \rangle$ only has distinct items, use `\seq_remove_duplicates:N` $\langle sequence\ variable \rangle$. This function is relatively slow, and to avoid performance issues one should only use it when necessary.

Some operations on a set $\langle seq\ var \rangle$ are straightforward. For instance, `\seq_count:N` $\langle seq\ var \rangle$ expands to the number of items, while `\seq_if_in:Nn(TF)` $\langle seq\ var \rangle$ $\{ \langle item \rangle \}$ tests if the $\langle item \rangle$ is in the set.

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

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

Removing an $\langle item \rangle$ from a set $\langle seq\ var \rangle$ can be done using `\seq_remove_all:Nn`,

```
\seq_remove_all:Nn  $\langle seq\ var \rangle$   $\{ \langle item \rangle \}$ 
```

The intersection of two sets $\langle seq\ var_1 \rangle$ and $\langle seq\ var_2 \rangle$ can be stored into $\langle seq\ var_3 \rangle$ by collecting items of $\langle seq\ var_1 \rangle$ which are in $\langle seq\ var_2 \rangle$.

```
\seq_clear:N  $\langle seq\ var_3 \rangle$ 
\seq_map_inline:Nn  $\langle seq\ var_1 \rangle$ 
{
  \seq_if_in:NnT  $\langle seq\ var_2 \rangle$   $\{ \#1 \}$ 
  { \seq_put_right:Nn  $\langle seq\ var_3 \rangle$   $\{ \#1 \}$  }
}
```

The code as written here only works if $\langle seq\ var_3 \rangle$ is different from the other two sequence variables. To cover all cases, items should first be collected in a sequence `\l_<pkg>_internal_seq`, then $\langle seq\ var_3 \rangle$ should be set equal to this internal sequence. The same remark applies to other set functions.

The union of two sets $\langle seq\ var_1 \rangle$ and $\langle seq\ var_2 \rangle$ can be stored into $\langle seq\ var_3 \rangle$ through

```
\seq_concat:NNN  $\langle seq\ var_3 \rangle$   $\langle seq\ var_1 \rangle$   $\langle seq\ var_2 \rangle$ 
\seq_remove_duplicates:N  $\langle seq\ var_3 \rangle$ 
```

or by adding items to (a copy of) $\langle seq\ var_1 \rangle$ one by one

```
\seq_set_eq:NN  $\langle seq\ var_3 \rangle$   $\langle seq\ var_1 \rangle$ 
\seq_map_inline:Nn  $\langle seq\ var_2 \rangle$ 
{
```

```

\seq_if_in:NnF <seq var3> {#1}
{ \seq_put_right:Nn <seq var3> {#1} }
}

```

The second approach is faster than the first when the $\langle seq\ var_2 \rangle$ is short compared to $\langle seq\ var_1 \rangle$.

The difference of two sets $\langle seq\ var_1 \rangle$ and $\langle seq\ var_2 \rangle$ can be stored into $\langle seq\ var_3 \rangle$ by removing items of the $\langle seq\ var_2 \rangle$ from (a copy of) the $\langle seq\ var_1 \rangle$ one by one.

```

\seq_set_eq:NN <seq var3> <seq var1>
\seq_map_inline:Nn <seq var2>
{ \seq_remove_all:Nn <seq var3> {#1} }

```

The symmetric difference of two sets $\langle seq\ var_1 \rangle$ and $\langle seq\ var_2 \rangle$ can be stored into $\langle seq\ var_3 \rangle$ by computing the difference between $\langle seq\ var_1 \rangle$ and $\langle seq\ var_2 \rangle$ and storing the result as $\backslash l_ \langle pkg \rangle_ internal_ seq$, then the difference between $\langle seq\ var_2 \rangle$ and $\langle seq\ var_1 \rangle$, and finally concatenating the two differences to get the symmetric differences.

```

\seq_set_eq:NN \l_<pkg>_internal_seq <seq var1>
\seq_map_inline:Nn <seq var2>
{ \seq_remove_all:Nn \l_<pkg>_internal_seq {#1} }
\seq_set_eq:NN <seq var3> <seq var2>
\seq_map_inline:Nn <seq var1>
{ \seq_remove_all:Nn <seq var3> {#1} }
\seq_concat:NNN <seq var3> <seq var3> \l_<pkg>_internal_seq

```

11 Constant and scratch sequences

$\backslash c_ empty_ seq$

Constant that is always empty.

New: 2012-07-02

$\backslash l_ tmpa_ seq$

$\backslash l_ tmpb_ seq$

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.

New: 2012-04-26

$\backslash g_ tmpa_ seq$

$\backslash g_ tmpb_ seq$

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.

New: 2012-04-26

12 Viewing sequences

<hr/> <code>\seq_show:N</code> <hr/>	<code>\seq_show:N</code> $\langle sequence \rangle$
<code>\seq_show:c</code> <hr/>	Displays the entries in the $\langle sequence \rangle$ in the terminal.
<hr/> <code>Updated: 2015-08-01</code> <hr/>	

13 Internal sequence functions

<hr/> <code>\s__seq</code> <hr/>	This scan mark (equal to <code>\scan_stop:</code>) marks the beginning of a sequence variable.
----------------------------------	---

<hr/> <code>__seq_item:n</code> ★ <hr/>	<code>__seq_item:n</code> $\{\langle item \rangle\}$ 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.
--	--

<hr/> <code>__seq_push_item_def:n</code> <hr/>	<code>__seq_push_item_def:n</code> $\{\langle code \rangle\}$
<code>__seq_push_item_def:x</code> <hr/>	Saves the definition of <code>__seq_item:n</code> and redefines it to accept one parameter and expand to $\langle code \rangle$. This function should always be balanced by use of <code>__seq_pop_item_def:.</code>

<hr/> <code>__seq_pop_item_def:</code> <hr/>	<code>__seq_pop_item_def:</code> Restores the definition of <code>__seq_item:n</code> most recently saved by <code>__seq_push_item_def:n</code> . This function should always be used in a balanced pair with <code>__seq_push_item_def:n</code> .
---	---

Part XIV

The l3clist 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:NN`. 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 \TeX category codes apply).

1 Creating and initialising comma lists

```
\clist_new:N
\clist_new:c
```

`\clist_new:N` \langle *comma list* \rangle

Creates a new \langle *comma list* \rangle or raises an error if the name is already taken. The declaration is global. The \langle *comma list* \rangle will initially contain no items.

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

`\clist_const:Nn` \langle *clist var* \rangle $\{\langle$ *comma list* $\rangle\}$

Creates a new constant \langle *clist var* \rangle or raises an error if the name is already taken. The value of the \langle *clist var* \rangle will be set globally to the \langle *comma list* \rangle .

New: 2014-07-05

```
\clist_clear:N
\clist_clear:c
\clist_gclear:N
\clist_gclear:c
```

`\clist_clear:N` \langle *comma list* \rangle

Clears all items from the \langle *comma list* \rangle .

<code>\clist_clear_new:N</code>	<code>\clist_clear_new:N</code> $\langle comma list \rangle$
<code>\clist_clear_new:c</code>	
<code>\clist_gclear_new:N</code>	Ensures that the $\langle comma list \rangle$ exists globally by applying <code>\clist_new:N</code> if necessary,
<code>\clist_gclear_new:c</code>	then applies <code>\clist_(g)clear:N</code> to leave the list empty.

<code>\clist_set_eq:NN</code>	<code>\clist_set_eq:NN</code> $\langle comma list_1 \rangle$ $\langle comma list_2 \rangle$
<code>\clist_set_eq:(cN Nc cc)</code>	Sets the content of $\langle comma list_1 \rangle$ equal to that of $\langle comma list_2 \rangle$.
<code>\clist_gset_eq:NN</code>	
<code>\clist_gset_eq:(cN Nc cc)</code>	

<code>\clist_set_from_seq:NN</code>	<code>\clist_set_from_seq:NN</code> $\langle comma list \rangle$ $\langle sequence \rangle$
<code>\clist_set_from_seq:(cN Nc cc)</code>	
<code>\clist_gset_from_seq:NN</code>	
<code>\clist_gset_from_seq:(cN Nc cc)</code>	

New: 2014-07-17

Converts the data in the $\langle sequence \rangle$ into a $\langle comma list \rangle$: the original $\langle sequence \rangle$ is unchanged. Items which contain either spaces or commas are surrounded by braces.

<code>\clist_concat:NNN</code>	<code>\clist_concat:NNN</code> $\langle comma list_1 \rangle$ $\langle comma list_2 \rangle$ $\langle comma list_3 \rangle$
<code>\clist_concat:ccc</code>	
<code>\clist_gconcat:NNN</code>	Concatenates the content of $\langle comma list_2 \rangle$ and $\langle comma list_3 \rangle$ together and saves the
<code>\clist_gconcat:ccc</code>	result in $\langle comma list_1 \rangle$. The items in $\langle comma list_2 \rangle$ will be placed at the left side of the

new comma list.

<code>\clist_if_exist_p:N</code> ★	<code>\clist_if_exist_p:N</code> $\langle comma list \rangle$
<code>\clist_if_exist_p:c</code> ★	<code>\clist_if_exist:NTF</code> $\langle comma list \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$
<code>\clist_if_exist:NTF</code> ★	Tests whether the $\langle comma list \rangle$ is currently defined. This does not check that the $\langle comma$
<code>\clist_if_exist:cTF</code> ★	$list \rangle$ really is a comma list.

New: 2012-03-03

2 Adding data to comma lists

<code>\clist_set:Nn</code>	<code>\clist_set:Nn</code> $\langle comma list \rangle$ $\{\langle item_1 \rangle, \dots, \langle item_n \rangle\}$
<code>\clist_set:(NV No Nx cn cV co cx)</code>	
<code>\clist_gset:Nn</code>	
<code>\clist_gset:(NV No Nx cn cV co cx)</code>	

New: 2011-09-06

Sets $\langle comma list \rangle$ to contain the $\langle items \rangle$, removing any previous content from the variable. Spaces are removed from both sides of each item.

<code>\clist_put_left:Nn</code>	<code>\clist_put_left:Nn <comma list> {\<item_1>, ..., \<item_n>}</code>
<code>\clist_put_left:(NV No Nx cn cV co cx)</code>	
<code>\clist_gput_left:Nn</code>	
<code>\clist_gput_left:(NV No Nx cn cV co cx)</code>	

Updated: 2011-09-05

Appends the $\langle items \rangle$ to the left of the $\langle comma list \rangle$. Spaces are removed from both sides of each item.

<code>\clist_put_right:Nn</code>	<code>\clist_put_right:Nn <comma list> {\<item_1>, ..., \<item_n>}</code>
<code>\clist_put_right:(NV No Nx cn cV co cx)</code>	
<code>\clist_gput_right:Nn</code>	
<code>\clist_gput_right:(NV No Nx cn cV co cx)</code>	

Updated: 2011-09-05

Appends the $\langle items \rangle$ to the right of the $\langle comma list \rangle$. 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.

<code>\clist_remove_duplicates:N</code>	<code>\clist_remove_duplicates:N <comma list></code>
<code>\clist_remove_duplicates:c</code>	
<code>\clist_gremove_duplicates:N</code>	
<code>\clist_gremove_duplicates:c</code>	

Removes duplicate items from the $\langle comma list \rangle$, leaving the left most copy of each item in the $\langle comma list \rangle$. The $\langle item \rangle$ comparison takes place on a token basis, as for `\tl_if_eq:nn(TF)`.

T_EXhackers note: This function iterates through every item in the $\langle comma list \rangle$ and does a comparison with the $\langle items \rangle$ already checked. It is therefore relatively slow with large comma lists. Furthermore, it will not work if any of the items in the $\langle comma list \rangle$ contains `{`, `}`, or `#` (assuming the usual T_EX category codes apply).

<code>\clist_remove_all:Nn</code>	<code>\clist_remove_all:Nn <comma list> {\<item>}</code>
<code>\clist_remove_all:cn</code>	
<code>\clist_gremove_all:Nn</code>	
<code>\clist_gremove_all:cn</code>	

Updated: 2011-09-06

T_EXhackers note: The $\langle item \rangle$ may not contain `{`, `}`, or `#` (assuming the usual T_EX category codes apply).

```
\clist_reverse:N
\clist_reverse:c
\clist_greverse:N
\clist_greverse:c
```

New: 2014-07-18

```
\clist_reverse:n
```

New: 2014-07-18

```
\clist_reverse:N <comma list>
```

Reverses the order of items stored in the *<comma list>*.

```
\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.

T_EXhackers 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.

4 Comma list conditionals

```
\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 ★
```

New: 2014-07-05

```
\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
```

Updated: 2011-09-06

```
\clist_if_in:NnTF <comma list> {<item>} {<true code>} {<false code>}
```

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`.

T_EXhackers note: The *<item>* may not contain `{`, `}`, or `#` (assuming the usual T_EX 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` $\langle comma\ list \rangle$ $\langle function \rangle$

Applies $\langle function \rangle$ to every $\langle item \rangle$ stored in the $\langle comma\ list \rangle$. The $\langle function \rangle$ will receive one argument for each iteration. The $\langle items \rangle$ are returned from left to right. The function `\clist_map_inline:Nn` is in general more efficient than `\clist_map_function:NN`. 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` $\langle comma\ list \rangle$ $\{ \langle inline\ function \rangle \}$

Applies $\langle inline\ function \rangle$ to every $\langle item \rangle$ stored within the $\langle comma\ list \rangle$. The $\langle inline\ function \rangle$ should consist of code which will receive the $\langle item \rangle$ as #1. One in line mapping can be nested inside another. The $\langle items \rangle$ are returned from left to right.

`\clist_map_variable:NNn`
`\clist_map_variable:cNn`
`\clist_map_variable:nNn`

Updated: 2012-06-29

`\clist_map_variable:NNn` $\langle comma\ list \rangle$ $\langle tl\ var. \rangle$ $\{ \langle function\ using\ tl\ var. \rangle \}$

Stores each entry in the $\langle comma\ list \rangle$ in turn in the $\langle tl\ var. \rangle$ and applies the $\langle function\ using\ tl\ var. \rangle$. The $\langle function \rangle$ will usually consist of code making use of the $\langle tl\ var. \rangle$, but this is not enforced. One variable mapping can be nested inside another. The $\langle items \rangle$ 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

<code>\clist_use:Nnnn</code> ★	<code>\clist_use:Nnnn <clist var> {<separator between two>}</code>
<code>\clist_use:cnnn</code> ★	<code>{<separator between more than two>} {<separator between final two>}</code>

New: 2013-05-26

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.

<code>\clist_use:Nn</code> ★	<code>\clist_use:Nn <clist var> {<separator>}</code>
<code>\clist_use:cn</code> ★	

New: 2013-05-26

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.

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.

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` $\langle comma list \rangle$ $\langle token list variable \rangle$

Stores the left-most item from a $\langle comma list \rangle$ in the $\langle token list variable \rangle$ without removing it from the $\langle comma list \rangle$. The $\langle token list variable \rangle$ is assigned locally. If the $\langle comma list \rangle$ is empty the $\langle token list variable \rangle$ will contain the marker value `\q_no_value`.

`\clist_get:NNTF`
`\clist_get:cNTF`
New: 2012-05-14

`\clist_get:NNTF` $\langle comma list \rangle$ $\langle token list variable \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

If the $\langle comma list \rangle$ is empty, leaves the $\langle false code \rangle$ in the input stream. The value of the $\langle token list variable \rangle$ is not defined in this case and should not be relied upon. If the $\langle comma list \rangle$ is non-empty, stores the top item from the $\langle comma list \rangle$ in the $\langle token list variable \rangle$ without removing it from the $\langle comma list \rangle$. The $\langle token list variable \rangle$ is assigned locally.

`\clist_pop:NN`
`\clist_pop:cN`
Updated: 2011-09-06

`\clist_pop:NN` $\langle comma list \rangle$ $\langle token list variable \rangle$

Pops the left-most item from a $\langle comma list \rangle$ into the $\langle token list variable \rangle$, *i.e.* removes the item from the comma list and stores it in the $\langle token list variable \rangle$. Both of the variables are assigned locally.

`\clist_gpop:NN`
`\clist_gpop:cN`

`\clist_gpop:NN` $\langle comma list \rangle$ $\langle token list variable \rangle$

Pops the left-most item from a $\langle comma list \rangle$ into the $\langle token list variable \rangle$, *i.e.* removes the item from the comma list and stores it in the $\langle token list variable \rangle$. The $\langle comma list \rangle$ is modified globally, while the assignment of the $\langle token list variable \rangle$ is local.

`\clist_pop:NNTF`
`\clist_pop:cNTF`
New: 2012-05-14

`\clist_pop:NNTF` $\langle comma list \rangle$ $\langle token list variable \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

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

`\clist_gpop:NNTF`
`\clist_gpop:cNTF`
New: 2012-05-14

`\clist_gpop:NNTF` $\langle comma list \rangle$ $\langle token list variable \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

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

<code>\clist_push:Nn</code>	<code>\clist_push:Nn <comma list> {<items>}</code>
<code>\clist_push:(NV No Nx cn cV co cx)</code>	
<code>\clist_gpush:Nn</code>	
<code>\clist_gpush:(NV No Nx cn cV co cx)</code>	

Adds the $\{\langle items \rangle\}$ to the top of the $\langle comma list \rangle$. Spaces are removed from both sides of each item.

8 Using a single item

<code>\clist_item:Nn</code> ★	<code>\clist_item:Nn <comma list> {<integer expression>}</code>
<code>\clist_item:cn</code> ★	
<code>\clist_item:nn</code> ★	

New: 2014-07-17

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

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

9 Viewing comma lists

<code>\clist_show:N</code>	<code>\clist_show:N <comma list></code>
<code>\clist_show:c</code>	

Updated: 2015-08-03

Displays the entries in the $\langle comma list \rangle$ in the terminal.

<code>\clist_show:n</code>	<code>\clist_show:n {<tokens>}</code>
----------------------------	---

Updated: 2013-08-03

Displays the entries in the comma list in the terminal.

10 Constant and scratch comma lists

<code>\c_empty_clist</code>	Constant that is always empty.
-----------------------------	--------------------------------

New: 2012-07-02

<code>\l_tmpa_clist</code>	
<code>\l_tmpb_clist</code>	

New: 2011-09-06

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.

<hr/>	
<code>\g_tmpa_clist</code>	Scratch comma lists for global assignment. These are never used by the kernel code, and so are safe for use with any \LaTeX 3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
<code>\g_tmpb_clist</code>	
<hr/>	
New: 2011-09-06	
<hr/>	

Part XV

The l3prop package

Property lists

L^AT_EX3 implements a “property list” data type, which contain an unordered list of entries each of which consists of a $\langle key \rangle$ and an associated $\langle value \rangle$. The $\langle key \rangle$ and $\langle value \rangle$ may both be any $\langle balanced\ text \rangle$. 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
```

```
\prop_new:N  $\langle property\ list \rangle$ 
```

Creates a new $\langle property\ list \rangle$ or raises an error if the name is already taken. The declaration is global. The $\langle property\ list \rangle$ will initially contain no entries.

```
\prop_clear:N
\prop_clear:c
\prop_gclear:N
\prop_gclear:c
```

```
\prop_clear:N  $\langle property\ list \rangle$ 
```

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
```

```
\prop_clear_new:N  $\langle property\ list \rangle$ 
```

Ensures that the $\langle property\ list \rangle$ exists globally by applying `\prop_new:N` if necessary, then applies `\prop_(g)clear:N` to leave the list empty.

```
\prop_set_eq:NN
\prop_set_eq:(cN|Nc|cc)
\prop_gset_eq:NN
\prop_gset_eq:(cN|Nc|cc)
```

```
\prop_set_eq:NN  $\langle property\ list_1 \rangle$   $\langle property\ list_2 \rangle$ 
```

Sets the content of $\langle property\ list_1 \rangle$ equal to that of $\langle property\ list_2 \rangle$.

2 Adding entries to property lists

<code>\prop_put:Nnn</code> <code>\prop_put:(NnV Nno Nnx NVn NVV Non Noo cnn cnV cno cnx cVn cVV con coo)</code> <code>\prop_gput:Nnn</code> <code>\prop_gput:(NnV Nno Nnx NVn NVV Non Noo cnn cnV cno cnx cVn cVV con coo)</code>	<code>\prop_put:Nnn <property list></code> <code>{<key>} {<value>}</code>
--	--

Updated: 2012-07-09

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>*.

<code>\prop_put_if_new:Nnn</code> <code>\prop_put_if_new:cnn</code> <code>\prop_gput_if_new:Nnn</code> <code>\prop_gput_if_new:cnn</code>	<code>\prop_put_if_new:Nnn <property list> {<key>} {<value>}</code> <p>If the <i><key></i> is present in the <i><property list></i> then no action is taken. If the <i><key></i> is not present in the <i><property list></i> then a new entry is added. Both the <i><key></i> and <i><value></i> may contain any <i><balanced text></i>. The <i><key></i> is stored after processing with <code>\tl_to_str:n</code>, meaning that category codes are ignored.</p>
--	---

3 Recovering values from property lists

<code>\prop_get:NnN</code> <code>\prop_get:(NVN NoN cnN cVN coN)</code>	<code>\prop_get:NnN <property list> {<key>} <tl var></code>
--	---

Updated: 2011-08-28

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 \TeX group. See also `\prop_get:NnNTF`.

<code>\prop_pop:NnN</code> <code>\prop_pop:(NoN cnN coN)</code>	<code>\prop_pop:NnN <property list> {<key>} <tl var></code> <p>Recovers the <i><value></i> stored with <i><key></i> from the <i><property list></i>, and places this in the <i><token list variable></i>. If the <i><key></i> is not found in the <i><property list></i> then the <i><token list variable></i> will contain the special marker <code>\q_no_value</code>. The <i><key></i> and <i><value></i> are then deleted from the property list. Both assignments are local. See also <code>\prop_pop:NnNTF</code>.</p>
--	---

Updated: 2011-08-18

<code>\prop_gpop:NnN</code> <code>\prop_gpop:(NoN cnN coN)</code>	<code>\prop_gpop:NnN <property list> {<key>} <tl var></code> <p>Recovers the <i><value></i> stored with <i><key></i> from the <i><property list></i>, and places this in the <i><token list variable></i>. If the <i><key></i> is not found in the <i><property list></i> then the <i><token list variable></i> will contain the special marker <code>\q_no_value</code>. The <i><key></i> and <i><value></i> are then deleted from the property list. The <i><property list></i> is modified globally, while the assignment of the <i><token list variable></i> is local. See also <code>\prop_gpop:NnNTF</code>.</p>
--	---

Updated: 2011-08-18

<code>\prop_item:Nn</code> ★	<code>\prop_item:Nn</code> $\langle property list \rangle$ $\{\langle key \rangle\}$
<code>\prop_item:cn</code> ★	Expands to the $\langle value \rangle$ corresponding to the $\langle key \rangle$ in the $\langle property list \rangle$. If the $\langle key \rangle$ is missing, this has an empty expansion.
New: 2014-07-17	

T_EXhackers note: This function is slower than the non-expandable analogue `\prop_get:NnN`. The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the $\langle value \rangle$ will not expand further when appearing in an x-type argument expansion.

4 Modifying property lists

<code>\prop_remove:Nn</code>	<code>\prop_remove:Nn</code> $\langle property list \rangle$ $\{\langle key \rangle\}$
<code>\prop_remove:(NV cn cV)</code>	Removes the entry listed under $\langle key \rangle$ from the $\langle property list \rangle$. If the $\langle key \rangle$ is not found in the $\langle property list \rangle$ no change occurs, <i>i.e.</i> there is no need to test for the existence of a key before deleting it.
<code>\prop_gremove:Nn</code>	
<code>\prop_gremove:(NV cn cV)</code>	
New: 2012-05-12	

5 Property list conditionals

<code>\prop_if_exist_p:N</code> ★	<code>\prop_if_exist_p:N</code> $\langle property list \rangle$
<code>\prop_if_exist_p:c</code> ★	<code>\prop_if_exist:NTF</code> $\langle property list \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$
<code>\prop_if_exist:N\underline{TF}</code> ★	Tests whether the $\langle property list \rangle$ is currently defined. This does not check that the $\langle property list \rangle$ really is a property list variable.
<code>\prop_if_exist:c\underline{TF}</code> ★	
New: 2012-03-03	

<code>\prop_if_empty_p:N</code> ★	<code>\prop_if_empty_p:N</code> $\langle property list \rangle$
<code>\prop_if_empty_p:c</code> ★	<code>\prop_if_empty:NTF</code> $\langle property list \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$
<code>\prop_if_empty:N\underline{TF}</code> ★	Tests if the $\langle property list \rangle$ is empty (containing no entries).
<code>\prop_if_empty:c\underline{TF}</code> ★	

<code>\prop_if_in_p:Nn</code> ★	<code>\prop_if_in:NnTF</code> $\langle property list \rangle$ $\{\langle key \rangle\}$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$
<code>\prop_if_in_p:(NV No cn cV co)</code> ★	
<code>\prop_if_in:Nn\underline{TF}</code> ★	
<code>\prop_if_in:(NV No cn cV co)\underline{TF}</code> ★	
Updated: 2011-09-15	

Tests if the $\langle key \rangle$ is present in the $\langle property list \rangle$, making the comparison using the method described by `\str_if_eq:nnTF`.

T_EXhackers note: This function iterates through every key–value pair in the $\langle property list \rangle$ and is therefore slower than using the non-expandable `\prop_get: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.

<code>\prop_get:NnNTF</code> <code>\prop_get:(NVN NoN cnN cVN coN)TF</code>	<code>\prop_get:NnNTF <property list> {<key>} <token list variable></code> <code>{<true code>} {<false code>}</code>
--	---

Updated: 2012-05-19

If the $\langle key \rangle$ is not present in the $\langle property list \rangle$, leaves the $\langle false code \rangle$ in the input stream. The value of the $\langle token list variable \rangle$ is not defined in this case and should not be relied upon. If the $\langle key \rangle$ is present in the $\langle property list \rangle$, stores the corresponding $\langle value \rangle$ in the $\langle token list variable \rangle$ without removing it from the $\langle property list \rangle$, then leaves the $\langle true code \rangle$ in the input stream. The $\langle token list variable \rangle$ is assigned locally.

<code>\prop_pop:NnNTF</code> <code>\prop_pop:cnNTF</code>	<code>\prop_pop:NnNTF <property list> {<key>} <token list variable> {<true code>}</code> <code>{<false code>}</code>
--	---

New: 2011-08-18
Updated: 2012-05-19

If the $\langle key \rangle$ is not present in the $\langle property list \rangle$, leaves the $\langle false code \rangle$ in the input stream. The value of the $\langle token list variable \rangle$ is not defined in this case and should not be relied upon. If the $\langle key \rangle$ is present in the $\langle property list \rangle$, pops the corresponding $\langle value \rangle$ in the $\langle token list variable \rangle$, *i.e.* removes the item from the $\langle property list \rangle$. Both the $\langle property list \rangle$ and the $\langle token list variable \rangle$ are assigned locally.

<code>\prop_gpop:NnNTF</code> <code>\prop_gpop:cnNTF</code>	<code>\prop_gpop:NnNTF <property list> {<key>} <token list variable> {<true code>}</code> <code>{<false code>}</code>
--	--

New: 2011-08-18
Updated: 2012-05-19

If the $\langle key \rangle$ is not present in the $\langle property list \rangle$, leaves the $\langle false code \rangle$ in the input stream. The value of the $\langle token list variable \rangle$ is not defined in this case and should not be relied upon. If the $\langle key \rangle$ is present in the $\langle property list \rangle$, pops the corresponding $\langle value \rangle$ in the $\langle token list variable \rangle$, *i.e.* removes the item from the $\langle property list \rangle$. The $\langle property list \rangle$ is modified globally, while the $\langle token list variable \rangle$ is assigned locally.

7 Mapping to property lists

<code>\prop_map_function:NN</code> ☆ <code>\prop_map_function:cN</code> ☆	<code>\prop_map_function:NN <property list> <function></code>
--	---

Updated: 2013-01-08

Applies $\langle function \rangle$ to every $\langle entry \rangle$ stored in the $\langle property list \rangle$. The $\langle function \rangle$ will receive two argument for each iteration: the $\langle key \rangle$ and associated $\langle value \rangle$. The order in which $\langle entries \rangle$ are returned is not defined and should not be relied upon.

<code>\prop_map_inline:Nn</code>	<code>\prop_map_inline:Nn <property list> {(inline function)}</code>
<code>\prop_map_inline:cn</code>	Applies <i><inline function></i> to every <i><entry></i> stored within the <i><property list></i> . The <i><inline function></i> should consist of code which will receive the <i><key></i> as #1 and the <i><value></i> as #2. The order in which <i><entries></i> are returned is not defined and should not be relied upon.
Updated: 2013-01-08	

<code>\prop_map_break:☆</code>	<code>\prop_map_break:</code>
Updated: 2012-06-29	Used to terminate a <code>\prop_map...</code> function before all entries in the <i><property list></i> 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 TeX errors.

<code>\prop_map_break:n ☆</code>	<code>\prop_map_break:n {(tokens)}</code>
Updated: 2012-06-29	Used to terminate a <code>\prop_map...</code> function before all entries in the <i><property list></i> have been processed, inserting the <i><tokens></i> 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 TeX errors.

8 Viewing property lists

<code>\prop_show:N</code>	<code>\prop_show:N <property list></code>
<code>\prop_show:c</code>	Displays the entries in the <i><property list></i> in the terminal.
Updated: 2015-08-01	

9 Scratch property lists

<code>\l_tmpa_prop</code>	Scratch property lists for local assignment. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
<code>\l_tmpb_prop</code>	
New: 2012-06-23	

<code>\g_tmpa_prop</code>	Scratch property lists for global assignment. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
<code>\g_tmpb_prop</code>	
New: 2012-06-23	

10 Constants

<code>\c_empty_prop</code>	A permanently-empty property list used for internal comparisons.
----------------------------	--

11 Internal property list functions

<code>\s__prop</code>	The internal token used at the beginning of property lists. This is also used after each $\langle key \rangle$ (see <code>__prop_pair:wn</code>).
-----------------------	---

<code>__prop_pair:wn</code>	<code>__prop_pair:wn $\langle key \rangle$ \s__prop {$\langle item \rangle$}</code>
	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.

<code>\l__prop_internal_tl</code>	Token list used to store new key–value pairs to be inserted by functions of the <code>\prop_put:Nnn</code> family.
-----------------------------------	--

<code>__prop_split:NnTF</code>	<code>__prop_split:NnTF $\langle property list \rangle$ {$\langle key \rangle$} {$\langle true code \rangle$} {$\langle false code \rangle$}</code>
Updated: 2013-01-08	Splits the $\langle property list \rangle$ at the $\langle key \rangle$, giving three token lists: the $\langle extract \rangle$ of $\langle property list \rangle$ before the $\langle key \rangle$, the $\langle value \rangle$ associated with the $\langle key \rangle$ and the $\langle extract \rangle$ of the $\langle property list \rangle$ after the $\langle value \rangle$. Both $\langle extracts \rangle$ retain the internal structure of a property list, and the concatenation of the two $\langle extracts \rangle$ is a property list. If the $\langle key \rangle$ is present in the $\langle property list \rangle$ then the $\langle true code \rangle$ is left in the input stream, with #1, #2, and #3 replaced by the first $\langle extract \rangle$, the $\langle value \rangle$, and the second extract. If the $\langle key \rangle$ is not present in the $\langle property list \rangle$ then the $\langle false code \rangle$ is left in the input stream, with no trailing material. Both $\langle true code \rangle$ and $\langle false code \rangle$ are used in the replacement text of a macro defined internally, hence macro parameter characters should be doubled, except #1, #2, and #3 which stand in the $\langle true code \rangle$ for the three extracts from the property list. The $\langle key \rangle$ comparison takes place as described for <code>\str_if_eq:nn</code> .

Part XVI

The l3box 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

<code>\box_new:N</code>	<code>\box_new:N <box></code>
<code>\box_new:c</code>	Creates a new $\langle box \rangle$ or raises an error if the name is already taken. The declaration is global. The $\langle box \rangle$ will initially be void.

<code>\box_clear:N</code>	<code>\box_clear:N <box></code>
<code>\box_clear:c</code>	Clears the content of the $\langle box \rangle$ by setting the box equal to <code>\c_void_box</code> .
<code>\box_gclear:N</code>	
<code>\box_gclear:c</code>	

<code>\box_clear_new:N</code>	<code>\box_clear_new:N <box></code>
<code>\box_clear_new:c</code>	Ensures that the $\langle box \rangle$ exists globally by applying <code>\box_new:N</code> if necessary, then applies <code>\box_(g)clear:N</code> to leave the $\langle box \rangle$ empty.
<code>\box_gclear_new:N</code>	
<code>\box_gclear_new:c</code>	

<code>\box_set_eq:NN</code>	<code>\box_set_eq:NN <box₁> <box₂></code>
<code>\box_set_eq:(cN Nc cc)</code>	Sets the content of $\langle box_1 \rangle$ equal to that of $\langle box_2 \rangle$.
<code>\box_gset_eq:NN</code>	
<code>\box_gset_eq:(cN Nc cc)</code>	

<code>\box_set_eq_clear:NN</code>	<code>\box_set_eq_clear:NN <box₁> <box₂></code>
<code>\box_set_eq_clear:(cN Nc cc)</code>	Sets the content of $\langle box_1 \rangle$ within the current TeX group equal to that of $\langle box_2 \rangle$, then clears $\langle box_2 \rangle$ globally.

<code>\box_gset_eq_clear:NN</code>	<code>\box_gset_eq_clear:NN <box₁> <box₂></code>
<code>\box_gset_eq_clear:(cN Nc cc)</code>	Sets the content of $\langle box_1 \rangle$ equal to that of $\langle box_2 \rangle$, then clears $\langle box_2 \rangle$. These assignments are global.

<code>\box_if_exist_p:N</code> ★	<code>\box_if_exist_p:N</code> $\langle box \rangle$
<code>\box_if_exist_p:c</code> ★	<code>\box_if_exist:N</code> $\langle box \rangle$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$
<code>\box_if_exist:N</code> ★	Tests whether the $\langle box \rangle$ is currently defined. This does not check that the $\langle box \rangle$ really is a box.
<code>\box_if_exist:c</code> ★	

New: 2012-03-03

2 Using boxes

<code>\box_use:N</code>	<code>\box_use:N</code> $\langle box \rangle$
<code>\box_use:c</code>	Inserts the current content of the $\langle box \rangle$ onto the current list for typesetting.

T_EXhackers note: This is the T_EX primitive `\copy`.

<code>\box_use_clear:N</code>	<code>\box_use_clear:N</code> $\langle box \rangle$
<code>\box_use_clear:c</code>	Inserts the current content of the $\langle box \rangle$ onto the current list for typesetting, then globally clears the content of the $\langle box \rangle$.

T_EXhackers note: This is the T_EX primitive `\box`.

<code>\box_move_right:nn</code>	<code>\box_move_right:nn</code> $\{\langle dimexpr \rangle\}$ $\{\langle box\ function \rangle\}$
<code>\box_move_left:nn</code>	This function operates in vertical mode, and inserts the material specified by the $\langle box\ function \rangle$ such that its reference point is displaced horizontally by the given $\langle dimexpr \rangle$ from the reference point for typesetting, to the right or left as appropriate. The $\langle box\ function \rangle$ should be a box operation such as <code>\box_use:N</code> $\langle box \rangle$ or a “raw” box specification such as <code>\vbox:n</code> $\{ xyz \}$.

<code>\box_move_up:nn</code>	<code>\box_move_up:nn</code> $\{\langle dimexpr \rangle\}$ $\{\langle box\ function \rangle\}$
<code>\box_move_down:nn</code>	This function operates in horizontal mode, and inserts the material specified by the $\langle box\ function \rangle$ such that its reference point is displaced vertical by the given $\langle dimexpr \rangle$ from the reference point for typesetting, up or down as appropriate. The $\langle box\ function \rangle$ should be a box operation such as <code>\box_use:N</code> $\langle box \rangle$ or a “raw” box specification such as <code>\vbox:n</code> $\{ xyz \}$.

3 Measuring and setting box dimensions

<code>\box_dp:N</code>	<code>\box_dp:N</code> $\langle box \rangle$
<code>\box_dp:c</code>	Calculates the depth (below the baseline) of the $\langle box \rangle$ in a form suitable for use in a $\langle dimension\ expression \rangle$.

T_EXhackers note: This is the T_EX primitive `\dp`.

<code>\box_ht:N</code>	<code>\box_ht:N <box></code>
<code>\box_ht:c</code>	Calculates the height (above the baseline) of the $\langle box \rangle$ in a form suitable for use in a $\langle dimension expression \rangle$.

T_EXhackers note: This is the T_EX primitive `\ht`.

<code>\box_wd:N</code>	<code>\box_wd:N <box></code>
<code>\box_wd:c</code>	Calculates the width of the $\langle box \rangle$ in a form suitable for use in a $\langle dimension expression \rangle$.

T_EXhackers note: This is the T_EX primitive `\wd`.

<code>\box_set_dp:Nn</code>	<code>\box_set_dp:Nn <box> {<dimension expression>}</code>
<code>\box_set_dp:cn</code>	Set the depth (below the baseline) of the $\langle box \rangle$ to the value of the $\{ \langle dimension expression \rangle \}$. This is a global assignment.

Updated: 2011-10-22

<code>\box_set_ht:Nn</code>	<code>\box_set_ht:Nn <box> {<dimension expression>}</code>
<code>\box_set_ht:cn</code>	Set the height (above the baseline) of the $\langle box \rangle$ to the value of the $\{ \langle dimension expression \rangle \}$. This is a global assignment.

Updated: 2011-10-22

<code>\box_set_wd:Nn</code>	<code>\box_set_wd:Nn <box> {<dimension expression>}</code>
<code>\box_set_wd:cn</code>	Set the width of the $\langle box \rangle$ to the value of the $\{ \langle dimension expression \rangle \}$. This is a global assignment.

Updated: 2011-10-22

4 Box conditionals

<code>\box_if_empty_p:N</code> ★	<code>\box_if_empty_p:N <box></code>
<code>\box_if_empty_p:c</code> ★	<code>\box_if_empty:NTF <box> {<true code>} {<false code>}</code>
<code>\box_if_empty:NTF</code> ★	Tests if $\langle box \rangle$ is a empty (equal to <code>\c_empty_box</code>).
<code>\box_if_empty:cTF</code> ★	

<code>\box_if_horizontal_p:N</code> ★	<code>\box_if_horizontal_p:N <box></code>
<code>\box_if_horizontal_p:c</code> ★	<code>\box_if_horizontal:NTF <box> {<true code>} {<false code>}</code>
<code>\box_if_horizontal:NTF</code> ★	Tests if $\langle box \rangle$ is a horizontal box.
<code>\box_if_horizontal:cTF</code> ★	

<code>\box_if_vertical_p:N</code> ★	<code>\box_if_vertical_p:N <box></code>
<code>\box_if_vertical_p:c</code> ★	<code>\box_if_vertical:NTF <box> {<true code>} {<false code>}</code>
<code>\box_if_vertical:NTF</code> ★	Tests if $\langle box \rangle$ is a vertical box.
<code>\box_if_vertical:cTF</code> ★	

5 The last box inserted

<hr/> <code>\box_set_to_last:N</code>	<code>\box_set_to_last:N</code> $\langle box \rangle$
<code>\box_set_to_last:c</code>	
<code>\box_gset_to_last:N</code>	Sets the $\langle box \rangle$ equal to the last item (box) added to the current partial list, removing the item from the list at the same time. When applied to the main vertical list, the $\langle box \rangle$ will always be void as it is not possible to recover the last added item.
<code>\box_gset_to_last:c</code>	

6 Constant boxes

<hr/> <code>\c_empty_box</code>	This is a permanently empty box, which is neither set as horizontal nor vertical.
<code>Updated: 2012-11-04</code>	

7 Scratch boxes

<hr/> <code>\l_tmpa_box</code>	Scratch boxes for local assignment. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
<code>\l_tmpb_box</code>	
<code>Updated: 2012-11-04</code>	

<hr/> <code>\g_tmpa_box</code>	Scratch boxes for global assignment. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
<code>\g_tmpb_box</code>	

8 Viewing box contents

<hr/> <code>\box_show:N</code>	<code>\box_show:N</code> $\langle box \rangle$
<code>\box_show:c</code>	Shows full details of the content of the $\langle box \rangle$ in the terminal.
<code>Updated: 2012-05-11</code>	
<hr/> <code>\box_show:Nnn</code>	<code>\box_show:Nnn</code> $\langle box \rangle$ $\langle intexpr_1 \rangle$ $\langle intexpr_2 \rangle$
<code>\box_show:cnn</code>	Display the contents of $\langle box \rangle$ in the terminal, showing the first $\langle intexpr_1 \rangle$ items of the box, and descending into $\langle intexpr_2 \rangle$ group levels.
<code>New: 2012-05-11</code>	
<hr/> <code>\box_log:N</code>	<code>\box_log:N</code> $\langle box \rangle$
<code>\box_log:c</code>	Writes full details of the content of the $\langle box \rangle$ to the log.
<code>New: 2012-05-11</code>	

<hr/> <code>\box_log:Nnn</code> <hr/>	<code>\box_log:Nnn <box> <intexpr₁> <intexpr₂></code>
<code>\box_log:cnn</code> <hr/>	Writes the contents of $\langle box \rangle$ to the log, showing the first $\langle intexpr_1 \rangle$ items of the box, and descending into $\langle intexpr_2 \rangle$ group levels.
<code>New: 2012-05-11</code> <hr/>	

9 Horizontal mode boxes

<hr/> <code>\hbox:n</code> <hr/>	<code>\hbox:n {\langle contents \rangle}</code>
	Typesets the $\langle contents \rangle$ into a horizontal box of natural width and then includes this box in the current list for typesetting.

<hr/> <code>\hbox_to_wd:nn</code> <hr/>	<code>\hbox_to_wd:nn {\langle dimexpr \rangle} {\langle contents \rangle}</code>
	Typesets the $\langle contents \rangle$ into a horizontal box of width $\langle dimexpr \rangle$ and then includes this box in the current list for typesetting.

<hr/> <code>\hbox_to_zero:n</code> <hr/>	<code>\hbox_to_zero:n {\langle contents \rangle}</code>
	Typesets the $\langle contents \rangle$ into a horizontal box of zero width and then includes this box in the current list for typesetting.

<hr/> <code>\hbox_set:Nn</code> <hr/>	<code>\hbox_set:Nn <box> {\langle contents \rangle}</code>
<code>\hbox_set:cn</code> <hr/>	Typesets the $\langle contents \rangle$ at natural width and then stores the result inside the $\langle box \rangle$.
<code>\hbox_gset:Nn</code> <hr/>	
<code>\hbox_gset:cn</code> <hr/>	

<hr/> <code>\hbox_set_to_wd:Nnn</code> <hr/>	<code>\hbox_set_to_wd:Nnn <box> {\langle dimexpr \rangle} {\langle contents \rangle}</code>
<code>\hbox_set_to_wd:cnn</code> <hr/>	Typesets the $\langle contents \rangle$ to the width given by the $\langle dimexpr \rangle$ and then stores the result inside the $\langle box \rangle$.
<code>\hbox_gset_to_wd:Nnn</code> <hr/>	
<code>\hbox_gset_to_wd:cnn</code> <hr/>	

<hr/> <code>\hbox_overlap_right:n</code> <hr/>	<code>\hbox_overlap_right:n {\langle contents \rangle}</code>
	Typesets the $\langle contents \rangle$ into a horizontal box of zero width such that material will protrude to the right of the insertion point.

<hr/> <code>\hbox_overlap_left:n</code> <hr/>	<code>\hbox_overlap_left:n {\langle contents \rangle}</code>
	Typesets the $\langle contents \rangle$ into a horizontal box of zero width such that material will protrude to the left of the insertion point.

<hr/> <code>\hbox_set:Nw</code> <hr/>	<code>\hbox_set:Nw <box> <contents> \hbox_set_end:</code>
<code>\hbox_set:cw</code> <hr/>	Typesets the $\langle contents \rangle$ at natural width and then stores the result inside the $\langle box \rangle$. In contrast to <code>\hbox_set:Nn</code> this function does not absorb the argument when finding the $\langle content \rangle$, and so can be used in circumstances where the $\langle content \rangle$ may not be a simple argument.
<code>\hbox_set_end:</code> <hr/>	
<code>\hbox_gset:Nw</code> <hr/>	
<code>\hbox_gset:cw</code> <hr/>	
<code>\hbox_gset_end:</code> <hr/>	

`\hbox_unpack:N`
`\hbox_unpack:c`

`\hbox_unpack:N` $\langle box \rangle$

Unpacks the content of the horizontal $\langle box \rangle$, retaining any stretching or shrinking applied when the $\langle box \rangle$ was set.

T_EXhackers note: This is the T_EX primitive `\unhcopy`.

`\hbox_unpack_clear:N`
`\hbox_unpack_clear:c`

`\hbox_unpack_clear:N` $\langle box \rangle$

Unpacks the content of the horizontal $\langle box \rangle$, retaining any stretching or shrinking applied when the $\langle box \rangle$ was set. The $\langle box \rangle$ is then cleared globally.

T_EXhackers note: This is the T_EX primitive `\unhbox`.

10 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`

`\vbox:n` $\{\langle contents \rangle\}$

Updated: 2011-12-18

Typesets the $\langle contents \rangle$ into a vertical box of natural height and includes this box in the current list for typesetting.

T_EXhackers note: This is the T_EX primitive `\vbox`.

`\vbox_top:n`

`\vbox_top:n` $\{\langle contents \rangle\}$

Updated: 2011-12-18

Typesets the $\langle contents \rangle$ into a vertical box of natural height and includes this box in the current list for typesetting. The baseline of the box will be equal to that of the *first* item added to the box.

T_EXhackers note: This is the T_EX primitive `\vtop`.

`\vbox_to_ht:nn`

`\vbox_to_ht:nn` $\{\langle dimexpr \rangle\} \{\langle contents \rangle\}$

Updated: 2011-12-18

Typesets the $\langle contents \rangle$ into a vertical box of height $\langle dimexpr \rangle$ and then includes this box in the current list for typesetting.

`\vbox_to_zero:n`

`\vbox_to_zero:n` $\{\langle contents \rangle\}$

Updated: 2011-12-18

Typesets the $\langle contents \rangle$ into a vertical box of zero height and then includes this box in the current list for typesetting.

<code>\vbox_set:Nn</code>	<code>\vbox_set:Nn <box> {<contents>}</code>
---------------------------	--

<code>\vbox_set:cn</code>

Typesets the $\langle contents \rangle$ at natural height and then stores the result inside the $\langle box \rangle$.

<code>\vbox_gset:Nn</code>

<code>\vbox_gset:cn</code>

Updated: 2011-12-18

<code>\vbox_set_top:Nn</code>

<code>\vbox_set_top:Nn <box> {<contents>}</code>
--

<code>\vbox_set_top:cn</code>

Typesets the $\langle contents \rangle$ at natural height and then stores the result inside the $\langle box \rangle$. The baseline of the box will be equal to that of the *first* item added to the box.

<code>\vbox_gset_top:Nn</code>

<code>\vbox_gset_top:cn</code>

Updated: 2011-12-18

<code>\vbox_set_to_ht:Nnn</code>

<code>\vbox_set_to_ht:Nnn <box> {<dimexpr>} {<contents>}</code>

<code>\vbox_set_to_ht:cnn</code>

Typesets the $\langle contents \rangle$ to the height given by the $\langle dimexpr \rangle$ and then stores the result inside the $\langle box \rangle$.

<code>\vbox_gset_to_ht:Nnn</code>

<code>\vbox_gset_to_ht:cnn</code>

Updated: 2011-12-18

<code>\vbox_set:Nw</code>

<code>\vbox_set:Nw <box> <contents> \vbox_set_end:</code>

<code>\vbox_set:cw</code>

Typesets the $\langle contents \rangle$ at natural height and then stores the result inside the $\langle box \rangle$. In contrast to `\vbox_set:Nn` this function does not absorb the argument when finding the $\langle content \rangle$, and so can be used in circumstances where the $\langle content \rangle$ may not be a simple argument.

<code>\vbox_gset:Nw</code>

<code>\vbox_gset:cw</code>

<code>\vbox_gset_end:</code>

Updated: 2011-12-18

<code>\vbox_set_split_to_ht:NNn</code>
--

<code>\vbox_set_split_to_ht:NNn <box₁> <box₂> {<dimexpr>}</code>
--

Updated: 2011-10-22

Sets $\langle box_1 \rangle$ to contain material to the height given by the $\langle dimexpr \rangle$ by removing content from the top of $\langle box_2 \rangle$ (which must be a vertical box).

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

<code>\vbox_unpack:N</code>

<code>\vbox_unpack:N <box></code>

<code>\vbox_unpack:c</code>

Unpacks the content of the vertical $\langle box \rangle$, retaining any stretching or shrinking applied when the $\langle box \rangle$ was set.

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

<code>\vbox_unpack_clear:N</code>

<code>\vbox_unpack:N <box></code>

<code>\vbox_unpack_clear:c</code>

Unpacks the content of the vertical $\langle box \rangle$, retaining any stretching or shrinking applied when the $\langle box \rangle$ was set. The $\langle box \rangle$ is then cleared globally.

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

11 Primitive box conditionals

`\if_hbox:N` ★ `\if_hbox:N` $\langle box \rangle$
 $\langle true\ code \rangle$
`\else:`
 $\langle false\ code \rangle$
`\fi:`

Tests if $\langle box \rangle$ is a horizontal box.

T_EXhackers note: This is the T_EX primitive `\ifhbox`.

`\if_vbox:N` ★ `\if_vbox:N` $\langle box \rangle$
 $\langle true\ code \rangle$
`\else:`
 $\langle false\ code \rangle$
`\fi:`

Tests if $\langle box \rangle$ is a vertical box.

T_EXhackers note: This is the T_EX primitive `\ifvbox`.

`\if_box_empty:N` ★ `\if_box_empty:N` $\langle box \rangle$
 $\langle true\ code \rangle$
`\else:`
 $\langle false\ code \rangle$
`\fi:`

Tests if $\langle box \rangle$ is an empty (void) box.

T_EXhackers note: This is the T_EX primitive `\ifvoid`.

Part XVII

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` $\langle coffin \rangle$

Creates a new $\langle coffin \rangle$ or raises an error if the name is already taken. The declaration is global. The $\langle coffin \rangle$ will initially be empty.

`\coffin_clear:N`

`\coffin_clear:c`

New: 2011-08-17

`\coffin_clear:N` $\langle coffin \rangle$

Clears the content of the $\langle coffin \rangle$ within the current \TeX group level.

`\coffin_set_eq:NN`

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

New: 2011-08-17

`\coffin_set_eq:NN` $\langle coffin_1 \rangle$ $\langle coffin_2 \rangle$

Sets both the content and poles of $\langle coffin_1 \rangle$ equal to those of $\langle coffin_2 \rangle$ within the current \TeX 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` $\langle box \rangle$

`\coffin_if_exist:NTF` $\langle box \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Tests whether the $\langle coffin \rangle$ is currently defined.

2 Setting coffin content and poles

All coffin functions create and manipulate coffins locally within the current \TeX group level.

`\hcoffin_set:Nn`

`\hcoffin_set:cn`

New: 2011-08-17

Updated: 2011-09-03

`\hcoffin_set:Nn` $\langle coffin \rangle$ $\{\langle material \rangle\}$

Typesets the $\langle material \rangle$ in horizontal mode, storing the result in the $\langle coffin \rangle$. The standard poles for the $\langle coffin \rangle$ are then set up based on the size of the typeset material.

<hr/> <code>\hcoffin_set:Nw</code> <code>\hcoffin_set:cw</code> <code>\hcoffin_set_end:</code> <hr/> New: 2011-09-10	<code>\hcoffin_set:Nw <coffin> <material> \hcoffin_set_end:</code> Typesets the $\langle material \rangle$ in horizontal mode, storing the result in the $\langle coffin \rangle$. The standard poles for the $\langle coffin \rangle$ are then set up based on the size of the typeset material. These functions are useful for setting the entire contents of an environment in a coffin.
<hr/> <code>\vcoffin_set:Nnn</code> <code>\vcoffin_set:cnn</code> <hr/> New: 2011-08-17 Updated: 2012-05-22	<code>\vcoffin_set:Nnn <coffin> {\width} {\material}</code> 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.
<hr/> <code>\vcoffin_set:Nnw</code> <code>\vcoffin_set:cnw</code> <code>\vcoffin_set_end:</code> <hr/> New: 2011-09-10 Updated: 2012-05-22	<code>\vcoffin_set:Nnw <coffin> {\width} <material> \vcoffin_set_end:</code> 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.
<hr/> <code>\coffin_set_horizontal_pole:Nnn</code> <code>\coffin_set_horizontal_pole:cnn</code> <hr/> New: 2012-07-20	<code>\coffin_set_horizontal_pole:Nnn <coffin> {\pole} {\offset}</code> 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.
<hr/> <code>\coffin_set_vertical_pole:Nnn</code> <code>\coffin_set_vertical_pole:cnn</code> <hr/> New: 2012-07-20	<code>\coffin_set_vertical_pole:Nnn <coffin> {\pole} {\offset}</code> 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

<hr/>	<hr/>
<code>\coffin_attach:NnnNnnnn</code>	<code>\coffin_attach:NnnNnnnn</code>
<code>\coffin_attach:(cnnNnnnn Nnnnnnn cnnnnnn)</code>	<code>\coffin_attach:(cnnNnnnn Nnnnnnn cnnnnnn)</code>
	<code>\coffin_1 \{ \coffin_1-pole_1 \} \{ \coffin_1-pole_2 \}</code>
	<code>\coffin_2 \{ \coffin_2-pole_1 \} \{ \coffin_2-pole_2 \}</code>
	<code>\{ \langle x-offset \rangle \} \{ \langle y-offset \rangle \}</code>

This function attaches $\langle coffin_2 \rangle$ to $\langle coffin_1 \rangle$ such that the bounding box of $\langle coffin_1 \rangle$ is not altered, *i.e.* $\langle coffin_2 \rangle$ can protrude outside of the bounding box of the coffin. The alignment is carried out by first calculating $\langle handle_1 \rangle$, the point of intersection of $\langle coffin_1-pole_1 \rangle$ and $\langle coffin_1-pole_2 \rangle$, and $\langle handle_2 \rangle$, the point of intersection of $\langle coffin_2-pole_1 \rangle$ and $\langle coffin_2-pole_2 \rangle$. $\langle coffin_2 \rangle$ is then attached to $\langle coffin_1 \rangle$ such that the relationship between $\langle handle_1 \rangle$ and $\langle handle_2 \rangle$ is described by the $\langle x-offset \rangle$ and $\langle y-offset \rangle$. The two offsets should be given as dimension expressions.

<hr/>	<hr/>
<code>\coffin_join:NnnNnnnn</code>	<code>\coffin_join:NnnNnnnn</code>
<code>\coffin_join:(cnnNnnnn Nnnnnnn cnnnnnn)</code>	<code>\coffin_join:(cnnNnnnn Nnnnnnn cnnnnnn)</code>
	<code>\coffin_1 \{ \coffin_1-pole_1 \} \{ \coffin_1-pole_2 \}</code>
	<code>\coffin_2 \{ \coffin_2-pole_1 \} \{ \coffin_2-pole_2 \}</code>
	<code>\{ \langle x-offset \rangle \} \{ \langle y-offset \rangle \}</code>

This function joins $\langle coffin_2 \rangle$ to $\langle coffin_1 \rangle$ such that the bounding box of $\langle coffin_1 \rangle$ may expand. The new bounding box will cover the area containing the bounding boxes of the two original coffins. The alignment is carried out by first calculating $\langle handle_1 \rangle$, the point of intersection of $\langle coffin_1-pole_1 \rangle$ and $\langle coffin_1-pole_2 \rangle$, and $\langle handle_2 \rangle$, the point of intersection of $\langle coffin_2-pole_1 \rangle$ and $\langle coffin_2-pole_2 \rangle$. $\langle coffin_2 \rangle$ is then attached to $\langle coffin_1 \rangle$ such that the relationship between $\langle handle_1 \rangle$ and $\langle handle_2 \rangle$ is described by the $\langle x-offset \rangle$ and $\langle y-offset \rangle$. The two offsets should be given as dimension expressions.

<hr/>	<hr/>
<code>\coffin_typeset:Nnnnn</code>	<code>\coffin_typeset:Nnnnn \langle coffin \rangle \{ \langle pole_1 \rangle \} \{ \langle pole_2 \rangle \}</code>
<code>\coffin_typeset:cnnnn</code>	<code>\{ \langle x-offset \rangle \} \{ \langle y-offset \rangle \}</code>

Updated: 2012-07-20

Typesetting is carried out by first calculating $\langle handle \rangle$, the point of intersection of $\langle pole_1 \rangle$ and $\langle pole_2 \rangle$. The coffin is then typeset in horizontal mode such that the relationship between the current reference point in the document and the $\langle handle \rangle$ is described by the $\langle x-offset \rangle$ and $\langle y-offset \rangle$. The two offsets should be given as dimension expressions. Typesetting a coffin is therefore analogous to carrying out an alignment where the “parent” coffin is the current insertion point.

4 Measuring coffins

<hr/>	<hr/>
<code>\coffin_dp:N</code>	<code>\coffin_dp:N \langle coffin \rangle</code>
<code>\coffin_dp:c</code>	

Calculates the depth (below the baseline) of the $\langle coffin \rangle$ in a form suitable for use in a $\langle dimension expression \rangle$.

<hr/> <code>\coffin_ht:N</code> <hr/>	<code>\coffin_ht:N</code> $\langle coffin \rangle$
<code>\coffin_ht:c</code> <hr/>	Calculates the height (above the baseline) of the $\langle coffin \rangle$ in a form suitable for use in a $\langle dimension expression \rangle$.
<hr/> <code>\coffin_wd:N</code> <hr/>	<code>\coffin_wd:N</code> $\langle coffin \rangle$
<code>\coffin_wd:c</code> <hr/>	Calculates the width of the $\langle coffin \rangle$ in a form suitable for use in a $\langle dimension expression \rangle$.

5 Coffin diagnostics

<hr/> <code>\coffin_display_handles:Nn</code> <hr/>	<code>\coffin_display_handles:Nn</code> $\langle coffin \rangle$ $\{ \langle color \rangle \}$
<code>\coffin_display_handles:cn</code> <hr/>	This function first calculates the intersections between all of the $\langle poles \rangle$ of the $\langle coffin \rangle$ to give a set of $\langle handles \rangle$. It then prints the $\langle coffin \rangle$ at the current location in the source, with the position of the $\langle handles \rangle$ marked on the coffin. The $\langle handles \rangle$ will be labelled as part of this process: the locations of the $\langle handles \rangle$ and the labels are both printed in the $\langle color \rangle$ specified.
Updated: 2011-09-02 <hr/>	

<hr/> <code>\coffin_mark_handle:Nnnn</code> <hr/>	<code>\coffin_mark_handle:Nnnn</code> $\langle coffin \rangle$ $\{ \langle pole_1 \rangle \}$ $\{ \langle pole_2 \rangle \}$ $\{ \langle color \rangle \}$
<code>\coffin_mark_handle:cnnn</code> <hr/>	This function first calculates the $\langle handle \rangle$ for the $\langle coffin \rangle$ as defined by the intersection of $\langle pole_1 \rangle$ and $\langle pole_2 \rangle$. It then marks the position of the $\langle handle \rangle$ on the $\langle coffin \rangle$. The $\langle handle \rangle$ will be labelled as part of this process: the location of the $\langle handle \rangle$ and the label are both printed in the $\langle color \rangle$ specified.
Updated: 2011-09-02 <hr/>	

<hr/> <code>\coffin_show_structure:N</code> <hr/>	<code>\coffin_show_structure:N</code> $\langle coffin \rangle$
<code>\coffin_show_structure:c</code> <hr/>	This function shows the structural information about the $\langle coffin \rangle$ in the terminal. The width, height and depth of the typeset material are given, along with the location of all of the poles of the coffin.
Updated: 2015-08-01 <hr/>	
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.	

5.1 Constants and variables

<hr/> <code>\c_empty_coffin</code> <hr/>	A permanently empty coffin.
<hr/> <code>\l_tmpa_coffin</code> <hr/>	Scratch coffins for local assignment. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
<code>\l_tmpb_coffin</code> <hr/>	
New: 2012-06-19	

Part XVIII

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

```
\color_group_begin:
```

```
...
```

```
\color_group_end:
```

Creates a color group: one used to “trap” color settings.

```
\color_ensure_current:
```

New: 2011-09-03

```
\color_ensure_current:
```

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 XIX

The l3msg package

Messages

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

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

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

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 `LaTeX` 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 {<module>} {<message>} {<text>} {<more text>}
```

Creates a *<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 at the time the message is used. An error will be raised if the *<message>* already exists.

<code>\msg_set:nnnn</code> <code>\msg_set:nnn</code> <code>\msg_gset:nnnn</code> <code>\msg_gset:nnn</code>	<code>\msg_set:nnnn {<module>} {<message>} {<text>} {<more text>}</code> <p>Sets up the text for a <i><message></i> for a given <i><module></i>. The message will be defined to first give <i><text></i> and then <i><more text></i> if the user requests it. If no <i><more text></i> is available then a standard text is given instead. Within <i><text></i> and <i><more text></i> four parameters (#1 to #4) can be used: these will be supplied at the time the message is used.</p>
--	---

<code>\msg_if_exist_p:nn</code> ★ <code>\msg_if_exist:nnTF</code> ★	<code>\msg_if_exist_p:nn {<module>} {<message>}</code> <code>\msg_if_exist:nnTF {<module>} {<message>} {<true code>} {<false code>}</code>
--	---

New: 2012-03-03

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

2 Contextual information for messages

<code>\msg_line_context:</code> ☆	<code>\msg_line_context:</code> <p>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.</p>
-----------------------------------	--

<code>\msg_line_number:</code> ★	<code>\msg_line_number:</code> <p>Prints the current line number when a message is given.</p>
----------------------------------	--

<code>\msg_fatal_text:n</code> ★	<code>\msg_fatal_text:n {<module>}</code> <p>Produces the standard text</p> <p style="margin-left: 40px;">Fatal <i><module></i> error</p> <p>This function can be redefined to alter the language in which the message is given, using #1 as the name of the <i><module></i> to be included.</p>
----------------------------------	--

<code>\msg_critical_text:n</code> ★	<code>\msg_critical_text:n {<module>}</code> <p>Produces the standard text</p> <p style="margin-left: 40px;">Critical <i><module></i> error</p> <p>This function can be redefined to alter the language in which the message is given, using #1 as the name of the <i><module></i> to be included.</p>
-------------------------------------	--

<code>\msg_error_text:n</code> ★	<code>\msg_error_text:n {<module>}</code> <p>Produces the standard text</p> <p style="margin-left: 40px;"><i><module></i> error</p> <p>This function can be redefined to alter the language in which the message is given, using #1 as the name of the <i><module></i> to be included.</p>
----------------------------------	--

<code>\msg_warning_text:n</code>	<code>\msg_warning_text:n {<module>}</code>
----------------------------------	---

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.

<code>\msg_info_text:n</code>	<code>\msg_info_text:n {<module>}</code>
-------------------------------	--

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.

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

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.

<code>\msg_fatal:nnnnnn</code>	<code>\msg_fatal:nnnnnn {<module>} {<message>} {<arg one>} {<arg two>} {<arg three>}</code>
<code>\msg_fatal:nnxxxx</code>	<code>{<arg four>}</code>
<code>\msg_fatal:nnnnn</code>	Issues <code><module> error <message></code> , passing <code><arg one></code> to <code><arg four></code> to the text-creating
<code>\msg_fatal:nnxxx</code>	functions. After issuing a fatal error the \TeX run will halt.
<code>\msg_fatal:nnnn</code>	
<code>\msg_fatal:nnxx</code>	
<code>\msg_fatal:nnn</code>	
<code>\msg_fatal:nnx</code>	
<code>\msg_fatal:nn</code>	

Updated: 2012-08-11

```

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

```

Updated: 2012-08-11

```

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

```

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

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

```

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

```

Updated: 2012-08-11

```

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

```

Issues $\langle module \rangle$ error $\langle message \rangle$, passing $\langle arg one \rangle$ to $\langle arg four \rangle$ to the text-creating functions. The error 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 $\langle module \rangle$ warning $\langle message \rangle$, passing $\langle arg one \rangle$ to $\langle arg four \rangle$ to the text-creating functions. The warning text 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:nnnnnn {\module} {\message} {\arg one} {\arg two} {\arg three} {\arg
four}

```

Issues $\langle module \rangle$ information $\langle message \rangle$, passing $\langle arg one \rangle$ to $\langle arg four \rangle$ to the text-creating functions. The information text will be added to the log file.

<hr/>	
<code>\msg_log:nnnnnn</code>	<code>\msg_log:nnnnnn {<module>} {<message>} {<arg one>} {<arg two>} {<arg three>} {<arg four>}</code>
<code>\msg_log:nnxxxx</code>	
<code>\msg_log:nnnnn</code>	Issues <i><module></i> information <i><message></i> , passing <i><arg one></i> to <i><arg four></i> to the text-creating functions. The information text will be added to the log file: the output is briefer than
<code>\msg_log:nnxxx</code>	<code>\msg_info:nnnnnn.</code>
<code>\msg_log:nnnn</code>	
<code>\msg_log:nnxx</code>	
<code>\msg_log:nnn</code>	
<code>\msg_log:nnx</code>	
<code>\msg_log:nn</code>	
<hr/>	
Updated: 2012-08-11	
<hr/>	
<code>\msg_none:nnnnnn</code>	<code>\msg_none:nnnnnn {<module>} {<message>} {<arg one>} {<arg two>} {<arg three>} {<arg four>}</code>
<code>\msg_none:nnxxxx</code>	
<code>\msg_none:nnnnn</code>	Does nothing: used as a message class to prevent any output at all (see the discussion of message redirection).
<code>\msg_none:nnxxx</code>	
<code>\msg_none:nnnn</code>	
<code>\msg_none:nnxx</code>	
<code>\msg_none:nnn</code>	
<code>\msg_none:nnx</code>	
<code>\msg_none:nn</code>	
<hr/>	
Updated: 2012-08-11	

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.

<hr/> <code>\msg_redirect_class:nn</code> <hr/>	<code>\msg_redirect_class:nn {<class one>} {<class two>}</code>
Updated: 2012-04-27	Changes the behaviour of messages of <i><class one></i> so that they are processed using the code for those of <i><class two></i> .
<hr/> <code>\msg_redirect_module:nnn</code> <hr/>	<code>\msg_redirect_module:nnn {<module>} {<class one>} {<class two>}</code>
Updated: 2012-04-27	Redirects message of <i><class one></i> for <i><module></i> to act as though they were from <i><class two></i> . Messages of <i><class one></i> from sources other than <i><module></i> 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 <i><module></i> could be turned off with:
	<code>\msg_redirect_module:nnn { module } { warning } { none }</code>
<hr/> <code>\msg_redirect_name:nnn</code> <hr/>	<code>\msg_redirect_name:nnn {<module>} {<message>} {<class>}</code>
Updated: 2012-04-27	Redirects a specific <i><message></i> from a specific <i><module></i> to act as a member of <i><class></i> of messages. No further redirection is performed. This function can be used to make a selected message “silent” without changing global parameters:
	<code>\msg_redirect_name:nnn { module } { annoying-message } { none }</code>

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.

<hr/> <code>\msg_interrupt:nnn</code> <hr/>	<code>\msg_interrupt:nnn {<first line>} {<text>} {<extra text>}</code>
<hr/> New: 2012-06-28 <hr/>	Interrupts the TeX run, issuing a formatted message comprising <i><first line></i> and <i><text></i> 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.

<hr/> <code>\msg_log:n</code> <hr/>	<code>\msg_log:n {<text>}</code>
<hr/> New: 2012-06-28 <hr/>	Writes to the log file with the <i><text></i> 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.

<hr/> <code>\msg_term:n</code> <hr/>	<code>\msg_term:n {<text>}</code>
<hr/> New: 2012-06-28 <hr/>	Writes to the terminal and log file with the <i><text></i> 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
```

Updated: 2011-08-16

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

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: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:nnnnnn {<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 T_EX 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:nnnnnn {<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.

<pre> __msg_kernel_warning:nnnnnn __msg_kernel_warning:nnxxxx __msg_kernel_warning:nnnnnn __msg_kernel_warning:nnxxx __msg_kernel_warning:nnnn __msg_kernel_warning:nnxx __msg_kernel_warning:nnn __msg_kernel_warning:nnx __msg_kernel_warning:nn </pre>	<pre> __msg_kernel_warning:nnnnnn {\module} {\message} {\arg one} {\arg two} {\arg three} {\arg four} </pre>
--	---

Updated: 2012-08-11

Issues kernel $\langle module \rangle$ warning $\langle message \rangle$, passing $\langle arg one \rangle$ to $\langle arg four \rangle$ to the text-creating functions. The warning text will be added to the log file, but the \TeX run will not be interrupted.

<pre> __msg_kernel_info:nnnnnn __msg_kernel_info:nnxxxx __msg_kernel_info:nnnnnn __msg_kernel_info:nnxxx __msg_kernel_info:nnnn __msg_kernel_info:nnxx __msg_kernel_info:nnn __msg_kernel_info:nnx __msg_kernel_info:nn </pre>	<pre> __msg_kernel_info:nnnnnn {\module} {\message} {\arg one} {\arg two} {\arg three} {\arg four} </pre>
---	--

Updated: 2012-08-11

Issues kernel $\langle module \rangle$ information $\langle message \rangle$, passing $\langle arg one \rangle$ to $\langle arg four \rangle$ to the text-creating functions. The information text will be added to the log file.

7 Expandable errors

In a few places, the \LaTeX 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.

<pre> __msg_kernel_expandable_error:nnnnnn __msg_kernel_expandable_error:nnnnn __msg_kernel_expandable_error:nnnn __msg_kernel_expandable_error:nnn __msg_kernel_expandable_error:nn </pre>	<pre> __msg_kernel_expandable_error:nnnnnn {\module} {\message} {\arg one} {\arg two} {\arg three} {\arg four} </pre>
--	--

New: 2011-11-23

Issues an error, passing $\langle arg one \rangle$ to $\langle arg four \rangle$ 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 {\error message}

```

New: 2011-08-11

Updated: 2011-08-13

Issues an “Undefined error” message from T_EX itself, and prints the $\langle error\ message\rangle$. The $\langle error\ message\rangle$ must be short: it is cropped at the end of one line.

T_EXhackers 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: \show-command

```

New: 2015-08-05

Causes the next $\langle show-command\rangle$ to send its output to the log file instead of the terminal. This allows for instance $\backslash cs_log:N$ to be defined as $\backslash __msg_log_next: \backslash cs_show:N$. The effect of this command lasts until the next use of $\backslash __msg_show_wrap:Nn$ or $\backslash __msg_show_wrap:n$ or $\backslash __msg_show_variable:NNNnn$, in other words until the next time the ε -T_EX primitive $\backslash showtokens$ 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 {\module} {\message} {\arg one} {\arg two}
\__msg_show_pre:(nnxxx|nnnnnV) {\arg three} {\arg four}

```

New: 2015-08-05

Prints the $\langle message\rangle$ from $\langle module\rangle$ in the terminal (or log file if $\backslash __msg_log_next:$ was issued) without formatting. Used in messages which print complex variable contents completely.

```

\__msg_show_variable:NNNnn \__msg_show_variable:NNNnn \variable \if-exist \if-empty {\msg} {\formatted
content}

```

New: 2015-08-04

If the $\langle variable\rangle$ does not exist according to $\langle if-exist\rangle$ (typically $\backslash cs_if_exist:N\TF$) then throw an error and do nothing more. Otherwise, if $\langle msg\rangle$ is not empty, display the message **LaTeX/kernel/show- $\langle msg\rangle$** with $\backslash token_to_str:N \langle variable\rangle$ as a first argument, and a second argument that is ? or empty depending on the result of $\langle if-empty\rangle$ (typically $\backslash tl_if_empty:N\TF$) on the $\langle variable\rangle$. Then display the $\langle formatted\ content\rangle$ by giving it as an argument to $\backslash __msg_show_wrap:n$.

<hr/> <code>_msg_show_wrap:Nn</code> <hr/>	<code>_msg_show_wrap:Nn <function> {<expression>}</code>
New: 2015-08-03 Updated: 2015-08-07	Shows or logs the <i><expression></i> (turned into a string), an equal sign, and the result of applying the <i><function></i> to the <i>{<expression>}</i> . For instance, if the <i><function></i> is <code>\int_eval:n</code> and the <i><expression></i> is <code>1+2</code> then this will log <code>> 1+2=3</code> . The case where the <i><function></i> is <code>\tl_to_str:n</code> is special: then the string representation of the <i><expression></i> is only logged once.

<hr/> <code>_msg_show_wrap:n</code> <hr/>	<code>_msg_show_wrap:n {<formatted text>}</code>
New: 2015-08-03	Shows or logs the <i><formatted text></i> . After expansion, unless it is empty, the <i><formatted text></i> must contain <code>></code> , and the part of <i><formatted text></i> before the first <code>></code> is removed. Failure to do so causes low-level T _E X errors.

<hr/> <code>_msg_show_item:n</code> <hr/>	<code>_msg_show_item:n <item></code>
<code>_msg_show_item:nn</code>	<code>_msg_show_item:nn <item-key> <item-value></code>
<hr/> <code>_msg_show_item_unbraced:nn</code> <hr/>	
Updated: 2012-09-09	

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_tl`

The text

This is a coding error.

used by kernel functions when erroneous programming input is encountered.

Part XX

The l3keys package

Key–value interfaces

The key–value method is a popular system for creating large numbers of settings for controlling function or package behaviour. The system normally results in input of the form

```
\MyModuleSetup{
  key-one = value one,
  key-two = value two
}
```

or

```
\MyModuleMacro[
  key-one = value one,
  key-two = value two
]{argument}
```

for the user.

The high level functions here are intended as a method to create key–value controls. Keys are themselves created using a key–value interface, minimising the number of functions and arguments required. Each key is created by setting one or more *properties* of the key:

```
\keys_define:nn { mymodule }
{
  key-one .code:n    = code including parameter #1,
  key-two .tl_set:N = \l_mymodule_store_tl
}
```

These values can then be set as with other key–value approaches:

```
\keys_set:nn { mymodule }
{
  key-one = value one,
  key-two = value two
}
```

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`. 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_tl { key }
\keys_define:nn { mymodule }
{
  \l_mymodule_tmp_tl .code:n = code
}

```

will create a key called `\l_mymodule_tmp_tl`, and not one called `key`.

1 Creating keys

```

\keys_define:nn \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

$\langle key \rangle$.bool_set:N = $\langle boolean \rangle$

Defines $\langle key \rangle$ to set $\langle boolean \rangle$ to $\langle value \rangle$ (which must be either **true** or **false**). If the variable does not exist, it will be created globally at the point that the key is set up.

```
.bool_set_inverse:N
.bool_set_inverse:c
.bool_gset_inverse:N
.bool_gset_inverse:c
```

New: 2011-08-28

Updated: 2013-07-08

$\langle key \rangle$.bool_set_inverse:N = $\langle boolean \rangle$

Defines $\langle key \rangle$ to set $\langle boolean \rangle$ to the logical inverse of $\langle value \rangle$ (which must be either **true** or **false**). If the $\langle boolean \rangle$ does not exist, it will be created globally at the point that the key is set up.

```
.choice:
```

$\langle key \rangle$.choice:

Sets $\langle key \rangle$ to act as a choice key. Each valid choice for $\langle key \rangle$ must then be created, as discussed in section 3.

```
.choices:nn
.choices:Vn
.choices:on
.choices:xn
```

New: 2011-08-21

Updated: 2013-07-10

$\langle key \rangle$.choices:nn = { $\langle choices \rangle$ } { $\langle code \rangle$ }

Sets $\langle key \rangle$ to act as a choice key, and defines a series $\langle choices \rangle$ which are implemented using the $\langle code \rangle$. Inside $\langle code \rangle$, `\l_keys_choice_tl` will be the name of the choice made, and `\l_keys_choice_int` will be the position of the choice in the list of $\langle choices \rangle$ (indexed from 1). Choices are discussed in detail in section 3.

<hr/> <code>.clist_set:N</code> <hr/>	<code><key> .clist_set:N = <comma list variable></code>
<code>.clist_set:c</code> <code>.clist_gset:N</code> <code>.clist_gset:c</code> <hr/>	Defines <code><key></code> to set <code><comma list variable></code> to <code><value></code> . 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.
<hr/> New: 2011-09-11 <hr/>	
<hr/> <code>.code:n</code> <hr/>	<code><key> .code:n = {<code>}</code>
<hr/> Updated: 2013-07-10 <hr/>	Stores the <code><code></code> for execution when <code><key></code> is used. The <code><code></code> can include one parameter (<code>#1</code>), which will be the <code><value></code> given for the <code><key></code> . The <code>x</code> -type variant will expand <code><code></code> at the point where the <code><key></code> is created.
<hr/> <code>.default:n</code> <code>.default:V</code> <code>.default:o</code> <code>.default:x</code> <hr/>	<code><key> .default:n = {<default>}</code> Creates a <code><default></code> value for <code><key></code> , which is used if no value is given. This will be used if only the key name is given, but not if a blank <code><value></code> is given:
<hr/> Updated: 2013-07-09 <hr/>	<pre> \keys_define:nn { mymodule } { key .code:n = Hello~#1, key .default:n = World } \keys_set:nn { mymodule } { key = Fred, % Prints 'Hello Fred' key, % Prints 'Hello World' key = , % Prints 'Hello ' } </pre>
	The default does not affect keys where values are required or forbidden. Thus a required value cannot be supplied by a default value, and giving a default value for a key which cannot take a value will not trigger an error.
<hr/> <code>.dim_set:N</code> <code>.dim_set:c</code> <code>.dim_gset:N</code> <code>.dim_gset:c</code> <hr/>	<code><key> .dim_set:N = <dimension></code> Defines <code><key></code> to set <code><dimension></code> to <code><value></code> (which must a dimension expression). If the variable does not exist, it will be created globally at the point that the key is set up.
<hr/> <code>.fp_set:N</code> <code>.fp_set:c</code> <code>.fp_gset:N</code> <code>.fp_gset:c</code> <hr/>	<code><key> .fp_set:N = <floating point></code> Defines <code><key></code> to set <code><floating point></code> to <code><value></code> (which must a floating point expression). If the variable does not exist, it will be created globally at the point that the key is set up.
<hr/> <code>.groups:n</code> <hr/>	<code><key> .groups:n = {<groups>}</code>
<hr/> New: 2013-07-14 <hr/>	Defines <code><key></code> as belonging to the <code><groups></code> declared. Groups provide a “secondary axis” for selectively setting keys, and are described in Section 6.

<hr/> <code>.initial:n</code> <hr/>	<code><key> .initial:n = {<value>}</code>
<code>.initial:V</code>	Initialises the <code><key></code> with the <code><value></code> , equivalent to
<code>.initial:o</code>	
<code>.initial:x</code> <hr/>	
Updated: 2013-07-09 <hr/>	<code>\keys_set:nn {<module>} { <key> = <value> }</code>
<hr/> <code>.int_set:N</code> <hr/>	<code><key> .int_set:N = <integer></code>
<code>.int_set:c</code>	Defines <code><key></code> to set <code><integer></code> to <code><value></code> (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.
<code>.int_gset:N</code>	
<code>.int_gset:c</code> <hr/>	
<hr/> <code>.meta:n</code> <hr/>	<code><key> .meta:n = {<keyval list>}</code>
Updated: 2013-07-10 <hr/>	Makes <code><key></code> a meta-key, which will set <code><keyval list></code> in one go. If <code><key></code> is given with a value at the time the key is used, then the value will be passed through to the subsidiary <code><keys></code> for processing (as #1).
<hr/> <code>.meta:nn</code> <hr/>	<code><key> .meta:nn = {<path>} {<keyval list>}</code>
New: 2013-07-10 <hr/>	Makes <code><key></code> a meta-key, which will set <code><keyval list></code> in one go using the <code><path></code> in place of the current one. If <code><key></code> is given with a value at the time the key is used, then the value will be passed through to the subsidiary <code><keys></code> for processing (as #1).
<hr/> <code>.multichoice:</code> <hr/>	<code><key> .multichoice:</code>
New: 2011-08-21 <hr/>	Sets <code><key></code> to act as a multiple choice key. Each valid choice for <code><key></code> must then be created, as discussed in section 3.
<hr/> <code>.multichoices:nn</code> <hr/>	<code><key> .multichoices:nn {<choices>} {<code>}</code>
<code>.multichoices:Vn</code>	Sets <code><key></code> to act as a multiple choice key, and defines a series <code><choices></code> which are implemented using the <code><code></code> . Inside <code><code></code> , <code>\l_keys_choice_tl</code> will be the name of the choice made, and <code>\l_keys_choice_int</code> will be the position of the choice in the list of <code><choices></code> (indexed from 1). Choices are discussed in detail in section 3.
<code>.multichoices:on</code>	
<code>.multichoices:xn</code> <hr/>	
New: 2011-08-21	Updated: 2013-07-10 <hr/>
Updated: 2013-07-10 <hr/>	
<hr/> <code>.skip_set:N</code> <hr/>	<code><key> .skip_set:N = <skip></code>
<code>.skip_set:c</code>	Defines <code><key></code> to set <code><skip></code> to <code><value></code> (which must be a skip expression). If the variable does not exist, it will be created globally at the point that the key is set up.
<code>.skip_gset:N</code>	
<code>.skip_gset:c</code> <hr/>	
<hr/> <code>.tl_set:N</code> <hr/>	<code><key> .tl_set:N = <token list variable></code>
<code>.tl_set:c</code>	Defines <code><key></code> to set <code><token list variable></code> to <code><value></code> . If the variable does not exist, it will be created globally at the point that the key is set up.
<code>.tl_gset:N</code>	
<code>.tl_gset:c</code> <hr/>	

<hr/> <code>.tl_set_x:N</code> <hr/>	<code><key> .tl_set_x:N = <token list variable></code>
<code>.tl_set_x:c</code>	Defines <code><key></code> to set <code><token list variable></code> to <code><value></code> , which will be subjected to an <code>x</code> -type expansion (<i>i.e.</i> using <code>\tl_set:Nx</code>). If the variable does not exist, it will be created globally at the point that the key is set up.
<code>.tl_gset_x:N</code>	
<code>.tl_gset_x:c</code> <hr/>	
<hr/> <code>.undefine:</code> <hr/>	<code><key> .undefine:</code>
<code>New: 2015-07-14</code> <hr/>	Removes the definition of the <code><key></code> within the current scope.
<hr/> <code>.value_forbidden:n</code> <hr/>	<code><key> .value_forbidden:n = true false</code>
<code>New: 2015-07-14</code> <hr/>	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.
<hr/> <code>.value_required:n</code> <hr/>	<code><key> .value_required:n = true false</code>
<code>New: 2015-07-14</code> <hr/>	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:nnxxx { mymodule } { unknown-choice }
    { key } % Name of choice key
    { choice-a , choice-b , choice-c } % Valid choices
    { \exp_not:n {#1} } % Invalid choice given
  %
  %
}

```

Multiple choices are created in a very similar manner to mutually-exclusive choices, using the properties `.multichoice:` and `.multichoices:nn`. As with mutually exclusive choices, multiple choices are define as sub-keys. Thus both

```

\keys_define:nn { mymodule }
{
  key .multichoices:nn =
    { choice-a, choice-b, choice-c }
    {
      You~gave~choice~'\tl_use:N \l_keys_choice_tl',~
      which~is~in~position~
      \int_use:N \l_keys_choice_int \c_space_tl
      in~the~list.
    }
}

```

and

```

\keys_define:nn { mymodule }
{
  key .multichoice:,
  key / choice-a .code:n = code-a,
  key / choice-b .code:n = code-b,
  key / choice-c .code:n = code-c,
}

```

are valid.

When a multiple choice key is set

```

\keys_set:nn { mymodule }
{
  key = { a , b , c } % 'key' defined as a multiple choice
}

```

each choice is applied in turn, equivalent to a `clist` mapping or to applying each value individually:

```
\keys_set:nn { mymodule }
{
  key = a ,
  key = b ,
  key = c ,
}
```

Thus each separate choice will have passed to it the `\l_keys_choice_tl` and `\l_keys_choice_int` in exactly the same way as described for `.choices:nn`.

4 Setting keys

`\keys_set:nn`
`\keys_set:(nV|nv|no)`

`\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.

`\l_keys_key_tl`
`\l_keys_path_tl`
`\l_keys_value_tl`

Updated: 2015-07-14

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>} <tl>
\keys_set_known:(nVN|nvN|noN)
\keys_set_known:nn
\keys_set_known:(nV|nv|no)
```

New: 2011-08-23

Updated: 2014-04-27

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 *<tl>* 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.

<code>\keys_set_filter:nnnN</code>	<code>\keys_set_filter:nnnN {<module>} {<groups>} {<keyval list>} <tl></code>
<code>\keys_set_filter:(nnVN nnvN nnoN)</code>	
<code>\keys_set_filter:nnn</code>	
<code>\keys_set_filter:(nnV nnv nno)</code>	

New: 2013-07-14

Updated: 2014-04-27

Actives 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 `<tl>` in a comma-separated form (*i.e.* an edited version of the `<keyval list>`). The `\keys_set_filter:nnn` version skips this stage.

Use of `\keys_set_filter:nnnN` can be nested, with the correct residual `<keyval list>` returned at each stage.

<code>\keys_set_groups:nnn</code>	<code>\keys_set_groups:nnn {<module>} {<groups>} {<keyval list>}</code>
<code>\keys_set_groups:(nnV nnv nno)</code>	

New: 2013-07-14

Actives 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

<code>\keys_if_exist_p:nn</code> ★	<code>\keys_if_exist_p:nn {<module>} {<key>}</code>
<code>\keys_if_exist:nnTF</code> ★	<code>\keys_if_exist:nnTF {<module>} {<key>} {<true code>} {<false code>}</code>

Tests if the `<key>` exists for `<module>`, *i.e.* if any code has been defined for `<key>`.

<code>\keys_if_choice_exist_p:nnn</code>	★	<code>\keys_if_choice_exist_p:nnn {\langle module \rangle} {\langle key \rangle} {\langle choice \rangle}</code>
<code>\keys_if_choice_exist:nnnTF</code>	★	<code>\keys_if_choice_exist:nnnTF {\langle module \rangle} {\langle key \rangle} {\langle choice \rangle} {\langle true code \rangle}</code>
		<code>{\langle false code \rangle}</code>

New: 2011-08-21

Tests if the $\langle choice \rangle$ is defined for the $\langle key \rangle$ within the $\langle module \rangle$, *i.e.* if any code has been defined for $\langle key \rangle / \langle choice \rangle$. The test is **false** if the $\langle key \rangle$ itself is not defined.

<code>\keys_show:nn</code>	<code>\keys_show:nn {\langle module \rangle} {\langle key \rangle}</code>
----------------------------	---

Updated: 2015-08-09

Shows the information associated to the $\langle key \rangle$ for a $\langle module \rangle$, including the function which is used to actually implement it.

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 $\langle key\text{--}value\text{ list} \rangle$ into $\langle keys \rangle$ and associated $\langle values \rangle$. 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 $\langle function_1 \rangle$ $\langle function_2 \rangle$ { $\langle key-value list \rangle$ }

Parses the $\langle key-value list \rangle$ into a series of $\langle keys \rangle$ and associated $\langle values \rangle$, or keys alone (if no $\langle value \rangle$ was given). $\langle function_1 \rangle$ should take one argument, while $\langle function_2 \rangle$ should absorb two arguments. After **\keyval_parse:NNn** has parsed the $\langle key-value list \rangle$, $\langle function_1 \rangle$ will be used to process keys given with no value and $\langle function_2 \rangle$ will be used to process keys given with a value. The order of the $\langle keys \rangle$ in the $\langle key-value list \rangle$ 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 $\langle key \rangle$ and $\langle value \rangle$, then one *outer* set of braces is removed from the $\langle key \rangle$ and $\langle value \rangle$ as part of the processing.

Part XXI

The l3file package

File and I/O operations

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

It is important to remember that when reading external files `TEX` 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 $\langle file\ name \rangle$ 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

<hr/> <code>\g_file_current_name_tl</code> <hr/>	Contains the name of the current <code>L^AT_EX</code> file. This variable should not be modified: it is intended for information only. It will be equal to <code>\c_sys_jobname_str</code> at the start of a <code>L^AT_EX</code> run and will be modified each time a file is read using <code>\file_input:n</code> .
<hr/> <code>\file_if_exist:nTF</code> <hr/> Updated: 2012-02-10	<code>\file_if_exist:nTF {$\langle file\ name \rangle$} {$\langle true\ code \rangle$} {$\langle false\ code \rangle$}</code> Searches for $\langle file\ name \rangle$ using the current <code>T_EX</code> search path and the additional paths controlled by <code>\file_path_include:n</code> .
<hr/> <code>\file_add_path:nN</code> <hr/> Updated: 2012-02-10	<code>\file_add_path:nN {$\langle file\ name \rangle$} $\langle tl\ var \rangle$</code> Searches for $\langle file\ name \rangle$ in the path as detailed for <code>\file_if_exist:nTF</code> , and if found sets the $\langle tl\ var \rangle$ the fully-qualified name of the file, <i>i.e.</i> the path and file name. If the file is not found then the $\langle tl\ var \rangle$ will contain the marker <code>\q_no_value</code> .
<hr/> <code>\file_input:n</code> <hr/> Updated: 2012-02-17	<code>\file_input:n {$\langle file\ name \rangle$}</code> Searches for $\langle file\ name \rangle$ in the path as detailed for <code>\file_if_exist:nTF</code> , and if found reads in the file as additional <code>L^AT_EX</code> 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.

<hr/> <code>\file_path_include:n</code> <hr/>	<code>\file_path_include:n {<path>}</code>
Updated: 2012-07-04	Adds $\langle path \rangle$ to 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>i.e.</i> , with x -type expansion except active characters.

<hr/> <code>\file_path_remove:n</code> <hr/>	<code>\file_path_remove:n {<path>}</code>
Updated: 2012-07-04	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>i.e.</i> , with x -type expansion except active characters.

<hr/> <code>\file_list:</code> <hr/>	<code>\file_list:</code>
	This function will list all files loaded using <code>\file_input:n</code> in the log file.

1.1 Input–output stream management

As T_EX 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.

<hr/> <code>\ior_new:N</code> <hr/>	<code>\ior_new:N <stream></code>
<code>\ior_new:c</code>	<code>\iow_new:N <stream></code>
<code>\iow_new:N</code>	Globally reserves the name of the $\langle stream \rangle$, either for reading or for writing as appropriate. The $\langle stream \rangle$ is not opened until the appropriate <code>\..._open:Nn</code> function is used. Attempting to use a $\langle stream \rangle$ which has not been opened is an error, and the $\langle stream \rangle$ will behave as the corresponding <code>\c_term_...</code>
<code>\iow_new:c</code>	
New: 2011-09-26	
Updated: 2011-12-27	

<hr/> <code>\ior_open:Nn</code> <hr/>	<code>\ior_open:Nn <stream> {<file name>}</code>
<code>\ior_open:cn</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 <code>\ior_close:N</code> instruction is given or the T _E X run ends.
Updated: 2012-02-10	

<hr/> <code>\ior_open:NnTF</code> <hr/>	<code>\ior_open:NnTF <stream> {<file name>} {<true code>} {<false code>}</code>
<code>\ior_open:cnTF</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 <code>\ior_close:N</code> instruction is given or the T _E X 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.
New: 2013-01-12	

<code>\iow_open:Nn</code>	<code>\iow_open:Nn <stream> {(file name)}</code>
---------------------------	--

<code>\iow_open:cn</code>

Updated: 2012-02-09

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 T_EX run ends. Opening a file for writing will clear any existing content in the file (*i.e.* writing is *not* additive).

<code>\ior_close:N</code>	<code>\ior_close:N <stream></code>
---------------------------	--

<code>\ior_close:c</code>	<code>\iow_close:N <stream></code>
---------------------------	--

<code>\iow_close:N</code>

<code>\iow_close:c</code>

Updated: 2012-07-31

Closes the $\langle stream \rangle$. Streams should always be closed when they are finished with as this ensures that they remain available to other programmers.

<code>\ior_list_streams:</code>	<code>\ior_list_streams:</code>
---------------------------------	---------------------------------

<code>\iow_list_streams:</code>	<code>\iow_list_streams:</code>
---------------------------------	---------------------------------

Updated: 2015-08-01

Displays a list of the file names associated with each open stream: intended for tracking down problems.

1.2 Reading from files

<code>\ior_get:NN</code>	<code>\ior_get:NN <stream> (token list variable)</code>
--------------------------	---

New: 2012-06-24

Function that reads one or more lines (until an equal number of left and right braces are found) from the input $\langle stream \rangle$ and stores the result locally in the $\langle token\ list \rangle$ variable. If the $\langle stream \rangle$ is not open, input is requested from the terminal. The material read from the $\langle stream \rangle$ will be tokenized by T_EX according to the category codes in force when the function is used. Note that any blank lines will be converted to the token `\par`. Therefore, if skipping blank lines is required 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
...
```

may be used. Also notice that if multiple lines are read to match braces then the resulting token list will contain `\par` tokens. As normal T_EX tokenization is in force, any lines which do not end in a comment character (usually `%`) 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 .`

T_EXhackers note: This protected macro expands to the T_EX primitive `\read` along with the `to` keyword.

\ior_get_str:NN

New: 2012-06-24

Updated: 2012-07-31

\ior_get_str:NN $\langle stream \rangle$ $\langle token\ list\ variable \rangle$

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

a b c

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**. However, the end-line character normally added by this primitive is not included in the result of **\ior_get_str:NN**.

\ior_if_eof_p:N ★**\ior_if_eof:NTF** ★Updated: 2012-02-10

\ior_if_eof_p:N $\langle stream \rangle$ **\ior_if_eof:NTF** $\langle stream \rangle$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$

Tests if the end of a $\langle stream \rangle$ has been reached during a reading operation. The test will also return a **true** value if the $\langle stream \rangle$ is not open.

2 Writing to files

\iow_now:Nn**\iow_now:(Nx|cn|cx)**Updated: 2012-06-05

\iow_now:Nn $\langle stream \rangle$ $\{\langle tokens \rangle\}$

This functions writes $\langle tokens \rangle$ to the specified $\langle stream \rangle$ immediately (*i.e.* the write operation is called on expansion of **\iow_now:Nn**).

\iow_log:n**\iow_log:x****\iow_log:n** $\{\langle tokens \rangle\}$

This function writes the given $\langle tokens \rangle$ to the log (transcript) file immediately: it is a dedicated version of **\iow_now:Nn**.

\iow_term:n**\iow_term:x****\iow_term:n** $\{\langle tokens \rangle\}$

This function writes the given $\langle tokens \rangle$ to the terminal file immediately: it is a dedicated version of **\iow_now:Nn**.

 $\backslash\mathrm{iow_shipout:Nn}$
 $\backslash\mathrm{iow_shipout:(Nx|cn|cx)}$

 $\backslash\mathrm{iow_shipout:Nn} \langle stream \rangle \{ \langle tokens \rangle \}$

This functions writes $\langle tokens \rangle$ to the specified $\langle stream \rangle$ when the current page is finalised (*i.e.* at shipout). The x -type variants expand the $\langle tokens \rangle$ at the point where the function is used but *not* when the resulting tokens are written to the $\langle stream \rangle$ (*cf.* $\backslash\mathrm{iow_shipout_x:Nn}$).

T_EXhackers note: When using `expl3` with a format other than L^AT_EX, new line characters inserted using $\backslash\mathrm{iow_newline:}$ or using the line-wrapping code $\backslash\mathrm{iow_wrap:nnnN}$ will not be recognized in the argument of $\backslash\mathrm{iow_shipout:Nn}$. This may lead to the insertion of additionnal unwanted line-breaks.

 $\backslash\mathrm{iow_shipout_x:Nn}$
 $\backslash\mathrm{iow_shipout_x:(Nx|cn|cx)}$

Updated: 2012-09-08

 $\backslash\mathrm{iow_shipout_x:Nn} \langle stream \rangle \{ \langle tokens \rangle \}$

This functions writes $\langle tokens \rangle$ to the specified $\langle stream \rangle$ when the current page is finalised (*i.e.* at shipout). The $\langle tokens \rangle$ are expanded at the time of writing in addition to any expansion when the function is used. This makes these functions suitable for including material finalised during the page building process (such as the page number integer).

T_EXhackers note: This is a wrapper around the T_EX primitive $\backslash\mathrm{write}$. When using `expl3` with a format other than L^AT_EX, new line characters inserted using $\backslash\mathrm{iow_newline:}$ or using the line-wrapping code $\backslash\mathrm{iow_wrap:nnnN}$ will not be recognized in the argument of $\backslash\mathrm{iow_shipout:Nn}$. This may lead to the insertion of additionnal unwanted line-breaks.

 $\backslash\mathrm{iow_char:N} \star$

 $\backslash\mathrm{iow_char:N} \backslash \langle char \rangle$

Inserts $\langle char \rangle$ into the output stream. Useful when trying to write difficult characters such as %, {, }, *etc.* in messages, for example:

$$\backslash\mathrm{iow_now:Nx} \backslash\mathrm{g_my_iow} \{ \backslash\mathrm{iow_char:N} \{ \text{ text } \backslash\mathrm{iow_char:N} \} \}$$

The function has no effect if writing is taking place without expansion (*e.g.* in the second argument of $\backslash\mathrm{iow_now:Nn}$).

 $\backslash\mathrm{iow_newline:} \star$

 $\backslash\mathrm{iow_newline:}$

Function to add a new line within the $\langle tokens \rangle$ written to a file. The function has no effect if writing is taking place without expansion (*e.g.* in the second argument of $\backslash\mathrm{iow_now:Nn}$).

T_EXhackers note: When using `expl3` with a format other than L^AT_EX, the character inserted by $\backslash\mathrm{iow_newline:}$ will not be recognized by T_EX, which may lead to the insertion of additionnal unwanted line-breaks. This issue only affects $\backslash\mathrm{iow_shipout:Nn}$, $\backslash\mathrm{iow_shipout_x:Nn}$ and direct uses of primitive operations.

2.1 Wrapping lines in output

`\iow_wrap:nnnN`

New: 2012-06-28
Updated: 2015-08-05

`\iow_wrap:nnnN` $\{\langle text \rangle\}$ $\{\langle run-on text \rangle\}$ $\{\langle set up \rangle\}$ $\langle function \rangle$

This function will wrap the $\langle text \rangle$ to a fixed number of characters per line. At the start of each line which is wrapped, the $\langle run-on text \rangle$ 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 $\langle run-on text \rangle$ for all lines except the first, for which the target number of characters is simply `\l_iow_line_count_int` since there is no run-on text. The $\langle text \rangle$ and $\langle run-on text \rangle$ are exhaustively expanded by the function, with the following substitutions:

- `\` 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 $\langle text \rangle$ (not the $\langle run-on text \rangle$).

Additional functions may be added to the wrapping by using the $\langle set up \rangle$, which is executed before the wrapping takes place: this may include overriding the substitutions listed.

Any expandable material in the $\langle text \rangle$ which is not to be expanded on wrapping should be converted to a string using `\token_to_str:N`, `\tl_to_str:n`, `\tl_to_str:N`, *etc.*

The result of the wrapping operation is passed as a braced argument to the $\langle function \rangle$, which will typically be a wrapper around a write operation. The output of `\iow_wrap:nnnN` (*i.e.* the argument passed to the $\langle function \rangle$) 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.

T_EXhackers note: Internally, `\iow_wrap:nnnN` carries out an x-type expansion on the $\langle text \rangle$ to expand it. This is done in such a way that `\exp_not:N` or `\exp_not:n` *could* be used to prevent expansion of material. However, this is less conceptually clear than conversion to a string, which is therefore the supported method for handling expandable material in the $\langle text \rangle$.

`\iow_indent:n`

New: 2011-09-21

`\iow_indent:n` $\{\langle text \rangle\}$

In the first argument of `\iow_wrap:nnnN` (for instance in messages), indents $\langle text \rangle$ by four spaces. This function will not cause a line break, and only affects lines which start within the scope of the $\langle text \rangle$. In case the indented $\langle text \rangle$ should appear on separate lines from the surrounding text, use `\` to force line breaks.

`\l_iow_line_count_int`

New: 2012-06-24

The maximum number of characters in a line to be written by the `\iow_wrap:nnnN` function. This value depends on the T_EX system in use: the standard value is 78, which is typically correct for unmodified T_EXlive and MiK_T_EX systems.

<hr/> <code>\c_catcode_other_space_tl</code> <hr/>	Token list containing one character with category code 12, (“other”), and character code 32 (space).
<hr/> New: 2011-09-05 <hr/>	

2.2 Constant input–output streams

<hr/> <code>\c_term_ior</code> <hr/>	Constant input stream for reading from the terminal. Reading from this stream using <code>\ior_get:MN</code> or similar will result in a prompt from T _E X of the form <code><tl>=</code>
--------------------------------------	---

<hr/> <code>\c_log_ior</code> <code>\c_term_ior</code> <hr/>	Constant output streams for writing to the log and to the terminal (plus the log), respectively.
---	--

2.3 Primitive conditionals

<hr/> <code>\if_eof:w</code> ★ <hr/>	<pre> \if_eof:w <stream> <true code> \else: <false code> \fi: </pre> <p>Tests if the <code><stream></code> returns “end of file”, which is true for non-existent files. The <code>\else:</code> branch is optional.</p>
--------------------------------------	---

T_EXhackers note: This is the T_EX primitive `\ifeof`.

2.4 Internal file functions and variables

<hr/> <code>\g__file_internal_ior</code> <hr/>	Used to test for the existence of files when opening.
<hr/> <code>\l__file_internal_name_tl</code> <hr/>	Used to return the full name of a file for internal use. This is set by <code>\file_if_exist:n(TF)</code> and <code>__file_if_exist:nT</code> , and the value may then be used to load a file directly provided no further operations intervene.
<hr/> <code>__file_name_sanitize:nn</code> <hr/>	<code>__file_name_sanitize:nn {<name>} {<tokens>}</code>
<hr/> New: 2012-02-09 <hr/>	Exhaustively-expands the <code><name></code> with the exception of any category <code><active></code> (catcode 13) tokens, which are not expanded. The list of <code><active></code> tokens is taken from <code>\l_char_active_seq</code> . The <code><sanitized name></code> is then inserted (in braces) after the <code><tokens></code> , 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:Nn` $\langle stream \rangle$ $\{\langle file\ name \rangle\}$

`__ior_open:No`

New: 2012-01-23

This function has identical syntax to the public version. However, it does not take precautions against active characters in the $\langle file\ name \rangle$, and it does not attempt to add a $\langle path \rangle$ to the $\langle file\ name \rangle$: it is therefore intended to be used by higher-level functions which have already fully expanded the $\langle file\ name \rangle$ and which need to perform multiple open or close operations. See for example the implementation of `\file_add_path:nN`,

`__iow_with:Nnn` `__iow_with:Nnn` $\langle integer \rangle$ $\{\langle value \rangle\}$ $\{\langle code \rangle\}$

New: 2014-08-23

If the $\langle integer \rangle$ is equal to the $\langle value \rangle$ then this function simply runs the $\langle code \rangle$. Otherwise it saves the current value of the $\langle integer \rangle$, sets it to the $\langle value \rangle$, runs the $\langle code \rangle$, and restores the $\langle integer \rangle$ 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 XXII

The l3fp package: floating points

A decimal floating point number is one which is stored as a significand and a separate exponent. The module implements expandably a wide set of arithmetic, trigonometric, and other operations on decimal floating point numbers, to be used within floating point expressions. Floating point expressions support the following operations with their usual precedence.

- Basic arithmetic: addition $x + y$, subtraction $x - y$, multiplication $x * y$, division x / y , square root \sqrt{x} , and parentheses.
- Comparison operators: $x < y$, $x \leq y$, $x > y$, $x \neq y$ etc.
- Boolean logic: 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 $\text{sin}d\ x$, $\text{cos}d\ x$, $\text{tan}d\ x$, $\text{cot}d\ x$, $\text{sec}d\ x$, $\text{csc}d\ x$ expecting their arguments in degrees.
- Inverse trigonometric functions: $\text{asin}\ x$, $\text{acos}\ x$, $\text{atan}\ x$, $\text{acot}\ x$, $\text{asec}\ x$, $\text{acsc}\ x$ giving a result in radians, and $\text{asind}\ x$, $\text{acosd}\ x$, $\text{atand}\ x$, $\text{acotd}\ x$, $\text{asecd}\ x$, $\text{acscd}\ x$ giving a result in degrees.
- (not yet) Hyperbolic functions and their inverse functions: $\sinh x$, $\cosh x$, $\tanh x$, $\coth x$, $\text{sech}\ x$, $\text{csch}\ x$, and $\text{asinh}\ x$, $\text{acosh}\ x$, $\text{atanh}\ x$, $\text{acoth}\ x$, $\text{asech}\ x$, $\text{acsch}\ x$.
- Extrema: $\max(x, y, \dots)$, $\min(x, y, \dots)$, $\text{abs}(x)$.
- 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”.
- Constants: `pi`, `deg` (one degree in radians).
- Dimensions, automatically expressed in points, e.g., `pc` is 12.
- Automatic conversion (no need for `\<type>_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.

```
\usepackage{xparse, siunitx}
\ExplSyntaxOn
\NewDocumentCommand { \calcnun } { m }
{ \num { \fp_to_scientific:n {#1} } }
\ExplSyntaxOff
\calcnun { 2 pi * sin ( 2.3 ^ 5 ) }
```

1 Creating and initialising floating point variables

<code>\fp_new:N</code>	<code>\fp_new:N <fp var></code>
<code>\fp_new:c</code>	Creates a new <i><fp var></i> or raises an error if the name is already taken. The declaration is global. The <i><fp var></i> will initially be +0.
Updated: 2012-05-08	
<code>\fp_const:Nn</code>	<code>\fp_const:Nn <fp var> {<floating point expression>}</code>
<code>\fp_const:cn</code>	Creates a new constant <i><fp var></i> or raises an error if the name is already taken. The <i><fp var></i> will be set globally equal to the result of evaluating the <i><floating point expression></i> .
Updated: 2012-05-08	
<code>\fp_zero:N</code>	<code>\fp_zero:N <fp var></code>
<code>\fp_zero:c</code>	Sets the <i><fp var></i> to +0.
<code>\fp_gzero:N</code>	
<code>\fp_gzero:c</code>	
Updated: 2012-05-08	
<code>\fp_zero_new:N</code>	<code>\fp_zero_new:N <fp var></code>
<code>\fp_zero_new:c</code>	Ensures that the <i><fp var></i> exists globally by applying <code>\fp_new:N</code> if necessary, then applies <code>\fp_(g)zero:N</code> to leave the <i><fp var></i> set to +0.
<code>\fp_gzero_new:N</code>	
<code>\fp_gzero_new:c</code>	
Updated: 2012-05-08	

2 Setting floating point variables

```
\fp_set:Nn
\fp_set:cn
\fp_gset:Nn
\fp_gset:cn
```

Updated: 2012-05-08

`\fp_set:Nn` $\langle fp\ var \rangle$ $\{\langle floating\ point\ expression \rangle\}$

Sets $\langle fp\ var \rangle$ equal to the result of computing the $\langle floating\ point\ expression \rangle$.

```
\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` $\langle fp\ var_1 \rangle$ $\langle fp\ var_2 \rangle$

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` $\langle fp\ var \rangle$ $\{\langle floating\ point\ expression \rangle\}$

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` $\langle fp\ var \rangle$ $\{\langle floating\ point\ expression \rangle\}$

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` $\{\langle floating\ point\ expression \rangle\}$

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` $\langle fp\ var \rangle$

`\fp_to_decimal:n` $\{\langle floating\ point\ expression \rangle\}$

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.

<code>\fp_to_dim:N</code>	★	<code>\fp_to_dim:N <fp var></code>
<code>\fp_to_dim:c</code>	★	<code>\fp_to_dim:n {<floating point expression>}</code>
<code>\fp_to_dim:n</code>	★	Evaluates the <i><floating point expression></i> and expresses the result as a dimension (in pt) suitable for use in dimension expressions. The output is identical to <code>\fp_to_decimal:n</code> , with an additional trailing pt. In particular, the result may be outside the range $[-2^{14} + 2^{-17}, 2^{14} - 2^{-17}]$ of valid T _E X dimensions, leading to overflow errors if used as a dimension. The values $\pm\infty$ and NaN trigger an “invalid operation” exception.

Updated: 2012-07-08

<code>\fp_to_int:N</code>	★	<code>\fp_to_int:N <fp var></code>
<code>\fp_to_int:c</code>	★	<code>\fp_to_int:n {<floating point expression>}</code>
<code>\fp_to_int:n</code>	★	Evaluates the <i><floating point expression></i> , 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 T _E X integers, leading to overflow errors if used in an integer expression. The values $\pm\infty$ and NaN trigger an “invalid operation” exception.

Updated: 2012-07-08

<code>\fp_to_scientific:N</code>	★	<code>\fp_to_scientific:N <fp var></code>
<code>\fp_to_scientific:c</code>	★	<code>\fp_to_scientific:n {<floating point expression>}</code>
<code>\fp_to_scientific:n</code>	★	Evaluates the <i><floating point expression></i> and expresses the result in scientific notation:

New: 2012-05-08
Updated: 2012-07-08

<optional -><digit>.<15 digits>e<optional sign><exponent>

The leading *<digit>* is non-zero except in the case of ± 0 . The values $\pm\infty$ and NaN trigger an “invalid operation” exception. The entire representation is in the form of a string (tokens of category code 12), *including* the **e**.

<code>\fp_to_tl:N</code>	★	<code>\fp_to_tl:N <fp var></code>
<code>\fp_to_tl:c</code>	★	<code>\fp_to_tl:n {<floating point expression>}</code>
<code>\fp_to_tl:n</code>	★	Evaluates the <i><floating point expression></i> and expresses the result in (almost) the shortest possible form. Numbers in the ranges $(0, 10^{-3})$ and $[10^{16}, \infty)$ are expressed in scientific notation with trailing zeros trimmed and no decimal separator when there is a single significant digit (see <code>\fp_to_scientific:n</code>). Numbers in the range $[10^{-3}, 10^{16})$ are expressed in a decimal notation without exponent, with trailing zeros trimmed, and no decimal separator for integer values (see <code>\fp_to_decimal:n</code>). Negative numbers start with <code>-</code> . The special values ± 0 , $\pm\infty$ and NaN are rendered as <code>0</code> , <code>-0</code> , <code>inf</code> , <code>-inf</code> , and <code>nan</code> respectively.

Updated: 2012-07-08

<code>\fp_use:N</code>	★	<code>\fp_use:N <fp var></code>
<code>\fp_use:c</code>	★	Inserts the value of the <i><fp var></i> into the input stream as a decimal number with no exponent. Leading or trailing zeros may be inserted to compensate for the exponent. Non-significant trailing zeros are trimmed. Integers are expressed without a decimal separator. The values $\pm\infty$ and NaN trigger an “invalid operation” exception. This function is identical to <code>\fp_to_decimal:N</code> .

Updated: 2012-07-08

4 Floating point conditionals

<code>\fp_if_exist_p:N</code> ★	<code>\fp_if_exist_p:N</code> $\langle fp\ var \rangle$
<code>\fp_if_exist_p:c</code> ★	<code>\fp_if_exist:N</code> $\langle fp\ var \rangle$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$
<code>\fp_if_exist:N</code> ★	Tests whether the $\langle fp\ var \rangle$ is currently defined. This does not check that the $\langle fp\ var \rangle$ really is a floating point variable.
<code>\fp_if_exist:c</code> ★	

Updated: 2012-05-08

<code>\fp_compare_p:nNn</code> ★	<code>\fp_compare_p:nNn</code> $\{\langle fpexpr_1 \rangle\}$ $\langle relation \rangle$ $\{\langle fpexpr_2 \rangle\}$
<code>\fp_compare:nNn</code> ★	<code>\fp_compare:nNn</code> $\{\langle fpexpr_1 \rangle\}$ $\langle relation \rangle$ $\{\langle fpexpr_2 \rangle\}$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$

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 either operand 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

```

<code>\fp_compare_p:n</code> ★ <code>\fp_compare:nTF</code> ★ <hr/> Updated: 2012-12-14	<code>\fp_compare_p:n</code> { $\langle fpexpr_1 \rangle$ $\langle relation_1 \rangle$... $\langle fpexpr_N \rangle$ $\langle relation_N \rangle$ $\langle fpexpr_{N+1} \rangle$ } <code>\fp_compare:nTF</code> { $\langle fpexpr_1 \rangle$ $\langle relation_1 \rangle$... $\langle fpexpr_N \rangle$ $\langle relation_N \rangle$ $\langle fpexpr_{N+1} \rangle$ } { $\langle true\ code \rangle$ } { $\langle false\ code \rangle$ }
---	--

Evaluates the $\langle floating\ point\ expressions \rangle$ as described for `\fp_eval:n` and compares consecutive result using the corresponding $\langle relation \rangle$, namely it compares $\langle intexpr_1 \rangle$ 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 of the 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 $>=$ (greater or equal), $!=$ (not equal), $!?$ or $<=>$ (comparable).

5 Floating point expression loops

<code>\fp_do_until:nNnn</code> ☆ <hr/> New: 2012-08-16	<code>\fp_do_until:nNnn</code> { $\langle fpexpr_1 \rangle$ } $\langle relation \rangle$ { $\langle fpexpr_2 \rangle$ } { $\langle code \rangle$ }
---	--

Places the $\langle code \rangle$ in the input stream for T_EX to process, and then evaluates the relationship between the two $\langle floating\ point\ expressions \rangle$ as described for `\fp_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**.

<code>\fp_do_while:nNnn</code> ☆	<code>\fp_do_while:nNnn {<fpexpr₁>} <relation> {<fpexpr₂>} {<code>}</code>
New: 2012-08-16	Places the <i><code></i> in the input stream for T _E X to process, and then evaluates the relationship between the two <i><floating point expressions></i> as described for <code>\fp_compare:nNnTF</code> . If the test is true then the <i><code></i> will be inserted into the input stream again and a loop will occur until the <i><relation></i> is false .
<code>\fp_until_do:nNnn</code> ☆	<code>\fp_until_do:nNnn {<fpexpr₁>} <relation> {<fpexpr₂>} {<code>}</code>
New: 2012-08-16	Evaluates the relationship between the two <i><floating point expressions></i> as described for <code>\fp_compare:nNnTF</code> , and then places the <i><code></i> in the input stream if the <i><relation></i> is false . After the <i><code></i> has been processed by T _E X the test will be repeated, and a loop will occur until the test is true .
<code>\fp_while_do:nNnn</code> ☆	<code>\fp_while_do:nNnn {<fpexpr₁>} <relation> {<fpexpr₂>} {<code>}</code>
New: 2012-08-16	Evaluates the relationship between the two <i><floating point expressions></i> as described for <code>\fp_compare:nNnTF</code> , and then places the <i><code></i> in the input stream if the <i><relation></i> is true . After the <i><code></i> has been processed by T _E X the test will be repeated, and a loop will occur until the test is false .
<code>\fp_do_until:nn</code> ☆	<code>\fp_do_until:nn { <fpexpr₁> <relation> <fpexpr₂> } {<code>}</code>
New: 2012-08-16	Places the <i><code></i> in the input stream for T _E X to process, and then evaluates the relationship between the two <i><floating point expressions></i> as described for <code>\fp_compare:nTF</code> . If the test is false then the <i><code></i> will be inserted into the input stream again and a loop will occur until the <i><relation></i> is true .
<code>\fp_do_while:nn</code> ☆	<code>\fp_do_while:nn { <fpexpr₁> <relation> <fpexpr₂> } {<code>}</code>
New: 2012-08-16	Places the <i><code></i> in the input stream for T _E X to process, and then evaluates the relationship between the two <i><floating point expressions></i> as described for <code>\fp_compare:nTF</code> . If the test is true then the <i><code></i> will be inserted into the input stream again and a loop will occur until the <i><relation></i> is false .
<code>\fp_until_do:nn</code> ☆	<code>\fp_until_do:nn { <fpexpr₁> <relation> <fpexpr₂> } {<code>}</code>
New: 2012-08-16	Evaluates the relationship between the two <i><floating point expressions></i> as described for <code>\fp_compare:nTF</code> , and then places the <i><code></i> in the input stream if the <i><relation></i> is false . After the <i><code></i> has been processed by T _E X the test will be repeated, and a loop will occur until the test is true .
<code>\fp_while_do:nn</code> ☆	<code>\fp_while_do:nn { <fpexpr₁> <relation> <fpexpr₂> } {<code>}</code>
New: 2012-08-16	Evaluates the relationship between the two <i><floating point expressions></i> as described for <code>\fp_compare:nTF</code> , and then places the <i><code></i> in the input stream if the <i><relation></i> is true . After the <i><code></i> has been processed by T _E X the test will be repeated, and a loop will occur until the test is false .

6 Some useful constants, and scratch variables

<hr/> <code>\c_zero_fp</code> <code>\c_minus_zero_fp</code> <hr/> New: 2012-05-08 <hr/>	Zero, with either sign.
<hr/> <code>\c_one_fp</code> <hr/> New: 2012-05-08 <hr/>	One as an fp: useful for comparisons in some places.
<hr/> <code>\c_inf_fp</code> <code>\c_minus_inf_fp</code> <hr/> New: 2012-05-08 <hr/>	Infinity, with either sign. These can be input directly in a floating point expression as <code>inf</code> and <code>-inf</code> .
<hr/> <code>\c_e_fp</code> <hr/> Updated: 2012-05-08 <hr/>	The value of the base of the natural logarithm, $e = \exp(1)$.
<hr/> <code>\c_pi_fp</code> <hr/> Updated: 2013-11-17 <hr/>	The value of π . This can be input directly in a floating point expression as <code>pi</code> .
<hr/> <code>\c_one_degree_fp</code> <hr/> New: 2012-05-08 Updated: 2013-11-17 <hr/>	The value of 1° in radians. Multiply an angle given in degrees by this value to obtain a result in radians. Note that trigonometric functions expecting an argument in radians or in degrees are both available. Within floating point expressions, this can be accessed as <code>deg</code> .
<hr/> <code>\l_tmpa_fp</code> <code>\l_tmpb_fp</code> <hr/>	Scratch floating points for local assignment. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
<hr/> <code>\g_tmpa_fp</code> <code>\g_tmpb_fp</code> <hr/>	Scratch floating points for global assignment. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

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 IEEE standard defines 5 types of exceptions.

- *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 almost any operation involving a NaN. This normally results in a NaN, except for conversion functions whose target type does not have a notion of NaN (e.g., `\fp_to_dim:n`).
- *Division by zero* occurs when dividing a non-zero number by 0, or when evaluating e.g., $\ln(0)$ or $\cot(0)$. This results in $\pm\infty$.
- *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 is associated a “flag”, which can be either *on* or *off*. By default, the “invalid operation” exception triggers an (expandable) error, and raises the corresponding flag. Other exceptions only raise the corresponding flag. The state of the flag can be tested and modified. The behaviour when an exception occurs can be modified (using `\fp_trap:nn`) to either produce an error and turn the flag on, or only turn the flag on, or do nothing at all.

<code>\fp_if_flag_on_p:n</code> ★	<code>\fp_if_flag_on_p:n {<exception>}</code>
<code>\fp_if_flag_on:nTF</code> ★	<code>\fp_if_flag_on:nTF {<exception>} {<true code>} {<false code>}</code>
New: 2012-08-08	Tests if the flag for the <code><exception></code> is on, which normally means the given <code><exception></code> has occurred. <i>This function is experimental, and may be altered or removed.</i>
<code>\fp_flag_off:n</code>	<code>\fp_flag_off:n {<exception>}</code>
New: 2012-08-08	Locally turns off the flag which indicates whether the <code><exception></code> has occurred. <i>This function is experimental, and may be altered or removed.</i>
<code>\fp_flag_on:n</code> ★	<code>\fp_flag_on:n {<exception>}</code>
New: 2012-08-08	Locally turns on the flag to indicate (or pretend) that the <code><exception></code> has occurred. Note that this function is expandable: it is used internally by <code>l3fp</code> to signal when exceptions do occur. <i>This function is experimental, and may be altered or removed.</i>

<code>\fp_trap:nn</code>	<code>\fp_trap:nn {⟨exception⟩} {⟨trap type⟩}</code>
New: 2012-07-19 Updated: 2012-08-08	<p>All occurrences of the <i>⟨exception⟩</i> (<code>invalid_operation</code>, <code>division_by_zero</code>, <code>overflow</code>, or <code>underflow</code>) within the current group are treated as <i>⟨trap type⟩</i>, which can be</p> <ul style="list-style-type: none"> • none: the <i>⟨exception⟩</i> will be entirely ignored, and leave no trace; • flag: the <i>⟨exception⟩</i> will turn the corresponding flag on when it occurs; • error: additionally, the <i>⟨exception⟩</i> 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.

8 Viewing floating points

<code>\fp_show:N</code>	<code>\fp_show:N ⟨fp var⟩</code>
<code>\fp_show:c</code>	<code>\fp_show:n {⟨floating point expression⟩}</code>
<code>\fp_show:n</code>	Evaluates the <i>⟨floating point expression⟩</i> and displays the result in the terminal.
New: 2012-05-08 Updated: 2015-08-07	

9 Floating point expressions

9.1 Input of floating point numbers

We support four types of floating point numbers:

- $\pm 0.d_1d_2 \dots d_{16} \cdot 10^n$, a normal floating point number, with $d_i \in [0, 9]$, $d_1 \neq 0$, and $|n| \leq 10000$;
- ± 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);

(*not yet*) subnormal numbers $\pm 0.d_1d_2 \dots d_{16} \cdot 10^{-10000}$ with $d_1 = 0$.

Normal floating point numbers are stored in base 10, with 16 significant figures.

On input, a normal floating point number consists of:

- *⟨sign⟩*: a possibly empty string of + and - characters;
- *⟨significand⟩*: a non-empty string of digits together with zero or one dot;
- *⟨exponent⟩* 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 0 characters, and an optional . character), 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.

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.

Note that **e-1** is not a representation of 10^{-1} , because it could be mistaken with the difference of “e” and 1. This is consistent with several other programming languages. However, in order to avoid confusions, **e-1** is not considered to be this difference either. To input the base of natural logarithms, use **exp(1)** or **\c_e_fp**.

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

$$\begin{aligned}\sin 2\pi &= \sin(2\pi) = 0, \\ 2^{2\max(3,4)} &= 2^{2\max(3,4)} = 256.\end{aligned}$$

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 $\langle operand_2 \rangle$ if $\langle operand_1 \rangle$ is true, and $\langle operand_3 \rangle$ if it is false (equal to ± 0). All three $\langle operands \rangle$ 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 $\langle operand_1 \rangle$ is true (non-zero), use that value, otherwise the value of $\langle operand_2 \rangle$. Both $\langle operands \rangle$ 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
=      {
>      \langle operand_1 \rangle \langle relation_1 \rangle
?      ...
Updated: 2013-12-14 \langle operand_N \rangle \langle relation_N \rangle
                    \langle operand_{N+1} \rangle
                    }

```

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 $\langle operand_i \rangle \langle relation_j \rangle$ are true, and $+0$ otherwise. All $\langle operands \rangle$ are evaluated in all cases. See `\fp_compare:nTF` for details.

```

+ \fp_eval:n { \langle operand_1 \rangle + \langle operand_2 \rangle }
- \fp_eval:n { \langle operand_1 \rangle - \langle operand_2 \rangle }

```

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 { \langle operand_1 \rangle * \langle operand_2 \rangle }
/ \fp_eval:n { \langle operand_1 \rangle / \langle operand_2 \rangle }

```

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 { + \langle operand \rangle }
- \fp_eval:n { - \langle operand \rangle }
! \fp_eval:n { ! \langle operand \rangle }

```

The unary $+$ does nothing, the unary $-$ changes the sign of the $\langle operand \rangle$, and $!$ $\langle operand \rangle$ 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 { \langle operand_1 \rangle ** \langle operand_2 \rangle }
^  \fp_eval:n { \langle operand_1 \rangle ^ \langle operand_2 \rangle }

```

Raises $\langle operand_1 \rangle$ to the power $\langle operand_2 \rangle$. This operation is right associative, hence $2 ** 2 ** 3$ equals $2^{2^3} = 256$. The “invalid operation” exception occurs if $\langle operand_1 \rangle$ is negative or -0 , and $\langle operand_2 \rangle$ is not an integer, unless the result is zero (in that case, the sign is chosen arbitrarily to be $+0$). “Division by zero” occurs when raising ± 0 to a strictly negative power. “Underflow” and “overflow” occur when appropriate.

```

abs \fp_eval:n { abs( \langle fpexpr \rangle ) }

```

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( \langle fpexpr \rangle ) }

```

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
```

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 $\langle fpexpr_2 \rangle < -9984$).

sin	<code>\fp_eval:n { sin(<fpexpr>) }</code>
cos	<code>\fp_eval:n { cos(<fpexpr>) }</code>
tan	<code>\fp_eval:n { tan(<fpexpr>) }</code>
cot	<code>\fp_eval:n { cot(<fpexpr>) }</code>
csc	<code>\fp_eval:n { csc(<fpexpr>) }</code>
sec	<code>\fp_eval:n { sec(<fpexpr>) }</code>

Updated: 2013-11-17

Computes the sine, cosine, tangent, cotangent, cosecant, or secant of the $\langle fpexpr \rangle$ given in radians. For arguments given in degrees, see **sind**, **cosd**, *etc.* Note that since π is irrational, $\sin(8\pi)$ is not quite zero, while its analog $\text{sind}(8 \times 180)$ 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	<code>\fp_eval:n { sind(<fpexpr>) }</code>
cosd	<code>\fp_eval:n { cosd(<fpexpr>) }</code>
tand	<code>\fp_eval:n { tand(<fpexpr>) }</code>
cotd	<code>\fp_eval:n { cotd(<fpexpr>) }</code>
cscd	<code>\fp_eval:n { cscd(<fpexpr>) }</code>
secd	<code>\fp_eval:n { secd(<fpexpr>) }</code>

New: 2013-11-02

Computes the sine, cosine, tangent, cotangent, cosecant, or secant of the $\langle fpexpr \rangle$ given in degrees. For arguments given in radians, see **sin**, **cos**, *etc.* Note that since π is irrational, $\sin(8\pi)$ is not quite zero, while its analog $\text{sind}(8 \times 180)$ 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	<code>\fp_eval:n { asin(<fpexpr>) }</code>
acos	<code>\fp_eval:n { acos(<fpexpr>) }</code>
acsc	<code>\fp_eval:n { acsc(<fpexpr>) }</code>
asec	<code>\fp_eval:n { asec(<fpexpr>) }</code>

New: 2013-11-02

Computes the arcsine, arccosine, arccosecant, or arcsecant of the $\langle fpexpr \rangle$ and returns the result in radians, in the range $[-\pi/2, \pi/2]$ for **asin** and **acsc** and $[0, \pi]$ for **acos** and **asec**. For a result in degrees, use **asind**, *etc.* If the argument of **asin** or **acos** lies outside the range $[-1, 1]$, or the argument of **acsc** or **asec** inside the range $(-1, 1)$, an “invalid operation” exception is raised. “Underflow” and “overflow” occur when appropriate.

<code>asind</code>	<code>\fp_eval:n { asind(<fpexpr>) }</code>
<code>acosd</code>	<code>\fp_eval:n { acosd(<fpexpr>) }</code>
<code>acscd</code>	<code>\fp_eval:n { acscd(<fpexpr>) }</code>
<code>asecd</code>	<code>\fp_eval:n { asecd(<fpexpr>) }</code>

New: 2013-11-02

Computes the arcsine, arccosine, arccosecant, or arcsecant of the $\langle fpexpr \rangle$ and returns the result in degrees, in the range $[-90, 90]$ for `asin` and `acsc` and $[0, 180]$ for `acos` and `asec`. For a result in radians, use `asin`, *etc.* If the argument of `asin` or `acos` lies outside the range $[-1, 1]$, or the argument of `acsc` or `asec` inside the range $(-1, 1)$, an “invalid operation” exception is raised. “Underflow” and “overflow” occur when appropriate.

<code>atan</code>	<code>\fp_eval:n { atan(<fpexpr>) }</code>
<code>acot</code>	<code>\fp_eval:n { atan(<fpexpr₁> , <fpexpr₂>) }</code>
	<code>\fp_eval:n { acot(<fpexpr>) }</code>
New: 2013-11-02	<code>\fp_eval:n { acot(<fpexpr₁> , <fpexpr₂>) }</code>

Those functions yield an angle in radians: `atand` and `acotd` are their analogs in degrees. The one-argument versions compute the arctangent or arccotangent of the $\langle fpexpr \rangle$: arctangent takes values in the range $[-\pi/2, \pi/2]$, and arccotangent in the range $[0, \pi]$. The two-argument arctangent computes the angle in polar coordinates of the point with Cartesian coordinates $(\langle fpexpr_2 \rangle, \langle fpexpr_1 \rangle)$: this is the arctangent of $\langle fpexpr_1 \rangle / \langle fpexpr_2 \rangle$, possibly shifted by π depending on the signs of $\langle fpexpr_1 \rangle$ and $\langle fpexpr_2 \rangle$. The two-argument arccotangent computes the angle in polar coordinates of the point $(\langle fpexpr_1 \rangle, \langle fpexpr_2 \rangle)$, equal to the arccotangent of $\langle fpexpr_1 \rangle / \langle fpexpr_2 \rangle$, possibly shifted by π . Both two-argument functions take values in the wider range $[-\pi, \pi]$. The ratio $\langle fpexpr_1 \rangle / \langle fpexpr_2 \rangle$ need not be defined for the two-argument arctangent: when both expressions yield ± 0 , or when both yield $\pm \infty$, the resulting angle is one of $\{\pm \pi/4, \pm 3\pi/4\}$ depending on signs. Only the “underflow” exception can occur.

<code>atand</code>	<code>\fp_eval:n { atand(<fpexpr>) }</code>
<code>acotd</code>	<code>\fp_eval:n { atand(<fpexpr₁> , <fpexpr₂>) }</code>
	<code>\fp_eval:n { acotd(<fpexpr>) }</code>
New: 2013-11-02	<code>\fp_eval:n { acotd(<fpexpr₁> , <fpexpr₂>) }</code>

Those functions yield an angle in degrees: `atand` and `acotd` are their analogs in radians. The one-argument versions compute the arctangent or arccotangent of the $\langle fpexpr \rangle$: arctangent takes values in the range $[-90, 90]$, and arccotangent in the range $[0, 180]$. The two-argument arctangent computes the angle in polar coordinates of the point with Cartesian coordinates $(\langle fpexpr_2 \rangle, \langle fpexpr_1 \rangle)$: this is the arctangent of $\langle fpexpr_1 \rangle / \langle fpexpr_2 \rangle$, possibly shifted by 180 depending on the signs of $\langle fpexpr_1 \rangle$ and $\langle fpexpr_2 \rangle$. The two-argument arccotangent computes the angle in polar coordinates of the point $(\langle fpexpr_1 \rangle, \langle fpexpr_2 \rangle)$, equal to the arccotangent of $\langle fpexpr_1 \rangle / \langle fpexpr_2 \rangle$, possibly shifted by 180. Both two-argument functions take values in the wider range $[-180, 180]$. The ratio $\langle fpexpr_1 \rangle / \langle fpexpr_2 \rangle$ need not be defined for the two-argument arctangent: when both expressions yield ± 0 , or when both yield $\pm \infty$, the resulting angle is one of $\{\pm 45, \pm 135\}$ depending on signs. Only the “underflow” exception can occur.

<hr/> sqrt <hr/>	<code>\fp_eval:n { sqrt(<fpexpr>) }</code>
<hr/> New: 2013-12-14 <hr/>	Computes the square root of the <i><fpexpr></i> . The “invalid operation” is raised when the <i><fpexpr></i> is negative; no other exception can occur. Special values yield $\sqrt{-0} = -0$, $\sqrt{+0} = +0$, $\sqrt{+\infty} = +\infty$ and $\sqrt{\text{NaN}} = \text{NaN}$.
<hr/> inf <hr/> nan <hr/>	The special values $+\infty$, $-\infty$, and NaN are represented as <code>inf</code> , <code>-inf</code> and <code>nan</code> (see <code>\c_-inf_fp</code> , <code>\c_minus_inf_fp</code> and <code>\c_nan_fp</code>).
<hr/> pi <hr/>	The value of π (see <code>\c_pi_fp</code>).
<hr/> deg <hr/>	The value of 1° in radians (see <code>\c_one_degree_fp</code>).
<hr/> em ex in pt pc cm mm dd cc nd nc bp sp <hr/>	Those units of measurement are equal to their values in <code>pt</code> , namely <div style="margin-left: 100px;"> $1\text{in} = 72.27\text{pt}$ $1\text{pt} = 1\text{pt}$ $1\text{pc} = 12\text{pt}$ $1\text{cm} = \frac{1}{2.54}\text{in} = 28.45275590551181\text{pt}$ $1\text{mm} = \frac{1}{25.4}\text{in} = 2.845275590551181\text{pt}$ $1\text{dd} = 0.376065\text{mm} = 1.07000856496063\text{pt}$ $1\text{cc} = 12\text{dd} = 12.84010277952756\text{pt}$ $1\text{nd} = 0.375\text{mm} = 1.066978346456693\text{pt}$ $1\text{nc} = 12\text{nd} = 12.80374015748031\text{pt}$ $1\text{bp} = \frac{1}{72}\text{in} = 1.00375\text{pt}$ $1\text{sp} = 2^{-16}\text{pt} = 1.52587890625e - 5\text{pt}$. </div>
	The values of the (font-dependent) units <code>em</code> and <code>ex</code> are gathered from T _E X when the surrounding floating point expression is evaluated.
<hr/> true false <hr/>	Other names for 1 and +0.
<hr/> \fp_abs:n ★ <hr/>	<code>\fp_abs:n {<floating point expression>}</code>
<hr/> New: 2012-05-14 Updated: 2012-07-08 <hr/>	Evaluates the <i><floating point expression></i> as described for <code>\fp_eval:n</code> 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, <code>abs()</code> can be used.

`\fp_max:nn` ★ `\fp_max:nn {<fp expression 1>} {<fp expression 2>}`

`\fp_min:nn` ★

New: 2012-09-26

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_+; if it receives a T_EX 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 {<fpexpr>} {<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`.
- Random numbers (pgfmath provides `\rnd`, `\rand`, `\random`), with seed reset at every `\fp_set:Nn`.
- 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. (Exclamation points mark important 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 T_EX “number too large” error.
- Subnormals are not implemented.
- The overflow trap receives the wrong argument in `l3fp-expo` (see `exp(1e5678)` in `m3fp-traps001`).

Possible optimizations/improvements.

- Document that `l3trial/l3fp-types` introduces tools for adding new types.
- In subsection 9.1, write a grammar.
- Fix the `TWO BARS` business with the index.
- 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 T_EX fp packages, the international standards,...
- Also take into account the “inexact” exception?
- Support multi-character prefix operators (*e.g.*, `@/` or whatever)? Perhaps for including comments inside the computation itself??

Part XXIII

The l3candidates package

Experimental additions to l3kernel

1 Important notice

This module provides a space in which functions can be added to l3kernel (expl3) while still being experimental.

As such, the functions here may not remain in their current form, or indeed at all, in l3kernel in the future.

In contrast to the material in l3experimental, the functions here are all *small* additions to the kernel. We encourage programmers to test them out and report back on the LaTeX-L mailing list.

Thus, if you intend to use any of these functions from the candidate module in a public package offered to others for productive use (e.g., being placed on CTAN) please consider the following points carefully:

- Be prepared that your public packages might require updating when such functions are being finalized.
- Consider informing us that you use a particular function in your public package, e.g., by discussing this on the LaTeX-L mailing list. This way it becomes easier to coordinate any updates necessary without a 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 l3basics

`\cs_log:N`
`\cs_log:c`

New: 2014-08-22
Updated: 2015-08-03

`\cs_log:N` $\langle control\ sequence \rangle$

Writes the definition of the $\langle control\ sequence \rangle$ in the log file. See also `\cs_show:N` which displays the result in the terminal.

`__kernel_register_log:N`
`__kernel_register_log:c`

Updated: 2015-08-03

`__kernel_register_log:N` $\langle register \rangle$

Used to write the contents of a TeX register to the log file in a form similar to `__kernel_register_show:N`.

3 Additions to l3box

3.1 Affine transformations

Affine transformations are changes which (informally) preserve straight lines. Simple translations are affine transformations, but are better handled in T_EX by doing the translation first, then inserting an unmodified box. On the other hand, rotation and resizing of boxed material can best be handled by modifying boxes. These transformations are described here.

<code>\box_resize:Nnn</code>	<code>\box_resize:Nnn <box> {<x-size>} {<y-size>}</code>
<code>\box_resize:cnn</code>	

Resize 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 vertical size (height plus depth) of the box. 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 negative y -sizes will result in a box a depth dependent on the height of the original box a height dependent on the depth. The resizing applies within the current T_EX group level.

<code>\box_resize_to_ht_plus_dp:Nn</code>	<code>\box_resize_to_ht_plus_dp:Nn <box> {<y-size>}</code>
<code>\box_resize_to_ht_plus_dp:cn</code>	

Resize the $\langle box \rangle$ to $\langle y-size \rangle$ vertically, scaling the horizontal size by the same amount ($\langle y-size \rangle$ is a dimension expression). The $\langle y-size \rangle$ is the vertical size (height plus depth) of the box. 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 size 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 negative y -sizes will result in a box with depth dependent on the height of the original box and height dependent on the depth of the original. The resizing applies within the current T_EX group level.

<code>\box_resize_to_ht:Nn</code>	<code>\box_resize_to_ht:Nn <box> {<y-size>}</code>
<code>\box_resize_to_ht:cn</code>	

Resize the $\langle box \rangle$ to $\langle y-size \rangle$ vertically, scaling the horizontal size by the same amount ($\langle y-size \rangle$ is a dimension expression). The $\langle y-size \rangle$ is the height only, not including depth, of the box. 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 size 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 negative y -sizes will result in a box with depth dependent on the height of the original box and height dependent on the depth of the original. The resizing applies within the current T_EX group level.

<code>\box_resize_to_wd:Nn</code>	<code>\box_resize_to_wd:Nn <box> {<x-size>}</code>
<code>\box_resize_to_wd:cn</code>	

Resize 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 size 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 negative y -sizes will result in a box a depth dependent on the height of the original box a height dependent on the depth. The resizing applies within the current \TeX group level.

<code>\box_resize_to_wd_and_ht:Nnn</code>	<code>\box_resize_to_wd_and_ht:Nnn <box> {<x-size>} {<y-size>}</code>
<code>\box_resize_to_wd_and_ht:cnn</code>	

New: 2014-07-03

Resize the $\langle box \rangle$ to a *height* of $\langle x-size \rangle$ horizontally and $\langle y-size \rangle$ vertically (both of the sizes are dimension expressions). The $\langle y-size \rangle$ is the *height* of the box, ignoring 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.

<code>\box_rotate:Nn</code>	<code>\box_rotate:Nn <box> {<angle>}</code>
<code>\box_rotate:cn</code>	

Rotates the $\langle box \rangle$ by $\langle angle \rangle$ (in degrees) anti-clockwise about its reference point. The reference point of the updated box 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.

<code>\box_scale:Nnn</code>	<code>\box_scale:Nnn <box> {<x-scale>} {<y-scale>}</code>
<code>\box_scale:cnn</code>	

Scales the $\langle box \rangle$ by factors $\langle x-scale \rangle$ and $\langle y-scale \rangle$ in the horizontal and vertical directions, respectively (both scales are integer expressions). The updated $\langle box \rangle$ 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 negative y -scales will result in a box a depth dependent on the height of the original box a height dependent on the depth. The resizing applies within the current \TeX group level.

3.2 Viewing part of a box

<code>\box_clip:N</code>	<code>\box_clip:N <box></code>
<code>\box_clip:c</code>	

Clips the $\langle box \rangle$ in the output so that only material inside the bounding box is displayed in the output. The updated $\langle box \rangle$ will be an hbox, irrespective of the nature of the $\langle box \rangle$ 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_ε graphics drivers!

T_EXhackers note: Clipping is implemented by the driver, and as such the full content of the box is placed in the output file. Thus clipping does not remove any information from the raw output, and hidden material can therefore be viewed by direct examination of the file.

<code>\box_trim:Nnnnn</code>	<code>\box_trim:Nnnnn <box> {\left} {\bottom} {\right} {\top}</code>
<code>\box_trim:cnnnn</code>	

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 `\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 T_EX group level. The behavior of the operation where the trims requested is greater than the size of the box is undefined.

<code>\box_viewport:Nnnnn</code>	<code>\box_viewport:Nnnnn <box> {\llx} {\lly} {\urx} {\ury}</code>
<code>\box_viewport:cnnnn</code>	

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 `\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 T_EX group level.

3.3 Internal variables

<code>\l__box_angle_fp</code>

The angle through which a box is rotated by `\box_rotate:Nn`, given in degrees counter-clockwise. This value is required by the underlying driver code in `l3driver` to carry out the driver-dependent part of box rotation.

<code>\l__box_cos_fp</code>
<code>\l__box_sin_fp</code>

The sine and cosine of the angle through which a box is rotated by `\box_rotate:Nn`: the values refer to the angle counter-clockwise. These values are required by the underlying driver code in `l3driver` to carry out the driver-dependent part of box rotation.

<hr/> <code>\l__box_scale_x_fp</code> <hr/>	The scaling factors by which a box is scaled by <code>\box_scale:Nnn</code> or <code>\box_resize:Nnn</code> . These values are required by the underlying driver code in <code>l3driver</code> to carry out the driver-dependent part of box rotation.
<code>\l__box_scale_y_fp</code> <hr/>	

<hr/> <code>\l__box_internal_box</code> <hr/>	Box used for affine transformations, which is used to contain rotated material when applying <code>\box_rotate:Nn</code> . This box must be correctly constructed for the driver-dependent code in <code>l3driver</code> to function correctly.
---	---

4 Additions to `l3clist`

<hr/> <code>\clist_log:N</code> <hr/>	<code>\clist_log:N</code> <i><comma list></i>
<code>\clist_log:c</code> <hr/>	Writes the entries in the <i><comma list></i> in the log file. See also <code>\clist_show:N</code> which displays the result in the terminal.
<small>New: 2014-08-22</small>	

<hr/> <code>\clist_log:n</code> <hr/>	<code>\clist_log:n</code> <i>{<tokens>}</i>
<code>\clist_log:c</code> <hr/>	Writes the entries in the comma list in the log file. See also <code>\clist_show:n</code> which displays the result in the terminal.
<small>New: 2014-08-22</small>	

5 Additions to `l3coffins`

<hr/> <code>\coffin_resize:Nnn</code> <hr/>	<code>\coffin_resize:Nnn</code> <i><coffin></i> <i>{<width>}</i> <i>{<total-height>}</i>
<code>\coffin_resize:cnn</code> <hr/>	Resized the <i><coffin></i> to <i><width></i> and <i><total-height></i> , both of which should be given as dimension expressions.

<hr/> <code>\coffin_rotate:Nn</code> <hr/>	<code>\coffin_rotate:Nn</code> <i><coffin></i> <i>{<angle>}</i>
<code>\coffin_rotate:cn</code> <hr/>	Rotates the <i><coffin></i> by the given <i><angle></i> (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.

<hr/> <code>\coffin_scale:Nnn</code> <hr/>	<code>\coffin_scale:Nnn</code> <i><coffin></i> <i>{<x-scale>}</i> <i>{<y-scale>}</i>
<code>\coffin_scale:cnn</code> <hr/>	Scales the <i><coffin></i> by a factors <i><x-scale></i> and <i><y-scale></i> in the horizontal and vertical directions, respectively. The two scale factors should be given as real numbers.

<hr/> <code>\coffin_log_structure:N</code> <hr/>	<code>\coffin_log_structure:N</code> <i><coffin></i>
<code>\coffin_log_structure:c</code> <hr/>	This function writes the structural information about the <i><coffin></i> in the log file. The width, height and depth of the typeset material are given, along with the location of all of the poles of the coffin. See also <code>\coffin_show_structure:N</code> which displays the result in the terminal.
<small>New: 2014-08-22</small>	

6 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 `LaTeX` 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_map_inline:Nn`

New: 2012-02-11

`\ior_map_inline:Nn ⟨stream⟩ {⟨inline function⟩}`

Applies the `⟨inline function⟩` to `⟨lines⟩` obtained by reading one or more lines (until an equal number of left and right braces are found) from the `⟨stream⟩`. 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_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:n`

New: 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_ior
{
  \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_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.

7 Additions to l3fp

`\fp_log:N``\fp_log:c``\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.

8 Additions to l3int

`\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.

<hr/>	<code>\int_log:n</code>	<code>\int_log:n {⟨integer expression⟩}</code>
<hr/>	New: 2014-08-22	Writes the result of evaluating the <i>⟨integer expression⟩</i> in the log file.
<hr/>	Updated: 2015-08-07	

9 Additions to l3keys

<hr/>	<code>\keys_log:nn</code>	<code>\keys_log:nn {⟨module⟩} {⟨key⟩}</code>
<hr/>	New: 2014-08-22	Writes in the log file the function which is used to actually implement a <i>⟨key⟩</i> for a <i>⟨module⟩</i> .

10 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.

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

New: 2015-08-06

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

11 Additions to l3prg

`\bool_log:N`
`\bool_log:c`

New: 2014-08-22
Updated: 2015-08-03

`\bool_log:N` $\langle boolean \rangle$
Writes the logical truth of the $\langle boolean \rangle$ in the log file.

`\bool_log:n`

New: 2014-08-22
Updated: 2015-08-07

`\bool_log:n` $\{\langle boolean\ expression \rangle\}$
Writes the logical truth of the $\langle boolean\ expression \rangle$ in the log file.

12 Additions to l3prop

`\prop_map_tokens:Nn` ☆
`\prop_map_tokens:cn` ☆

`\prop_map_tokens:Nn` $\langle property\ list \rangle$ $\{\langle code \rangle\}$
Analogue of `\prop_map_function:NN` which maps several tokens instead of a single function. The $\langle code \rangle$ receives each key-value pair in the $\langle property\ list \rangle$ 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 $\langle key \rangle$ and the $\langle value \rangle$ as its three arguments. For that specific task, `\prop_item:Nn` is faster.

`\prop_log:N`
`\prop_log:c`

New: 2014-08-12

`\prop_log:N` $\langle property\ list \rangle$
Writes the entries in the $\langle property\ list \rangle$ in the log file.

13 Additions to l3seq

`\seq_mapthread_function:NNN` ☆ `\seq_mapthread_function:NNN` $\langle seq_1 \rangle$ $\langle seq_2 \rangle$ $\langle function \rangle$
`\seq_mapthread_function:(NcN|cNN|ccN)` ☆

Applies $\langle function \rangle$ to every pair of items $\langle seq_1\text{-}item \rangle$ – $\langle seq_2\text{-}item \rangle$ from the two sequences, returning items from both sequences from left to right. The $\langle function \rangle$ 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`

`\seq_set_filter:NNn` $\langle sequence_1 \rangle$ $\langle sequence_2 \rangle$ $\{ \langle inline\ boolexpr \rangle \}$

Evaluates the $\langle inline\ boolexpr \rangle$ for every $\langle item \rangle$ stored within the $\langle sequence_2 \rangle$. The $\langle inline\ boolexpr \rangle$ will receive the $\langle item \rangle$ as **#1**. The sequence of all $\langle items \rangle$ for which the $\langle inline\ boolexpr \rangle$ evaluated to **true** is assigned to $\langle sequence_1 \rangle$.

T_EXhackers note: Contrarily to other mapping functions, `\seq_map_break:` cannot be used in this function, and will lead to low-level T_EX errors.

`\seq_set_map:NNn`
`\seq_gset_map:NNn`

New: 2011-12-22

`\seq_set_map:NNn` $\langle sequence_1 \rangle$ $\langle sequence_2 \rangle$ $\{ \langle inline\ function \rangle \}$

Applies $\langle inline\ function \rangle$ to every $\langle item \rangle$ stored within the $\langle sequence_2 \rangle$. The $\langle inline\ function \rangle$ should consist of code which will receive the $\langle item \rangle$ as **#1**. The sequence resulting from x-expanding $\langle inline\ function \rangle$ applied to each $\langle item \rangle$ is assigned to $\langle sequence_1 \rangle$. As such, the code in $\langle inline\ function \rangle$ should be expandable.

T_EXhackers note: Contrarily to other mapping functions, `\seq_map_break:` cannot be used in this function, and will lead to low-level T_EX errors.

`\seq_log:N`
`\seq_log:c`

New: 2014-08-12

`\seq_log:N` $\langle sequence \rangle$

Writes the entries in the $\langle sequence \rangle$ in the log file.

14 Additions to l3skip

`\skip_split_finite_else_action:nnNN`

`\skip_split_finite_else_action:nnNN` $\{ \langle skipexpr \rangle \}$ $\{ \langle action \rangle \}$
 $\langle dimen_1 \rangle$ $\langle dimen_2 \rangle$

Checks if the $\langle skipexpr \rangle$ contains finite glue. If it does then it assigns $\langle dimen_1 \rangle$ the stretch component and $\langle dimen_2 \rangle$ the shrink component. If it contains infinite glue set $\langle dimen_1 \rangle$ and $\langle dimen_2 \rangle$ to 0pt and place **#2** into the input stream: this is usually an error or warning message of some sort.

`\dim_log:N`
`\dim_log:c`

New: 2014-08-22
Updated: 2015-08-03

`\dim_log:N` $\langle dimension \rangle$

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` $\{ \langle dimension\ expression \rangle \}$

Writes the result of evaluating the $\langle dimension\ expression \rangle$ in the log file.

<hr/> <code>\skip_log:N</code> <code>\skip_log:c</code> <hr/>	<code>\skip_log:N <skip></code> Writes the value of the <code><skip></code> in the log file.
New: 2014-08-22 Updated: 2015-08-03	
<hr/> <code>\skip_log:n</code> <hr/>	<code>\skip_log:n {<skip expression>}</code> Writes the result of evaluating the <code><skip expression></code> in the log file.
New: 2014-08-22 Updated: 2015-08-07	
<hr/> <code>\muskip_log:N</code> <code>\muskip_log:c</code> <hr/>	<code>\muskip_log:N <muskip></code> Writes the value of the <code><muskip></code> in the log file.
New: 2014-08-22 Updated: 2015-08-03	
<hr/> <code>\muskip_log:n</code> <hr/>	<code>\muskip_log:n {<muskip expression>}</code> Writes the result of evaluating the <code><muskip expression></code> in the log file.
New: 2014-08-22 Updated: 2015-08-07	

15 Additions to l3tl

<hr/> <code>\tl_if_single_token_p:n</code> ★ <code>\tl_if_single_token:nTF</code> ★ <hr/>	<code>\tl_if_single_token_p:n {<token list>}</code> <code>\tl_if_single_token:nTF {<token list>} {<true code>} {<false code>}</code> Tests if the token list consists of exactly one token, <i>i.e.</i> is either a single space character or a single “normal” token. Token groups (<code>{...}</code>) are not single tokens.
--	---

<hr/> <code>\tl_reverse_tokens:n</code> ★ <hr/>	<code>\tl_reverse_tokens:n {<tokens>}</code> This function, which works directly on T _E X tokens, reverses the order of the <code><tokens></code> : 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, <code>\tl_reverse_tokens:n {a~{b()}}</code> leaves <code>{() (b)~a</code> in the input stream. This function requires two steps of expansion.
---	---

T_EXhackers 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.

<hr/> <code>\tl_count_tokens:n</code> ★ <hr/>	<code>\tl_count_tokens:n {<tokens>}</code> Counts the number of T _E X tokens in the <code><tokens></code> 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 <code>a~{bc}</code> is 6. This function requires three expansions, giving an <i><integer denotation></i> .
---	---

<code>\tl_lower_case:n</code>	★	<code>\tl_upper_case:n</code>	<code>{\tokens}</code>
<code>\tl_lower_case:nn</code>	★	<code>\tl_upper_case:nn</code>	<code>{\language}{\tokens}</code>
<code>\tl_upper_case:n</code>	★	These functions are intended to be applied to input which may be regarded broadly as “text”. They traverse the <code>\tokens</code> 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 <code>\tokens</code> are normalized and become <code>{</code> and <code>}</code> , respectively.	
<code>\tl_upper_case:nn</code>	★		
<code>\tl_mixed_case:n</code>	★		
<code>\tl_mixed_case:nn</code>	★		

New: 2014-06-30
Updated: 2015-05-07

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_tl_case_change_math_tl`

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_tl_case_change_math_tl`, 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_tl_case_change_exclude_tl`

Case changing can be prevented by using any command on the list `\l_tl_case_change_exclude_tl`. 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_tl_case_change_exclude_tl { \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 $\text{\LaTeX 2}_{\epsilon}$ the commands `\cite`, `\ensuremath`, `\label` and `\ref` are automatically included in the list for exclusion from case changing.

In package mode, the case change system will also convert text stored using the $\text{\LaTeX 2}_{\epsilon}$ “LICR” approach. This will upper/lower case tokens as implemented for the font encodings T1, T2, T5 and LGR (see the behaviour of the $\text{\LaTeX 2}_{\epsilon}$ command `\MakeUppercase`). Note that these commands will automatically be protected from expansion.

“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_tl_case_change_exclude_tl` 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_tl_mixed_case_ignore_tl`

The list of characters to ignore when searching for the first “letter” in mixed-casing is determined by `\l_tl_mixed_change_ignore_tl`. 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, when used with `pdfTeX` *only* the characters `a–zA–Z` are modified. When used with `XYTeX` or `LuaTeX` a full range of Unicode transformations are enabled. Specifically, the standard mappings here follow those defined by the [Unicode Consortium](#) in `UnicodeData.txt` and `SpecialCasing.txt`. Note that in some cases, `pdfTeX` can interpret the input to a case change but not generate the correct output (for example in the mapping `i` to `I-dot` in Turkish): in these cases the input is left unchanged.

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 `<language>` argument, and follow Unicode Consortium guidelines. Currently, the languages recognised for special handling are as follows.

- Azeri and Turkish (`az` and `tr`). The case pairs `I/i-dotless` and `I-dot/i` are activated for these languages. The combining dot mark is removed when 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 (`nl`). 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 `pdfTeX`.

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).

`\tl_set_from_file:Nnn`
`\tl_set_from_file:cnn`
`\tl_gset_from_file:Nnn`
`\tl_gset_from_file:cnn`

`\tl_set_from_file:Nnn <tl> {<setup>} {<filename>}`

Defines `<tl>` 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_log:N
\tl_log:c
```

New: 2014-08-22
Updated: 2015-08-01

```
\tl_log:n
```

New: 2014-08-22

```
\tl_set_from_file_x:Nnn <tl> {<setup>} {<filename>}
```

Defines $\langle tl \rangle$ to the contents of $\langle filename \rangle$, expanding the contents of the file as it is read. Category codes and other definitions may need to be set appropriately via the $\langle setup \rangle$ argument.

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

Writes the content of the $\langle tl var \rangle$ in the log file. See also $\backslash tl_show:N$ which displays the result in the terminal.

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

Writes the $\langle token list \rangle$ in the log file. See also $\backslash tl_show:n$ which displays the result in the terminal.

16 Additions to **l3tokens**

```
\char_set_active_eq:NN
\char_gset_active_eq:NN
```

Updated: 2015-09-01

```
\char_set_active_eq:nN
\char_gset_active_eq:nN
```

New: 2015-09-02

```
\char_set_active_eq:NN <char> <function>
```

Sets the behaviour of the $\langle char \rangle$ in situations where it is active (category code 13) to be equivalent to that of the $\langle function \rangle$. The category code of the $\langle char \rangle$ is *unchanged* by this process. The $\langle function \rangle$ may itself be an active character.

```
\char_set_active_eq:nN {<integer expression>} <function>
```

Sets the behaviour of the $\langle char \rangle$ which has character code as given by the $\langle integer expression \rangle$ in situations where it is active (category code 13) to be equivalent to that of the $\langle function \rangle$. The category code of the $\langle char \rangle$ is *unchanged* by this process. The $\langle function \rangle$ may itself be an active character.

`\char_generate:nn` ★

New: 2015-09-09

`\char_generate:nn` $\{\langle charcode \rangle\}$ $\{\langle catcode \rangle\}$

Generates a character token of the given $\langle charcode \rangle$ and $\langle catcode \rangle$ (both of which may be integer expressions). The $\langle catcode \rangle$ may be one of

- 1 (begin group)
- 2 (end group)
- 3 (math toggle)
- 4 (alignment)
- 6 (parameter)
- 7 (math superscript)
- 8 (math subscript)
- 11 (letter)
- 12 (other)

and other values will raise an error.

The $\langle catcode \rangle$ may be any one valid for the engine in use. Note however that for X_YTeX releases prior to 0.99992 only the 8-bit range (0 to 255) is accepted due to engine limitations.

`\peek_N_type:TF`

Updated: 2012-12-20

`\peek_N_type:TF` $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$

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 XXIV

The l3sys 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 T_EX starts.

T_EXhackers note: This copies the contents of the primitive `\jobname`. It is a constant that is set by T_EX 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.

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

2.1 Engine

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

New: 2015-09-07

`\c_sys_engine_str`

New: 2015-09-19

`\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 -pT_EX and ε -upT_EX as expl3 requires the ε -T_EX extensions. Each conditional is true for *exactly one* supported engine. In particular, `\sys_if_engine ptex_p:` is true for ε -pT_EX but false for ε -upT_EX.

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

2.2 Output format

<code>\sys_if_output_dvi_p:</code>	★	<code>\sys_if_output_dvi:TF</code>	{ <i>(true code)</i> } { <i>(false code)</i> }
------------------------------------	---	------------------------------------	--

<code>\sys_if_output_dvi:</code>	★	<code><i>TF</i></code>
----------------------------------	---	------------------------

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

<code>\sys_if_output_pdf:</code>	★	<code><i>TF</i></code>
----------------------------------	---	------------------------

New: 2015-09-19

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

<code>\c_sys_output_str</code>

New: 2015-09-19

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

Part XXV

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.

<code>\lua_now_x:n</code>	★	<code>\lua_now:n {⟨token list⟩}</code>
---------------------------	---	--

<code>\lua_now:n</code>	★
-------------------------	---

New: 2015-06-29

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.

<code>\lua_shipout_x:n</code>	<code>\lua_shipout:n {⟨token list⟩}</code>
-------------------------------	--

<code>\lua_shipout:n</code>

New: 2015-06-30

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”.

<hr/>	
<code>\lua_escape_x:n</code> ★	<code>\lua_escape:n {⟨<i>token list</i>⟩}</code>
<code>\lua_escape:n</code> ★	
<hr/>	
New: 2015-06-29	Converts the <i>⟨token list⟩</i> 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 <code>\n</code> and <code>\r</code> , respectively.
<hr/>	

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.

Part XXVI

The l3drivers 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, L^AT_EX3 is aware of the following drivers:

- **pdfmode**: The “driver” for direct PDF output by *both* pdfTeX and LuaTeX (no separate driver is used in this case: the engine deals with PDF creation itself).
- **dvips**: The dvips program, which works in conjugation with pdfTeX or LuaTeX in DVI mode.
- **dvipdfmx**: The dvipdfmx program, which works in conjugation with pdfTeX or LuaTeX in DVI mode.
- **xdvipdfmx**: The driver used by X_YTeX.

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”: several variable values must be in the correct locations for the driver code to function.

1 Box clipping

`_driver_box_use_clip:N`

New: 2011-11-11

`_driver_box_use_clip:N` $\langle box \rangle$

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

<code>__driver_box_rotate_begin:</code>	<code>__driver_box_rotate_begin:</code>
<code>__driver_box_rotate_end:</code>	<code>\box_use:N \l__box_internal_box</code>
	<code>__driver_box_rotate_end:</code>

New: 2011-09-01

Updated: 2013-12-27

Rotates the $\langle box material \rangle$ anti-clockwise around the current insertion point. The angle of rotation (in degrees counter-clockwise) and the sine and cosine of this angle should be stored in `\l__box_angle_fp`, `\l__box_sin_fp` and `\l__box_cos_fp`, respectively. Typically, the box material inserted between the beginning and end markers will be stored in `\l__box_internal_box`: this fact is required by some drivers to obtain the correct output.

<code>__driver_box_scale_begin:</code>	<code>__driver_box_scale_begin:</code>
<code>__driver_box_scale_end:</code>	$\langle box material \rangle$
	<code>__driver_box_scale_end:</code>

New: 2011-09-02

Updated: 2013-12-27

Scales the $\langle box material \rangle$ (which should be either a `\box_use:N` or `\hbox:n` construct). The $\langle box material \rangle$ is scaled by the values stored in `\l__box_scale_x_fp` and `\l__box_scale_y_fp` in the horizontal and vertical directions, respectively. This function is also reused when resizing boxes: at a driver level, only scalings are supported and so the higher-level code must convert the absolute sizes to scale factors.

3 Color support

<code>__driver_color_ensure_current:</code>	<code>__driver_color_ensure_current:</code>
--	--

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.

Part XXVII

Implementation

1 l3bootstrap implementation

- $\langle *initex | package \rangle$
- $\langle @@=expl \rangle$

1.1 Format-specific code

The very first thing to do is to bootstrap the `iniTeX` system so that everything else will actually work. `TeX` does not start with some pretty basic character codes set up.

```
3 <*initex>
4 \catcode '\{ = 1 \relax
5 \catcode '\} = 2 \relax
6 \catcode '\# = 6 \relax
7 \catcode '\^ = 7 \relax
8 </initex>
```

Tab characters should not show up in the code, but to be on the safe side.

```
9 <*initex>
10 \catcode '\^^I = 10 \relax
11 </initex>
```

For `LuaTeX`, the extra primitives need to be enabled. This is not needed in package mode: plain `TeX` and `ConTeXt` have the primitives enabled while `LATεTeX` has them with the prefix `luatex` (which is handled in `l3names`).

```
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 `LATεTeX` format may not have the raw `\Umath` primitive names available. We fix that globally: it should cause no issues. Older `LuaTeX` versions do not have a pre-built table of the primitive names here so sort one out ourselves. These end up globally-defined but at that is true with a newer format anyway and as they all start `\U` this should be reasonably safe.

```
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 with XeTeX and LuaTeX

Only pdfTeX has a primitive called `\pdfstrcmp`. The XeTeX version is just `\strcmp`, so there is some shuffling to do. As this is still a real primitive, using the pdfTeX name is “safe”.

```
38 \begingroup\expandafter\expandafter\expandafter\endgroup
39 \expandafter\ifx\csname pdfstrcmp\endcsname\relax
40 \let\pdfstrcmp\strcmp
41 \fi
```

If LuaTeX is in use then no primitive `\pdfstrcmp` is available. However, it can be emulated using some Lua code. In earlier versions of the code, the `pdfstrcmp` package was loaded to do this task. However, that raises some issues in “generic” (it fails with ConTeXt MkIV), and also adds a hardly-needed dependency. Note that LuaTeX prior to version 0.36 is not supported by expl3: here that means simply skipping the definition, which will then be picked up later. This definition may need to be done twice: one “now” and once at the start of every job. The latter can occur in package mode if for example a custom format is being constructed. To achieve this while not requiring a separate file, the Lua code is saved into a macro then used twice. (In the long term, the Lua code here may be best moved to a separate file.)

No macro definition is given just yet: that is left until `l3basics`.

```
42 \begingroup
43 \expandafter\ifx\csname directlua\endcsname\relax
44 \else
45   \ifnum\luatexversion<36 %
46   \else
47     \catcode'\_ =11 %
48     \catcode'\:=11 %
49     \def\tempa
50     {%
51       l3kernel = l3kernel or { }
52       function l3kernel.strptime(A, B)
53         if A == B then
54           tex.write("0")
55         elseif A < B then
56           tex.write("-1")
57         else
58           tex.write("1")
59         end
60       end
61     }
62     \directlua{\tempa}
```

A test for LuaTeX in IniTeX mode.

```
63 \ifnum 0%
64 \directlua
65 {%
66   if status.ini_version then
67     tex.write("1")
```

```

68         end
69     }>0 %
70     \global\everyjob\expandafter
71     {%
72         \the\expandafter\everyjob
73         \expandafter\lua_now_x:n\expandafter{\tempa}%
74     }
75 \fi
76 \fi
77 \fi
78 \endgroup

```

1.3 Emulating \Ucharchar in LuaTeX

Expandably creating character tokens can be done in macros but for the full Unicode range requires engine support in practical terms. That is available in XeTeX as `\Ucharcat` but in LuaTeX some Lua code is required. That is set up here, following the model above. Note that we do not try to emulate the syntax of the primitive (doable with LuaTeX 0.80 or later, but not required by our use case).

One minor wrinkle is that a catcode table is needed to return values. For format mode, long-term we can simply take an arbitrary one and make sure this is never allocated. In package mode use the same approach (and table) as `ucharcat`: look for an allocator and use it if available, or use a table unlikely to clash with any other use. We use the same table as `ucharcat` but as this is “disposable” we are safe.

```

79 \begingroup
80 \expandafter\ifx\csname directlua\endcsname\relax
81 \else
82     \ifnum\luatexversion<70 %
83     \else
84 \<initex>
85     % TEMP
86     \catcode'\@=11 %
87     \chardef\ucharcat@table="8000 %
88 \</initex>
89 \<*package>
90     \begingroup\expandafter\expandafter\expandafter\endgroup
91     \expandafter\ifx\csname newcatcodetable\endcsname\relax
92     \directlua{tex.enableprimitives("",{"initcatcodetable"})}
93     \chardef\ucharcat@table"7000 %
94     \initcatcodetable\ucharcat@table
95 \else
96     \newcatcodetable\ucharcat@table
97 \fi
98 \</package>
99 \catcode'\_ =11 %
100 \catcode'\: =11 %
101 \def\tempa
102     {%

```

```

103         l3kernel = l3kernel or { }
104         local utf8_char = unicode.utf8.char
105         function l3kernel.charcat(charcode, catcode)
106             tex.setcatcode(\number\ucharcat@table, charcode, catcode)
107             tex.sprint(\number\ucharcat@table, utf8_char(charcode))
108         end
109     }
110     \directlua{\tempa}
111     \ifnum 0%
112         \directlua
113         {%
114             if status.ini_version then
115                 tex.write("1")
116             end
117         }>0 %
118     \global\everyjob\expandafter
119     {%
120         \the\expandafter\everyjob
121         \expandafter\lua_now_x:n\expandafter{\tempa}%
122     }
123     \fi
124     \fi
125     \fi
126 \endgroup

```

1.4 Engine requirements

The code currently requires functionality equivalent to `\pdfstrcmp` in addition to ε -TeX. This is picked up by testing for the `\pdfstrcmp` primitive or a version of LuaTeX capable of emulating it.

```

127 \begingroup
128   \def\next{\endgroup}%
129   \def\ShortText{Required primitives not found}%
130   \def\LongText%
131   {%
132     LaTeX3 requires the e-TeX primitives and \string\pdfstrcmp.\LineBreak
133     \LineBreak
134     These are available in the engines\LineBreak
135     - pdfTeX v1.30\LineBreak
136     - XeTeX v0.9994\LineBreak
137     - LuaTeX v0.40\LineBreak
138     - e-(u)pTeX mid-2012\LineBreak
139     or later.\LineBreak
140     \LineBreak
141   }%
142   \ifnum0%
143     \expandafter\ifx\csname pdfstrcmp\endcsname\relax\else 1\fi
144     \expandafter\ifx\csname directlua\endcsname\relax
145     \else

```

```

146     \ifnum\luatexversion<36 \else 1\fi
147     \fi
148     =0 %
149     \newlinechar'\^^J %
150 <*initex>
151     \def\LineBreak{^^J}%
152     \edef\next
153     {%
154         \errhelp
155         {%
156             \LongText
157             For pdfTeX and XeTeX the '-etex' command-line switch is also
158             needed.\LineBreak
159             \LineBreak
160             Format building will abort!\LineBreak
161         }%
162         \errmessage{\ShortText}%
163         \endgroup
164         \noexpand\end
165     }%
166 </initex>
167 <*package>
168     \def\LineBreak{\noexpand\MessageBreak}%
169     \expandafter\ifx\csname PackageError\endcsname\relax
170     \def\LineBreak{^^J}%
171     \def\PackageError#1#2#3%
172     {%
173         \errhelp{#3}%
174         \errmessage{#1 Error: #2}
175     }%
176 \fi
177 \edef\next
178 {%
179     \noexpand\PackageError{expl3}{\ShortText}
180     {\LongText Loading of expl3 will abort!}%
181     \endgroup
182     \noexpand\endinput
183 }%
184 </package>
185 \fi
186 \next

```

1.5 Extending allocators

In format mode, allocating registers is handled by `l3alloc`. 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.

```

187 <*package>
188 \begingroup
189   \def\@tempa{LaTeX2e}
190   \def\next{}
191   \ifx\fmtname\@tempa
192     \expandafter\ifx\csname extrafloats\endcsname\relax
193       \def\next
194         {%
195           \RequirePackage{etex}%
196           \csname reserveinserts\endcsname{32}%
197         }
198     \fi
199   \fi
200 \expandafter\endgroup
201 \next
202 </package>

```

1.6 The $\text{\LaTeX}3$ 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.

```

203 \protected\def\ExplSyntaxOff{}
204 <*package>
205 \protected\edef\ExplSyntaxOff
206   {%
207     \protected\def\ExplSyntaxOff{}%
208     \catcode 9 = \the\catcode 9\relax
209     \catcode 32 = \the\catcode 32\relax
210     \catcode 34 = \the\catcode 34\relax
211     \catcode 38 = \the\catcode 38\relax
212     \catcode 58 = \the\catcode 58\relax
213     \catcode 94 = \the\catcode 94\relax
214     \catcode 95 = \the\catcode 95\relax

```

```

215 \catcode 124 = \the\catcode 124\relax
216 \catcode 126 = \the\catcode 126\relax
217 \endlinechar = \the\endlinechar\relax
218 \chardef\csname\detokenize{l__kernel_expl_bool}\endcsname = 0\relax
219 }
220 </package>

```

(End definition for \ExplSyntaxOff. This function is documented on page 7.)

The code environment is now set up.

```

221 \catcode 9 = 9\relax
222 \catcode 32 = 9\relax
223 \catcode 34 = 12\relax
224 \catcode 58 = 11\relax
225 \catcode 94 = 7\relax
226 \catcode 95 = 11\relax
227 \catcode 124 = 12\relax
228 \catcode 126 = 10\relax
229 \endlinechar = 32\relax

```

\l__kernel_expl_bool The status for experimental code syntax: this is on at present.

```

230 \chardef\l__kernel_expl_bool = 1 ~

```

(End definition for \l__kernel_expl_bool. This variable is documented on page 8.)

\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!

```

231 \protected \def \ExplSyntaxOn
232 {
233   \bool_if:NF \l__kernel_expl_bool
234   {
235     \cs_set_protected_nopar:Npx \ExplSyntaxOff
236     {
237       \char_set_catcode:nn { 9 } { \char_value_catcode:n { 9 } }
238       \char_set_catcode:nn { 32 } { \char_value_catcode:n { 32 } }
239       \char_set_catcode:nn { 34 } { \char_value_catcode:n { 34 } }
240       \char_set_catcode:nn { 38 } { \char_value_catcode:n { 38 } }
241       \char_set_catcode:nn { 58 } { \char_value_catcode:n { 58 } }
242       \char_set_catcode:nn { 94 } { \char_value_catcode:n { 94 } }
243       \char_set_catcode:nn { 95 } { \char_value_catcode:n { 95 } }
244       \char_set_catcode:nn { 124 } { \char_value_catcode:n { 124 } }
245       \char_set_catcode:nn { 126 } { \char_value_catcode:n { 126 } }
246       \tex_endlinechar:D =
247       \tex_the:D \tex_endlinechar:D \scan_stop:
248       \bool_set_false:N \l__kernel_expl_bool
249       \cs_set_protected_nopar:Npn \ExplSyntaxOff { }
250     }
251   }
252   \char_set_catcode_ignore:n { 9 } % tab

```

```

253 \char_set_catcode_ignore:n { 32 } % space
254 \char_set_catcode_other:n { 34 } % double quote
255 \char_set_catcode_alignment:n { 38 } % ampersand
256 \char_set_catcode_letter:n { 58 } % colon
257 \char_set_catcode_math_superscript:n { 94 } % circumflex
258 \char_set_catcode_letter:n { 95 } % underscore
259 \char_set_catcode_other:n { 124 } % pipe
260 \char_set_catcode_space:n { 126 } % tilde
261 \tex_endlinechar:D = 32 \scan_stop:
262 \bool_set_true:N \l__kernel_expl_bool
263 }

```

(End definition for `\ExplSyntaxOn`. This function is documented on page 7.)

```

264 \</initex | package>

```

2 l3names implementation

```

265 \<*initex | package>

```

No prefix substitution here.

```

266 \<@@=>

```

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).

`\tex_undefined:D` This function does not exist at all, but is the name used by the plain T_EX format for an undefined function. So it should be marked here as “taken”.

(End definition for `\tex_undefined:D`. This function is documented on page ??.)

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.

```

267 \let \tex_global:D \global
268 \let \tex_let:D \let

```

Everything is inside a (rather long) group, which keeps `__kernel_primitive:NN` trapped.

```

269 \begingroup

```

`__kernel_primitive:NN` A temporary function to actually do the renaming. This also allows the original names to be removed in format mode.

```

270 \long \def \__kernel_primitive:NN #1#2
271 {
272   \tex_global:D \tex_let:D #2 #1
273 \<*initex>
274   \tex_global:D \tex_let:D #1 \tex_undefined:D
275 \</initex>
276 }

```

(End definition for `_kernel_primitive:NN`.)

To allow extracting “just the names”, a bit of DocStrip fiddling.

```
277 </initex | package>
278 <*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.

```
279 \_kernel\_primitive:NN \                \tex_space:D
280 \_kernel\_primitive:NN \/                \tex_italiccorrection:D
281 \_kernel\_primitive:NN \-                \tex_hyphen:D
```

Now all the other primitives.

```
282 \_kernel\_primitive:NN \above                \tex_above:D
283 \_kernel\_primitive:NN \abovedisplayshortskip \tex_abovedisplayshortskip:D
284 \_kernel\_primitive:NN \abovedisplayskip       \tex_abovedisplayskip:D
285 \_kernel\_primitive:NN \abovewithdelims        \tex_abovewithdelims:D
286 \_kernel\_primitive:NN \accent                 \tex_accent:D
287 \_kernel\_primitive:NN \adjdemerits            \tex_adjdemerits:D
288 \_kernel\_primitive:NN \advance                \tex_advance:D
289 \_kernel\_primitive:NN \afterassignment         \tex_afterassignment:D
290 \_kernel\_primitive:NN \aftergroup             \tex_aftergroup:D
291 \_kernel\_primitive:NN \atop                   \tex_atop:D
292 \_kernel\_primitive:NN \atopwithdelims         \tex_atopwithdelims:D
293 \_kernel\_primitive:NN \badness                \tex_badness:D
294 \_kernel\_primitive:NN \baselineskip           \tex_baselineskip:D
295 \_kernel\_primitive:NN \batchmode              \tex_batchmode:D
296 \_kernel\_primitive:NN \begingroup             \tex_begingroup:D
297 \_kernel\_primitive:NN \belowdisplayshortskip \tex_belowdisplayshortskip:D
298 \_kernel\_primitive:NN \belowdisplayskip       \tex_belowdisplayskip:D
299 \_kernel\_primitive:NN \binoppenalty           \tex_binoppenalty:D
300 \_kernel\_primitive:NN \botmark                \tex_botmark:D
301 \_kernel\_primitive:NN \box                   \tex_box:D
302 \_kernel\_primitive:NN \boxmaxdepth            \tex_boxmaxdepth:D
303 \_kernel\_primitive:NN \brokenpenalty          \tex_brokenpenalty:D
304 \_kernel\_primitive:NN \catcode                \tex_catcode:D
305 \_kernel\_primitive:NN \char                   \tex_char:D
306 \_kernel\_primitive:NN \chardef                \tex_chardef:D
307 \_kernel\_primitive:NN \cleaders               \tex_cleaders:D
308 \_kernel\_primitive:NN \closein                \tex_closein:D
309 \_kernel\_primitive:NN \closeout               \tex_closeout:D
310 \_kernel\_primitive:NN \clubpenalty            \tex_clubpenalty:D
311 \_kernel\_primitive:NN \copy                   \tex_copy:D
312 \_kernel\_primitive:NN \count                 \tex_count:D
313 \_kernel\_primitive:NN \countdef               \tex_countdef:D
314 \_kernel\_primitive:NN \cr                    \tex_cr:D
315 \_kernel\_primitive:NN \crrcr                 \tex_crrcr:D
316 \_kernel\_primitive:NN \csname                \tex_csname:D
317 \_kernel\_primitive:NN \day                   \tex_day:D
```

318	_kernel_primitive:NN	\deadcycles	\tex_deadcycles:D
319	_kernel_primitive:NN	\def	\tex_def:D
320	_kernel_primitive:NN	\defaultthyphenchar	\tex_defaultthyphenchar:D
321	_kernel_primitive:NN	\defaultskewchar	\tex_defaultskewchar:D
322	_kernel_primitive:NN	\delcode	\tex_delcode:D
323	_kernel_primitive:NN	\delimiter	\tex_delimiter:D
324	_kernel_primitive:NN	\delimiterfactor	\tex_delimiterfactor:D
325	_kernel_primitive:NN	\delimitershortfall	\tex_delimitershortfall:D
326	_kernel_primitive:NN	\dimen	\tex_dimen:D
327	_kernel_primitive:NN	\dimendef	\tex_dimendef:D
328	_kernel_primitive:NN	\discretionary	\tex_discretionary:D
329	_kernel_primitive:NN	\displayindent	\tex_displayindent:D
330	_kernel_primitive:NN	\displaylimits	\tex_displaylimits:D
331	_kernel_primitive:NN	\displaystyle	\tex_displaystyle:D
332	_kernel_primitive:NN	\displaywidowpenalty	\tex_displaywidowpenalty:D
333	_kernel_primitive:NN	\displaywidth	\tex_displaywidth:D
334	_kernel_primitive:NN	\divide	\tex_divide:D
335	_kernel_primitive:NN	\doublehyphendemerits	\tex_doublehyphendemerits:D
336	_kernel_primitive:NN	\dp	\tex_dp:D
337	_kernel_primitive:NN	\dump	\tex_dump:D
338	_kernel_primitive:NN	\edef	\tex_edef:D
339	_kernel_primitive:NN	\else	\tex_else:D
340	_kernel_primitive:NN	\emergencystretch	\tex_emergencystretch:D
341	_kernel_primitive:NN	\end	\tex_end:D
342	_kernel_primitive:NN	\endcsname	\tex_endcsname:D
343	_kernel_primitive:NN	\endgroup	\tex_endgroup:D
344	_kernel_primitive:NN	\endinput	\tex_endinput:D
345	_kernel_primitive:NN	\endlinechar	\tex_endlinechar:D
346	_kernel_primitive:NN	\eqno	\tex_eqno:D
347	_kernel_primitive:NN	\errhelp	\tex_errhelp:D
348	_kernel_primitive:NN	\errmessage	\tex_errmessage:D
349	_kernel_primitive:NN	\errorcontextlines	\tex_errorcontextlines:D
350	_kernel_primitive:NN	\errorstopmode	\tex_errorstopmode:D
351	_kernel_primitive:NN	\escapechar	\tex_escapechar:D
352	_kernel_primitive:NN	\everycr	\tex_everycr:D
353	_kernel_primitive:NN	\everydisplay	\tex_everydisplay:D
354	_kernel_primitive:NN	\everyhbox	\tex_everyhbox:D
355	_kernel_primitive:NN	\everyjob	\tex_everyjob:D
356	_kernel_primitive:NN	\everymath	\tex_everymath:D
357	_kernel_primitive:NN	\everypar	\tex_everypar:D
358	_kernel_primitive:NN	\everyvbox	\tex_everyvbox:D
359	_kernel_primitive:NN	\exhyphenpenalty	\tex_exhyphenpenalty:D
360	_kernel_primitive:NN	\expandafter	\tex_expandafter:D
361	_kernel_primitive:NN	\fam	\tex_fam:D
362	_kernel_primitive:NN	\fi	\tex_fi:D
363	_kernel_primitive:NN	\finalhyphendemerits	\tex_finalhyphendemerits:D
364	_kernel_primitive:NN	\firstmark	\tex_firstmark:D
365	_kernel_primitive:NN	\floatingpenalty	\tex_floatingpenalty:D
366	_kernel_primitive:NN	\font	\tex_font:D
367	_kernel_primitive:NN	\fontdimen	\tex_fontdimen:D

368	_kernel_primitive:NN	\fontname	\tex_fontname:D
369	_kernel_primitive:NN	\futurelet	\tex_futurelet:D
370	_kernel_primitive:NN	\gdef	\tex_gdef:D
371	_kernel_primitive:NN	\global	\tex_global:D
372	_kernel_primitive:NN	\globaldefs	\tex_globaldefs:D
373	_kernel_primitive:NN	\halign	\tex_halign:D
374	_kernel_primitive:NN	\hangingafter	\tex_hangingafter:D
375	_kernel_primitive:NN	\hangindent	\tex_hangindent:D
376	_kernel_primitive:NN	\hbadness	\tex_hbadness:D
377	_kernel_primitive:NN	\hbox	\tex_hbox:D
378	_kernel_primitive:NN	\hfil	\tex_hfil:D
379	_kernel_primitive:NN	\hfill	\tex_hfill:D
380	_kernel_primitive:NN	\hfilneg	\tex_hfilneg:D
381	_kernel_primitive:NN	\hfuzz	\tex_hfuzz:D
382	_kernel_primitive:NN	\hoffset	\tex_hoffset:D
383	_kernel_primitive:NN	\holdinginserts	\tex_holdinginserts:D
384	_kernel_primitive:NN	\hrule	\tex_hrule:D
385	_kernel_primitive:NN	\hsize	\tex_hsize:D
386	_kernel_primitive:NN	\hskip	\tex_hskip:D
387	_kernel_primitive:NN	\hss	\tex_hss:D
388	_kernel_primitive:NN	\ht	\tex_ht:D
389	_kernel_primitive:NN	\hyphenation	\tex_hyphenation:D
390	_kernel_primitive:NN	\hyphenchar	\tex_hyphenchar:D
391	_kernel_primitive:NN	\hyphenpenalty	\tex_hyphenpenalty:D
392	_kernel_primitive:NN	\if	\tex_if:D
393	_kernel_primitive:NN	\ifcase	\tex_ifcase:D
394	_kernel_primitive:NN	\ifcat	\tex_ifcat:D
395	_kernel_primitive:NN	\ifdim	\tex_ifdim:D
396	_kernel_primitive:NN	\ifeof	\tex_ifeof:D
397	_kernel_primitive:NN	\iffalse	\tex_iffalse:D
398	_kernel_primitive:NN	\ifhbox	\tex_ifhbox:D
399	_kernel_primitive:NN	\ifhmode	\tex_ifhmode:D
400	_kernel_primitive:NN	\ifinner	\tex_ifinner:D
401	_kernel_primitive:NN	\ifmmode	\tex_ifmmode:D
402	_kernel_primitive:NN	\ifnum	\tex_ifnum:D
403	_kernel_primitive:NN	\ifodd	\tex_ifodd:D
404	_kernel_primitive:NN	\iftrue	\tex_iftrue:D
405	_kernel_primitive:NN	\ifvbox	\tex_ifvbox:D
406	_kernel_primitive:NN	\ifvmode	\tex_ifvmode:D
407	_kernel_primitive:NN	\ifvoid	\tex_ifvoid:D
408	_kernel_primitive:NN	\ifx	\tex_ifx:D
409	_kernel_primitive:NN	\ignorespaces	\tex_ignorespaces:D
410	_kernel_primitive:NN	\immediate	\tex_immediate:D
411	_kernel_primitive:NN	\indent	\tex_indent:D
412	_kernel_primitive:NN	\input	\tex_input:D
413	_kernel_primitive:NN	\inputlineno	\tex_inputlineno:D
414	_kernel_primitive:NN	\insert	\tex_insert:D
415	_kernel_primitive:NN	\insertpenalties	\tex_insertpenalties:D
416	_kernel_primitive:NN	\interlinepenalty	\tex_interlinepenalty:D
417	_kernel_primitive:NN	\jobname	\tex_jobname:D

418	_kernel_primitive:NN	\kern	\tex_kern:D
419	_kernel_primitive:NN	\language	\tex_language:D
420	_kernel_primitive:NN	\lastbox	\tex_lastbox:D
421	_kernel_primitive:NN	\lastkern	\tex_lastkern:D
422	_kernel_primitive:NN	\lastpenalty	\tex_lastpenalty:D
423	_kernel_primitive:NN	\lastskip	\tex_lastskip:D
424	_kernel_primitive:NN	\lccode	\tex_lccode:D
425	_kernel_primitive:NN	\leaders	\tex_leaders:D
426	_kernel_primitive:NN	\left	\tex_left:D
427	_kernel_primitive:NN	\lefthyphenmin	\tex_lefthyphenmin:D
428	_kernel_primitive:NN	\leftskip	\tex_leftskip:D
429	_kernel_primitive:NN	\leqno	\tex_leqno:D
430	_kernel_primitive:NN	\let	\tex_let:D
431	_kernel_primitive:NN	\limits	\tex_limits:D
432	_kernel_primitive:NN	\linepenalty	\tex_linepenalty:D
433	_kernel_primitive:NN	\lineskip	\tex_lineskip:D
434	_kernel_primitive:NN	\lineskiplimit	\tex_lineskiplimit:D
435	_kernel_primitive:NN	\long	\tex_long:D
436	_kernel_primitive:NN	\looseness	\tex_looseness:D
437	_kernel_primitive:NN	\lower	\tex_lower:D
438	_kernel_primitive:NN	\lowercase	\tex_lowercase:D
439	_kernel_primitive:NN	\mag	\tex_mag:D
440	_kernel_primitive:NN	\mark	\tex_mark:D
441	_kernel_primitive:NN	\mathaccent	\tex_mathaccent:D
442	_kernel_primitive:NN	\mathbin	\tex_mathbin:D
443	_kernel_primitive:NN	\mathchar	\tex_mathchar:D
444	_kernel_primitive:NN	\mathchardef	\tex_mathchardef:D
445	_kernel_primitive:NN	\mathchoice	\tex_mathchoice:D
446	_kernel_primitive:NN	\mathclose	\tex_mathclose:D
447	_kernel_primitive:NN	\mathcode	\tex_mathcode:D
448	_kernel_primitive:NN	\mathinner	\tex_mathinner:D
449	_kernel_primitive:NN	\mathop	\tex_mathop:D
450	_kernel_primitive:NN	\mathopen	\tex_mathopen:D
451	_kernel_primitive:NN	\mathord	\tex_mathord:D
452	_kernel_primitive:NN	\mathpunct	\tex_mathpunct:D
453	_kernel_primitive:NN	\mathrel	\tex_mathrel:D
454	_kernel_primitive:NN	\mathsurround	\tex_mathsurround:D
455	_kernel_primitive:NN	\maxdeadcycles	\tex_maxdeadcycles:D
456	_kernel_primitive:NN	\maxdepth	\tex_maxdepth:D
457	_kernel_primitive:NN	\meaning	\tex_meaning:D
458	_kernel_primitive:NN	\medmuskip	\tex_medmuskip:D
459	_kernel_primitive:NN	\message	\tex_message:D
460	_kernel_primitive:NN	\mkern	\tex_mkern:D
461	_kernel_primitive:NN	\month	\tex_month:D
462	_kernel_primitive:NN	\moveleft	\tex_moveleft:D
463	_kernel_primitive:NN	\moveright	\tex_moveright:D
464	_kernel_primitive:NN	\mskip	\tex_mskip:D
465	_kernel_primitive:NN	\multiply	\tex_multiply:D
466	_kernel_primitive:NN	\muskip	\tex_muskip:D
467	_kernel_primitive:NN	\muskipdef	\tex_muskipdef:D

468	_kernel_primitive:NN	\newlinechar	\tex_newlinechar:D
469	_kernel_primitive:NN	\noalign	\tex_noalign:D
470	_kernel_primitive:NN	\noboundary	\tex_noboundary:D
471	_kernel_primitive:NN	\noexpand	\tex_noexpand:D
472	_kernel_primitive:NN	\noindent	\tex_noindent:D
473	_kernel_primitive:NN	\nolimits	\tex_nolimits:D
474	_kernel_primitive:NN	\nonscript	\tex_nonscript:D
475	_kernel_primitive:NN	\nonstopmode	\tex_nonstopmode:D
476	_kernel_primitive:NN	\nullldelimiterspace	\tex_nullldelimiterspace:D
477	_kernel_primitive:NN	\nullfont	\tex_nullfont:D
478	_kernel_primitive:NN	\number	\tex_number:D
479	_kernel_primitive:NN	\omit	\tex_omit:D
480	_kernel_primitive:NN	\openin	\tex_openin:D
481	_kernel_primitive:NN	\openout	\tex_openout:D
482	_kernel_primitive:NN	\or	\tex_or:D
483	_kernel_primitive:NN	\outer	\tex_outer:D
484	_kernel_primitive:NN	\output	\tex_output:D
485	_kernel_primitive:NN	\outputpenalty	\tex_outputpenalty:D
486	_kernel_primitive:NN	\over	\tex_over:D
487	_kernel_primitive:NN	\overfullrule	\tex_overfullrule:D
488	_kernel_primitive:NN	\overline	\tex_overline:D
489	_kernel_primitive:NN	\overwithdelims	\tex_overwithdelims:D
490	_kernel_primitive:NN	\pagedepth	\tex_pagedepth:D
491	_kernel_primitive:NN	\pagefilllstretch	\tex_pagefilllstretch:D
492	_kernel_primitive:NN	\pagefillstretch	\tex_pagefillstretch:D
493	_kernel_primitive:NN	\pagefilstretch	\tex_pagefilstretch:D
494	_kernel_primitive:NN	\pagegoal	\tex_pagegoal:D
495	_kernel_primitive:NN	\pageshrink	\tex_pageshrink:D
496	_kernel_primitive:NN	\pagestretch	\tex_pagestretch:D
497	_kernel_primitive:NN	\pagetotal	\tex_pagetotal:D
498	_kernel_primitive:NN	\par	\tex_par:D
499	_kernel_primitive:NN	\parfillskip	\tex_parfillskip:D
500	_kernel_primitive:NN	\parindent	\tex_parindent:D
501	_kernel_primitive:NN	\parshape	\tex_parshape:D
502	_kernel_primitive:NN	\parskip	\tex_parskip:D
503	_kernel_primitive:NN	\patterns	\tex_patterns:D
504	_kernel_primitive:NN	\pausing	\tex_pausing:D
505	_kernel_primitive:NN	\penalty	\tex_penalty:D
506	_kernel_primitive:NN	\postdisplaypenalty	\tex_postdisplaypenalty:D
507	_kernel_primitive:NN	\predisdisplaypenalty	\tex_predisdisplaypenalty:D
508	_kernel_primitive:NN	\predisplaysize	\tex_predisplaysize:D
509	_kernel_primitive:NN	\pretolerance	\tex_pretolerance:D
510	_kernel_primitive:NN	\prevdepth	\tex_prevdepth:D
511	_kernel_primitive:NN	\prevgraf	\tex_prevgraf:D
512	_kernel_primitive:NN	\radical	\tex_radical:D
513	_kernel_primitive:NN	\raise	\tex_raise:D
514	_kernel_primitive:NN	\read	\tex_read:D
515	_kernel_primitive:NN	\relax	\tex_relax:D
516	_kernel_primitive:NN	\relpenalty	\tex_relpenalty:D
517	_kernel_primitive:NN	\right	\tex_right:D

518	_kernel_primitive:NN	\righthyphenmin	\tex_righthyphenmin:D
519	_kernel_primitive:NN	\rightskip	\tex_rightskip:D
520	_kernel_primitive:NN	\romannumeral	\tex_romannumeral:D
521	_kernel_primitive:NN	\scriptfont	\tex_scriptfont:D
522	_kernel_primitive:NN	\scriptscriptfont	\tex_scriptscriptfont:D
523	_kernel_primitive:NN	\scriptscriptstyle	\tex_scriptscriptstyle:D
524	_kernel_primitive:NN	\scriptspace	\tex_scriptspace:D
525	_kernel_primitive:NN	\scriptstyle	\tex_scriptstyle:D
526	_kernel_primitive:NN	\scrollmode	\tex_scrollmode:D
527	_kernel_primitive:NN	\setbox	\tex_setbox:D
528	_kernel_primitive:NN	\setlanguage	\tex_setlanguage:D
529	_kernel_primitive:NN	\sfcode	\tex_sfcode:D
530	_kernel_primitive:NN	\shipout	\tex_shipout:D
531	_kernel_primitive:NN	\show	\tex_show:D
532	_kernel_primitive:NN	\showbox	\tex_showbox:D
533	_kernel_primitive:NN	\showboxbreadth	\tex_showboxbreadth:D
534	_kernel_primitive:NN	\showboxdepth	\tex_showboxdepth:D
535	_kernel_primitive:NN	\showlists	\tex_showlists:D
536	_kernel_primitive:NN	\showthe	\tex_showthe:D
537	_kernel_primitive:NN	\skewchar	\tex_skewchar:D
538	_kernel_primitive:NN	\skip	\tex_skip:D
539	_kernel_primitive:NN	\skipdef	\tex_skipdef:D
540	_kernel_primitive:NN	\spacefactor	\tex_spacefactor:D
541	_kernel_primitive:NN	\spaceskip	\tex_spaceskip:D
542	_kernel_primitive:NN	\span	\tex_span:D
543	_kernel_primitive:NN	\special	\tex_special:D
544	_kernel_primitive:NN	\splitbotmark	\tex_splitbotmark:D
545	_kernel_primitive:NN	\splitfirstmark	\tex_splitfirstmark:D
546	_kernel_primitive:NN	\splitmaxdepth	\tex_splitmaxdepth:D
547	_kernel_primitive:NN	\splittopskip	\tex_splittopskip:D
548	_kernel_primitive:NN	\string	\tex_string:D
549	_kernel_primitive:NN	\tabskip	\tex_tabskip:D
550	_kernel_primitive:NN	\textfont	\tex_textfont:D
551	_kernel_primitive:NN	\textstyle	\tex_textstyle:D
552	_kernel_primitive:NN	\the	\tex_the:D
553	_kernel_primitive:NN	\thickmuskip	\tex_thickmuskip:D
554	_kernel_primitive:NN	\thinmuskip	\tex_thinmuskip:D
555	_kernel_primitive:NN	\time	\tex_time:D
556	_kernel_primitive:NN	\toks	\tex_toks:D
557	_kernel_primitive:NN	\toksdef	\tex_toksdef:D
558	_kernel_primitive:NN	\tolerance	\tex_tolerance:D
559	_kernel_primitive:NN	\topmark	\tex_topmark:D
560	_kernel_primitive:NN	\topskip	\tex_topskip:D
561	_kernel_primitive:NN	\tracingcommands	\tex_tracingcommands:D
562	_kernel_primitive:NN	\tracinglostchars	\tex_tracinglostchars:D
563	_kernel_primitive:NN	\tracingmacros	\tex_tracingmacros:D
564	_kernel_primitive:NN	\tracingonline	\tex_tracingonline:D
565	_kernel_primitive:NN	\tracingoutput	\tex_tracingoutput:D
566	_kernel_primitive:NN	\tracingpages	\tex_tracingpages:D
567	_kernel_primitive:NN	\tracingparagraphs	\tex_tracingparagraphs:D

568	_kernel_primitive:NN	\tracingrestores	\tex_tracingrestores:D
569	_kernel_primitive:NN	\tracingstats	\tex_tracingstats:D
570	_kernel_primitive:NN	\uccode	\tex_uccode:D
571	_kernel_primitive:NN	\uchyph	\tex_uchyph:D
572	_kernel_primitive:NN	\underline	\tex_underline:D
573	_kernel_primitive:NN	\unhbox	\tex_unhbox:D
574	_kernel_primitive:NN	\unhcopy	\tex_unhcopy:D
575	_kernel_primitive:NN	\unkern	\tex_unkern:D
576	_kernel_primitive:NN	\unpenalty	\tex_unpenalty:D
577	_kernel_primitive:NN	\unskip	\tex_unskip:D
578	_kernel_primitive:NN	\unvbox	\tex_unvbox:D
579	_kernel_primitive:NN	\unvcopy	\tex_unvcopy:D
580	_kernel_primitive:NN	\uppercase	\tex_uppercase:D
581	_kernel_primitive:NN	\vadjust	\tex_vadjust:D
582	_kernel_primitive:NN	\valign	\tex_valign:D
583	_kernel_primitive:NN	\vbadness	\tex_vbadness:D
584	_kernel_primitive:NN	\vbox	\tex_vbox:D
585	_kernel_primitive:NN	\vcenter	\tex_vcenter:D
586	_kernel_primitive:NN	\vfil	\tex_vfil:D
587	_kernel_primitive:NN	\vfill	\tex_vfill:D
588	_kernel_primitive:NN	\vfilneg	\tex_vfilneg:D
589	_kernel_primitive:NN	\vfuzz	\tex_vfuzz:D
590	_kernel_primitive:NN	\voffset	\tex_voffset:D
591	_kernel_primitive:NN	\vrule	\tex_vrule:D
592	_kernel_primitive:NN	\vsize	\tex_vsize:D
593	_kernel_primitive:NN	\vskip	\tex_vskip:D
594	_kernel_primitive:NN	\vsplit	\tex_vsplit:D
595	_kernel_primitive:NN	\vss	\tex_vss:D
596	_kernel_primitive:NN	\vtop	\tex_vtop:D
597	_kernel_primitive:NN	\wd	\tex_wd:D
598	_kernel_primitive:NN	\widowpenalty	\tex_widowpenalty:D
599	_kernel_primitive:NN	\write	\tex_write:D
600	_kernel_primitive:NN	\xdef	\tex_xdef:D
601	_kernel_primitive:NN	\xleaders	\tex_xleaders:D
602	_kernel_primitive:NN	\xspaceskip	\tex_xspaceskip:D
603	_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_`.

604	_kernel_primitive:NN	\beginL	\etex_beginL:D
605	_kernel_primitive:NN	\beginR	\etex_beginR:D
606	_kernel_primitive:NN	\botmarks	\etex_botmarks:D
607	_kernel_primitive:NN	\clubpenalties	\etex_clubpenalties:D
608	_kernel_primitive:NN	\currentgrouplevel	\etex_currentgrouplevel:D
609	_kernel_primitive:NN	\currentgrouptype	\etex_currentgrouptype:D
610	_kernel_primitive:NN	\currentifbranch	\etex_currentifbranch:D
611	_kernel_primitive:NN	\currentiflevel	\etex_currentiflevel:D
612	_kernel_primitive:NN	\currentifttype	\etex_currentifttype:D
613	_kernel_primitive:NN	\detokenize	\etex_detokenize:D
614	_kernel_primitive:NN	\dimexpr	\etex_dimexpr:D

615	_kernel_primitive:NN	\displaywidowpenalties	\etex_displaywidowpenalties:D
616	_kernel_primitive:NN	\endL	\etex_endL:D
617	_kernel_primitive:NN	\endR	\etex_endR:D
618	_kernel_primitive:NN	\eTeXrevision	\etex_eTeXrevision:D
619	_kernel_primitive:NN	\eTeXversion	\etex_eTeXversion:D
620	_kernel_primitive:NN	\everyeof	\etex_everyeof:D
621	_kernel_primitive:NN	\firstmarks	\etex_firstmarks:D
622	_kernel_primitive:NN	\fontchardp	\etex_fontchardp:D
623	_kernel_primitive:NN	\fontcharht	\etex_fontcharht:D
624	_kernel_primitive:NN	\fontcharic	\etex_fontcharic:D
625	_kernel_primitive:NN	\fontcharwd	\etex_fontcharwd:D
626	_kernel_primitive:NN	\glueexpr	\etex_glueexpr:D
627	_kernel_primitive:NN	\glueshrink	\etex_glueshrink:D
628	_kernel_primitive:NN	\glueshrinkorder	\etex_glueshrinkorder:D
629	_kernel_primitive:NN	\gluestretch	\etex_gluestretch:D
630	_kernel_primitive:NN	\gluestretchorder	\etex_gluestretchorder:D
631	_kernel_primitive:NN	\gluetomu	\etex_gluetomu:D
632	_kernel_primitive:NN	\ifcsname	\etex_ifcsname:D
633	_kernel_primitive:NN	\ifdefined	\etex_ifdefined:D
634	_kernel_primitive:NN	\iffontchar	\etex_iffontchar:D
635	_kernel_primitive:NN	\interactionmode	\etex_interactionmode:D
636	_kernel_primitive:NN	\interlinepenalties	\etex_interlinepenalties:D
637	_kernel_primitive:NN	\lastlinefit	\etex_lastlinefit:D
638	_kernel_primitive:NN	\lastnodetype	\etex_lastnodetype:D
639	_kernel_primitive:NN	\marks	\etex_marks:D
640	_kernel_primitive:NN	\middle	\etex_middle:D
641	_kernel_primitive:NN	\muexpr	\etex_muexpr:D
642	_kernel_primitive:NN	\mutoglu	\etex_mutoglu:D
643	_kernel_primitive:NN	\numexpr	\etex_numexpr:D
644	_kernel_primitive:NN	\pagediscards	\etex_pagediscards:D
645	_kernel_primitive:NN	\parshapedimen	\etex_parshapedimen:D
646	_kernel_primitive:NN	\parshapeindent	\etex_parshapeindent:D
647	_kernel_primitive:NN	\parshapelength	\etex_parshapelength:D
648	_kernel_primitive:NN	\predisplaydirection	\etex_predisplaydirection:D
649	_kernel_primitive:NN	\protected	\etex_protected:D
650	_kernel_primitive:NN	\readline	\etex_readline:D
651	_kernel_primitive:NN	\savinghyphcodes	\etex_savinghyphcodes:D
652	_kernel_primitive:NN	\savingvdiscards	\etex_savingvdiscards:D
653	_kernel_primitive:NN	\scantokens	\etex_scantokens:D
654	_kernel_primitive:NN	\showgroups	\etex_showgroups:D
655	_kernel_primitive:NN	\showifs	\etex_showifs:D
656	_kernel_primitive:NN	\showtokens	\etex_showtokens:D
657	_kernel_primitive:NN	\splitbotmarks	\etex_splitbotmarks:D
658	_kernel_primitive:NN	\splitdiscards	\etex_splitdiscards:D
659	_kernel_primitive:NN	\splitfirstmarks	\etex_splitfirstmarks:D
660	_kernel_primitive:NN	\TeXXeTstate	\etex_TeXXeTstate:D
661	_kernel_primitive:NN	\topmarks	\etex_topmarks:D
662	_kernel_primitive:NN	\tracingassigns	\etex_tracingassigns:D
663	_kernel_primitive:NN	\tracinggroups	\etex_tracinggroups:D
664	_kernel_primitive:NN	\tracingifs	\etex_tracingifs:D

```

665 \__kernel_primitive:NN \tracingnesting \etex_tracingnesting:D
666 \__kernel_primitive:NN \tracingscantokens \etex_tracingscantokens:D
667 \__kernel_primitive:NN \unexpanded \etex_unexpanded:D
668 \__kernel_primitive:NN \unless \etex_unless:D
669 \__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.

```

670 \__kernel_primitive:NN \pdfannot \pdfTEX_pdfannot:D
671 \__kernel_primitive:NN \pdfcatalog \pdfTEX_pdfcatalog:D
672 \__kernel_primitive:NN \pdfcompresslevel \pdfTEX_pdfcompresslevel:D
673 \__kernel_primitive:NN \pdfcolorstack \pdfTEX_pdfcolorstack:D
674 \__kernel_primitive:NN \pdfcolorstackinit \pdfTEX_pdfcolorstackinit:D
675 \__kernel_primitive:NN \pdfcreationdate \pdfTEX_pdfcreationdate:D
676 \__kernel_primitive:NN \pdfdecimaldigits \pdfTEX_pdfdecimaldigits:D
677 \__kernel_primitive:NN \pdfdest \pdfTEX_pdfdest:D
678 \__kernel_primitive:NN \pdfdestmargin \pdfTEX_pdfdestmargin:D
679 \__kernel_primitive:NN \pdfendlink \pdfTEX_pdfendlink:D
680 \__kernel_primitive:NN \pdfendthread \pdfTEX_pdfendthread:D
681 \__kernel_primitive:NN \pdffontattr \pdfTEX_pdffontattr:D
682 \__kernel_primitive:NN \pdffontname \pdfTEX_pdffontname:D
683 \__kernel_primitive:NN \pdffontobjnum \pdfTEX_pdffontobjnum:D
684 \__kernel_primitive:NN \pdfgamma \pdfTEX_pdfgamma:D
685 \__kernel_primitive:NN \pdfimageapplygamma \pdfTEX_pdfimageapplygamma:D
686 \__kernel_primitive:NN \pdfimagegamma \pdfTEX_pdfimagegamma:D
687 \__kernel_primitive:NN \pdfgentounicode \pdfTEX_pdfgentounicode:D
688 \__kernel_primitive:NN \pdfglyphtounicode \pdfTEX_pdfglyphtounicode:D
689 \__kernel_primitive:NN \pdfhorigin \pdfTEX_pdfhorigin:D
690 \__kernel_primitive:NN \pdfimagehicolor \pdfTEX_pdfimagehicolor:D
691 \__kernel_primitive:NN \pdfimageresolution \pdfTEX_pdfimageresolution:D
692 \__kernel_primitive:NN \pdfincludechars \pdfTEX_pdfincludechars:D
693 \__kernel_primitive:NN \pdfinclusioncopyfonts \pdfTEX_pdfinclusioncopyfonts:D
694 \__kernel_primitive:NN \pdfinclusionerrorlevel \pdfTEX_pdfinclusionerrorlevel:D
695 \__kernel_primitive:NN \pdfinfo \pdfTEX_pdfinfo:D
696 \__kernel_primitive:NN \pdflastannot \pdfTEX_pdflastannot:D
697 \__kernel_primitive:NN \pdflastlink \pdfTEX_pdflastlink:D
698 \__kernel_primitive:NN \pdflastobj \pdfTEX_pdflastobj:D
699 \__kernel_primitive:NN \pdflastxform \pdfTEX_pdflastxform:D
700 \__kernel_primitive:NN \pdflastximage \pdfTEX_pdflastximage:D
701 \__kernel_primitive:NN \pdflastximagecolordepth \pdfTEX_pdflastximagecolordepth:D
702 \__kernel_primitive:NN \pdflastximagepages \pdfTEX_pdflastximagepages:D
703 \__kernel_primitive:NN \pdflinkmargin \pdfTEX_pdflinkmargin:D
704 \__kernel_primitive:NN \pdfliteral \pdfTEX_pdfliteral:D
705 \__kernel_primitive:NN \pdfminorversion \pdfTEX_pdfminorversion:D
706 \__kernel_primitive:NN \pdfnames \pdfTEX_pdfnames:D
707 \__kernel_primitive:NN \pdfobj \pdfTEX_pdfobj:D

```

708	_kernel_primitive:NN	\pdfobjcompresslevel	\pdftex_pdfobjcompresslevel:D
709	_kernel_primitive:NN	\pdfoutline	\pdftex_pdfoutline:D
710	_kernel_primitive:NN	\pdfoutput	\pdftex_pdfoutput:D
711	_kernel_primitive:NN	\pdfpageattr	\pdftex_pdfpageattr:D
712	_kernel_primitive:NN	\pdfpagebox	\pdftex_pdfpagebox:D
713	_kernel_primitive:NN	\pdfpageheight	\pdftex_pdfpageheight:D
714	_kernel_primitive:NN	\pdfpageref	\pdftex_pdfpageref:D
715	_kernel_primitive:NN	\pdfpageresources	\pdftex_pdfpageresources:D
716	_kernel_primitive:NN	\pdfpagesattr	\pdftex_pdfpagesattr:D
717	_kernel_primitive:NN	\pdfpagewidth	\pdftex_pdfpagewidth:D
718	_kernel_primitive:NN	\pdfrefobj	\pdftex_pdfrefobj:D
719	_kernel_primitive:NN	\pdfrefxform	\pdftex_pdfrefxform:D
720	_kernel_primitive:NN	\pdfrefximage	\pdftex_pdfrefximage:D
721	_kernel_primitive:NN	\pdfrestore	\pdftex_pdfrestore:D
722	_kernel_primitive:NN	\pdfretval	\pdftex_pdfretval:D
723	_kernel_primitive:NN	\pdfsave	\pdftex_pdfsave:D
724	_kernel_primitive:NN	\pdfsetmatrix	\pdftex_pdfsetmatrix:D
725	_kernel_primitive:NN	\pdfstartlink	\pdftex_pdfstartlink:D
726	_kernel_primitive:NN	\pdfstartthread	\pdftex_pdfstartthread:D
727	_kernel_primitive:NN	\pdfthread	\pdftex_pdfthread:D
728	_kernel_primitive:NN	\pdfthreadmargin	\pdftex_pdfthreadmargin:D
729	_kernel_primitive:NN	\pdftrailer	\pdftex_pdftrailer:D
730	_kernel_primitive:NN	\pdfuniquestring	\pdftex_pdfuniquestring:D
731	_kernel_primitive:NN	\pdfvorigin	\pdftex_pdfvorigin:D
732	_kernel_primitive:NN	\pdfxform	\pdftex_pdfxform:D
733	_kernel_primitive:NN	\pdfxformattr	\pdftex_pdfxformattr:D
734	_kernel_primitive:NN	\pdfxformname	\pdftex_pdfxformname:D
735	_kernel_primitive:NN	\pdfxformresources	\pdftex_pdfxformresources:D
736	_kernel_primitive:NN	\pdfximage	\pdftex_pdfximage:D
737	_kernel_primitive:NN	\pdfximagebbox	\pdftex_pdfximagebbox:D

While these are not.

738	_kernel_primitive:NN	\ifpdfprimitive	\pdftex_ifprimitive:D
739	_kernel_primitive:NN	\pdfadjustspacing	\pdftex_adjustspacing:D
740	_kernel_primitive:NN	\pdfcopyfont	\pdftex_copyfont:D
741	_kernel_primitive:NN	\pdfdraftmode	\pdftex_draftmode:D
742	_kernel_primitive:NN	\pdfeachlinedepth	\pdftex_eachlinedepth:D
743	_kernel_primitive:NN	\pdfeachlineheight	\pdftex_eachlineheight:D
744	_kernel_primitive:NN	\pdffirstlineheight	\pdftex_firstlineheight:D
745	_kernel_primitive:NN	\pdffontexpand	\pdftex_fontexpand:D
746	_kernel_primitive:NN	\pdffontsize	\pdftex_fontsize:D
747	_kernel_primitive:NN	\pdfignoreddimen	\pdftex_ignoreddimen:D
748	_kernel_primitive:NN	\pdfinserttht	\pdftex_inserttht:D
749	_kernel_primitive:NN	\pdflastlinedepth	\pdftex_lastlinedepth:D
750	_kernel_primitive:NN	\pdflastxpos	\pdftex_lastxpos:D
751	_kernel_primitive:NN	\pdflastypos	\pdftex_lastypos:D
752	_kernel_primitive:NN	\pdfmapfile	\pdftex_mapfile:D
753	_kernel_primitive:NN	\pdfmapline	\pdftex_mapline:D
754	_kernel_primitive:NN	\pdfnoligatures	\pdftex_noligatures:D
755	_kernel_primitive:NN	\pdfnormaldeviate	\pdftex_normaldeviate:D

756	<code>__kernel_primitive:NN \pdfpkmode</code>	<code>\pdfutex_pkmode:D</code>
757	<code>__kernel_primitive:NN \pdfpkresolution</code>	<code>\pdfutex_pkresolution:D</code>
758	<code>__kernel_primitive:NN \pdfprimitive</code>	<code>\pdfutex_primitive:D</code>
759	<code>__kernel_primitive:NN \pdfprotrudechars</code>	<code>\pdfutex_protrudechars:D</code>
760	<code>__kernel_primitive:NN \pdfpxdimen</code>	<code>\pdfutex_pxdimen:D</code>
761	<code>__kernel_primitive:NN \pdfrandomseed</code>	<code>\pdfutex_randomseed:D</code>
762	<code>__kernel_primitive:NN \pdfsavepos</code>	<code>\pdfutex_savepos:D</code>
763	<code>__kernel_primitive:NN \pdfstrcmp</code>	<code>\pdfutex_strcmp:D</code>
764	<code>__kernel_primitive:NN \pdfsetrandomseed</code>	<code>\pdfutex_setrandomseed:D</code>
765	<code>__kernel_primitive:NN \pdfshellescape</code>	<code>\pdfutex_shellescape:D</code>
766	<code>__kernel_primitive:NN \pdftracingfonts</code>	<code>\pdfutex_tracingfonts:D</code>
767	<code>__kernel_primitive:NN \pdfuniformdeviate</code>	<code>\pdfutex_uniformdeviate:D</code>

The version primitives are not related to PDF mode but are related to pdfTeX so retain the full prefix.

768	<code>__kernel_primitive:NN \pdfutexbanner</code>	<code>\pdfutex_pdfutexbanner:D</code>
769	<code>__kernel_primitive:NN \pdfutexrevision</code>	<code>\pdfutex_pdfutexrevision:D</code>
770	<code>__kernel_primitive:NN \pdfutexversion</code>	<code>\pdfutex_pdfutexversion:D</code>

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!)

771	<code>__kernel_primitive:NN \efcode</code>	<code>\pdfutex_efcode:D</code>
772	<code>__kernel_primitive:NN \ifincsname</code>	<code>\pdfutex_ifincsname:D</code>
773	<code>__kernel_primitive:NN \leftmarginkern</code>	<code>\pdfutex_leftmarginkern:D</code>
774	<code>__kernel_primitive:NN \letterspacefont</code>	<code>\pdfutex_letterspacefont:D</code>
775	<code>__kernel_primitive:NN \lpcode</code>	<code>\pdfutex_lpcode:D</code>
776	<code>__kernel_primitive:NN \quitvmode</code>	<code>\pdfutex_quitvmode:D</code>
777	<code>__kernel_primitive:NN \rightmarginkern</code>	<code>\pdfutex_rightmarginkern:D</code>
778	<code>__kernel_primitive:NN \rpcode</code>	<code>\pdfutex_rpcode:D</code>
779	<code>__kernel_primitive:NN \synctex</code>	<code>\pdfutex_synctex:D</code>
780	<code>__kernel_primitive:NN \tagcode</code>	<code>\pdfutex_tagcode:D</code>

X_YTeX-specific primitives. Note that X_YTeX'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.

781	<code>__kernel_primitive:NN \suppressfontnotfounderror</code>	<code>\xetex_suppressfontnotfounderror:D</code>
782	<code>__kernel_primitive:NN \XeTeXcharclass</code>	<code>\xetex_charclass:D</code>
783	<code>__kernel_primitive:NN \XeTeXcharglyph</code>	<code>\xetex_charglyph:D</code>
784	<code>__kernel_primitive:NN \XeTeXcountfeatures</code>	<code>\xetex_countfeatures:D</code>
785	<code>__kernel_primitive:NN \XeTeXcountglyphs</code>	<code>\xetex_countglyphs:D</code>
786	<code>__kernel_primitive:NN \XeTeXcountselectors</code>	<code>\xetex_countselectors:D</code>
787	<code>__kernel_primitive:NN \XeTeXcountvariations</code>	<code>\xetex_countvariations:D</code>
788	<code>__kernel_primitive:NN \XeTeXdefaultencoding</code>	<code>\xetex_defaultencoding:D</code>
789	<code>__kernel_primitive:NN \XeTeXdashbreakstate</code>	<code>\xetex_dashbreakstate:D</code>
790	<code>__kernel_primitive:NN \XeTeXfeaturecode</code>	<code>\xetex_featurecode:D</code>
791	<code>__kernel_primitive:NN \XeTeXfeaturename</code>	<code>\xetex_featurename:D</code>
792	<code>__kernel_primitive:NN \XeTeXfindfeaturebyname</code>	<code>\xetex_findfeaturebyname:D</code>
793	<code>__kernel_primitive:NN \XeTeXfindselectorbyname</code>	<code>\xetex_findselectorbyname:D</code>
794	<code>__kernel_primitive:NN \XeTeXfindvariationbyname</code>	<code>\xetex_findvariationbyname:D</code>

795	_kernel_primitive:NN	\XeTeXfirstfontchar	\xetex_firstfontchar:D
796	_kernel_primitive:NN	\XeTeXfonttype	\xetex_fonttype:D
797	_kernel_primitive:NN	\XeTeXglyph	\xetex_glyph:D
798	_kernel_primitive:NN	\XeTeXglyphbounds	\xetex_glyphbounds:D
799	_kernel_primitive:NN	\XeTeXglyphindex	\xetex_glyphindex:D
800	_kernel_primitive:NN	\XeTeXglyphname	\xetex_glyphname:D
801	_kernel_primitive:NN	\XeTeXinputencoding	\xetex_inputencoding:D
802	_kernel_primitive:NN	\XeTeXinputnormalization	\xetex_inputnormalization:D
803	_kernel_primitive:NN	\XeTeXinterchartokenstate	\xetex_interchartokenstate:D
804	_kernel_primitive:NN	\XeTeXinterchartoks	\xetex_interchartoks:D
805	_kernel_primitive:NN	\XeTeXisdefaultselector	\xetex_isdefaultselector:D
806	_kernel_primitive:NN	\XeTeXisexclusivefeature	\xetex_isexclusivefeature:D
807	_kernel_primitive:NN	\XeTeXlastfontchar	\xetex_lastfontchar:D
808	_kernel_primitive:NN	\XeTeXlinebreakskip	\xetex_linebreakskip:D
809	_kernel_primitive:NN	\XeTeXlinebreaklocale	\xetex_linebreaklocale:D
810	_kernel_primitive:NN	\XeTeXlinebreakpenalty	\xetex_linebreakpenalty:D
811	_kernel_primitive:NN	\XeTeXOTcountfeatures	\xetex_OTcountfeatures:D
812	_kernel_primitive:NN	\XeTeXOTcountlanguages	\xetex_OTcountlanguages:D
813	_kernel_primitive:NN	\XeTeXOTcountscripts	\xetex_OTcountscripts:D
814	_kernel_primitive:NN	\XeTeXOTfeaturetag	\xetex_OTfeaturetag:D
815	_kernel_primitive:NN	\XeTeXOTlanguagetag	\xetex_OTlanguagetag:D
816	_kernel_primitive:NN	\XeTeXOTscripttag	\xetex_OTscripttag:D
817	_kernel_primitive:NN	\XeTeXpdffile	\xetex_pdffile:D
818	_kernel_primitive:NN	\XeTeXpdfpagecount	\xetex_pdfpagecount:D
819	_kernel_primitive:NN	\XeTeXpicfile	\xetex_picfile:D
820	_kernel_primitive:NN	\XeTeXselectorname	\xetex_selectorname:D
821	_kernel_primitive:NN	\XeTeXtracingfonts	\xetex_tracingfonts:D
822	_kernel_primitive:NN	\XeTeXupwardsmode	\xetex_upwardsmode:D
823	_kernel_primitive:NN	\XeTeXuseglyphmetrics	\xetex_useglyphmetrics:D
824	_kernel_primitive:NN	\XeTeXvariation	\xetex_variation:D
825	_kernel_primitive:NN	\XeTeXvariationdefault	\xetex_variationdefault:D
826	_kernel_primitive:NN	\XeTeXvariationmax	\xetex_variationmax:D
827	_kernel_primitive:NN	\XeTeXvariationmin	\xetex_variationmin:D
828	_kernel_primitive:NN	\XeTeXvariationname	\xetex_variationname:D

The version primitives retain XeTeX.

829	_kernel_primitive:NN	\XeTeXrevision	\xetex_XeTeXrevision:D
830	_kernel_primitive:NN	\XeTeXversion	\xetex_XeTeXversion:D

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. Primitives which are aliases are covered only once, so for example `\pdfpageheight` covers `\pageheight` as well.

831	_kernel_primitive:NN	\alignmark	\luatex_alignmark:D
832	_kernel_primitive:NN	\aligntab	\luatex_aligntab:D
833	_kernel_primitive:NN	\attribute	\luatex_attribute:D
834	_kernel_primitive:NN	\attributedef	\luatex_attributedef:D
835	_kernel_primitive:NN	\catcodetable	\luatex_catcodetable:D
836	_kernel_primitive:NN	\clearmarks	\luatex_clearmarks:D
837	_kernel_primitive:NN	\crampeddisplaystyle	\luatex_crampeddisplaystyle:D

838	_kernel_primitive:NN	\crampedscriptscriptstyle	\luatex_crampedscriptscriptstyle:D
839	_kernel_primitive:NN	\crampedscriptstyle	\luatex_crampedscriptstyle:D
840	_kernel_primitive:NN	\crampedtextstyle	\luatex_crampedtextstyle:D
841	_kernel_primitive:NN	\directlua	\luatex_directlua:D
842	_kernel_primitive:NN	\expanded	\luatex_expanded:D
843	_kernel_primitive:NN	\fontid	\luatex_fontid:D
844	_kernel_primitive:NN	\formatname	\luatex_formatname:D
845	_kernel_primitive:NN	\gleaders	\luatex_gleaders:D
846	_kernel_primitive:NN	\initcatcodetable	\luatex_initcatcodetable:D
847	_kernel_primitive:NN	\latelua	\luatex_latelua:D
848	_kernel_primitive:NN	\luaescapestring	\luatex_luaescapestring:D
849	_kernel_primitive:NN	\luafunction	\luatex_luafunction:D
850	_kernel_primitive:NN	\luastartup	\luatex_luastartup:D
851	_kernel_primitive:NN	\luatexdatestamp	\luatex_luatexdatestamp:D
852	_kernel_primitive:NN	\luatexrevision	\luatex_luatexrevision:D
853	_kernel_primitive:NN	\luatexversion	\luatex_luatexversion:D
854	_kernel_primitive:NN	\mathstyle	\luatex_mathstyle:D
855	_kernel_primitive:NN	\nokerns	\luatex_nokerns:D
856	_kernel_primitive:NN	\noligs	\luatex_noligs:D
857	_kernel_primitive:NN	\outputbox	\luatex_outputbox:D
858	_kernel_primitive:NN	\pageleftoffset	\luatex_pageleftoffset:D
859	_kernel_primitive:NN	\pagetopoffset	\luatex_pagetopoffset:D
860	_kernel_primitive:NN	\postexhyphenchar	\luatex_postexhyphenchar:D
861	_kernel_primitive:NN	\posthyphenchar	\luatex_posthyphenchar:D
862	_kernel_primitive:NN	\preexhyphenchar	\luatex_preexhyphenchar:D
863	_kernel_primitive:NN	\prehyphenchar	\luatex_prehyphenchar:D
864	_kernel_primitive:NN	\savecatcodetable	\luatex_savecatcodetable:D
865	_kernel_primitive:NN	\scantextokens	\luatex_scantextokens:D
866	_kernel_primitive:NN	\suppressifcsnameerror	\luatex_suppressifcsnameerror:D
867	_kernel_primitive:NN	\suppresslongerror	\luatex_suppresslongerror:D
868	_kernel_primitive:NN	\suppressmathparerror	\luatex_suppressmathparerror:D
869	_kernel_primitive:NN	\suppressoutererror	\luatex_suppressoutererror: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.

870	_kernel_primitive:NN	\bodydir	\luatex_bodydir:D
871	_kernel_primitive:NN	\boxdir	\luatex_boxdir:D
872	_kernel_primitive:NN	\leftghost	\luatex_leftghost:D
873	_kernel_primitive:NN	\localbrokenpenalty	\luatex_localbrokenpenalty:D
874	_kernel_primitive:NN	\localinterlinepenalty	\luatex_localinterlinepenalty:D
875	_kernel_primitive:NN	\lcalleftbox	\luatex_lcalleftbox:D
876	_kernel_primitive:NN	\lcalrightbox	\luatex_lcalrightbox:D
877	_kernel_primitive:NN	\mathdir	\luatex_mathdir:D
878	_kernel_primitive:NN	\pagebottomoffset	\luatex_pagebottomoffset:D
879	_kernel_primitive:NN	\pagedir	\luatex_pagedir:D
880	_kernel_primitive:NN	\pagerightoffset	\luatex_pagerightoffset:D
881	_kernel_primitive:NN	\pardir	\luatex_pardir:D
882	_kernel_primitive:NN	\rightghost	\luatex_rightghost:D
883	_kernel_primitive:NN	\textdir	\luatex_textdir:D

The set of Unicode math primitives were introduced by X_gTeX and LuaTeX in a somewhat complex fashion: a few first as \XeTeX... which were then renamed with LuaTeX having a lot more. These names now all start \U... and mainly \Umath.... 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 TeX90 primitive or where it really seems required we keep the math part of the name.

884	__kernel_primitive:NN \Uchar	\utex_char:D
885	__kernel_primitive:NN \Ucharcat	\utex_charcat:D
886	__kernel_primitive:NN \Udelcode	\utex_delcode:D
887	__kernel_primitive:NN \Udelcodenum	\utex_delcodenum:D
888	__kernel_primitive:NN \Udelimiter	\utex_delimiter:D
889	__kernel_primitive:NN \Udelimiterover	\utex_delimiterover:D
890	__kernel_primitive:NN \Udelimiterunder	\utex_delimiterunder:D
891	__kernel_primitive:NN \Umathaccent	\utex_mathaccent:D
892	__kernel_primitive:NN \Umathaxis	\utex_mathaxis:D
893	__kernel_primitive:NN \Umathbinbinspacing	\utex_binbinspacing:D
894	__kernel_primitive:NN \Umathbinclosespacing	\utex_binclosespacing:D
895	__kernel_primitive:NN \Umathbininnerspacing	\utex_bininnerspacing:D
896	__kernel_primitive:NN \Umathbinopenspacing	\utex_binopenspacing:D
897	__kernel_primitive:NN \Umathbinopspacing	\utex_binopspacing:D
898	__kernel_primitive:NN \Umathbinordspacing	\utex_binordspacing:D
899	__kernel_primitive:NN \Umathbinpunctspacing	\utex_binpunctspacing:D
900	__kernel_primitive:NN \Umathbinrelspacing	\utex_binrelspacing:D
901	__kernel_primitive:NN \Umathchar	\utex_mathchar:D
902	__kernel_primitive:NN \Umathchardef	\utex_mathchardef:D
903	__kernel_primitive:NN \Umathcharnum	\utex_mathcharnum:D
904	__kernel_primitive:NN \Umathcharnumdef	\utex_mathcharnumdef:D
905	__kernel_primitive:NN \Umathclosebinspacing	\utex_closebinspacing:D
906	__kernel_primitive:NN \Umathcloseclosespacing	\utex_closeclosespacing:D
907	__kernel_primitive:NN \Umathcloseinnerspacing	\utex_closeinnerspacing:D
908	__kernel_primitive:NN \Umathcloseopenspacing	\utex_closeopenspacing:D
909	__kernel_primitive:NN \Umathcloseopspacing	\utex_closeopspacing:D
910	__kernel_primitive:NN \Umathcloseordspacing	\utex_closeordspacing:D
911	__kernel_primitive:NN \Umathclosepunctspacing	\utex_closepunctspacing:D
912	__kernel_primitive:NN \Umathcloserelspacing	\utex_closerelspacing:D
913	__kernel_primitive:NN \Umathcode	\utex_mathcode:D
914	__kernel_primitive:NN \Umathcodenum	\utex_mathcodenum:D
915	__kernel_primitive:NN \Umathconnectoroverlapmin	\utex_connectoroverlapmin:D
916	__kernel_primitive:NN \Umathfractiondelsize	\utex_fractiondelsize:D
917	__kernel_primitive:NN \Umathfractiondenomdown	\utex_fractiondenomdown:D
918	__kernel_primitive:NN \Umathfractiondenomvgap	\utex_fractiondenomvgap:D
919	__kernel_primitive:NN \Umathfractionnumup	\utex_fractionnumup:D
920	__kernel_primitive:NN \Umathfractionnumvgap	\utex_fractionnumvgap:D
921	__kernel_primitive:NN \Umathfractionrule	\utex_fractionrule:D
922	__kernel_primitive:NN \Umathinnerbinspacing	\utex_innerbinspacing:D
923	__kernel_primitive:NN \Umathinnerclosespacing	\utex_innerclosespacing:D
924	__kernel_primitive:NN \Umathinnerinnerspacing	\utex_innerinnerspacing:D
925	__kernel_primitive:NN \Umathinneropenspacing	\utex_inneropenspacing:D
926	__kernel_primitive:NN \Umathinneropspacing	\utex_inneropspacing:D

927	_kernel_primitive:NN	\Umathinnerordspacing	\utex_innerordspacing:D
928	_kernel_primitive:NN	\Umathinnerpunctspacing	\utex_innerpunctspacing:D
929	_kernel_primitive:NN	\Umathinnerrelspacing	\utex_innerrelspacing:D
930	_kernel_primitive:NN	\Umathlimitabovebgap	\utex_limitabovebgap:D
931	_kernel_primitive:NN	\Umathlimitabovekern	\utex_limitabovekern:D
932	_kernel_primitive:NN	\Umathlimitabovevgap	\utex_limitabovevgap:D
933	_kernel_primitive:NN	\Umathlimitbelowbgap	\utex_limitbelowbgap:D
934	_kernel_primitive:NN	\Umathlimitbelowkern	\utex_limitbelowkern:D
935	_kernel_primitive:NN	\Umathlimitbelowvgap	\utex_limitbelowvgap:D
936	_kernel_primitive:NN	\Umathopbinspacing	\utex_opbinspacing:D
937	_kernel_primitive:NN	\Umathopclosespacing	\utex_opclosespacing:D
938	_kernel_primitive:NN	\Umathopenbinspacing	\utex_openbinspacing:D
939	_kernel_primitive:NN	\Umathopenclosespacing	\utex_openclosespacing:D
940	_kernel_primitive:NN	\Umathopeninnerspacing	\utex_openinnerspacing:D
941	_kernel_primitive:NN	\Umathopenopenspacing	\utex_openopenspacing:D
942	_kernel_primitive:NN	\Umathopenopspacing	\utex_openopspacing:D
943	_kernel_primitive:NN	\Umathopenordspacing	\utex_openordspacing:D
944	_kernel_primitive:NN	\Umathopenpunctspacing	\utex_openpunctspacing:D
945	_kernel_primitive:NN	\Umathopenrelspacing	\utex_openrelspacing:D
946	_kernel_primitive:NN	\Umathoperatorsize	\utex_operatorsize:D
947	_kernel_primitive:NN	\Umathopinnerspacing	\utex_opinnerspacing:D
948	_kernel_primitive:NN	\Umathopopenspacing	\utex_opopenspacing:D
949	_kernel_primitive:NN	\Umathopopspacing	\utex_opopspacing:D
950	_kernel_primitive:NN	\Umathopordspacing	\utex_opordspacing:D
951	_kernel_primitive:NN	\Umathoppunctspacing	\utex_oppunctspacing:D
952	_kernel_primitive:NN	\Umathoprelspacing	\utex_oprelspacing:D
953	_kernel_primitive:NN	\Umathordbinspacing	\utex_ordbinspacing:D
954	_kernel_primitive:NN	\Umathordclosespacing	\utex_ordclosespacing:D
955	_kernel_primitive:NN	\Umathordinnerspacing	\utex_ordinnerspacing:D
956	_kernel_primitive:NN	\Umathordopenspacing	\utex_ordopenspacing:D
957	_kernel_primitive:NN	\Umathordopspacing	\utex_ordopspacing:D
958	_kernel_primitive:NN	\Umathordordspacing	\utex_ordordspacing:D
959	_kernel_primitive:NN	\Umathordpunctspacing	\utex_ordpunctspacing:D
960	_kernel_primitive:NN	\Umathordrelspacing	\utex_ordrelspacing:D
961	_kernel_primitive:NN	\Umathoverbarkern	\utex_overbarkern:D
962	_kernel_primitive:NN	\Umathoverbarrule	\utex_overbarrule:D
963	_kernel_primitive:NN	\Umathoverbarvgap	\utex_overbarvgap:D
964	_kernel_primitive:NN	\Umathoverdelimitervgap	\utex_overdelimitervgap:D
965	_kernel_primitive:NN	\Umathoverdelimitervgap	\utex_overdelimitervgap:D
966	_kernel_primitive:NN	\Umathpunctbinspacing	\utex_punctbinspacing:D
967	_kernel_primitive:NN	\Umathpunctclosespacing	\utex_punctclosespacing:D
968	_kernel_primitive:NN	\Umathpunctinnerspacing	\utex_punctinnerspacing:D
969	_kernel_primitive:NN	\Umathpunctopenspacing	\utex_punctopenspacing:D
970	_kernel_primitive:NN	\Umathpunctopspacing	\utex_punctopspacing:D
971	_kernel_primitive:NN	\Umathpunctordspacing	\utex_punctordspacing:D
972	_kernel_primitive:NN	\Umathpunctpunctspacing	\utex_punctpunctspacing:D
973	_kernel_primitive:NN	\Umathpunctrelspacing	\utex_punctrelspacing:D
974	_kernel_primitive:NN	\Umathquad	\utex_quad:D
975	_kernel_primitive:NN	\Umathradicaldegreeafter	\utex_radicaldegreeafter:D
976	_kernel_primitive:NN	\Umathradicaldegreebefore	\utex_radicaldegreebefore:D

977	_kernel_primitive:NN	\Umathradicaldegreeraise	\utex_radicaldegreeraise:D
978	_kernel_primitive:NN	\Umathradicalkern	\utex_radicalkern:D
979	_kernel_primitive:NN	\Umathradicalrule	\utex_radicalrule:D
980	_kernel_primitive:NN	\Umathradicalvgap	\utex_radicalvgap:D
981	_kernel_primitive:NN	\Umathrelbinspacing	\utex_relbinspacing:D
982	_kernel_primitive:NN	\Umathrelclosespacing	\utex_relclosespacing:D
983	_kernel_primitive:NN	\Umathrelinnerspacing	\utex_relinnerspacing:D
984	_kernel_primitive:NN	\Umathrelopenspacing	\utex_relopenspacing:D
985	_kernel_primitive:NN	\Umathrellopspacing	\utex_rellopspacing:D
986	_kernel_primitive:NN	\Umathrelordspacing	\utex_relordspacing:D
987	_kernel_primitive:NN	\Umathrelpunctspacing	\utex_relpunctspacing:D
988	_kernel_primitive:NN	\Umathrelrelspacing	\utex_relrelspacing:D
989	_kernel_primitive:NN	\Umathspaceafterscript	\utex_spaceafterscript:D
990	_kernel_primitive:NN	\Umathstackdenomdown	\utex_stackdenomdown:D
991	_kernel_primitive:NN	\Umathstacknumup	\utex_stacknumup:D
992	_kernel_primitive:NN	\Umathstackvgap	\utex_stackvgap:D
993	_kernel_primitive:NN	\Umathsubshiftdown	\utex_subshiftdown:D
994	_kernel_primitive:NN	\Umathsubshiftdrop	\utex_subshiftdrop:D
995	_kernel_primitive:NN	\Umathsubsupshiftdown	\utex_subsupshiftdown:D
996	_kernel_primitive:NN	\Umathsubsupvgap	\utex_subsupvgap:D
997	_kernel_primitive:NN	\Umathsubtopmax	\utex_subtopmax:D
998	_kernel_primitive:NN	\Umathsupbottommin	\utex_supbottommin:D
999	_kernel_primitive:NN	\Umathsupshiftdrop	\utex_supshiftdrop:D
1000	_kernel_primitive:NN	\Umathsupshiftup	\utex_supshiftup:D
1001	_kernel_primitive:NN	\Umathsupsubbottommax	\utex_supsubbottommax:D
1002	_kernel_primitive:NN	\Umathunderbarkern	\utex_underbarkern:D
1003	_kernel_primitive:NN	\Umathunderbarrule	\utex_underbarrule:D
1004	_kernel_primitive:NN	\Umathunderbarvgap	\utex_underbarvgap:D
1005	_kernel_primitive:NN	\Umathunderdelimiterbgap	\utex_underdelimiterbgap:D
1006	_kernel_primitive:NN	\Umathunderdelimitervgap	\utex_underdelimitervgap:D
1007	_kernel_primitive:NN	\Uoverdelimiter	\utex_overdelimiter:D
1008	_kernel_primitive:NN	\Uradical	\utex_radical:D
1009	_kernel_primitive:NN	\Uroot	\utex_root:D
1010	_kernel_primitive:NN	\Ustack	\utex_stack:D
1011	_kernel_primitive:NN	\Ustartdisplaymath	\utex_startdisplaymath:D
1012	_kernel_primitive:NN	\Ustartmath	\utex_startmath:D
1013	_kernel_primitive:NN	\Ustopdisplaymath	\utex_stopdisplaymath:D
1014	_kernel_primitive:NN	\Ustopmath	\utex_stopmath:D
1015	_kernel_primitive:NN	\Usubscript	\utex_subscript:D
1016	_kernel_primitive:NN	\Usuperscript	\utex_superscript:D
1017	_kernel_primitive:NN	\Uunderdelimiter	\utex_underdelimiter:D

Primitives from p_T_EX.

1018	_kernel_primitive:NN	\autospacing	\ptex_autospacing:D
1019	_kernel_primitive:NN	\autoxspacing	\ptex_autoxspacing:D
1020	_kernel_primitive:NN	\dtou	\ptex_dtou:D
1021	_kernel_primitive:NN	\euc	\ptex_euc:D
1022	_kernel_primitive:NN	\ifdbbox	\ptex_ifdbbox:D
1023	_kernel_primitive:NN	\ifddir	\ptex_ifddir:D
1024	_kernel_primitive:NN	\ifmdir	\ptex_ifmdir:D

```

1025 \__kernel_primitive:NN \iftbox \ptex_iftbox:D
1026 \__kernel_primitive:NN \iftdir \ptex_iftdir:D
1027 \__kernel_primitive:NN \ifybox \ptex_ifybox:D
1028 \__kernel_primitive:NN \ifydir \ptex_ifydir:D
1029 \__kernel_primitive:NN \inhibitglue \ptex_inhibitglue:D
1030 \__kernel_primitive:NN \inhibitxspcode \ptex_inhibitxspcode:D
1031 \__kernel_primitive:NN \jcharwidowpenalty \ptex_jcharwidowpenalty:D
1032 \__kernel_primitive:NN \jfam \ptex_jfam:D
1033 \__kernel_primitive:NN \jfont \ptex_jfont:D
1034 \__kernel_primitive:NN \jis \ptex_jis:D
1035 \__kernel_primitive:NN \kanjiskip \ptex_kanjiskip:D
1036 \__kernel_primitive:NN \kansuji \ptex_kansuji:D
1037 \__kernel_primitive:NN \kansujichar \ptex_kansujichar:D
1038 \__kernel_primitive:NN \kcatcode \ptex_kcatcode:D
1039 \__kernel_primitive:NN \kuten \ptex_kuten:D
1040 \__kernel_primitive:NN \noautospadding \ptex_noautospadding:D
1041 \__kernel_primitive:NN \noautoxspacing \ptex_noautoxspacing:D
1042 \__kernel_primitive:NN \postbreakpenalty \ptex_postbreakpenalty:D
1043 \__kernel_primitive:NN \prebreakpenalty \ptex_prebreakpenalty:D
1044 \__kernel_primitive:NN \showmode \ptex_showmode:D
1045 \__kernel_primitive:NN \sjis \ptex_sjis:D
1046 \__kernel_primitive:NN \tate \ptex_tate:D
1047 \__kernel_primitive:NN \tbaselineshift \ptex_tbaselineshift:D
1048 \__kernel_primitive:NN \tfont \ptex_tfont:D
1049 \__kernel_primitive:NN \xkanjiskip \ptex_xkanjiskip:D
1050 \__kernel_primitive:NN \xspcode \ptex_xspcode:D
1051 \__kernel_primitive:NN \ybaselineshift \ptex_ybaselineshift:D
1052 \__kernel_primitive:NN \yoko \ptex_yoko:D

```

Primitives from upTeX.

```

1053 \__kernel_primitive:NN \disablecjktoken \uptex_disablecjktoken:D
1054 \__kernel_primitive:NN \enablecjktoken \uptex_enablecjktoken:D
1055 \__kernel_primitive:NN \forcecjktoken \uptex_forcecjktoken:D
1056 \__kernel_primitive:NN \kchar \uptex_kchar:D
1057 \__kernel_primitive:NN \kchardef \uptex_kchardef:D
1058 \__kernel_primitive:NN \kuten \uptex_kuten:D
1059 \__kernel_primitive:NN \ucs \uptex_ucs:D

```

End of the “just the names” part of the source.

```

1060 </initex | names | package>
1061 <*initex | package>

```

The job is done: close the group (using the primitive renamed!).

```

1062 \tex_endgroup:D

```

L^AT_EX 2_ε will have moved a few primitives, so these are sorted out. A convenient test for L^AT_EX 2_ε is the \@@end saved primitive.

```

1063 <*package>
1064 \etex_ifdefined:D \@@end
1065 \tex_let:D \tex_end:D \@@end
1066 \tex_let:D \tex_everydisplay:D \frozen@everydisplay

```

```

1067 \tex_let:D \tex_everymath:D          \frozen@everymath
1068 \tex_let:D \tex_hyphen:D             \@hyph
1069 \tex_let:D \tex_input:D              @@input
1070 \tex_let:D \tex_italiccorrection:D   @@italiccorr
1071 \tex_let:D \tex_underline:D          @@underline
1072 \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!

```

1073 \etex_ifdefined:D \luatexcatcodetable
1074 \tex_let:D \luatex_alignmark:D          \luatexalignmark
1075 \tex_let:D \luatex_aligntab:D           \luatexaligntab
1076 \tex_let:D \luatex_attribute:D          \luatexattribute
1077 \tex_let:D \luatex_attributedef:D       \luatexattributedef
1078 \tex_let:D \luatex_catcodetable:D       \luatexcatcodetable
1079 \tex_let:D \luatex_clearmarks:D         \luatexclearmarks
1080 \tex_let:D \luatex_crampeddisplaystyle:D \luatexcrampeddisplaystyle
1081 \tex_let:D \luatex_crampedscriptscriptstyle:D \luatexcrampedscriptscriptstyle
1082 \tex_let:D \luatex_crampedscriptstyle:D \luatexcrampedscriptstyle
1083 \tex_let:D \luatex_crampedtextstyle:D   \luatexcrampedtextstyle
1084 \tex_let:D \luatex_fontid:D              \luatexfontid
1085 \tex_let:D \luatex_formatname:D         \luatexformatname
1086 \tex_let:D \luatex_gleaders:D           \luatexgleaders
1087 \tex_let:D \luatex_initcatcodetable:D   \luatexinitcatcodetable
1088 \tex_let:D \luatex_latelua:D            \luatexlatelua
1089 \tex_let:D \luatex_luaescapestring:D    \luatexluaescapestring
1090 \tex_let:D \luatex_luafunction:D        \luatexluafunction
1091 \tex_let:D \luatex_luastartup:D         \luatexluastartup
1092 \tex_let:D \luatex_mathstyle:D          \luatexmathstyle
1093 \tex_let:D \luatex_nokerns:D            \luatexnokerns
1094 \tex_let:D \luatex_noligs:D             \luatexnoligs
1095 \tex_let:D \luatex_outputbox:D          \luatexoutputbox
1096 \tex_let:D \luatex_pageleftoffset:D     \luatexpageleftoffset
1097 \tex_let:D \luatex_pagetopoffset:D      \luatexpagetopoffset
1098 \tex_let:D \luatex_postexhyphenchar:D   \luatexpostexhyphenchar
1099 \tex_let:D \luatex_postthyphenchar:D    \luatexpostthyphenchar
1100 \tex_let:D \luatex_preexhyphenchar:D    \luatexpreexhyphenchar
1101 \tex_let:D \luatex_prehyphenchar:D      \luatexprehyphenchar
1102 \tex_let:D \luatex_savecatcodetable:D    \luatexsavecatcodetable
1103 \tex_let:D \luatex_scantextokens:D       \luatexscantextokens
1104 \tex_let:D \luatex_suppressifcsnameerror:D \luatexsuppressifcsnameerror
1105 \tex_let:D \luatex_suppresslongerror:D   \luatexsuppresslongerror
1106 \tex_let:D \luatex_suppressmathparerror:D \luatexsuppressmathparerror
1107 \tex_let:D \luatex_suppressoutererror:D  \luatexsuppressoutererror
1108 \tex_let:D \utex_char:D                  \luatexUchar
1109 \tex_let:D \xetex_suppressfontnotfounderror:D \luatexsuppressfontnotfounderror

```

Which also covers those slightly odd ones.

```

1110 \tex_let:D \luatex_bodydir:D            \luatexbodydir

```

```

1111 \tex_let:D \luatex_boxdir:D \luatexboxdir
1112 \tex_let:D \luatex_chardp:D \luatexchardp
1113 \tex_let:D \luatex_charht:D \luatexcharht
1114 \tex_let:D \luatex_charit:D \luatexcharit
1115 \tex_let:D \luatex_charwd:D \luatexcharwd
1116 \tex_let:D \luatex_leftghost:D \luatexleftghost
1117 \tex_let:D \luatex_localbrokenpenalty:D \luatexlocalbrokenpenalty
1118 \tex_let:D \luatex_localinterlinepenalty:D \luatexlocalinterlinepenalty
1119 \tex_let:D \luatex_localleftbox:D \luatexlocalleftbox
1120 \tex_let:D \luatex_localrightbox:D \luatexlocalrightbox
1121 \tex_let:D \luatex_mathdir:D \luatexmathdir
1122 \tex_let:D \luatex_pagebottomoffset:D \luatexpagebottomoffset
1123 \tex_let:D \luatex_pagedir:D \luatexpagedir
1124 \tex_let:D \luatex_pageheight:D \luatexpageheight
1125 \tex_let:D \luatex_pagerightoffset:D \luatexpagerightoffset
1126 \tex_let:D \luatex_pagewidth:D \luatexpagewidth
1127 \tex_let:D \luatex_pardir:D \luatexpardir
1128 \tex_let:D \luatex_rightghost:D \luatexrightghost
1129 \tex_let:D \luatex_textdir:D \luatextextdir
1130 \tex_fi:D

```

Tidy up some X_ƎT_ƎX renames and the fact that some format-building processes leave a couple of questionable decisions about. (There may at some stage be LuaT_ƎX renames of the same sort to address.)

```

1131 \etex_unless:D \etex_ifdefined:D \pdfTeX_ifprimitive:D
1132 \tex_expandafter:D \tex_let:D
1133 \tex_csname:D pdfTeX_ifprimitive:D \tex_expandafter:D \tex_endcsname:D
1134 \tex_csname:D ifprimitive \tex_endcsname:D
1135 \tex_let:D pdfTeX_primitive:D \primitive
1136 \tex_let:D pdfTeX_shellescape:D \shellescape
1137 \tex_fi:D

```

Only pdfT_ƎX and LuaT_ƎX define \pdfmapfile and \pdfmapline.

```

1138 \tex_ifnum:D 0
1139 \etex_ifdefined:D \pdfTeX_pdfTeXversion:D 1 \tex_fi:D
1140 \etex_ifdefined:D \luatex luatexversion:D 1 \tex_fi:D
1141 = 0 %
1142 \tex_let:D pdfTeX_mapfile:D \tex_undefined:D
1143 \tex_let:D pdfTeX_mapline:D \tex_undefined:D
1144 \tex_fi:D

```

Older X_ƎT_ƎX versions use \XeT_ƎX as the prefix for the Unicode math primitives it knows. That is tidied up here (we support X_ƎT_ƎX versions from 0.9994 but this change was in 0.9999).

```

1145 \etex_ifdefined:D \XeTeXdelcode
1146 \tex_let:D \utex_delcode:D \XeTeXdelcode
1147 \tex_let:D \utex_delcodenum:D \XeTeXdelcodenum
1148 \tex_let:D \utex_delimiter:D \XeTeXdelimiter
1149 \tex_let:D \utex_mathaccent:D \XeTeXmathaccent
1150 \tex_let:D \utex_mathchar:D \XeTeXmathchar
1151 \tex_let:D \utex_mathchardef:D \XeTeXmathchardef

```

```

1152 \tex_let:D \utex_mathcharnum:D \XeTeXmathcharnum
1153 \tex_let:D \utex_mathcharnumdef:D \XeTeXmathcharnumdef
1154 \tex_let:D \utex_mathcode:D \XeTeXmathcode
1155 \tex_let:D \utex_mathcodenum:D \XeTeXmathcodenum
1156 \tex_fi:D

```

Up to v0.80, LuaT_{EX} defines the pdfT_{EX} version data: rather confusing.

```

1157 \etex_ifdefined:D \luatex luatexversion:D
1158 \tex_let:D \pdfTEXpdfTEXbanner:D \tex_undefined:D
1159 \tex_let:D \pdfTEXpdfTEXbanner:D \tex_undefined:D
1160 \tex_let:D \pdfTEXpdfTEXversion:D \tex_undefined:D
1161 \tex_fi:D

```

For ConT_{EX}t, 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 ConT_{EX}t.

```

1162 \etex_ifdefined:D \normalend
1163 \tex_let:D \tex_end:D \normalend
1164 \tex_let:D \tex_everyjob:D \normaleveryjob
1165 \tex_let:D \tex_input:D \normalinput
1166 \tex_let:D \tex_language:D \normallanguage
1167 \tex_let:D \tex_mathop:D \normalmathop
1168 \tex_let:D \tex_month:D \normalmonth
1169 \tex_let:D \tex_outer:D \normalouter
1170 \tex_let:D \tex_over:D \normalover
1171 \tex_let:D \tex_vcenter:D \normalvcenter
1172 \tex_let:D \etex_unexpanded:D \normalunexpanded
1173 \tex_let:D \luatex_expanded:D \normalexpanded
1174 \tex_fi:D
1175 \etex_ifdefined:D \normalitaliccorrection
1176 \tex_let:D \tex_hoffset:D \normalhoffset
1177 \tex_let:D \tex_italiccorrection:D \normalitaliccorrection
1178 \tex_let:D \tex_voffset:D \normalvoffset
1179 \tex_let:D \etex_showtokens:D \normalshowtokens
1180 \tex_let:D \luatex_bodydir:D \spac_directions_normal_body_dir
1181 \tex_let:D \luatex_pagedir:D \spac_directions_normal_page_dir
1182 \tex_fi:D
1183 \etex_ifdefined:D \normalleft
1184 \tex_let:D \tex_left:D \normalleft
1185 \tex_let:D \tex_middle:D \normalmiddle
1186 \tex_let:D \tex_right:D \normalright
1187 \tex_fi:D
1188 </package>
1189 </initex | package>

```

3 l3basics implementation

```

1190 <*initex | package>

```

3.1 Renaming some TeX primitives (again)

Having given all the TeX 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.⁴

```

\if_true: Then some conditionals.
\if_false: 1191 \tex_let:D \if_true:          \tex_iftrue:D
\or:       1192 \tex_let:D \if_false:      \tex_iffalse:D
\else:     1193 \tex_let:D \or:            \tex_or:D
\fi:       1194 \tex_let:D \else:          \tex_else:D
\reverse_if:N 1195 \tex_let:D \fi:          \tex_fi:D
\if:w      1196 \tex_let:D \reverse_if:N    \etex_unless:D
\if_charcode:w 1197 \tex_let:D \if:w        \tex_if:D
\if_catcode:w 1198 \tex_let:D \if_charcode:w \tex_if:D
\if_meaning:w 1199 \tex_let:D \if_catcode:w \tex_ifcat:D
            1200 \tex_let:D \if_meaning:w      \tex_ifx:D

```

(End definition for \if_true: and others. These functions are documented on page 23.)

```

\if_mode_math: TeX lets us detect some if its modes.
\if_mode_horizontal: 1201 \tex_let:D \if_mode_math:      \tex_ifmmode:D
\if_mode_vertical:   1202 \tex_let:D \if_mode_horizontal: \tex_ifhmode:D
\if_mode_inner:      1203 \tex_let:D \if_mode_vertical:   \tex_ifvmode:D
                    1204 \tex_let:D \if_mode_inner:      \tex_ifinner:D

```

(End definition for \if_mode_math: and others. These functions are documented on page 24.)

```

\if_cs_exist:N Building csnames and testing if control sequences exist.
\if_cs_exist:w 1205 \tex_let:D \if_cs_exist:N      \etex_ifdefined:D
\cs:w         1206 \tex_let:D \if_cs_exist:w      \etex_ifcurname:D
\cs_end:      1207 \tex_let:D \cs:w              \tex_csname:D
            1208 \tex_let:D \cs_end:                \tex_endcurname:D

```

(End definition for \if_cs_exist:N and others. These functions are documented on page 24.)

```

\exp_after:wN The five \exp_ functions are used in the l3expan module where they are described.
\exp_not:N    1209 \tex_let:D \exp_after:wN      \tex_expandafter:D
\exp_not:n    1210 \tex_let:D \exp_not:N          \tex_noexpand:D
            1211 \tex_let:D \exp_not:n          \etex_unexpanded:D
            1212 \tex_let:D \exp:w            \tex_romannumeral:D
            1213 \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 32.)

```

\token_to_meaning:N Examining a control sequence or token.
\cs_meaning:N       1214 \tex_let:D \token_to_meaning:N \tex_meaning:D
                    1215 \tex_let:D \cs_meaning:N   \tex_meaning:D

```

⁴This renaming gets expensive in terms of csname usage, an alternative scheme would be to just use the \tex...:D name in the cases where no good alternative exists.

(End definition for `\token_to_meaning:N` and `\cs_meaning:N`. These functions are documented on page 56.)

`\tl_to_str:n` Making strings.

`\token_to_str:N`

```

1216 \tex_let:D \tl_to_str:n      \etex_detokenize:D
1217 \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 102.)

`\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:`

```

1218 \tex_let:D \scan_stop:      \tex_relax:D
1219 \tex_let:D \group_begin:    \tex_begingroup:D
1220 \tex_let:D \group_end:      \tex_endgroup:D

```

(End definition for `\scan_stop:`, `\group_begin:`, and `\group_end:`. These functions are documented on page 10.)

`\if_int_compare:w` For integers.

`__int_to_roman:w`

```

1221 \tex_let:D \if_int_compare:w \tex_ifnum:D
1222 \tex_let:D \__int_to_roman:w  \tex_romannumeral:D

```

(End definition for `\if_int_compare:w` and `__int_to_roman:w`. These functions are documented on page 77.)

`\group_insert_after:N` Adding material after the end of a group.

```

1223 \tex_let:D \group_insert_after:N \tex_aftergroup:D

```

(End definition for `\group_insert_after:N`. This function is documented on page 10.)

`\exp_args:Nc` Discussed in `l3expan`, but needed much earlier.

`\exp_args:cc`

```

1224 \tex_long:D \tex_def:D \exp_args:Nc #1#2
1225 { \exp_after:wN #1 \cs:w #2 \cs_end: }
1226 \tex_long:D \tex_def:D \exp_args:cc #1#2
1227 { \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 29.)

`\token_to_meaning:c` A small number of variants defined by hand. Some of the necessary functions (`\use_i:nn`, `\use_ii:nn`, and `\exp_args:NNc`) are not defined at that point yet, but will be defined before those variants are used. The `\cs_meaning:c` command must check for an undefined control sequence to avoid defining it mistakenly.

```

1228 \tex_def:D \token_to_str:c { \exp_args:Nc \token_to_str:N }
1229 \tex_long:D \tex_def:D \cs_meaning:c #1
1230 {
1231   \if_cs_exist:w #1 \cs_end:
1232   \exp_after:wN \use_i:nn
1233   \else:
1234   \exp_after:wN \use_ii:nn
1235   \fi:
1236   { \exp_args:Nc \cs_meaning:N {#1} }

```

```

1237     { \tl_to_str:n {undefined} }
1238   }
1239   \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 ??.)

3.2 Defining some constants

`\c_minus_one` `\c_zero` `\c_sixteen` We need the constants `\c_minus_one` and `\c_sixteen` now for writing information to the log and the terminal and `\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! The actual allocation mechanism is in `l3alloc`, and works such that the first available count register is 10.

```

1240 <*package>
1241 \tex_let:D \c_minus_one \m@ne
1242 </package>
1243 <*initex>
1244 \tex_countdef:D \c_minus_one = 10 ~
1245 \c_minus_one = -1 ~
1246 </initex>
1247 \tex_chardef:D \c_sixteen = 16 ~
1248 \tex_chardef:D \c_zero = 0 ~

```

(End definition for `\c_minus_one`, `\c_zero`, and `\c_sixteen`. These variables are documented on page 76.)

`\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`.

```

1249 \etex_ifdefined:D \luatex luatexversion:D
1250 \tex_chardef:D \c_max_register_int = 65 535 ~
1251 \tex_else:D
1252 \tex_mathchardef:D \c_max_register_int = 32 767 ~
1253 \tex_fi:D

```

(End definition for `\c_max_register_int`. This variable is documented on page 76.)

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.

```

1254 \tex_let:D \cs_set_nopar:Npn \tex_def:D
1255 \tex_let:D \cs_set_nopar:Npx \tex_edef:D
1256 \etex_protected:D \cs_set_nopar:Npn \cs_set:Npn
1257 { \tex_long:D \cs_set_nopar:Npn }

```

```

1258 \etex_protected:D \cs_set_nopar:Npn \cs_set:Npx
1259 { \tex_long:D \cs_set_nopar:Npx }
1260 \etex_protected:D \cs_set_nopar:Npn \cs_set_protected_nopar:Npn
1261 { \etex_protected:D \cs_set_nopar:Npn }
1262 \etex_protected:D \cs_set_nopar:Npn \cs_set_protected_nopar:Npx
1263 { \etex_protected:D \cs_set_nopar:Npx }
1264 \cs_set_protected_nopar:Npn \cs_set_protected:Npn
1265 { \etex_protected:D \tex_long:D \cs_set_nopar:Npn }
1266 \cs_set_protected_nopar:Npn \cs_set_protected:Npx
1267 { \etex_protected:D \tex_long:D \cs_set_nopar:Npx }

```

(End definition for `\cs_set_nopar:Npn` and others. These functions are documented on page 13.)

```

\cs_gset_nopar:Npn Global versions of the above functions.
\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
1268 \tex_let:D \cs_gset_nopar:Npn \tex_gdef:D
1269 \tex_let:D \cs_gset_nopar:Npx \tex_xdef:D
1270 \cs_set_protected_nopar:Npn \cs_gset:Npn
1271 { \tex_long:D \cs_gset_nopar:Npn }
1272 \cs_set_protected_nopar:Npn \cs_gset:Npx
1273 { \tex_long:D \cs_gset_nopar:Npx }
1274 \cs_set_protected_nopar:Npn \cs_gset_protected_nopar:Npn
1275 { \etex_protected:D \cs_gset_nopar:Npn }
1276 \cs_set_protected_nopar:Npn \cs_gset_protected_nopar:Npx
1277 { \etex_protected:D \cs_gset_nopar:Npx }
1278 \cs_set_protected_nopar:Npn \cs_gset_protected:Npn
1279 { \etex_protected:D \tex_long:D \cs_gset_nopar:Npn }
1280 \cs_set_protected_nopar:Npn \cs_gset_protected:Npx
1281 { \etex_protected:D \tex_long:D \cs_gset_nopar:Npx }

```

(End definition for `\cs_gset_nopar:Npn` and others. These functions are documented on page 13.)

3.4 Selecting tokens

`\l__exp_internal_tl` Scratch token list variable for `\l3expan`, used by `\use:x`, used in defining conditionals. We don't use `tl` methods because `\l3basics` is loaded earlier.

```

1282 \cs_set_nopar:Npn \l__exp_internal_tl { }

```

(End definition for `\l__exp_internal_tl`. This variable is documented on page 35.)

`\use:c` This macro grabs its argument and returns a csname from it.

```

1283 \cs_set:Npn \use:c #1 { \cs:w #1 \cs_end: }

```

(End definition for `\use:c`. This function is documented on page 18.)

`\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 `\l3expan`.

```

1284 \cs_set_protected:Npn \use:x #1
1285 {
1286   \cs_set_nopar:Npx \l__exp_internal_tl {#1}
1287   \l__exp_internal_tl
1288 }

```

(End definition for `\use:x`. This function is documented on page 21.)

`\use:n` These macros grab their arguments and returns them back to the input (with outer braces removed).
`\use:nn`
`\use:nnn` 1289 `\cs_set:Npn \use:n #1 {#1}`
`\use:nnnn` 1290 `\cs_set:Npn \use:nn #1#2 {#1#2}`
1291 `\cs_set:Npn \use:nnn #1#2#3 {#1#2#3}`
1292 `\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 19.)

`\use_i:nn` The equivalent to L^AT_EX 2_ε's `\@firstoftwo` and `\@secondoftwo`.
`\use_ii:nn` 1293 `\cs_set:Npn \use_i:nn #1#2 {#1}`
1294 `\cs_set:Npn \use_ii:nn #1#2 {#2}`

(End definition for `\use_i:nn` and `\use_ii:nn`. These functions are documented on page 20.)

`\use_i:nnn` We also need something for picking up arguments from a longer list.
`\use_ii:nnn` 1295 `\cs_set:Npn \use_i:nnn #1#2#3 {#1}`
`\use_iii:nnn` 1296 `\cs_set:Npn \use_ii:nnn #1#2#3 {#2}`
`\use_i_ii:nnn` 1297 `\cs_set:Npn \use_iii:nnn #1#2#3 {#3}`
`\use_i:nnnn` 1298 `\cs_set:Npn \use_i_ii:nnn #1#2#3 {#1#2}`
`\use_ii:nnnn` 1299 `\cs_set:Npn \use_i:nnnn #1#2#3#4 {#1}`
`\use_iii:nnnn` 1300 `\cs_set:Npn \use_ii:nnnn #1#2#3#4 {#2}`
`\use_iv:nnnn` 1301 `\cs_set:Npn \use_iii:nnnn #1#2#3#4 {#3}`
1302 `\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 20.)

`\use_none_delimit_by_q_nil:w` Functions that gobble everything until they see either `\q_nil`, `\q_stop`, or `\q_`
`\use_none_delimit_by_q_stop:w` recursion_stop, respectively.
`\use_none_delimit_by_q_recursion_stop:w` 1303 `\cs_set:Npn \use_none_delimit_by_q_nil:w #1 \q_nil { }`
1304 `\cs_set:Npn \use_none_delimit_by_q_stop:w #1 \q_stop { }`
1305 `\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 21.)

`\use_i_delimit_by_q_nil:nw` Same as above but execute first argument after gobbling. Very useful when you need to
`\use_i_delimit_by_q_stop:nw` skip the rest of a mapping sequence but want an easy way to control what should be
`\use_i_delimit_by_q_recursion_stop:nw` expanded next.
1306 `\cs_set:Npn \use_i_delimit_by_q_nil:nw #1#2 \q_nil {#1}`
1307 `\cs_set:Npn \use_i_delimit_by_q_stop:nw #1#2 \q_stop {#1}`
1308 `\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 21.)

3.5 Gobbling tokens from input

`\use_none:n`
`\use_none:nn`
`\use_none:nnn`
`\use_none:nnnn`
`\use_none:nnnnn`
`\use_none:nnnnnn`
`\use_none:nnnnnnn`
`\use_none:nnnnnnnn`
`\use_none:nnnnnnnnn`

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.

1309	<code>\cs_set:Npn \use_none:n</code>	<code>#1</code>	<code>{ }</code>
1310	<code>\cs_set:Npn \use_none:nn</code>	<code>#1#2</code>	<code>{ }</code>
1311	<code>\cs_set:Npn \use_none:nnn</code>	<code>#1#2#3</code>	<code>{ }</code>
1312	<code>\cs_set:Npn \use_none:nnnn</code>	<code>#1#2#3#4</code>	<code>{ }</code>
1313	<code>\cs_set:Npn \use_none:nnnnn</code>	<code>#1#2#3#4#5</code>	<code>{ }</code>
1314	<code>\cs_set:Npn \use_none:nnnnnn</code>	<code>#1#2#3#4#5#6</code>	<code>{ }</code>
1315	<code>\cs_set:Npn \use_none:nnnnnnn</code>	<code>#1#2#3#4#5#6#7</code>	<code>{ }</code>
1316	<code>\cs_set:Npn \use_none:nnnnnnnn</code>	<code>#1#2#3#4#5#6#7#8</code>	<code>{ }</code>
1317	<code>\cs_set:Npn \use_none:nnnnnnnnn</code>	<code>#1#2#3#4#5#6#7#8#9</code>	<code>{ }</code>

(End definition for `\use_none:n` and others. These functions are documented on page 21.)

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:`
`\prg_return_false:`

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.

```

1318 \cs_set_nopar:Npn \prg_return_true:
1319 { \exp_after:wN \use_i:nn \exp:w }
1320 \cs_set_nopar:Npn \prg_return_false:
1321 { \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 39.)

```
\prg_set_conditional:Npnn
\prg_new_conditional:Npnn
\prg_set_protected_conditional:Npnn
\prg_new_protected_conditional:Npnn
\__prg_generate_conditional_parm:nnNpnn
```

The user functions for the types using parameter text from the programmer. The various functions only differ by which function is used for the assignment. For those `Npnn` type functions, we must grab the parameter text, reading everything up to a left brace before continuing. Then split the base function into name and signature, and feed `{<name>}{<signature>}{<boolean>}{<set or new>}{<maybe protected>}{<parameters>}{TF,...}{<code>}` to the auxiliary function responsible for defining all conditionals.

```
1322 \cs_set_protected_nopar:Npn \prg_set_conditional:Npnn
1323 { \__prg_generate_conditional_parm:nnNpnn { set } { } }
1324 \cs_set_protected_nopar:Npn \prg_new_conditional:Npnn
1325 { \__prg_generate_conditional_parm:nnNpnn { new } { } }
1326 \cs_set_protected_nopar:Npn \prg_set_protected_conditional:Npnn
1327 { \__prg_generate_conditional_parm:nnNpnn { set } { _protected } }
1328 \cs_set_protected_nopar:Npn \prg_new_protected_conditional:Npnn
1329 { \__prg_generate_conditional_parm:nnNpnn { new } { _protected } }
1330 \cs_set_protected:Npn \__prg_generate_conditional_parm:nnNpnn #1#2#3#4#
1331 {
1332   \__cs_split_function:NN #3 \__prg_generate_conditional:nnNnnnnn
1333   {#1} {#2} {#4}
1334 }
```

(End definition for `\prg_set_conditional:Npnn` and others. These functions are documented on page 37.)

```
\prg_set_conditional:Nnn
\prg_new_conditional:Nnn
\prg_set_protected_conditional:Nnn
\prg_new_protected_conditional:Nnn
\__prg_generate_conditional_count:nnNnn
\__prg_generate_conditional_count:nnNnnnn
```

The user functions for the types automatically inserting the correct parameter text based on the signature. The various functions only differ by which function is used for the assignment. Split the base function into name and signature. The second auxiliary generates the parameter text from the number of letters in the signature. Then feed `{<name>}{<signature>}{<boolean>}{<set or new>}{<maybe protected>}{<parameters>}{TF,...}{<code>}` to the auxiliary function responsible for defining all conditionals. If the `<signature>` has more than 9 letters, the definition is aborted since T_EX macros have at most 9 arguments. The erroneous case where the function name contains no colon is captured later.

```
1335 \cs_set_protected_nopar:Npn \prg_set_conditional:Nnn
1336 { \__prg_generate_conditional_count:nnNnn { set } { } }
1337 \cs_set_protected_nopar:Npn \prg_new_conditional:Nnn
1338 { \__prg_generate_conditional_count:nnNnn { new } { } }
1339 \cs_set_protected_nopar:Npn \prg_set_protected_conditional:Nnn
1340 { \__prg_generate_conditional_count:nnNnn { set } { _protected } }
1341 \cs_set_protected_nopar:Npn \prg_new_protected_conditional:Nnn
1342 { \__prg_generate_conditional_count:nnNnn { new } { _protected } }
1343 \cs_set_protected:Npn \__prg_generate_conditional_count:nnNnn #1#2#3
1344 {
```

```

1345     \_cs_split_function:NN #3 \_prg_generate_conditional_count:nnNnnnn
1346     {#1} {#2}
1347   }
1348 \cs_set_protected:Npn \_prg_generate_conditional_count:nnNnnnn #1#2#3#4#5
1349 {
1350   \_cs_parm_from_arg_count:nnF
1351   { \_prg_generate_conditional:nnNnnnn {#1} {#2} #3 {#4} {#5} }
1352   { \tl_count:n {#2} }
1353   {
1354     \_msg_kernel_error:nxxx { kernel } { bad-number-of-arguments }
1355     { \token_to_str:c { #1 : #2 } }
1356     { \tl_count:n {#2} }
1357     \use_none:nn
1358   }
1359 }

```

(End definition for `\prg_set_conditional:Nnn` and others. These functions are documented on page 37.)

`_prg_generate_conditional:nnNnnnn`
`_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.

```

1360 \cs_set_protected:Npn \_prg_generate_conditional:nnNnnnnn #1#2#3#4#5#6#7#8
1361 {
1362   \if_meaning:w \c_false_bool #3
1363   \_msg_kernel_error:nnx { kernel } { missing-colon }
1364   { \token_to_str:c {#1} }
1365   \exp_after:wN \use_none:nn
1366   \fi:
1367   \use:x
1368   {
1369     \exp_not:N \_prg_generate_conditional:nnnnnnw
1370     \exp_not:n { {#4} {#5} {#1} {#2} {#6} {#8} }
1371     \tl_to_str:n {#7}
1372     \exp_not:n { , \q_recursion_tail , \q_recursion_stop }
1373   }
1374 }

```

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.

```

1375 \cs_set_protected:Npn \_prg_generate_conditional:nnnnnnw #1#2#3#4#5#6#7 ,

```

```

1376 {
1377   \if_meaning:w \q_recursion_tail #7
1378   \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
1379   \fi:
1380   \use:c { __prg_generate_ #7 _form:wnnnnnn }
1381   \tl_if_empty:nF {#7}
1382   {
1383     \__msg_kernel_error:nxxx
1384     { kernel } { conditional-form-unknown }
1385     {#7} { \token_to_str:c { #3 : #4 } }
1386   }
1387   \use_none:nnnnnnn
1388   \q_stop
1389   {#1} {#2} {#3} {#4} {#5} {#6}
1390   \__prg_generate_conditional:nnnnnnw {#1} {#2} {#3} {#4} {#5} {#6}
1391 }

```

(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.

```

1392 \cs_set_protected:Npn \__prg_generate_p_form:wnnnnnn
1393   #1 \q_stop #2#3#4#5#6#7
1394 {
1395   \if_meaning:w \scan_stop: #3 \scan_stop:
1396   \exp_after:wN \use_i:nn
1397   \else:
1398   \exp_after:wN \use_ii:nn
1399   \fi:
1400   {
1401     \exp_args:cc { cs_ #2 #3 :Npn } { #4 _p: #5 } #6
1402     { #7 \exp_end: \c_true_bool \c_false_bool }
1403   }
1404   {
1405     \__msg_kernel_error:nxx { kernel } { protected-predicate }
1406     { \token_to_str:c { #4 _p: #5 } }
1407   }
1408 }
1409 \cs_set_protected:Npn \__prg_generate_T_form:wnnnnnn
1410   #1 \q_stop #2#3#4#5#6#7
1411 {
1412   \exp_args:cc { cs_ #2 #3 :Npn } { #4 : #5 T } #6
1413   { #7 \exp_end: \use:n \use_none:n }
1414 }

```

```

1415 \cs_set_protected:Npn \__prg_generate_F_form:wnnnnnnn
1416   #1 \q_stop #2#3#4#5#6#7
1417   {
1418     \exp_args:cc { cs_ #2 #3 :Npn } { #4 : #5 F } #6
1419     { #7 \exp_end: { } }
1420   }
1421 \cs_set_protected:Npn \__prg_generate_TF_form:wnnnnnnn
1422   #1 \q_stop #2#3#4#5#6#7
1423   {
1424     \exp_args:cc { cs_ #2 #3 :Npn } { #4 : #5 TF } #6
1425     { #7 \exp_end: { } }
1426   }

```

(End definition for __prg_generate_p_form:wnnnnnnn and others.)

\prg_set_eq_conditional:NNn The setting-equal functions. Split both functions and feed $\{\langle name_1 \rangle\} \{\langle signature_1 \rangle\}$
\prg_new_eq_conditional:NNn $\langle boolean_1 \rangle \{\langle name_2 \rangle\} \{\langle signature_2 \rangle\} \langle boolean_2 \rangle \langle copying\ function \rangle \langle conditions \rangle$, \q_–
 __prg_set_eq_conditional:NNn recursion_tail , \q_recursion_stop to a first auxiliary.

```

1427 \cs_set_protected_nopar:Npn \prg_set_eq_conditional:NNn
1428   { \__prg_set_eq_conditional:NNNn \cs_set_eq:cc }
1429 \cs_set_protected_nopar:Npn \prg_new_eq_conditional:NNn
1430   { \__prg_set_eq_conditional:NNNn \cs_new_eq:cc }
1431 \cs_set_protected:Npn \__prg_set_eq_conditional:NNNn #1#2#3#4
1432   {
1433     \use:x
1434     {
1435       \exp_not:N \__prg_set_eq_conditional:nnNnnNNw
1436       \__cs_split_function:NN #2 \prg_do_nothing:
1437       \__cs_split_function:NN #3 \prg_do_nothing:
1438       \exp_not:N #1
1439       \tl_to_str:n {#4}
1440       \exp_not:n { , \q_recursion_tail , \q_recursion_stop }
1441     }
1442   }

```

(End definition for \prg_set_eq_conditional:NNn and \prg_new_eq_conditional:NNn. These functions are documented on page 39.)

__prg_set_eq_conditional:nnNnnNNw Split the function to be defined, and setup a manual clist loop over argument #6 of the
 __prg_set_eq_conditional_loop:nnnnNw first auxiliary. The second auxiliary receives twice three arguments coming from splitting
 __prg_set_eq_conditional_p_form:nnn the function to be defined and the function to copy. Make sure that both functions
 __prg_set_eq_conditional_TF_form:nnn contained a colon, otherwise we don't know how to build conditionals, hence abort. Call
 __prg_set_eq_conditional_T_form:nnn the looping macro, with arguments $\{\langle name_1 \rangle\} \{\langle signature_1 \rangle\} \{\langle name_2 \rangle\} \{\langle signature_2 \rangle\}$
 __prg_set_eq_conditional_F_form:nnn $\langle copying\ function \rangle$ and followed by the comma list. At each step in the loop, make sure
 that the conditional form we copy is defined, and copy it, otherwise abort.

```

1443 \cs_set_protected:Npn \__prg_set_eq_conditional:nnNnnNNw #1#2#3#4#5#6
1444   {
1445     \if_meaning:w \c_false_bool #3
1446     \__msg_kernel_error:nnx { kernel } { missing-colon }
1447     { \token_to_str:c {#1} }

```

```

1448     \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
1449   \fi:
1450   \if_meaning:w \c_false_bool #6
1451     \_msg_kernel_error:nxx { kernel } { missing-colon }
1452     { \token_to_str:c {#4} }
1453     \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
1454   \fi:
1455   \_prg_set_eq_conditional_loop:nnnnNw {#1} {#2} {#4} {#5}
1456 }
1457 \cs_set_protected:Npn \_prg_set_eq_conditional_loop:nnnnNw #1#2#3#4#5#6 ,
1458 {
1459   \if_meaning:w \q_recursion_tail #6
1460   \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
1461   \fi:
1462   \use:c { \_prg_set_eq_conditional_ #6 _form:wNnnnn }
1463   \tl_if_empty:nF {#6}
1464   {
1465     \_msg_kernel_error:nxxx
1466     { kernel } { conditional-form-unknown }
1467     {#6} { \token_to_str:c { #1 : #2 } }
1468   }
1469   \use_none:nnnnnn
1470   \q_stop
1471   #5 {#1} {#2} {#3} {#4}
1472   \_prg_set_eq_conditional_loop:nnnnNw {#1} {#2} {#3} {#4} #5
1473 }
1474 \cs_set:Npn \_prg_set_eq_conditional_p_form:wNnnnn #1 \q_stop #2#3#4#5#6
1475 {
1476   \_chk_if_exist_cs:c { #5 _p : #6 }
1477   #2 { #3 _p : #4 } { #5 _p : #6 }
1478 }
1479 \cs_set:Npn \_prg_set_eq_conditional_TF_form:wNnnnn #1 \q_stop #2#3#4#5#6
1480 {
1481   \_chk_if_exist_cs:c { #5 : #6 TF }
1482   #2 { #3 : #4 TF } { #5 : #6 TF }
1483 }
1484 \cs_set:Npn \_prg_set_eq_conditional_T_form:wNnnnn #1 \q_stop #2#3#4#5#6
1485 {
1486   \_chk_if_exist_cs:c { #5 : #6 T }
1487   #2 { #3 : #4 T } { #5 : #6 T }
1488 }
1489 \cs_set:Npn \_prg_set_eq_conditional_F_form:wNnnnn #1 \q_stop #2#3#4#5#6
1490 {
1491   \_chk_if_exist_cs:c { #5 : #6 F }
1492   #2 { #3 : #4 F } { #5 : #6 F }
1493 }

```

(End definition for `_prg_set_eq_conditional:nnNnnNNw` and `_prg_set_eq_conditional_loop:nnnnNw`.)

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
1494 \tex_chardef:D \c_true_bool = 1 ~
1495 \tex_chardef:D \c_false_bool = 0 ~
```

(End definition for `\c_true_bool` and `\c_false_bool`. These variables are documented on page 22.)

3.7 Dissecting a control sequence

```
\cs_to_str:N This converts a control sequence into the character string of its name, removing the
__cs_to_str:N leading escape character. This turns out to be a non-trivial matter as there are different
__cs_to_str:w 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__` yields the escape character itself and a space. The character codes are different, thus the `\if:w` test is false, and TeX reads `__cs_to_str:N` after turning the following control sequence into a string; this auxiliary removes the escape character, and stops the expansion of the initial `\tex_romannumeral:D`. The second case is that the escape character is not printable. Then the `\if:w` test is unfinished after reading a the space from `\token_to_str:N__`, and the auxiliary `__cs_to_str:w` is expanded, feeding – as a second character for the test; the test is false, and TeX skips to `\fi:`, then performs `\token_to_str:N`, and stops the `\tex_romannumeral:D` with `\c_zero`. 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_romannumeral:D` then sees 0, which is not a terminated number, followed by the escape character, a space, which is removed, terminating the expansion of `\tex_romannumeral:D`. In all three cases, `\cs_to_str:N` takes two expansion steps to be fully expanded.

```
1496 \cs_set_nopar:Npn \cs_to_str:N
1497 {
```

We implement the expansion scheme using `\tex_romannumeral: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.

```

1498 \tex_romannumeral:D
1499 \if:w \token_to_str:N \ \_cs_to_str:w \fi:
1500 \exp_after:wN \_cs_to_str:N \token_to_str:N
1501 }
1502 \cs_set:Npn \_cs_to_str:N #1 { \c_zero }
1503 \cs_set:Npn \_cs_to_str:w #1 \_cs_to_str:N
1504 { - \_int_value:w \fi: \exp_after:wN \c_zero }

```

(End definition for `\cs_to_str:N`. This function is documented on page 19.)

```

\_cs_split_function:NN
\_cs_split_function_auxi:w
\_cs_split_function_auxii:w

```

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).

```

1505 \cs_set:Npx \_cs_split_function:NN #1
1506 {
1507   \exp_not:N \exp_after:wN \exp_not:N \exp_after:wN
1508   \exp_not:N \exp_after:wN \exp_not:N \_cs_split_function_auxi:w
1509   \exp_not:N \cs_to_str:N #1 \exp_not:N \q_mark \c_true_bool
1510   \token_to_str:N : \exp_not:N \q_mark \c_false_bool
1511   \exp_not:N \q_stop
1512 }
1513 \use:x
1514 {
1515   \cs_set:Npn \exp_not:N \_cs_split_function_auxi:w
1516     ##1 \token_to_str:N : ##2 \exp_not:N \q_mark ##3##4 \exp_not:N \q_stop ##5
1517 }
1518 { \_cs_split_function_auxii:w #5 #1 \q_mark \q_stop {#2} #3 }
1519 \cs_set:Npn \_cs_split_function_auxii:w #1#2 \q_mark #3 \q_stop
1520 { #1 {#2} }

```

(End definition for `_cs_split_function:NN`.)

`_cs_get_function_name:N` Simple wrappers.

```

\cs_get_function_signature:N
1521 \cs_set:Npn \_cs_get_function_name:N #1
1522 { \_cs_split_function:NN #1 \use_i:nnn }
1523 \cs_set:Npn \_cs_get_function_signature:N #1
1524 { \_cs_split_function:NN #1 \use_ii: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:`
`\cs_if_exist:NTF` or `\fi:` as TeX will only ever skip input in case the token tested against is `\scan_stop:`.
`\cs_if_exist:cTF`

```

1525 \prg_set_conditional:Npnn \cs_if_exist:N #1 { p , T , F , TF }
1526 {
1527   \if_meaning:w #1 \scan_stop:
1528   \prg_return_false:
1529   \else:
1530     \if_cs_exist:N #1
1531     \prg_return_true:
1532     \else:
1533       \prg_return_false:
1534     \fi:
1535   \fi:
1536 }

```

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.

```

1537 \prg_set_conditional:Npnn \cs_if_exist:c #1 { p , T , F , TF }
1538 {
1539   \if_cs_exist:w #1 \cs_end:
1540   \exp_after:wN \use_i:nn
1541   \else:
1542     \exp_after:wN \use_ii:nn
1543   \fi:
1544   {
1545     \exp_after:wN \if_meaning:w \cs:w #1 \cs_end: \scan_stop:
1546     \prg_return_false:
1547     \else:
1548       \prg_return_true:
1549     \fi:
1550   }

```

```

1551     \prg_return_false:
1552 }

```

(End definition for `\cs_if_exist:NTF` and `\cs_if_exist:cTF`. These functions are documented on page 23.)

`\cs_if_free_p:N` The logical reversal of the above.

```

\cs_if_free_p:c 1553 \prg_set_conditional:Npnn \cs_if_free:N #1 { p , T , F , TF }
\cs_if_free:NTF 1554 {
\cs_if_free:cTF 1555     \if_meaning:w #1 \scan_stop:
1556     \prg_return_true:
1557     \else:
1558     \if_cs_exist:N #1
1559     \prg_return_false:
1560     \else:
1561     \prg_return_true:
1562     \fi:
1563 \fi:
1564 }
1565 \prg_set_conditional:Npnn \cs_if_free:c #1 { p , T , F , TF }
1566 {
1567     \if_cs_exist:w #1 \cs_end:
1568     \exp_after:wN \use_i:nn
1569     \else:
1570     \exp_after:wN \use_ii:nn
1571     \fi:
1572     {
1573     \exp_after:wN \if_meaning:w \cs:w #1 \cs_end: \scan_stop:
1574     \prg_return_true:
1575     \else:
1576     \prg_return_false:
1577     \fi:
1578     }
1579     { \prg_return_true: }
1580 }

```

(End definition for `\cs_if_free:NTF` and `\cs_if_free:cTF`. These functions are documented on page 23.)

`\cs_if_exist_use:NTF` The `\cs_if_exist_use:...` functions cannot be implemented as conditionals because
`\cs_if_exist_use:cTF` the true branch must leave both the control sequence itself and the true code in the input
`\cs_if_exist_use:N` stream. For the `c` variants, we are careful not to put the control sequence in the hash
`\cs_if_exist_use:c` table if it does not exist.

```

1581 \cs_set:Npn \cs_if_exist_use:NTF #1#2
1582 { \cs_if_exist:NTF #1 { #1 #2 } }
1583 \cs_set:Npn \cs_if_exist_use:NF #1
1584 { \cs_if_exist:NTF #1 { #1 } }
1585 \cs_set:Npn \cs_if_exist_use:NT #1 #2
1586 { \cs_if_exist:NTF #1 { #1 #2 } { } }
1587 \cs_set:Npn \cs_if_exist_use:N #1

```

```

1588 { \cs_if_exist:NTF #1 { #1 } { } }
1589 \cs_set:Npn \cs_if_exist_use:cTF #1#2
1590 { \cs_if_exist:cTF {#1} { \use:c {#1} #2 } }
1591 \cs_set:Npn \cs_if_exist_use:cF #1
1592 { \cs_if_exist:cTF {#1} { \use:c {#1} } }
1593 \cs_set:Npn \cs_if_exist_use:cT #1#2
1594 { \cs_if_exist:cTF {#1} { \use:c {#1} #2 } { } }
1595 \cs_set:Npn \cs_if_exist_use:c #1
1596 { \cs_if_exist:cTF {#1} { \use:c {#1} } { } }

```

(End definition for `\cs_if_exist_use:NTF` and `\cs_if_exist_use:cTF`. These functions are documented on page 18.)

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 `l3io`.

```

1597 \cs_set_protected_nopar:Npn \iow_log:x
1598 { \tex_immediate:D \tex_write:D \c_minus_one }
1599 \cs_set_protected_nopar:Npn \iow_term:x
1600 { \tex_immediate:D \tex_write:D \c_sixteen }

```

(End definition for `\iow_log:x` and `\iow_term:x`. These functions are documented on page ??.)

`__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 `l3expan` later on but `__chk_suspend_log:` and `__chk_resume_log:` are not used before that point.

```

1601 <*initex>
1602 \cs_set_protected_nopar:Npn __chk_log:x { \use_none:n }
1603 \cs_set_protected_nopar:Npn __chk_suspend_log: { }
1604 \cs_set_protected_nopar:Npn __chk_resume_log: { }
1605 </initex>
1606 <*package>
1607 \tex_ifodd:D \l@expl@log@functions@bool
1608 \cs_set_protected_nopar:Npn __chk_log:x { \iow_log:x }
1609 \cs_set_protected_nopar:Npn __chk_suspend_log:
1610 {
1611 \cs_set_protected_nopar:Npx __chk_resume_log:

```

```

1612     {
1613       \cs_set_protected_nopar:Npn \__chk_resume_log:
1614         { \exp_not:o { \__chk_resume_log: } }
1615       \cs_set_protected_nopar:Npn \__chk_log:x
1616         { \exp_not:o { \__chk_log:x } }
1617     }
1618     \cs_set_protected_nopar:Npn \__chk_log:x { \use_none:n }
1619   }
1620   \cs_set_protected_nopar:Npn \__chk_resume_log: { }
1621 \else:
1622   \cs_set_protected_nopar:Npn \__chk_log:x { \use_none:n }
1623   \cs_set_protected_nopar:Npn \__chk_suspend_log: { }
1624   \cs_set_protected_nopar:Npn \__chk_resume_log: { }
1625 \fi:
1626 </package>

```

(End definition for __chk_log:x, __chk_suspend_log:, and __chk_resume_log:.)

`__msg_kernel_error:nxx` If an internal error occurs before L^AT_EX3 has loaded l3msg then the code should issue a usable if terse error message and halt. This can only happen if a coding error is made by the team, so this is a reasonable response. Setting the `\newlinechar` is needed, to turn `^^J` into a proper line break in plain T_EX.

```

1627 \cs_set_protected:Npn \__msg_kernel_error:nxx #1#2#3#4
1628 {
1629   \tex_newlinechar:D = '\^^J \tex_relax:D
1630   \tex_errmessage:D
1631   {
1632     !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!~! ^^J
1633     Argh,~internal~LaTeX3~error! ^^J ^^J
1634     Module ~ #1 , ~ message-name~"#2": ^^J
1635     Arguments~'#3'~and~'#4' ^^J ^^J
1636     This~is~one~for~The~LaTeX3~Project:~bailing-out
1637   }
1638   \tex_end:D
1639 }
1640 \cs_set_protected:Npn \__msg_kernel_error:nxx #1#2#3
1641 { \__msg_kernel_error:nxx {#1} {#2} {#3} { } }
1642 \cs_set_protected:Npn \__msg_kernel_error:nn #1#2
1643 { \__msg_kernel_error:nxx {#1} {#2} { } { } }

```

(End definition for __msg_kernel_error:nxx, __msg_kernel_error:nxx, and __msg_kernel_error:nn.)

`\msg_line_context:` Another one from l3msg which will be altered later.

```

1644 \cs_set_nopar:Npn \msg_line_context:
1645 { on~line~ \tex_the:D \tex_inputlineno:D }

```

(End definition for \msg_line_context:. This function is documented on page 161.)

`__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 $\langle csname \rangle$ is undefined or $\backslash scan_stop:$. 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 $\backslash if...$ type function!

```

1646 \cs_set_protected:Npn \__chk_if_free_cs:N #1
1647 {
1648   \cs_if_free:NF #1
1649   {
1650     \__msg_kernel_error:nxxx { kernel } { command-already-defined }
1651     { \token_to_str:N #1 } { \token_to_meaning:N #1 }
1652   }
1653 }
1654 \*package
1655 \tex_ifodd:D \l@expl@log@functions@bool
1656 \cs_set_protected:Npn \__chk_if_free_cs:N #1
1657 {
1658   \cs_if_free:NF #1
1659   {
1660     \__msg_kernel_error:nxxx { kernel } { command-already-defined }
1661     { \token_to_str:N #1 } { \token_to_meaning:N #1 }
1662   }
1663   \__chk_log:x { Defining~\token_to_str:N #1~ \msg_line_context: }
1664 }
1665 \fi:
1666 \*package
1667 \cs_set_protected_nopar:Npn \__chk_if_free_cs:c
1668 { \exp_args:Nc \__chk_if_free_cs:N }

```

(End definition for $\backslash_chk_if_free_cs:N$ and $\backslash_chk_if_free_cs:c$.)

$\backslash_chk_if_exist_var:N$ Create the checking function for variable definitions when the option is set.

```

1669 \*package
1670 \tex_ifodd:D \l@expl@check@declarations@bool
1671 \cs_set_protected:Npn \__chk_if_exist_var:N #1
1672 {
1673   \cs_if_exist:NF #1
1674   {
1675     \__msg_kernel_error:nxx { check } { non-declared-variable }
1676     { \token_to_str:N #1 }
1677   }
1678 }
1679 \fi:
1680 \*package

```

(End definition for $\backslash_chk_if_exist_var:N$.)

$\backslash_chk_if_exist_cs:N$ This function issues an error message when the control sequence in its argument does not exist.

```

1681 \cs_set_protected:Npn \__chk_if_exist_cs:N #1
1682 {
1683   \cs_if_exist:NF #1

```

```

1684     {
1685         \_msg_kernel_error:nnx { kernel } { command-not-defined }
1686         { \token_to_str:N #1 }
1687     }
1688 }
1689 \cs_set_protected_nopar:Npn \_chk_if_exist_cs:c
1690 { \exp_args:Nc \_chk_if_exist_cs:N }

```

(End definition for _chk_if_exist_cs:N and _chk_if_exist_cs:c.)

3.10 More new definitions

Function which check that the control sequence is free before defining it.

```

\cs_new_nopar:Npn
\cs_new_nopar:Npx
  \cs_new:Npn
    \cs_new:Npx
\cs_new_protected_nopar:Npn
\cs_new_protected_nopar:Npx
  \cs_new_protected:Npn
    \cs_new_protected:Npx
      \_cs_tmp:w
1691 \cs_set:Npn \_cs_tmp:w #1#2
1692 {
1693     \cs_set_protected:Npn #1 ##1
1694     {
1695         \_chk_if_free_cs:N ##1
1696         #2 ##1
1697     }
1698 }
1699 \_cs_tmp:w \cs_new_nopar:Npn          \cs_gset_nopar:Npn
1700 \_cs_tmp:w \cs_new_nopar:Npx          \cs_gset_nopar:Npx
1701 \_cs_tmp:w \cs_new:Npn                 \cs_gset:Npn
1702 \_cs_tmp:w \cs_new:Npx                 \cs_gset:Npx
1703 \_cs_tmp:w \cs_new_protected_nopar:Npn \cs_gset_protected_nopar:Npn
1704 \_cs_tmp:w \cs_new_protected_nopar:Npx \cs_gset_protected_nopar:Npx
1705 \_cs_tmp:w \cs_new_protected:Npn       \cs_gset_protected:Npn
1706 \_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 12.)

\cs_set_nopar:cpn Like \cs_set_nopar:Npn and \cs_new_nopar:Npn, except that the first argument consists of the sequence of characters that should be used to form the name of the desired control sequence (the c stands for csname argument, see the expansion module). Global versions are also provided.

\cs_set_nopar:cpn \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.

```

1707 \cs_set:Npn \_cs_tmp:w #1#2
1708 { \cs_new_protected_nopar:Npn #1 { \exp_args:Nc #2 } }
1709 \_cs_tmp:w \cs_set_nopar:cpn   \cs_set_nopar:Npn
1710 \_cs_tmp:w \cs_set_nopar:cpx   \cs_set_nopar:Npx
1711 \_cs_tmp:w \cs_gset_nopar:cpn  \cs_gset_nopar:Npn
1712 \_cs_tmp:w \cs_gset_nopar:cpx  \cs_gset_nopar:Npx
1713 \_cs_tmp:w \cs_new_nopar:cpn   \cs_new_nopar:Npn
1714 \_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 ??.)

`\cs_set:cpn` Variants of the `\cs_set:Npn` versions which make a csname out of the first arguments.
`\cs_set:cpx` We may also do this globally.

```

\cs_gset:cpn 1715 \__cs_tmp:w \cs_set:cpn \cs_set:Npn
\cs_gset:cpx 1716 \__cs_tmp:w \cs_set:cpx \cs_set:Npx
\cs_new:cpn 1717 \__cs_tmp:w \cs_gset:cpn \cs_gset:Npn
\cs_new:cpx 1718 \__cs_tmp:w \cs_gset:cpx \cs_gset:Npx
1719 \__cs_tmp:w \cs_new:cpn \cs_new:Npn
1720 \__cs_tmp:w \cs_new:cpx \cs_new:Npx

```

(End definition for `\cs_set:cpn` and others. These functions are documented on page ??.)

`\cs_set_protected_nopar:cpn` Variants of the `\cs_set_protected_nopar:Npn` versions which make a csname out of
`\cs_set_protected_nopar:cpx` the first arguments. We may also do this globally.

```

\cs_gset_protected_nopar:cpn 1721 \__cs_tmp:w \cs_set_protected_nopar:cpn \cs_set_protected_nopar:Npn
\cs_gset_protected_nopar:cpx 1722 \__cs_tmp:w \cs_set_protected_nopar:cpx \cs_set_protected_nopar:Npx
\cs_new_protected_nopar:cpn 1723 \__cs_tmp:w \cs_gset_protected_nopar:cpn \cs_gset_protected_nopar:Npn
\cs_new_protected_nopar:cpx 1724 \__cs_tmp:w \cs_gset_protected_nopar:cpx \cs_gset_protected_nopar:Npx
1725 \__cs_tmp:w \cs_new_protected_nopar:cpn \cs_new_protected_nopar:Npn
1726 \__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 ??.)

`\cs_set_protected:cpn` Variants of the `\cs_set_protected:Npn` versions which make a csname out of the first
`\cs_set_protected:cpx` arguments. We may also do this globally.

```

\cs_gset_protected:cpn 1727 \__cs_tmp:w \cs_set_protected:cpn \cs_set_protected:Npn
\cs_gset_protected:cpx 1728 \__cs_tmp:w \cs_set_protected:cpx \cs_set_protected:Npx
\cs_new_protected:cpn 1729 \__cs_tmp:w \cs_gset_protected:cpn \cs_gset_protected:Npn
\cs_new_protected:cpx 1730 \__cs_tmp:w \cs_gset_protected:cpx \cs_gset_protected:Npx
1731 \__cs_tmp:w \cs_new_protected:cpn \cs_new_protected:Npn
1732 \__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 ??.)

3.11 Copying definitions

`\cs_set_eq:NN` These macros allow us to copy the definition of a control sequence to another control
`\cs_set_eq:cN` sequence.

`\cs_set_eq:Nc` The = sign allows us to define funny char tokens like = itself or `_` with this function.
`\cs_set_eq:cc` For the definition of `\c_space_char{~}` to work we need the ~ after the =.

`\cs_gset_eq:NN` `\cs_set_eq:NN` is long to avoid problems with a literal argument of `\par`. While
`\cs_gset_eq:cN` `\cs_new_eq:NN` will probably never be correct with a first argument of `\par`, define it
`\cs_gset_eq:Nc` long in order to throw an “already defined” error rather than “runaway argument”.
`\cs_gset_eq:cc`

```

1733 \cs_new_protected:Npn \cs_set_eq:NN #1 { \tex_let:D #1 =~ }
1734 \cs_new_protected_nopar:Npn \cs_set_eq:cN { \exp_args:Nc \cs_set_eq:NN }
1735 \cs_new_protected_nopar:Npn \cs_set_eq:Nc { \exp_args:NNc \cs_set_eq:NN }
1736 \cs_new_protected_nopar:Npn \cs_set_eq:cc { \exp_args:Ncc \cs_set_eq:NN }
1737 \cs_new_protected_nopar:Npn \cs_gset_eq:NN { \tex_global:D \cs_set_eq:NN }
1738 \cs_new_protected_nopar:Npn \cs_gset_eq:Nc { \exp_args:NNc \cs_gset_eq:NN }

```

```

1739 \cs_new_protected_nopar:Npn \cs_gset_eq:cN { \exp_args:Nc \cs_gset_eq:NN }
1740 \cs_new_protected_nopar:Npn \cs_gset_eq:cc { \exp_args:Ncc \cs_gset_eq:NN }
1741 \cs_new_protected:Npn \cs_new_eq:NN #1
1742 {
1743   \__chk_if_free_cs:N #1
1744   \tex_global:D \cs_set_eq:NN #1
1745 }
1746 \cs_new_protected_nopar:Npn \cs_new_eq:cN { \exp_args:Nc \cs_new_eq:NN }
1747 \cs_new_protected_nopar:Npn \cs_new_eq:Nc { \exp_args:Nnc \cs_new_eq:NN }
1748 \cs_new_protected_nopar:Npn \cs_new_eq:cc { \exp_args:Ncc \cs_new_eq:NN }

```

(End definition for `\cs_set_eq:NN` and others. These functions are documented on page 17.)

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 T_EX conditionals in case `#1` is unbalanced in this matter.

`\cs_undefine:c`

```

1749 \cs_new_protected:Npn \cs_undefine:N #1
1750 { \cs_gset_eq:NN #1 \tex_undefined:D }
1751 \cs_new_protected:Npn \cs_undefine:c #1
1752 {
1753   \if_cs_exist:w #1 \cs_end:
1754     \exp_after:wN \use:n
1755   \else:
1756     \exp_after:wN \use_none:n
1757   \fi:
1758   { \cs_gset_eq:cN {#1} \tex_undefined:D }
1759 }

```

(End definition for `\cs_undefine:N` and `\cs_undefine:c`. These functions are documented on page 17.)

3.13 Generating parameter text from argument count

`__cs_parm_from_arg_count:nnF`
`__cs_parm_from_arg_count_test:nnF`

L^AT_EX3 provides shorthands to define control sequences and conditionals with a simple parameter text, derived directly from the signature, or more generally from knowing the number of arguments, between 0 and 9. This function expands to its first argument, untouched, followed by a brace group containing the parameter text `{#1...#n}`, where n is the result of evaluating the second argument (as described in `\int_eval:n`). If the second argument gives a result outside the range $[0, 9]$, the third argument is returned instead, normally an error message. Some of the functions use here are not defined yet, but will be defined before this function is called.

```

1760 \cs_set_protected:Npn \__cs_parm_from_arg_count:nnF #1#2
1761 {
1762   \exp_args:Nx \__cs_parm_from_arg_count_test:nnF
1763   {
1764     \exp_after:wN \exp_not:n
1765     \if_case:w \__int_eval:w #2 \__int_eval_end:

```

```

1766         { }
1767         \or: { ##1 }
1768         \or: { ##1##2 }
1769         \or: { ##1##2##3 }
1770         \or: { ##1##2##3##4 }
1771         \or: { ##1##2##3##4##5 }
1772         \or: { ##1##2##3##4##5##6 }
1773         \or: { ##1##2##3##4##5##6##7 }
1774         \or: { ##1##2##3##4##5##6##7##8 }
1775         \or: { ##1##2##3##4##5##6##7##8##9 }
1776         \else: { \c_false_bool }
1777         \fi:
1778     }
1779     {#1}
1780 }
1781 \cs_set_protected:Npn \__cs_parm_from_arg_count_test:nnF #1#2
1782 {
1783     \if_meaning:w \c_false_bool #1
1784     \exp_after:wN \use_ii:nn
1785     \else:
1786     \exp_after:wN \use_i:nn
1787     \fi:
1788     { #2 {#1} }
1789 }

```

(End definition for __cs_parm_from_arg_count: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.

```

1790 \cs_new:Npn \__cs_count_signature:N #1
1791 { \int_eval:n { \__cs_split_function:NN #1 \__cs_count_signature:nnN } }
1792 \cs_new:Npn \__cs_count_signature:nnN #1#2#3
1793 {
1794     \if_meaning:w \c_true_bool #3
1795     \tl_count:n {#2}
1796     \else:
1797     \c_minus_one
1798     \fi:
1799 }
1800 \cs_new_nopar:Npn \__cs_count_signature:c
1801 { \exp_args:Nc \__cs_count_signature:N }

```

(End definition for __cs_count_signature:N and __cs_count_signature:c.)

\cs_generate_from_arg_count:NNnn 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

\cs_generate_from_arg_count:cNnn

\cs_generate_from_arg_count:Ncnn

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

```

1802 \cs_new_protected:Npn \cs_generate_from_arg_count:NNnn #1#2#3#4
1803 {
1804   \__cs_parm_from_arg_count:nnF { \use:nnn #2 #1 } {#3}
1805   {
1806     \_msg_kernel_error:nxxx { kernel } { bad-number-of-arguments }
1807     { \token_to_str:N #1 } { \int_eval:n {#3} }
1808   }
1809   {#4}
1810 }
```

A variant form we need right away, plus one which is used elsewhere but which is most logically created here.

```

1811 \cs_new_protected_nopar:Npn \cs_generate_from_arg_count:cNnn
1812 { \exp_args:Nc \cs_generate_from_arg_count:NNnn }
1813 \cs_new_protected_nopar:Npn \cs_generate_from_arg_count:Ncnn
1814 { \exp_args:NNc \cs_generate_from_arg_count:NNnn }
```

(End definition for \cs_generate_from_arg_count:NNnn, \cs_generate_from_arg_count:cNnn, and \cs_generate_from_arg_count:Ncnn. These functions are documented on page 16.)

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}`.

We want to define `\cs_set:Nn` as

```

\cs_set:Nn \cs_set:Nx \cs_set_protected:Npn \cs_set:Nn #1#2
\cs_set_nopar:Nn \cs_set_nopar:Nx {
\cs_set_protected:Nn \cs_generate_from_arg_count:NNnn #1 \cs_set:Npn
\cs_set_protected:Nx { \__cs_count_signature:N #1 } {#2}
\cs_set_protected_nopar:Nn }
\cs_set_protected_nopar:Nx
\cs_gset:Nn \cs_gset:Nx
\cs_gset_nopar:Nn \cs_gset_nopar:Nx
\cs_gset_protected:Nn \cs_new_protected_nopar:cpx { cs_ #1 : #2 }
\cs_gset_protected:Nx {
\cs_gset_protected_nopar:Nn \exp_not:N \__cs_generate_from_signature:NNn
\cs_gset_protected_nopar:Nx \exp_after:wN \exp_not:N \cs:w cs_ #1 : #3 \cs_end:
\cs_new:Nn 1815 \cs_set:Npn \__cs_tmp:w #1#2#3
\cs_new:Nx 1816 {
\cs_new_nopar:Nn 1817 \cs_new_protected_nopar:cpx { cs_ #1 : #2 }
\cs_new_protected:Nn 1818 {
\cs_new_protected_nopar:Nn 1819 \exp_not:N \__cs_generate_from_signature:NNn
\cs_new_protected_nopar:Nx 1820 \exp_after:wN \exp_not:N \cs:w cs_ #1 : #3 \cs_end:
\cs_new:Nn 1821 }
\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
```

```

1822 }
1823 \cs_new_protected:Npn \__cs_generate_from_signature:NNn #1#2
1824 {
1825   \__cs_split_function:NN #2 \__cs_generate_from_signature:nnNNNn
1826   #1 #2
1827 }
1828 \cs_new_protected:Npn \__cs_generate_from_signature:nnNNNn #1#2#3#4#5#6
1829 {
1830   \bool_if:NTF #3
1831   {
1832     \cs_generate_from_arg_count:NNnn
1833     #5 #4 { \tl_count:n {#2} } {#6}
1834   }
1835   {
1836     \__msg_kernel_error:nnx { kernel } { missing-colon }
1837     { \token_to_str:N #5 }
1838   }
1839 }

```

Then we define the 24 variants beginning with N.

```

1840 \__cs_tmp:w { set } { Nn } { Npn }
1841 \__cs_tmp:w { set } { Nx } { Npx }
1842 \__cs_tmp:w { set_nopar } { Nn } { Npn }
1843 \__cs_tmp:w { set_nopar } { Nx } { Npx }
1844 \__cs_tmp:w { set_protected } { Nn } { Npn }
1845 \__cs_tmp:w { set_protected } { Nx } { Npx }
1846 \__cs_tmp:w { set_protected_nopar } { Nn } { Npn }
1847 \__cs_tmp:w { set_protected_nopar } { Nx } { Npx }
1848 \__cs_tmp:w { gset } { Nn } { Npn }
1849 \__cs_tmp:w { gset } { Nx } { Npx }
1850 \__cs_tmp:w { gset_nopar } { Nn } { Npn }
1851 \__cs_tmp:w { gset_nopar } { Nx } { Npx }
1852 \__cs_tmp:w { gset_protected } { Nn } { Npn }
1853 \__cs_tmp:w { gset_protected } { Nx } { Npx }
1854 \__cs_tmp:w { gset_protected_nopar } { Nn } { Npn }
1855 \__cs_tmp:w { gset_protected_nopar } { Nx } { Npx }
1856 \__cs_tmp:w { new } { Nn } { Npn }
1857 \__cs_tmp:w { new } { Nx } { Npx }
1858 \__cs_tmp:w { new_nopar } { Nn } { Npn }
1859 \__cs_tmp:w { new_nopar } { Nx } { Npx }
1860 \__cs_tmp:w { new_protected } { Nn } { Npn }
1861 \__cs_tmp:w { new_protected } { Nx } { Npx }
1862 \__cs_tmp:w { new_protected_nopar } { Nn } { Npn }
1863 \__cs_tmp:w { new_protected_nopar } { Nx } { Npx }

```

(End definition for \cs_set:Nn and others. These functions are documented on page 15.)

```

\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

```

```

1866 \cs_new_protected_nopar:cpx { cs_ #1 : c #2 }
1867 {
1868   \exp_not:N \exp_args:Nc
1869   \exp_after:wN \exp_not:N \cs:w cs_ #1 : N #2 \cs_end:
1870 }
1871 }
1872 \__cs_tmp:w { set } { n }
1873 \__cs_tmp:w { set } { x }
1874 \__cs_tmp:w { set_nopar } { n }
1875 \__cs_tmp:w { set_nopar } { x }
1876 \__cs_tmp:w { set_protected } { n }
1877 \__cs_tmp:w { set_protected } { x }
1878 \__cs_tmp:w { set_protected_nopar } { n }
1879 \__cs_tmp:w { set_protected_nopar } { x }
1880 \__cs_tmp:w { gset } { n }
1881 \__cs_tmp:w { gset } { x }
1882 \__cs_tmp:w { gset_nopar } { n }
1883 \__cs_tmp:w { gset_nopar } { x }
1884 \__cs_tmp:w { gset_protected } { n }
1885 \__cs_tmp:w { gset_protected } { x }
1886 \__cs_tmp:w { gset_protected_nopar } { n }
1887 \__cs_tmp:w { gset_protected_nopar } { x }
1888 \__cs_tmp:w { new } { n }
1889 \__cs_tmp:w { new } { x }
1890 \__cs_tmp:w { new_nopar } { n }
1891 \__cs_tmp:w { new_nopar } { x }
1892 \__cs_tmp:w { new_protected } { n }
1893 \__cs_tmp:w { new_protected } { x }
1894 \__cs_tmp:w { new_protected_nopar } { n }
1895 \__cs_tmp:w { new_protected_nopar } { x }

```

(End definition for \cs_set:cn and others. These functions are documented on page ??.)

3.16 Checking control sequence equality

\cs_if_eq_p:NN Check if two control sequences are identical.

```

\cs_if_eq_p:cN 1896 \prg_new_conditional:Npnn \cs_if_eq:NN #1#2 { p , T , F , TF }
\cs_if_eq_p:Nc 1897 {
\cs_if_eq_p:cc 1898   \if_meaning:w #1#2
\cs_if_eq:NNTF 1899   \prg_return_true: \else: \prg_return_false: \fi:
\cs_if_eq:cNTF 1900 }
\cs_if_eq:NcTF 1901 \cs_new_nopar:Npn \cs_if_eq_p:cN { \exp_args:Nc \cs_if_eq_p:NN }
\cs_if_eq:NcTF 1902 \cs_new_nopar:Npn \cs_if_eq:cNTF { \exp_args:Nc \cs_if_eq:NNTF }
\cs_if_eq:ccTF 1903 \cs_new_nopar:Npn \cs_if_eq:cNT { \exp_args:Nc \cs_if_eq:NNTF }
1904 \cs_new_nopar:Npn \cs_if_eq:cNF { \exp_args:Nc \cs_if_eq:NNF }
1905 \cs_new_nopar:Npn \cs_if_eq_p:Nc { \exp_args:NNc \cs_if_eq_p:NN }
1906 \cs_new_nopar:Npn \cs_if_eq:NcTF { \exp_args:NNc \cs_if_eq:NNTF }
1907 \cs_new_nopar:Npn \cs_if_eq:NcT { \exp_args:NNc \cs_if_eq:NNTF }
1908 \cs_new_nopar:Npn \cs_if_eq:NcF { \exp_args:NNc \cs_if_eq:NNF }
1909 \cs_new_nopar:Npn \cs_if_eq_p:cc { \exp_args:Ncc \cs_if_eq_p:NN }

```

```

1910 \cs_new_nopar:Npn \cs_if_eq:ccTF { \exp_args:Ncc \cs_if_eq:NNTF }
1911 \cs_new_nopar:Npn \cs_if_eq:ccT { \exp_args:Ncc \cs_if_eq:NNT }
1912 \cs_new_nopar:Npn \cs_if_eq:ccF { \exp_args:Ncc \cs_if_eq:NNF }

```

(End definition for `\cs_if_eq:NNTF` and others. These functions are documented on page 23.)

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 $\langle variable \rangle = \langle value \rangle$.

```

1913 \cs_new_protected:Npn \__kernel_register_show:N #1
1914 {
1915   \__msg_show_variable:NNNnn #1 \cs_if_exist:NTF ? { }
1916   { > ~ \token_to_str:N #1 = \tex_the:D #1 }
1917 }
1918 \cs_new_protected_nopar:Npn \__kernel_register_show:c
1919 { \exp_args:Nc \__kernel_register_show:N }

```

(End definition for `__kernel_register_show:N` and `__kernel_register_show:c`.)

`\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. The `\cs_show:c` command converts its argument to a control sequence within a group to avoid showing `\relax` for undefined control sequences.

`\cs_show:c`

```

1920 \cs_new_protected:Npn \cs_show:N #1
1921 { \__msg_show_wrap:n { > ~ \token_to_str:N #1 = \cs_meaning:N #1 } }
1922 \cs_new_protected_nopar:Npn \cs_show:c
1923 { \group_begin: \exp_args:NNc \group_end: \cs_show:N }

```

(End definition for `\cs_show:N` and `\cs_show:c`. These functions are documented on page 17.)

3.18 Doing nothing functions

`\prg_do_nothing:` This does not fit anywhere else!

```

1924 \cs_new_nopar:Npn \prg_do_nothing: { }

```

(End definition for `\prg_do_nothing:`. This function is documented on page 10.)

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 the user decides to break out. This is done by putting the code that must be performed as an argument of `__prg_break_point:Nn`. The breaking functions are then defined to jump to that point and perform the argument of `__prg_break_point:Nn`, before the user's code (if any). There is a check that we close the correct loop, otherwise we continue breaking.

`__prg_map_break:Nn`

```

1925 \cs_new_eq:NN \__prg_break_point:Nn \use_ii:nn

```

```

1926 \cs_new:Npn \__prg_map_break:Nn #1#2#3 \__prg_break_point:Nn #4#5
1927 {
1928     #5
1929     \if_meaning:w #1 #4
1930     \exp_after:wN \use_iii:nnn
1931     \fi:
1932     \__prg_map_break:Nn #1 {#2}
1933 }

```

(End definition for `__prg_break_point:Nn` and `__prg_map_break:Nn`. These functions are documented on page 45.)

`__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.

```

1934 \cs_new_eq:NN \__prg_break_point: \prg_do_nothing:
1935 \cs_new:Npn \__prg_break: #1 \__prg_break_point: { }
1936 \cs_new:Npn \__prg_break:n #1#2 \__prg_break_point: {#1}

```

(End definition for `__prg_break_point:.` This function is documented on page 45.)

```

1937 </initex | package>

```

4 l3expan implementation

```

1938 <*initex | package>
1939 <@@=exp>

```

`\exp_after:wN` These are defined in `l3basics`.

`\exp_not:N`
`\exp_not:n`

(End definition for `\exp_after:wN`. This function is documented on page 32.)

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_nopar: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_tl` 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_tl`. This variable is documented on page 35.)

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 $\backslash::\langle Z \rangle$ always has signature $\#1\backslash::\#2\#3$ where $\#1$ holds the remaining argument manipulations to be performed, $\backslash::$ 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 $\backslash::p$, which has to grab an argument delimited by a left brace.

$\backslash_exp_arg_next:nnn$ $\#1$ 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.

```
1940 \cs_new:Npn \_exp_arg_next:nnn #1#2#3 { #2 \::: { #3 {#1} } }
1941 \cs_new:Npn \_exp_arg_next:Nnn #1#2#3 { #2 \::: { #3 #1 } }
```

(End definition for $\backslash_exp_arg_next:nnn$.)

$\backslash:::$ The end marker is just another name for the identity function.

```
1942 \cs_new:Npn \::: #1 {#1}
```

(End definition for $\backslash::$.)

$\backslash::n$ This function is used to skip an argument that doesn't need to be expanded.

```
1943 \cs_new:Npn \::n #1 \::: #2#3 { #1 \::: { #2 {#3} } }
```

(End definition for $\backslash::n$.)

$\backslash::N$ This function is used to skip an argument that consists of a single token and doesn't need to be expanded.

```
1944 \cs_new:Npn \::N #1 \::: #2#3 { #1 \::: {#2#3} }
```

(End definition for $\backslash::N$.)

$\backslash::p$ 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.

```
1945 \cs_new:Npn \::p #1 \::: #2#3# { #1 \::: {#2#3} }
```

(End definition for $\backslash::p$.)

$\backslash::c$ This function is used to skip an argument that is turned into a control sequence without expansion.

```
1946 \cs_new:Npn \::c #1 \::: #2#3
1947 { \exp_after:wN \_exp_arg_next:Nnn \cs:w #3 \cs_end: {#1} {#2} }
```

(End definition for $\backslash::c$.)

$\backslash::o$ This function is used to expand an argument once.

```
1948 \cs_new:Npn \::o #1 \::: #2#3
1949 { \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 T_EX 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 T_EX has not tried to execute any of the non-expandable tokens. This is what differentiates this function from the `x` argument type.

```

1950 \cs_new:Npn \::f #1 \::: #2#3
1951 {
1952   \exp_after:wN \__exp_arg_next:nnn
1953   \exp_after:wN { \exp:w \exp_end_continue_f:w #3 }
1954   {#1} {#2}
1955 }
1956 \use:nn { \cs_new_eq:NN \exp_stop_f: } { ~ }
```

(End definition for \::f.)

\::x This function is used to expand an argument fully.

```

1957 \cs_new_protected:Npn \::x #1 \::: #2#3
1958 {
1959   \cs_set_nopar:Npx \l__exp_internal_tl { {#3} }
1960   \exp_after:wN \__exp_arg_next:nnn \l__exp_internal_tl {#1} {#2}
1961 }
```

(End definition for \::x.)

\::v These functions return the value of a register, i.e., one of `tl`, `clist`, `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.

```

1962 \cs_new:Npn \::V #1 \::: #2#3
1963 {
1964   \exp_after:wN \__exp_arg_next:nnn
1965   \exp_after:wN { \exp:w \__exp_eval_register:N #3 }
1966   {#1} {#2}
1967 }
1968 \cs_new:Npn \::v # 1\::: #2#3
1969 {
1970   \exp_after:wN \__exp_arg_next:nnn
1971   \exp_after:wN { \exp:w \__exp_eval_register:c {#3} }
1972   {#1} {#2}
1973 }
```

(End definition for \::v.)

```

\__exp_eval_register:N This function evaluates a register. Now a register might exist as one of two things: A
\__exp_eval_register:c parameter-less macro or a built-in TEX register such as \count. For the TEX registers we
\__exp_eval_error_msg:w 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:.

```

```

1974 \cs_new:Npn \__exp_eval_register:N #1
1975 {
1976   \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 T_EX 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.

```

1977   \if_meaning:w \scan_stop: #1
1978   \__exp_eval_error_msg:w
1979   \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 T_EX 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.

```

1980   \else:
1981   \exp_after:wN \use_i_ii:nnn
1982   \fi:
1983   \exp_after:wN \exp_end: \tex_the:D #1
1984 }
1985 \cs_new:Npn \__exp_eval_register:c #1
1986 { \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}

```

```

1987 \cs_new:Npn \__exp_eval_error_msg:w #1 \tex_the:D #2
1988 {
1989     \fi:
1990     \fi:
1991     \_msg_kernel_expandable_error:nnn { kernel } { bad-variable } {#2}
1992     \exp_end:
1993 }

```

(End definition for `__exp_eval_register:N` and `__exp_eval_register:c`.)

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!

```

1994 \cs_new:Npn \exp_args:No #1#2 { \exp_after:wN #1 \exp_after:wN {#2} }
1995 \cs_new:Npn \exp_args:NNo #1#2#3
1996 { \exp_after:wN #1 \exp_after:wN #2 \exp_after:wN {#3} }
1997 \cs_new:Npn \exp_args:NNNo #1#2#3#4
1998 { \exp_after:wN #1 \exp_after:wN#2 \exp_after:wN #3 \exp_after:wN {#4} }

```

(End definition for `\exp_args:No`. This function is documented on page 29.)

`\exp_args:Nc` In l3basics.
`\exp_args:cc` (End definition for `\exp_args:Nc` and `\exp_args:cc`. These functions are documented on page 29.)

`\exp_args:NNc` Here are the functions that turn their argument into csnames but are expandable.
`\exp_args:Ncc`
`\exp_args:Nccc`

```

1999 \cs_new:Npn \exp_args:NNc #1#2#3
2000 { \exp_after:wN #1 \exp_after:wN #2 \cs:w # 3\cs_end: }
2001 \cs_new:Npn \exp_args:Ncc #1#2#3
2002 { \exp_after:wN #1 \cs:w #2 \exp_after:wN \cs_end: \cs:w #3 \cs_end: }
2003 \cs_new:Npn \exp_args:Nccc #1#2#3#4
2004 {
2005     \exp_after:wN #1
2006     \cs:w #2 \exp_after:wN \cs_end:
2007     \cs:w #3 \exp_after:wN \cs_end:
2008     \cs:w #4 \cs_end:
2009 }

```

(End definition for `\exp_args:NNc`, `\exp_args:Ncc`, and `\exp_args:Nccc`. These functions are documented on page 30.)

`\exp_args:Nf`
`\exp_args:Nv`
`\exp_args:Nv`

```

2010 \cs_new:Npn \exp_args:Nf #1#2
2011 { \exp_after:wN #1 \exp_after:wN { \exp:w \exp_end_continue_f:w #2 } }
2012 \cs_new:Npn \exp_args:Nv #1#2
2013 {
2014     \exp_after:wN #1 \exp_after:wN
2015     { \exp:w \__exp_eval_register:c {#2} }

```

```

2016     }
2017 \cs_new:Npn \exp_args:NV #1#2
2018 {
2019     \exp_after:wN #1 \exp_after:wN
2020     { \exp:w \__exp_eval_register:N #2 }
2021 }

```

(End definition for `\exp_args:Nf`, `\exp_args:NV`, and `\exp_args:Nv`. These functions are documented on page 29.)

`\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 2022 \cs_new:Npn \exp_args:NNf #1#2#3
\exp_args:Ncf 2023 {
\exp_args:Nco 2024     \exp_after:wN #1
2025     \exp_after:wN #2
2026     \exp_after:wN { \exp:w \exp_end_continue_f:w #3 }
2027 }
2028 \cs_new:Npn \exp_args:NNv #1#2#3
2029 {
2030     \exp_after:wN #1
2031     \exp_after:wN #2
2032     \exp_after:wN { \exp:w \__exp_eval_register:c {#3} }
2033 }
2034 \cs_new:Npn \exp_args:NNV #1#2#3
2035 {
2036     \exp_after:wN #1
2037     \exp_after:wN #2
2038     \exp_after:wN { \exp:w \__exp_eval_register:N #3 }
2039 }
2040 \cs_new:Npn \exp_args:Nco #1#2#3
2041 {
2042     \exp_after:wN #1
2043     \cs:w #2 \exp_after:wN \cs_end:
2044     \exp_after:wN {#3}
2045 }
2046 \cs_new:Npn \exp_args:Ncf #1#2#3
2047 {
2048     \exp_after:wN #1
2049     \cs:w #2 \exp_after:wN \cs_end:
2050     \exp_after:wN { \exp:w \exp_end_continue_f:w #3 }
2051 }
2052 \cs_new:Npn \exp_args:NVV #1#2#3
2053 {
2054     \exp_after:wN #1
2055     \exp_after:wN { \exp:w \exp_after:wN
2056         \__exp_eval_register:N \exp_after:wN #2 \exp_after:wN }
2057     \exp_after:wN { \exp:w \__exp_eval_register:N #3 }
2058 }

```

(End definition for `\exp_args:NNV` and others. These functions are documented on page ??.)

A few more that we can hand-tune.

```

\exp_args:Ncco 2059 \cs_new:Npn \exp_args:NNNV #1#2#3#4
\exp_args:NcNc 2060 {
\exp_args:NcNo 2061   \exp_after:wN #1
\exp_args:NNNV 2062   \exp_after:wN #2
2063   \exp_after:wN #3
2064   \exp_after:wN { \exp:w \__exp_eval_register:N #4 }
2065 }
2066 \cs_new:Npn \exp_args:NcNc #1#2#3#4
2067 {
2068   \exp_after:wN #1
2069   \cs:w #2 \exp_after:wN \cs_end:
2070   \exp_after:wN #3
2071   \cs:w #4 \cs_end:
2072 }
2073 \cs_new:Npn \exp_args:NcNo #1#2#3#4
2074 {
2075   \exp_after:wN #1
2076   \cs:w #2 \exp_after:wN \cs_end:
2077   \exp_after:wN #3
2078   \exp_after:wN {#4}
2079 }
2080 \cs_new:Npn \exp_args:Ncco #1#2#3#4
2081 {
2082   \exp_after:wN #1
2083   \cs:w #2 \exp_after:wN \cs_end:
2084   \cs:w #3 \exp_after:wN \cs_end:
2085   \exp_after:wN {#4}
2086 }

```

(End definition for `\exp_args:Ncco` and others. These functions are documented on page ??.)

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
2087 \cs_new_protected_nopar:Npn \exp_args:Nx { \::x \::: }

```

(End definition for `\exp_args:Nx`. This function is documented on page 30.)

Here are the actual function definitions, using the helper functions above.

```

\exp_args:Nnc 2088 \cs_new_nopar:Npn \exp_args:Nnc { \::n \::c \::: }
\exp_args:Nfo 2089 \cs_new_nopar:Npn \exp_args:Nfo { \::f \::o \::: }
\exp_args:Nff 2090 \cs_new_nopar:Npn \exp_args:Nff { \::f \::f \::: }
\exp_args:Nnf 2091 \cs_new_nopar:Npn \exp_args:Nnf { \::n \::f \::: }
\exp_args:NnV
\exp_args:Noo
\exp_args:Nof
\exp_args:Noc
\exp_args:NNx
\exp_args:Ncx
\exp_args:Nnx
\exp_args:Nox
\exp_args:Nxo
\exp_args:Nxx

```

```

2092 \cs_new_nopar:Npn \exp_args:Nno { \::n \::o \::: }
2093 \cs_new_nopar:Npn \exp_args:NnV { \::n \::V \::: }
2094 \cs_new_nopar:Npn \exp_args:Noo { \::o \::o \::: }
2095 \cs_new_nopar:Npn \exp_args:Nof { \::o \::f \::: }
2096 \cs_new_nopar:Npn \exp_args:Noc { \::o \::c \::: }
2097 \cs_new_protected_nopar:Npn \exp_args:NNx { \::N \::x \::: }
2098 \cs_new_protected_nopar:Npn \exp_args:Ncx { \::c \::x \::: }
2099 \cs_new_protected_nopar:Npn \exp_args:Nnx { \::n \::x \::: }
2100 \cs_new_protected_nopar:Npn \exp_args:Nox { \::o \::x \::: }
2101 \cs_new_protected_nopar:Npn \exp_args:Nxo { \::x \::o \::: }
2102 \cs_new_protected_nopar:Npn \exp_args:Nxx { \::x \::x \::: }

```

(End definition for `\exp_args:Nnc` and others. These functions are documented on page ??.)

```

\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
2103 \cs_new_nopar:Npn \exp_args:NNno { \::N \::n \::o \::: }
2104 \cs_new_nopar:Npn \exp_args:NNoo { \::N \::o \::o \::: }
2105 \cs_new_nopar:Npn \exp_args:Nnnc { \::n \::n \::c \::: }
2106 \cs_new_nopar:Npn \exp_args:Nnno { \::n \::n \::o \::: }
2107 \cs_new_nopar:Npn \exp_args:Nooo { \::o \::o \::o \::: }
2108 \cs_new_protected_nopar:Npn \exp_args:NNNx { \::N \::N \::x \::: }
2109 \cs_new_protected_nopar:Npn \exp_args:NNnx { \::N \::n \::x \::: }
2110 \cs_new_protected_nopar:Npn \exp_args:NNox { \::N \::o \::x \::: }
2111 \cs_new_protected_nopar:Npn \exp_args:Nnnx { \::n \::n \::x \::: }
2112 \cs_new_protected_nopar:Npn \exp_args:Nnox { \::n \::o \::x \::: }
2113 \cs_new_protected_nopar:Npn \exp_args:Nccx { \::c \::c \::x \::: }
2114 \cs_new_protected_nopar:Npn \exp_args:Ncnx { \::c \::n \::x \::: }
2115 \cs_new_protected_nopar:Npn \exp_args:Noox { \::o \::o \::x \::: }

```

(End definition for `\exp_args:NNno` and others. These functions are documented on page ??.)

4.4 Last-unbraced versions

`__exp_arg_last_unbraced:nn` There are a few places where the last argument needs to be available unbraced. First some helper macros.

```

\::f_unbraced
\::o_unbraced
\::V_unbraced
\::v_unbraced
\::x_unbraced
2116 \cs_new:Npn \__exp_arg_last_unbraced:nn #1#2 { #2#1 }
2117 \cs_new:Npn \::f_unbraced \::: #1#2
2118 {
2119   \exp_after:wN \__exp_arg_last_unbraced:nn
2120   \exp_after:wN { \exp:w \exp_end_continue_f:w #2 } {#1}
2121 }
2122 \cs_new:Npn \::o_unbraced \::: #1#2
2123 { \exp_after:wN \__exp_arg_last_unbraced:nn \exp_after:wN {#2} {#1} }
2124 \cs_new:Npn \::V_unbraced \::: #1#2
2125 {
2126   \exp_after:wN \__exp_arg_last_unbraced:nn
2127   \exp_after:wN { \exp:w \__exp_eval_register:N #2 } {#1}
2128 }
2129 \cs_new:Npn \::v_unbraced \::: #1#2
2130 {

```

```

2131 \exp_after:wN \_exp_arg_last_unbraced:nn
2132 \exp_after:wN { \exp:w \_exp_eval_register:c {#2} } {#1}
2133 }
2134 \cs_new_protected:Npn \::x_unbraced \::: #1#2
2135 {
2136 \cs_set_nopar:Npx \l__exp_internal_tl { \exp_not:n {#1} #2 }
2137 \l__exp_internal_tl
2138 }

```

(End definition for _exp_arg_last_unbraced:nn.)

\exp_last_unbraced:NV Now the business end: most of these are hand-tuned for speed, but the general system is
 \exp_last_unbraced:Nv in place.

```

\exp_last_unbraced:Nf \cs_new:Npn \exp_last_unbraced:NV #1#2
\exp_last_unbraced:Nv \exp_last_unbraced:Nv #1#2 { \exp_after:wN #1 \exp:w \_exp_eval_register:N #2 }
\exp_last_unbraced:Nco \cs_new:Npn \exp_last_unbraced:Nv #1#2
\exp_last_unbraced:NcV \exp_last_unbraced:NcV #1#2 { \exp_after:wN #1 \exp:w \_exp_eval_register:c {#2} }
\exp_last_unbraced:NNV \cs_new:Npn \exp_last_unbraced:Nv #1#2 { \exp_after:wN #1 #2 }
\exp_last_unbraced:NNo \cs_new:Npn \exp_last_unbraced:Nf #1#2
\exp_last_unbraced:NNNV \exp_last_unbraced:Nf #1#2 { \exp_after:wN #1 \exp:w \exp_end_continue_f:w #2 }
\exp_last_unbraced:NNNo \cs_new:Npn \exp_last_unbraced:Nco #1#2#3
\exp_last_unbraced:Nno \exp_last_unbraced:Nco #1#2#3 { \exp_after:wN #1 \cs:w #2 \exp_after:wN \cs_end: #3 }
\exp_last_unbraced:Noo \cs_new:Npn \exp_last_unbraced:NcV #1#2#3
\exp_last_unbraced:Nfo \exp_last_unbraced:NcV #1#2#3 {
2149 {
2150 \exp_after:wN #1
2151 \cs:w #2 \exp_after:wN \cs_end:
2152 \exp:w \_exp_eval_register:N #3
2153 }
2154 \cs_new:Npn \exp_last_unbraced:NNV #1#2#3
2155 {
2156 \exp_after:wN #1
2157 \exp_after:wN #2
2158 \exp:w \_exp_eval_register:N #3
2159 }
2160 \cs_new:Npn \exp_last_unbraced:NNo #1#2#3
2161 { \exp_after:wN #1 \exp_after:wN #2 #3 }
2162 \cs_new:Npn \exp_last_unbraced:NNNV #1#2#3#4
2163 {
2164 \exp_after:wN #1
2165 \exp_after:wN #2
2166 \exp_after:wN #3
2167 \exp:w \_exp_eval_register:N #4
2168 }
2169 \cs_new:Npn \exp_last_unbraced:NNNo #1#2#3#4
2170 { \exp_after:wN #1 \exp_after:wN #2 \exp_after:wN #3 #4 }
2171 \cs_new_nopar:Npn \exp_last_unbraced:Nno { \::n \::o_unbraced \::: }
2172 \cs_new_nopar:Npn \exp_last_unbraced:Noo { \::o \::o_unbraced \::: }
2173 \cs_new_nopar:Npn \exp_last_unbraced:Nfo { \::f \::o_unbraced \::: }
2174 \cs_new_nopar:Npn \exp_last_unbraced:NnNo { \::n \::N \::o_unbraced \::: }
2175 \cs_new_protected_nopar:Npn \exp_last_unbraced:Nx { \::x_unbraced \::: }

```

(End definition for `\exp_last_unbraced:Nv`. This function is documented on page ??.)

```

\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.

```

2176 \cs_new:Npn \exp_last_two_unbraced:Noo #1#2#3
2177 { \exp_after:wN \__exp_last_two_unbraced:noN \exp_after:wN {#3} {#2} #1 }
2178 \cs_new:Npn \__exp_last_two_unbraced:noN #1#2#3
2179 { \exp_after:wN #3 #2 #1 }

```

(End definition for `\exp_last_two_unbraced:Noo`. This function is documented on page 32.)

4.5 Preventing expansion

```

\exp_not:o
\exp_not:c 2180 \cs_new:Npn \exp_not:o #1 { \etex_unexpanded:D \exp_after:wN {#1} }
\exp_not:f 2181 \cs_new:Npn \exp_not:c #1 { \exp_after:wN \exp_not:N \cs:w #1 \cs_end: }
\exp_not:V 2182 \cs_new:Npn \exp_not:f #1
\exp_not:v 2183 { \etex_unexpanded:D \exp_after:wN { \exp:w \exp_end_continue_f:w #1 } }
2184 \cs_new:Npn \exp_not:V #1
2185 {
2186   \etex_unexpanded:D \exp_after:wN
2187   { \exp:w \__exp_eval_register:N #1 }
2188 }
2189 \cs_new:Npn \exp_not:v #1
2190 {
2191   \etex_unexpanded:D \exp_after:wN
2192   { \exp:w \__exp_eval_register:c {#1} }
2193 }

```

(End definition for `\exp_not:o`. This function is documented on page 33.)

4.6 Controlled expansion

```

\exp:w To trigger a sequence of “arbitrary” many expansions we need a method to invoke TEX’s
\exp_end: expansion mechanism in such a way that a) we are able to stop it in a controlled manner
\exp_end_continue_f:w and b) that the result of what triggered the expansion in the first place is null, i.e., that
\exp_end_continue_f:nw 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_romannumeral:D` expands the tokens following it when looking for a number and that its expansion is null if that number turns out to be zero or negative. So we use that to start the expansion sequence.

```

2194 %\cs_new_eq:NN \exp:w \tex_romannumeral:D

```

So to stop the expansion sequence in a controlled way all we need to provide is `\c_zero` as part of expanded tokens. As this is an integer constant it will immediately stop `\tex_romannumerl:D`'s search for a number.

```
2195 %\cs_new_eq:NN \exp_end: \c_zero
```

(Note that according to our specification all tokens we expand initiated by `\exp:w` are supposed to be expandable (as well as their replacement text in the expansion) so we will not encounter a “number” that actually result in a roman numeral being generated. Or if we do then the programmer made a mistake.)

If on the other hand we want to stop the initial expansion sequence but continue with an f-type expansion we provide the alphabetic constant `^^@` that also represents 0 but this time T_EX's syntax for a *number* will continue searching for an optional space (and it will continue expansion doing that) — see T_EXbook page 269 for details.

```
2196 \tex_catcode:D '^^@=13
```

```
2197 \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.⁵

```
2198 \cs_new:Npn ^^@{\expansionERROR}
```

```
2199 \cs_new:Npn \exp_end_continue_f:nw #1 { '^^@ #1 }
```

```
2200 \tex_catcode:D '^^@=15
```

(End definition for `\exp:w`. This function is documented on page 35.)

4.7 Defining function variants

```
2201 <@@=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_nopar:Npx` or `\cs_new_protected_nopar: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.

```
2202 \cs_new_protected:Npn \cs_generate_variant:Nn #1#2
```

```
2203 {
```

```
2204   \__chk_if_exist_cs:N #1
```

```
2205   \__cs_generate_variant:N #1
```

```
2206   \exp_after:wN \__cs_split_function:NN
```

```
2207   \exp_after:wN #1
```

```
2208   \exp_after:wN \__cs_generate_variant:nnNN
```

```
2209   \exp_after:wN #1
```

```
2210   \tl_to_str:n {#2} , \scan_stop: , \q_recursion_stop
```

```
2211 }
```

⁵Need to get a real error message.

(End definition for `\cs_generate_variant:Nn`. This function is documented on page 27.)

```

__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 T_EX 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 first occurrence of `macro`. The `ww` auxiliary removes everything in the meaning string after the first `ma`. We use `ma` rather than the full `macro` because the meaning of the `\firstmark` primitive (and four others) can contain an arbitrary string after a leading `firstmark:`. Then, look for `pr` in the part we extracted: no need to look for anything longer: the only strings we can have are an empty string, `\long`, `\protected`, `\protected\long`, `\first`, `\top`, `\bot`, `\splittop`, or `\splitbot`, with `\` replaced by the appropriate escape character. If `pr` appears in the part before `ma`, the first `\q_mark` is taken as an argument of the `wwNw` auxiliary, and #3 is `\cs_new_protected_nopar:Npx`, otherwise it is `\cs_new_nopar:Npx`.

```

2212 \cs_new_protected:Npx __cs_generate_variant:N #1
2213 {
2214   \exp_not:N \exp_after:wN \exp_not:N \if_meaning:w
2215   \exp_not:N \exp_not:N #1 #1
2216   \cs_set_eq:NN \exp_not:N __cs_tmp:w \cs_new_protected_nopar:Npx
2217   \exp_not:N \else:
2218   \exp_not:N \exp_after:wN \exp_not:N __cs_generate_variant:ww
2219   \exp_not:N \token_to_meaning:N #1 \tl_to_str:n { ma }
2220   \exp_not:N \q_mark
2221   \exp_not:N \q_mark \cs_new_protected_nopar:Npx
2222   \tl_to_str:n { pr }
2223   \exp_not:N \q_mark \cs_new_nopar:Npx
2224   \exp_not:N \q_stop
2225   \exp_not:N \fi:
2226 }
2227 \use:x
2228 {
2229   \cs_new_protected:Npn \exp_not:N __cs_generate_variant:ww
2230   ##1 \tl_to_str:n { ma } ##2 \exp_not:N \q_mark
2231 }
2232 { __cs_generate_variant:wwNw #1 }
2233 \use:x
2234 {
2235   \cs_new_protected:Npn \exp_not:N __cs_generate_variant:wwNw
2236   ##1 \tl_to_str:n { pr } ##2 \exp_not:N \q_mark
2237   ##3 ##4 \exp_not:N \q_stop
2238 }
2239 { \cs_set_eq:NN __cs_tmp:w #3 }

```

(End definition for `__cs_generate_variant:N`.)

```

__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.

```
2240 \cs_new_protected:Npn \__cs_generate_variant:nnNN #1#2#3#4
2241 {
2242   \if_meaning:w \c_false_bool #3
2243     \__msg_kernel_error:nxx { kernel } { missing-colon }
2244     { \token_to_str:c {#1} }
2245     \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
2246   \fi:
2247   \__cs_generate_variant:Nnnw #4 {#1}{#2}
2248 }
```

(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`.

```

2249 \cs_new_protected:Npn \__cs_generate_variant:Nnnw #1#2#3#4 ,
2250 {
2251   \if_meaning:w \scan_stop: #4
2252   \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
2253   \fi:
2254   \use:x
2255   {
2256     \exp_not:N \__cs_generate_variant:wwNN
2257     \__cs_generate_variant_loop:nNwN { }
2258     #4
2259     \__cs_generate_variant_loop_end:nwwwNNnn
2260     \q_mark
2261     #3 ~
2262     { ~ { } \fi: \__cs_generate_variant_loop_long:wNNnn } ~
2263     { }
2264     \q_stop
2265     \exp_not:N #1 {#2} {#4}
2266   }
2267   \__cs_generate_variant:Nnnw #1 {#2} {#3}
2268 }

```

(End definition for `__cs_generate_variant:Nnnw`.)

<code>__cs_generate_variant_loop:nNwN</code>	#1 :	Last few (consecutive) letters common between the base and variant (in fact, <code>__cs_generate_variant_same:N</code> <i><letter></i> for each letter).
<code>__cs_generate_variant_loop_same:w</code>	#2 :	Next variant letter.
<code>__cs_generate_variant_loop_end:nwwwNNnn</code>	#3 :	Remainder of variant form.
<code>__cs_generate_variant_loop_long:wNNnn</code>	#4 :	Next base letter.
<code>__cs_generate_variant_loop_invalid:NNwNNnn</code>		

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.

```

2269 \cs_new:Npn \_cs_generate_variant_loop:nNwN #1#2#3 \q_mark #4
2270 {
2271   \if:w #2 #4
2272     \exp_after:wN \_cs_generate_variant_loop_same:w
2273   \else:
2274     \if:w N #4 \else:
2275       \if:w n #4 \else:
2276         \_cs_generate_variant_loop_invalid:NNwNNnn #4#2
2277       \fi:
2278     \fi:
2279   \fi:
2280   #1
2281   \prg_do_nothing:
2282   #2
2283   \_cs_generate_variant_loop:nNwN { } #3 \q_mark
2284 }
2285 \cs_new:Npn \_cs_generate_variant_loop_same:w
2286   #1 \prg_do_nothing: #2#3#4
2287 {
2288   #3 { #1 \_cs_generate_variant_same:N #2 }
2289 }
2290 \cs_new:Npn \_cs_generate_variant_loop_end:nwwwNNnn
2291   #1#2 \q_mark #3 ~ #4 \q_stop #5#6#7#8
2292 {
2293   \scan_stop: \scan_stop: \fi:
2294   \exp_not:N \q_mark
2295   \exp_not:N \q_stop
2296   \exp_not:N #6
2297   \exp_not:c { #7 : #8 #1 #3 }
2298 }
2299 \cs_new:Npn \_cs_generate_variant_loop_long:wNNnn #1 \q_stop #2#3#4#5
2300 {
2301   \exp_not:n
2302   {
2303     \q_mark

```

```

2304         \_msg_kernel_error:nxxx { kernel } { variant-too-long }
2305         {#5} { \token_to_str:N #3 }
2306         \use_none:nnnn
2307         \q_stop
2308         #3
2309         #3
2310     }
2311 }
2312 \cs_new:Npn \__cs_generate_variant_loop_invalid:NNwNNnn
2313     #1#2 \fi: \fi: \fi: #3 \q_stop #4#5#6#7
2314 {
2315     \fi: \fi: \fi:
2316     \exp_not:n
2317     {
2318         \q_mark
2319         \_msg_kernel_error:nxxxx { kernel } { invalid-variant }
2320         {#7} { \token_to_str:N #5 } {#1} {#2}
2321         \use_none:nnnn
2322         \q_stop
2323         #5
2324         #5
2325     }
2326 }

```

(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.

```

2327 \cs_new:Npn \__cs_generate_variant_same:N #1
2328 {
2329     \if:w N #1
2330         N
2331     \else:
2332         \if:w p #1
2333             p
2334         \else:
2335             n
2336         \fi:
2337     \fi:
2338 }

```

(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_nopar:Npx. Then define the variant by combining the \exp_args:N #3 variant and the base function.

```

2339 \cs_new_protected:Npn \__cs_generate_variant:wwNN
2340     #1 \q_mark #2 \q_stop #3#4

```

```

2341 {
2342   #2
2343   \cs_if_free:NTF #4
2344   {
2345     \group_begin:
2346     \__cs_generate_internal_variant:n {#1}
2347     \__cs_tmp:w #4 { \exp_not:c { exp_args:N #1 } \exp_not:N #3 }
2348     \group_end:
2349   }
2350   {
2351     \__chk_log:x
2352     {
2353       Variant~\token_to_str:N #4~%
2354       already~defined;~ not~ changing~ it~ \msg_line_context:
2355     }
2356   }
2357 }

```

(End definition for __cs_generate_variant:wwNN.)

_cs_generate_internal_variant:n
 _cs_generate_internal_variant:wwnw
 _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.

```

2358 \cs_new_protected:Npx \__cs_generate_internal_variant:n #1
2359 {
2360   \exp_not:N \__cs_generate_internal_variant:wwnNwnn
2361   #1 \exp_not:N \q_mark
2362   { \cs_set_eq:NN \exp_not:N \__cs_tmp:w \cs_new_protected_nopar:Npx }
2363   \cs_new_protected_nopar:cpx
2364   \token_to_str:N x \exp_not:N \q_mark
2365   { }
2366   \cs_new_nopar:cpx
2367   \exp_not:N \q_stop
2368   { exp_args:N #1 }
2369   {
2370     \exp_not:N \__cs_generate_internal_variant_loop:n #1
2371     { : \exp_not:N \use_i:nn }
2372   }
2373 }
2374 \use:x
2375 {
2376   \cs_new_protected:Npn \exp_not:N \__cs_generate_internal_variant:wwnNwnn
2377   ##1 \token_to_str:N x ##2 \exp_not:N \q_mark
2378   ##3 ##4 ##5 \exp_not:N \q_stop ##6 ##7
2379 }
2380 {
2381   #3
2382   \cs_if_free:cT {#6} { #4 {#6} {#7} }
2383 }

```

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.

```

2384 \cs_new:Npn \__cs_generate_internal_variant_loop:n #1
2385 {
2386   \exp_after:wN \exp_not:N \cs:w :: #1 \cs_end:
2387   \__cs_generate_internal_variant_loop:n
2388 }

```

(End definition for `__cs_generate_internal_variant:n`.)

```

2389 </initex | package>

```

5 l3prg implementation

The following test files are used for this code: `m3prg001.lvt`, `m3prg002.lvt`, `m3prg003.lvt`.

```

2390 <*initex | package>

```

5.1 Primitive conditionals

`\if_bool:N` Those two primitive TeX conditionals are synonyms.
`\if_predicate:w`

```

2391 \cs_new_eq:NN \if_bool:N \tex_ifodd:D
2392 \cs_new_eq:NN \if_predicate:w \tex_ifodd:D

```

(End definition for `\if_bool:N`. This function is documented on page 44.)

5.2 Defining a set of conditional functions

These are all defined in `l3basics`, as they are needed “early”. This is just a reminder!

(End definition for `\prg_set_conditional:Npnn` and others. These functions are documented on page 37.)

5.3 The boolean data type

Boolean variables have to be initiated when they are created. Other than that there is not much to say here.

```

2394 \cs_new_protected:Npn \bool_new:N #1 { \cs_new_eq:NN #1 \c_false_bool }
2395 \cs_generate_variant:Nn \bool_new:N { c }

```

(End definition for `\bool_new:N` and `\bool_new:c`. These functions are documented on page 39.)

```

\bool_set_true:N Setting is already pretty easy.
\bool_set_true:c 2396 \cs_new_protected:Npn \bool_set_true:N #1
\bool_gset_true:N 2397 { \cs_set_eq:NN #1 \c_true_bool }
\bool_gset_true:c 2398 \cs_new_protected:Npn \bool_set_false:N #1
\bool_set_false:N 2399 { \cs_set_eq:NN #1 \c_false_bool }
\bool_set_false:c 2400 \cs_new_protected:Npn \bool_gset_true:N #1
\bool_gset_false:N 2401 { \cs_gset_eq:NN #1 \c_true_bool }
\bool_gset_false:c 2402 \cs_new_protected:Npn \bool_gset_false:N #1
2403 { \cs_gset_eq:NN #1 \c_false_bool }
2404 \cs_generate_variant:Nn \bool_set_true:N { c }
2405 \cs_generate_variant:Nn \bool_set_false:N { c }
2406 \cs_generate_variant:Nn \bool_gset_true:N { c }
2407 \cs_generate_variant:Nn \bool_gset_false:N { c }

```

(End definition for `\bool_set_true:N` and others. These functions are documented on page 40.)

```

\bool_set_eq:NN The usual copy code.
\bool_set_eq:cN 2408 \cs_new_eq:NN \bool_set_eq:NN \cs_set_eq:NN
\bool_set_eq:Nc 2409 \cs_new_eq:NN \bool_set_eq:Nc \cs_set_eq:Nc
\bool_set_eq:cc 2410 \cs_new_eq:NN \bool_set_eq:cN \cs_set_eq:cN
\bool_gset_eq:NN 2411 \cs_new_eq:NN \bool_set_eq:cc \cs_set_eq:cc
\bool_gset_eq:cN 2412 \cs_new_eq:NN \bool_gset_eq:NN \cs_gset_eq:NN
\bool_gset_eq:Nc 2413 \cs_new_eq:NN \bool_gset_eq:Nc \cs_gset_eq:Nc
\bool_gset_eq:cN 2414 \cs_new_eq:NN \bool_gset_eq:cN \cs_gset_eq:cN
\bool_gset_eq:cc 2415 \cs_new_eq:NN \bool_gset_eq:cc \cs_gset_eq:cc

```

(End definition for `\bool_set_eq:NN` and others. These functions are documented on page 40.)

```

\bool_set:Nn This function evaluates a boolean expression and assigns the first argument the meaning
\bool_set:cn \c_true_bool or \c_false_bool.
\bool_gset:Nn 2416 \cs_new_protected:Npn \bool_set:Nn #1#2
\bool_gset:cn 2417 { \tex_chardef:D #1 = \bool_if_p:n {#2} }
2418 \cs_new_protected:Npn \bool_gset:Nn #1#2
2419 { \tex_global:D \tex_chardef:D #1 = \bool_if_p:n {#2} }
2420 \cs_generate_variant:Nn \bool_set:Nn { c }
2421 \cs_generate_variant:Nn \bool_gset:Nn { c }

```

(End definition for `\bool_set:Nn` and `\bool_set:cn`. These functions are documented on page 40.)

Booleans are not based on token lists but do need checking: this code complements similar material in `l3tl`.

```

2422 <*package>
2423 \if_bool:N \l@expl@check@declarations@bool
2424 \cs_set_protected:Npn \bool_set_true:N #1
2425 {
2426   \__chk_if_exist_var:N #1
2427   \cs_set_eq:NN #1 \c_true_bool
2428 }
2429 \cs_set_protected:Npn \bool_set_false:N #1
2430 {
2431   \__chk_if_exist_var:N #1

```

```

2432     \cs_set_eq:NN #1 \c_false_bool
2433   }
2434   \cs_set_protected:Npn \bool_gset_true:N #1
2435   {
2436     \__chk_if_exist_var:N #1
2437     \cs_gset_eq:NN #1 \c_true_bool
2438   }
2439   \cs_set_protected:Npn \bool_gset_false:N #1
2440   {
2441     \__chk_if_exist_var:N #1
2442     \cs_gset_eq:NN #1 \c_false_bool
2443   }
2444   \cs_set_protected:Npn \bool_set_eq:NN #1
2445   {
2446     \__chk_if_exist_var:N #1
2447     \cs_set_eq:NN #1
2448   }
2449   \cs_set_protected:Npn \bool_gset_eq:NN #1
2450   {
2451     \__chk_if_exist_var:N #1
2452     \cs_gset_eq:NN #1
2453   }
2454   \cs_set_protected:Npn \bool_set:Nn #1#2
2455   {
2456     \__chk_if_exist_var:N #1
2457     \tex_chardef:D #1 = \bool_if_p:n {#2}
2458   }
2459   \cs_set_protected:Npn \bool_gset:Nn #1#2
2460   {
2461     \__chk_if_exist_var:N #1
2462     \tex_global:D \tex_chardef:D #1 = \bool_if_p:n {#2}
2463   }
2464   \fi:
2465   \</package>

```

\bool_if_p:N Straight forward here. We could optimize here if we wanted to as the boolean can just be input directly.

```

\bool_if_p:c
\bool_if:NTF
\bool_if:cTF
2466   \prg_new_conditional:Npnn \bool_if:N #1 { p , T , F , TF }
2467   {
2468     \if_meaning:w \c_true_bool #1
2469     \prg_return_true:
2470   \else:
2471     \prg_return_false:
2472   \fi:
2473 }
2474 \cs_generate_variant:Nn \bool_if_p:N { c }
2475 \cs_generate_variant:Nn \bool_if:NT { c }
2476 \cs_generate_variant:Nn \bool_if:NF { c }
2477 \cs_generate_variant:Nn \bool_if:NTF { c }

```

(End definition for `\bool_if:N` and `\bool_if:cTF`. These functions are documented on page 40.)

```

\bool_show:N Show the truth value of the boolean, as true or false.
\bool_show:c 2478 \cs_new_protected:Npn \bool_show:N #1
\bool_show:n 2479 {
  2480   \__msg_show_variable:NNNnn #1 \bool_if_exist:NTF ? { }
  2481   { > ~ \token_to_str:N #1 = \__bool_to_str:n {#1} }
  2482 }
  2483 \cs_new_protected_nopar:Npn \bool_show:n
  2484 { \__msg_show_wrap:Nn \__bool_to_str:n }
  2485 \cs_new:Npn \__bool_to_str:n #1
  2486 { \bool_if:nTF {#1} { true } { false } }
  2487 \cs_generate_variant:Nn \bool_show:N { c }

```

(End definition for `\bool_show:N`, `\bool_show:c`, and `\bool_show:n`. These functions are documented on page 40.)

`\l_tmpa_bool` A few booleans just if you need them.

```

\l_tmpb_bool 2488 \bool_new:N \l_tmpa_bool
\g_tmpa_bool 2489 \bool_new:N \l_tmpb_bool
\g_tmppb_bool 2490 \bool_new:N \g_tmpa_bool
                2491 \bool_new:N \g_tmppb_bool

```

(End definition for `\l_tmpa_bool` and others. These variables are documented on page 40.)

```

\bool_if_exist_p:N Copies of the cs functions defined in l3basics.
\bool_if_exist_p:c 2492 \prg_new_eq_conditional:NNn \bool_if_exist:N \cs_if_exist:N
\bool_if_exist:NTF 2493 { TF , T , F , p }
\bool_if_exist:cTF 2494 \prg_new_eq_conditional:NNn \bool_if_exist:c \cs_if_exist:c
                2495 { TF , T , F , p }

```

(End definition for `\bool_if_exist:N` and `\bool_if_exist:cTF`. These functions are documented on page 40.)

5.4 Boolean expressions

`\bool_if_p:n` Evaluating the truth value of a list of predicates is done using an input syntax somewhat similar to the one found in other programming languages with `(` and `)` for grouping, `!` for logical “Not”, `&&` for logical “And” and `||` for logical “Or”. 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 $\langle true \rangle$ or $\langle false \rangle$.

The Eval function then contains a post-processing operation which grabs the instruction following the predicate. This is either And, Or or Close. In each case the truth value is used to determine where to go next. The following situations can arise:

$\langle true \rangle$ **And** Current truth value is true, logical And seen, continue with GetNext to examine truth value of next boolean (sub-)expression.

$\langle false \rangle$ **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 $\langle false \rangle$.

$\langle true \rangle$ **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 $\langle true \rangle$.

$\langle false \rangle$ **Or** Current truth value is false, logical Or seen, continue with GetNext to examine truth value of next boolean (sub-)expression.

$\langle true \rangle$ **Close** Current truth value is true, Close seen, return $\langle true \rangle$.

$\langle false \rangle$ **Close** Current truth value is false, Close seen, return $\langle false \rangle$.

We introduce an additional Stop operation with the same semantics as the Close operation.

$\langle true \rangle$ **Stop** Current truth value is true, return $\langle true \rangle$.

$\langle false \rangle$ **Stop** Current truth value is false, return $\langle false \rangle$.

The reasons for this follow below.

```

2496 \prg_new_conditional:Npnn \bool_if:n #1 { T , F , TF }
2497 {
2498   \if_predicate:w \bool_if_p:n {#1}
2499   \prg_return_true:
2500   \else:
2501     \prg_return_false:
2502   \fi:
2503 }
```

(End definition for `\bool_if:nTF`. This function is documented on page 42.)

```

\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 $\langle true \rangle ||$ is seen, but to the next `||` or closing parenthesis when $\langle false \rangle \&\&$ 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.

```

2504 \cs_new:Npn \bool_if_p:n #1
2505 {
2506   \group_align_safe_begin:
2507   \__bool_if_left_parentheses:wwn \q_nil
2508   #1 \q_mark { }
2509   ( \q_mark { \__bool_if_right_parentheses:wwn \q_nil }
2510   ) \q_mark { \__bool_if_or:wwn \q_nil }
2511   || \q_mark \__bool_if_parse:NNNww
2512   \q_stop
2513 }
2514 \cs_new:Npn \__bool_if_left_parentheses:wwn #1 \q_nil #2 ( #3 \q_mark #4
2515 { #4 \__bool_if_left_parentheses:wwn #1 #2 (( \q_nil #3 \q_mark {#4} }
2516 \cs_new:Npn \__bool_if_right_parentheses:wwn #1 \q_nil #2 ) #3 \q_mark #4
2517 { #4 \__bool_if_right_parentheses:wwn #1 #2 )) \q_nil #3 \q_mark {#4} }
2518 \cs_new:Npn \__bool_if_or:wwn #1 \q_nil #2 || #3 \q_mark #4
2519 { #4 \__bool_if_or:wwn #1 #2 )||( \q_nil #3 \q_mark {#4} }

```

(End definition for `\bool_if_p:n`. This function is documented on page ??.)

`__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.

```

2520 \cs_new:Npn \__bool_if_parse:NNNww #1#2#3#4 \q_mark #5 \q_stop
2521 {
2522   \__bool_get_next:NN \use_i:nn (( #4 )) S
2523 }

```

(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_ii: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.

```

2524 \cs_new:Npn \__bool_get_next:NN #1#2
2525 {
2526   \use:c

```

```

2527     {
2528         __bool_
2529         \if_meaning:w !#2 ! \else: \if_meaning:w (#2 ( \else: p \fi: \fi:
2530         :Nw
2531     }
2532     #1 #2
2533 }

```

(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.

```

2534 \cs_new:cpn { __bool_!:Nw } #1#2
2535 { \exp_after:wN \__bool_get_next:NN #1 \use_ii: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.

```

2536 \cs_new:cpn { __bool_(:Nw } #1#2
2537 {
2538     \exp_after:wN \__bool_choose:NNN \exp_after:wN #1
2539     \__int_value:w \__bool_get_next:NN \use_i:nn
2540 }

```

(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.

```

2541 \cs_new:cpn { __bool_p:Nw } #1
2542 { \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.

```

2543 \cs_new:Npn \__bool_choose:NNN #1#2#3
2544 {
2545     \use:c
2546     {
2547         __bool_ #3 _
2548         #1 #2 { \if_meaning:w 0 #2 1 \else: 0 \fi: }
2549         :w
2550     }
2551 }

```

(End definition for `__bool_choose:NNN`.)

`__bool_)_0:w` Closing a group is just about returning the result. The Stop operation is similar except
`__bool_)_1:w` it closes the special alignment group before returning the boolean.

`__bool_S_0:w` 2552 `\cs_new_nopar:cpn { __bool_)_0:w } { \c_false_bool }`
`__bool_S_1:w` 2553 `\cs_new_nopar:cpn { __bool_)_1:w } { \c_true_bool }`
2554 `\cs_new_nopar:cpn { __bool_S_0:w } { \group_align_safe_end: \c_false_bool }`
2555 `\cs_new_nopar:cpn { __bool_S_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` 2556 `\cs_new_nopar:cpn { __bool_&_1:w } & { __bool_get_next:NN \use_i:nn }`
2557 `\cs_new_nopar:cpn { __bool_|_0:w } | { __bool_get_next:NN \use_i:nn }`

(End definition for `__bool_&_1:w`.)

`__bool_&_0:w` When the truth value has already been decided, we have to throw away the remainder
`__bool_|_1:w` of the current group as we are doing minimal evaluation. This is slightly tricky as there
`__bool_eval_skip_to_end_auxi:Nw` are no braces so we have to play match the `()` manually.

`__bool_eval_skip_to_end_auxii:Nw` 2558 `\cs_new_nopar:cpn { __bool_&_0:w } &`
`__bool_eval_skip_to_end_auxiii:Nw` 2559 `{ __bool_eval_skip_to_end_auxi:Nw \c_false_bool }`
2560 `\cs_new_nopar:cpn { __bool_|_1:w } |`
2561 `{ __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.

```
2562 %% (
2563 \cs_new:Npn \__bool_eval_skip_to_end_auxi:Nw #1#2 )
2564 {
2565     \__bool_eval_skip_to_end_auxii:Nw #1#2 ( % )
2566     \q_no_value \q_stop
2567     {#2}
2568 }
```

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

```
2569 \cs_new:Npn \__bool_eval_skip_to_end_auxii:Nw #1#2 ( #3#4 \q_stop #5 % )
2570 {
2571     \quark_if_no_value:NTF #3
2572     {#1}
2573     { \__bool_eval_skip_to_end_auxiii:Nw #1 #5 }
2574 }
```

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!

```
2575 \cs_new:Npn \__bool_eval_skip_to_end_auxiii:Nw #1#2 ( #3 )
2576 { % (
2577     \__bool_eval_skip_to_end_auxi:Nw #1#3 )
2578 }
```

(End definition for __bool_&_0:w.)

\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.

```
2579 \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 42.)

\bool_xor_p:nn Exclusive or. If the boolean expressions have same truth value, return false, otherwise return true.

```

2580 \cs_new:Npn \bool_xor_p:nn #1#2
2581 {
2582   \int_compare:nNnTF { \bool_if_p:n {#1} } = { \bool_if_p:n {#2} }
2583     \c_false_bool
2584     \c_true_bool
2585 }

```

(End definition for \bool_xor_p:nn. This function is documented on page 42.)

5.5 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.

\bool_while_do:cn

\bool_until_do:Nn

\bool_until_do:cn

```

2586 \cs_new:Npn \bool_while_do:Nn #1#2
2587 { \bool_if:NT #1 { #2 \bool_while_do:Nn #1 {#2} } }
2588 \cs_new:Npn \bool_until_do:Nn #1#2
2589 { \bool_if:NF #1 { #2 \bool_until_do:Nn #1 {#2} } }
2590 \cs_generate_variant:Nn \bool_while_do:Nn { c }
2591 \cs_generate_variant:Nn \bool_until_do:Nn { c }

```

(End definition for \bool_while_do:Nn and \bool_while_do:cn. These functions are documented on page 43.)

\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.

\bool_do_while:cn

\bool_do_until:Nn

\bool_do_until:cn

```

2592 \cs_new:Npn \bool_do_while:Nn #1#2
2593 { #2 \bool_if:NT #1 { \bool_do_while:Nn #1 {#2} } }
2594 \cs_new:Npn \bool_do_until:Nn #1#2
2595 { #2 \bool_if:NF #1 { \bool_do_until:Nn #1 {#2} } }
2596 \cs_generate_variant:Nn \bool_do_while:Nn { c }
2597 \cs_generate_variant:Nn \bool_do_until:Nn { c }

```

(End definition for \bool_do_while:Nn and \bool_do_while:cn. These functions are documented on page 42.)

\bool_while_do:nn Loop functions with the test either before or after the first body expansion.

\bool_do_while:nn

\bool_until_do:nn

\bool_do_until:nn

```

2598 \cs_new:Npn \bool_while_do:nn #1#2
2599 {
2600   \bool_if:nT {#1}
2601   {
2602     #2
2603     \bool_while_do:nn {#1} {#2}
2604   }
2605 }
2606 \cs_new:Npn \bool_do_while:nn #1#2
2607 {
2608   #2

```

```

2609     \bool_if:nT {#1} { \bool_do_while:nn {#1} {#2} }
2610   }
2611   \cs_new:Npn \bool_until_do:nn #1#2
2612   {
2613     \bool_if:nF {#1}
2614     {
2615       #2
2616       \bool_until_do:nn {#1} {#2}
2617     }
2618   }
2619   \cs_new:Npn \bool_do_until:nn #1#2
2620   {
2621     #2
2622     \bool_if:nF {#1} { \bool_do_until:nn {#1} {#2} }
2623   }

```

(End definition for `\bool_while_do:nn` and others. These functions are documented on page 43.)

5.6 Producing multiple copies

```

2624 <@@=prg>

```

\prg_replicate:nn

__prg_replicate:N

__prg_replicate_first:N

__prg_replicate_

__prg_replicate_0:n

__prg_replicate_1:n

__prg_replicate_2:n

__prg_replicate_3:n

__prg_replicate_4:n

__prg_replicate_5:n

__prg_replicate_6:n

__prg_replicate_7:n

__prg_replicate_8:n

__prg_replicate_9:n

__prg_replicate_first_-:n

__prg_replicate_first_0:n

__prg_replicate_first_1:n

__prg_replicate_first_2:n

__prg_replicate_first_3:n

__prg_replicate_first_4:n

__prg_replicate_first_5:n

__prg_replicate_first_6:n

__prg_replicate_first_7:n

__prg_replicate_first_8:n

__prg_replicate_first_9:n

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.

```

2625 \cs_new:Npn \prg_replicate:nn #1
2626 {
2627   \exp:w
2628   \exp_after:wN __prg_replicate_first:N
2629   \__int_value:w \__int_eval:w #1 \__int_eval_end:
2630   \cs_end:

```


5.7 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 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.

```
2672 \prg_new_conditional:Npnn \mode_if_vertical: { p , T , F , TF }
2673 { \if_mode_vertical: \prg_return_true: \else: \prg_return_false: \fi: }
```

(End definition for `\mode_if_vertical:TF`. This function is documented on page 44.)

`\mode_if_horizontal_p:` For testing horizontal mode.

```
\mode_if_horizontal:TF 2674 \prg_new_conditional:Npnn \mode_if_horizontal: { p , T , F , TF }
2675 { \if_mode_horizontal: \prg_return_true: \else: \prg_return_false: \fi: }
```

(End definition for `\mode_if_horizontal:TF`. This function is documented on page 44.)

`\mode_if_inner_p:` For testing inner mode.

```
\mode_if_inner:TF 2676 \prg_new_conditional:Npnn \mode_if_inner: { p , T , F , TF }
2677 { \if_mode_inner: \prg_return_true: \else: \prg_return_false: \fi: }
```

(End definition for `\mode_if_inner:TF`. This function is documented on page 44.)

`\mode_if_math_p:` For testing math mode. At the beginning of an alignment cell, this should be used only inside a non-expandable function.

```
\mode_if_math:TF 2678 \prg_new_conditional:Npnn \mode_if_math: { p , T , F , TF }
2679 { \if_mode_math: \prg_return_true: \else: \prg_return_false: \fi: }
```

(End definition for `\mode_if_math:TF`. This function is documented on page 44.)

5.8 Internal programming functions

`\group_align_safe_begin:` TeX's alignment structures present many problems. As Knuth says himself in *TeX: 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 TeX 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 TeXbook*... 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.

```
2680 \cs_new_nopar:Npn \group_align_safe_begin:
2681 { \if_int_compare:w \if_false: { \fi: ' } = \c_zero \fi: }
2682 \cs_new_nopar:Npn \group_align_safe_end:
2683 { \if_int_compare:w '{ = \c_zero } \fi: }
```

(End definition for `\group_align_safe_begin:` and `\group_align_safe_end:.`)

2684 `<@@=prg>`

`\g__prg_map_int` A nesting counter for mapping.

2685 `\int_new:N \g__prg_map_int`

(End definition for `\g__prg_map_int`. This variable is documented on page 45.)

`__prg_break_point:Nn` These are defined in `l3basics`, as they are needed “early”. This is just a reminder that is the case!
`__prg_map_break:Nn`

(End definition for `__prg_break_point:Nn`. This function is documented on page 45.)

`__prg_break_point:` Also done in `l3basics` as in format mode these are needed within `l3alloc`.

`__prg_break:`

`__prg_break:n` (End definition for `__prg_break_point:.` This function is documented on page 45.)

5.9 Deprecated functions

`\scan_align_safe_stop:` Deprecated 2015-08-01 for removal after 2016-12-31.

2686 `\cs_new_protected_nopar:Npn \scan_align_safe_stop: { }`

(End definition for `\scan_align_safe_stop:.`)

2687 `</initex | package>`

6 l3quark implementation

The following test files are used for this code: `m3quark001.lvt`.

2688 `<*initex | package>`

6.1 Quarks

2689 `<@@=quark>`

`\quark_new:N` Allocate a new quark.

2690 `\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 47.)

`\q_nil` Some “public” quarks. `\q_stop` is an “end of argument” marker, `\q_nil` is a empty value
`\q_mark` and `\q_no_value` marks an empty argument.

`\q_no_value`

2691 `\quark_new:N \q_nil`

`\q_stop`

2692 `\quark_new:N \q_mark`

2693 `\quark_new:N \q_no_value`

2694 `\quark_new:N \q_stop`

(End definition for `\q_nil` and others. These variables are documented on page 47.)

`\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.

```
2695 \quark_new:N \q_recursion_tail
2696 \quark_new:N \q_recursion_stop
```

(End definition for `\q_recursion_tail` and `\q_recursion_stop`. These variables are documented on page 48.)

`\quark_if_recursion_tail_stop:N` When doing recursions, it is easy to spend a lot of time testing if the end marker has been found. To avoid this, a dedicated end marker is used each time a recursion is set up. Thus if the marker is found everything can be wrapper up and finished off. The simple case is when the test can guarantee that only a single token is being tested. In this case, there is just a dedicated copy of the standard quark test. Both a gobbling version and one inserting end code are provided.

```
2697 \cs_new:Npn \quark_if_recursion_tail_stop:N #1
2698 {
2699   \if_meaning:w \q_recursion_tail #1
2700   \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
2701   \fi:
2702 }
2703 \cs_new:Npn \quark_if_recursion_tail_stop_do:Nn #1
2704 {
2705   \if_meaning:w \q_recursion_tail #1
2706   \exp_after:wN \use_i_delimit_by_q_recursion_stop:nw
2707   \else:
2708   \exp_after:wN \use_none:n
2709   \fi:
2710 }
```

(End definition for `\quark_if_recursion_tail_stop:N`. This function is documented on page 48.)

`\quark_if_recursion_tail_stop:n` See `\quark_if_nil:nTF` for the details. Expanding `__quark_if_recursion_tail:w`
`\quark_if_recursion_tail_stop:o` once in front of the tokens chosen here gives an empty result if and only if `#1` is exactly
`\quark_if_recursion_tail_stop_do:nn` `\q_recursion_tail`.

```
\quark_if_recursion_tail_stop_do:nn
\__quark_if_recursion_tail:w
2711 \cs_new:Npn \quark_if_recursion_tail_stop:n #1
2712 {
2713   \tl_if_empty:oTF
2714   { \__quark_if_recursion_tail:w {} #1 {} ?! \q_recursion_tail ??! }
2715   { \use_none_delimit_by_q_recursion_stop:w }
2716   { }
2717 }
2718 \cs_new:Npn \quark_if_recursion_tail_stop_do:nn #1
2719 {
2720   \tl_if_empty:oTF
2721   { \__quark_if_recursion_tail:w {} #1 {} ?! \q_recursion_tail ??! }
2722   { \use_i_delimit_by_q_recursion_stop:nw }
2723   { \use_none:n }
```

```

2724 }
2725 \cs_new:Npn \__quark_if_recursion_tail:w
2726   #1 \q_recursion_tail #2 ? #3 ?! { #1 #2 }
2727 \cs_generate_variant:Nn \quark_if_recursion_tail_stop:n { o }
2728 \cs_generate_variant:Nn \quark_if_recursion_tail_stop_do:nn { o }

```

(End definition for `\quark_if_recursion_tail_stop:n` and `\quark_if_recursion_tail_stop:o`. These functions are documented on page 48.)

`__quark_if_recursion_tail_break:NN` `\quark_if_recursion_tail_stop... functions. Break the mapping using #2.`
`__quark_if_recursion_tail_break:nN`

```

2729 \cs_new:Npn \__quark_if_recursion_tail_break:NN #1#2
2730 {
2731   \if_meaning:w \q_recursion_tail #1
2732     \exp_after:wN #2
2733   \fi:
2734 }
2735 \cs_new:Npn \__quark_if_recursion_tail_break:nN #1#2
2736 {
2737   \tl_if_empty:oTF
2738     { \__quark_if_recursion_tail:w {} #1 {} ?! \q_recursion_tail ??! }
2739     { #2 }
2740     { }
2741 }

```

(End definition for `__quark_if_recursion_tail_break:NN`. This function is documented on page 49.)

```

\quark_if_nil_p:N    Here we test if we found a special quark as the first argument. We better start with
\quark_if_nil:NTF    \q_no_value as the first argument since the whole thing may otherwise loop if #1 is
\quark_if_no_value_p:N    wrongly given a string like aabc instead of a single token.6
\quark_if_no_value_p:c   
\quark_if_no_value:NTF   
\quark_if_no_value:cTF   
2742 \prg_new_conditional:Nnn \quark_if_nil:N { p, T , F , TF }
2743 {
2744   \if_meaning:w \q_nil #1
2745     \prg_return_true:
2746   \else:
2747     \prg_return_false:
2748   \fi:
2749 }
2750 \prg_new_conditional:Nnn \quark_if_no_value:N { p, T , F , TF }
2751 {
2752   \if_meaning:w \q_no_value #1
2753     \prg_return_true:
2754   \else:
2755     \prg_return_false:
2756   \fi:
2757 }
2758 \cs_generate_variant:Nn \quark_if_no_value_p:N { c }
2759 \cs_generate_variant:Nn \quark_if_no_value:NT { c }

```

⁶It may still loop in special circumstances however!

```

2760 \cs_generate_variant:Nn \quark_if_no_value:NF { c }
2761 \cs_generate_variant:Nn \quark_if_no_value:NTF { c }

```

(End definition for `\quark_if_nil:NTF`. This function is documented on page 47.)

`\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.

```

2762 \prg_new_conditional:Nnn \quark_if_nil:n { p , T , F , TF }
2763 {
2764   \__tl_if_empty_return:o
2765   { \__quark_if_nil:w {} #1 {} ? ! \q_nil ? ? ! }
2766 }
2767 \cs_new:Npn \__quark_if_nil:w #1 \q_nil #2 ? #3 ? ! { #1 #2 }
2768 \prg_new_conditional:Nnn \quark_if_no_value:n { p , T , F , TF }
2769 {
2770   \__tl_if_empty_return:o
2771   { \__quark_if_no_value:w {} #1 {} ? ! \q_no_value ? ? ! }
2772 }
2773 \cs_new:Npn \__quark_if_no_value:w #1 \q_no_value #2 ? #3 ? ! { #1 #2 }
2774 \cs_generate_variant:Nn \quark_if_nil_p:n { V , o }
2775 \cs_generate_variant:Nn \quark_if_nil:nTF { V , o }
2776 \cs_generate_variant:Nn \quark_if_nil:nT { V , o }
2777 \cs_generate_variant:Nn \quark_if_nil:nF { V , o }

```

(End definition for `\quark_if_nil:nTF`, `\quark_if_nil:VTF`, and `\quark_if_nil:oTF`. These functions are documented on page 47.)

`\q__tl_act_mark` These private quarks are needed by `l3tl`, but that is loaded before the quark module, hence their definition is deferred.

```

2778 \quark_new:N \q__tl_act_mark
2779 \quark_new:N \q__tl_act_stop

```

(End definition for `\q__tl_act_mark` and `\q__tl_act_stop`. These variables are documented on page ??.)

6.2 Scan marks

```

2780 <@@=scan>

```

`\g__scan_marks_tl` The list of all scan marks currently declared.

```

2781 \tl_new:N \g__scan_marks_tl

```

(End definition for `\g__scan_marks_tl`. This variable is documented on page ??.)

`__scan_new:N` Check whether the variable is already a scan mark, then declare it to be equal to `\scan_stop`: globally.

```

2782 \cs_new_protected:Npn \__scan_new:N #1
2783 {
2784   \tl_if_in:NnTF \g__scan_marks_tl { #1 }
2785   {
2786     \__msg_kernel_error:nxx { kernel } { scanmark-already-defined }
2787     { \token_to_str:N #1 }
2788   }
2789   {
2790     \tl_gput_right:Nn \g__scan_marks_tl {#1}
2791     \cs_new_eq:NN #1 \scan_stop:
2792   }
2793 }
```

(End definition for `__scan_new:N`.)

`\s__stop` We only declare one scan mark here, more can be defined by specific modules.

```

2794 \__scan_new:N \s__stop
```

(End definition for `\s__stop`. This variable is documented on page 50.)

`__use_none_delimit_by_s__stop:w` Similar to `\use_none_delimit_by_q_stop:w`.

```

2795 \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.

```

2796 \__scan_new:N \s__seq
```

(End definition for `\s__seq`. This variable is documented on page 130.)

```

2797 </initex | package>
```

7 l3token implementation

```

2798 <*initex | package>
```

```

2799 <@@=char>
```

7.1 Character tokens

`\char_set_catcode:n` Category code changes. Besides setting the catcode using `\tex_catcode:D`, this function keeps up to date the sequences `\l_char_active_seq`, `\l_char_special_seq`, and in package mode `\dospecials` (for plain \TeX and $\mathrm{L}\mathrm{A}\mathrm{T}\mathrm{E}\mathrm{X}\,2_{\epsilon}$) and `\@sanitize` ($\mathrm{L}\mathrm{A}\mathrm{T}\mathrm{E}\mathrm{X}\,2_{\epsilon}$ only) as well. The first list contains active characters, and the other three contain a single-character control sequence. These are constructed from the given character code `#1` using the `\lowercase` primitive. In package mode there are difficulties because `\+`

`\char_value_catcode:n`

`\char_show_value_catcode:n`

```

  \__char_set_catcode:Nn
  \__char_set_catcode:NnNNN
  \__char_set_catcode:NNN
```

and the active \sim L are `\outer` macros. This pushes us to only construct the single-character control sequence and the active character with character code `#1` if they are necessary for the update. As a start, construct an “other” (catcode 12) with character code `#1`. The second part of the `:Nn` auxiliary takes care of `\l_char_active_seq`: if the new catcode is 13 (active), then the active character is constructed through `\lowercase` and added to the sequence unless it is already in there. The first part is more intricate and distinguishes the case where the original catcode is special (different from letter or other, since for ϵ -TeX’s `\numexpr`, $11/2 = 12/2 = 6$) or not. If it the catcode was special, the single-character control sequence may need to be removed from the token lists and sequence, otherwise it may need to be added to the lists.

The addition or removal is done by the `:NnNN` auxiliary, which performs the catcode assignment (note here that the arguments of `\char_set_catcode:n` are each evaluated exactly once), then tests whether the new catcode is special or not. If that changed compared to the previous catcode, then some work is needed: pass to the `:NN` auxiliary the single-character control sequence, a function acting on token lists, and a function acting on sequences. Defining this auxiliary is tricky: the odd `f`-expansion expands the two `\tl_if_exist:NT` before the definition is performed, and the macro parameter token `#` (we are not yet doing a definition) stops the `f`-expansion. In L^AT_EX 2_ε, `\@sanitize` differs from `\dospecials` in that it should not list begin-group and end-group character tokens. For this, patch `__char_set_catcode:Nn` by copying its code using its `o`-expansion with arguments `#1` and `#2` (braces are needed because each argument is literally two tokens, a macro parameter character and a digit), and appending to it some code that removes the single-character control sequence from `\@sanitize` if the (new) catcode is 1 or 2.

```

2800 \group_begin:
2801 \tex_catcode:D '@ = \c_eleven
2802 \tex_catcode:D \c_zero = \c_twelve
2803 \cs_new_protected:Npn \char_set_catcode:n #1#2
2804 {
2805   \group_begin:
2806   \char_set_lccode:n { 0 } {#1}
2807   \tex_lowercase:D
2808   { \group_end: \__char_set_catcode:Nn ^~@ } {#2}
2809 }
2810 \tex_catcode:D \c_zero = \c_thirteen
2811 \cs_new_protected:Npn \__char_set_catcode:Nn #1#2
2812 {
2813   \int_compare:nNnTF { \tex_catcode:D '#1 / \c_two } = \c_six
2814   {
2815     \__char_set_catcode:NnNN #1 {#2} \int_compare:nNnF
2816     \tl_put_right:Nn \seq_put_right:Nn
2817   }
2818   {
2819     \__char_set_catcode:NnNN #1 {#2} \int_compare:nNnT
2820     \tl_remove_all:Nn \seq_remove_all:Nn
2821   }
2822   \int_compare:nNnT { \tex_catcode:D '#1 } = \c_thirteen
2823   {

```

```

2824     \group_begin:
2825     \char_set_lccode:n { 0 } { '#1 }
2826     \tex_lowercase:D
2827     {
2828     \group_end:
2829     \seq_if_in:NnF \l_char_active_seq { ^^@ }
2830     { \seq_put_right:Nn \l_char_active_seq { ^^@ } }
2831     }
2832   }
2833 }
2834 \cs_new_protected:Npn \__char_set_catcode:NnNNN #1#2#3#4#5
2835 {
2836   \tex_catcode:D '#1 = \__int_eval:w #2 \__int_eval_end:
2837   #3 { \tex_catcode:D '#1 / \c_two } = \c_six
2838   {
2839     \group_begin: \exp_args:Nnc \group_end:
2840     \__char_set_catcode:NNN {#1} #4 #5
2841   }
2842 }
2843 \exp_args:Nnf \use:n
2844 { \cs_new_protected:Npn \__char_set_catcode:NNN #1#2#3 }
2845 {
2846   <*package>
2847   \tl_if_exist:NT \dospecials
2848   {
2849     \tl_if_exist:NT \@sanitize { #2 \@sanitize { \@makeother #1 } }
2850     #2 \dospecials { \do #1 }
2851   }
2852   </package>
2853   #3 \l_char_special_seq {#1}
2854 }
2855 <*package>
2856 \tl_if_exist:NT \@sanitize
2857 {
2858   \exp_args:Nno \use:n
2859   { \cs_gset_protected:Npn \__char_set_catcode:Nn #1#2 }
2860   {
2861     \__char_set_catcode:Nn {#1} {#2}
2862     \int_compare:nNnT { \tex_catcode:D '#1 / \c_two } = \c_one
2863     {
2864       \group_begin: \exp_args:NNNx \group_end:
2865       \tl_remove_all:Nn \@sanitize
2866       { \exp_not:N \@makeother \exp_not:c {#1} }
2867     }
2868   }
2869 }
2870 </package>
2871 \cs_new:Npn \char_value_catcode:n #1
2872 { \tex_the:D \tex_catcode:D \__int_eval:w #1 \__int_eval_end: }
2873 \cs_new_protected:Npn \char_show_value_catcode:n #1

```

```

2874 { \_msg_show_wrap:n { > ~ \char_value_catcode:n {#1} } }
2875 \group_end:

```

(End definition for `\char_set_catcode:nn`. This function is documented on page 53.)

```

\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

2876 \cs_new_protected:Npn \char_set_catcode_escape:N #1
2877 { \char_set_catcode:nn { '#1 } \c_zero }
2878 \cs_new_protected:Npn \char_set_catcode_group_begin:N #1
2879 { \char_set_catcode:nn { '#1 } \c_one }
2880 \cs_new_protected:Npn \char_set_catcode_group_end:N #1
2881 { \char_set_catcode:nn { '#1 } \c_two }
2882 \cs_new_protected:Npn \char_set_catcode_math_toggle:N #1
2883 { \char_set_catcode:nn { '#1 } \c_three }
2884 \cs_new_protected:Npn \char_set_catcode_alignment:N #1
2885 { \char_set_catcode:nn { '#1 } \c_four }
2886 \cs_new_protected:Npn \char_set_catcode_end_line:N #1
2887 { \char_set_catcode:nn { '#1 } \c_five }
2888 \cs_new_protected:Npn \char_set_catcode_parameter:N #1
2889 { \char_set_catcode:nn { '#1 } \c_six }
2890 \cs_new_protected:Npn \char_set_catcode_math_superscript:N #1
2891 { \char_set_catcode:nn { '#1 } \c_seven }
2892 \cs_new_protected:Npn \char_set_catcode_math_subscript:N #1
2893 { \char_set_catcode:nn { '#1 } \c_eight }
2894 \cs_new_protected:Npn \char_set_catcode_ignore:N #1
2895 { \char_set_catcode:nn { '#1 } \c_nine }
2896 \cs_new_protected:Npn \char_set_catcode_space:N #1
2897 { \char_set_catcode:nn { '#1 } \c_ten }
2898 \cs_new_protected:Npn \char_set_catcode_letter:N #1
2899 { \char_set_catcode:nn { '#1 } \c_eleven }
2900 \cs_new_protected:Npn \char_set_catcode_other:N #1
2901 { \char_set_catcode:nn { '#1 } \c_twelve }
2902 \cs_new_protected:Npn \char_set_catcode_active:N #1
2903 { \char_set_catcode:nn { '#1 } \c_thirteen }
2904 \cs_new_protected:Npn \char_set_catcode_comment:N #1
2905 { \char_set_catcode:nn { '#1 } \c_fourteen }
2906 \cs_new_protected:Npn \char_set_catcode_invalid:N #1
2907 { \char_set_catcode:nn { '#1 } \c_fifteen }

```

(End definition for `\char_set_catcode_escape:N` and others. These functions are documented on page 52.)

```

\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

2908 \cs_new_protected:Npn \char_set_catcode_escape:n #1
2909 { \char_set_catcode:nn {#1} \c_zero }
2910 \cs_new_protected:Npn \char_set_catcode_group_begin:n #1
2911 { \char_set_catcode:nn {#1} \c_one }
2912 \cs_new_protected:Npn \char_set_catcode_group_end:n #1
2913 { \char_set_catcode:nn {#1} \c_two }
2914 \cs_new_protected:Npn \char_set_catcode_math_toggle:n #1

```

```

2915 { \char_set_catcode:nn {#1} \c_three }
2916 \cs_new_protected:Npn \char_set_catcode_alignment:n #1
2917 { \char_set_catcode:nn {#1} \c_four }
2918 \cs_new_protected:Npn \char_set_catcode_end_line:n #1
2919 { \char_set_catcode:nn {#1} \c_five }
2920 \cs_new_protected:Npn \char_set_catcode_parameter:n #1
2921 { \char_set_catcode:nn {#1} \c_six }
2922 \cs_new_protected:Npn \char_set_catcode_math_superscript:n #1
2923 { \char_set_catcode:nn {#1} \c_seven }
2924 \cs_new_protected:Npn \char_set_catcode_math_subscript:n #1
2925 { \char_set_catcode:nn {#1} \c_eight }
2926 \cs_new_protected:Npn \char_set_catcode_ignore:n #1
2927 { \char_set_catcode:nn {#1} \c_nine }
2928 \cs_new_protected:Npn \char_set_catcode_space:n #1
2929 { \char_set_catcode:nn {#1} \c_ten }
2930 \cs_new_protected:Npn \char_set_catcode_letter:n #1
2931 { \char_set_catcode:nn {#1} \c_eleven }
2932 \cs_new_protected:Npn \char_set_catcode_other:n #1
2933 { \char_set_catcode:nn {#1} \c_twelve }
2934 \cs_new_protected:Npn \char_set_catcode_active:n #1
2935 { \char_set_catcode:nn {#1} \c_thirteen }
2936 \cs_new_protected:Npn \char_set_catcode_comment:n #1
2937 { \char_set_catcode:nn {#1} \c_fourteen }
2938 \cs_new_protected:Npn \char_set_catcode_invalid:n #1
2939 { \char_set_catcode:nn {#1} \c_fifteen }

```

(End definition for `\char_set_catcode_escape:n` and others. These functions are documented on page 53.)

```

\char_set_mathcode:nn Pretty repetitive, but necessary!
\char_value_mathcode:n
\char_show_value_mathcode:n
\char_set_lccode:nn
\char_value_lccode:n
\char_show_value_lccode:n
\char_set_uccode:nn
\char_value_uccode:n
\char_show_value_uccode:n
\char_set_sfcode:nn
\char_value_sfcode:n
\char_show_value_sfcode:n
2940 \cs_new_protected:Npn \char_set_mathcode:nn #1#2
2941 {
2942   \tex_mathcode:D \__int_eval:w #1 \__int_eval_end:
2943   = \__int_eval:w #2 \__int_eval_end:
2944 }
2945 \cs_new:Npn \char_value_mathcode:n #1
2946 { \tex_the:D \tex_mathcode:D \__int_eval:w #1 \__int_eval_end: }
2947 \cs_new_protected:Npn \char_show_value_mathcode:n #1
2948 { \__msg_show_wrap:n { > ~ \char_value_mathcode:n {#1} } }
2949 \cs_new_protected:Npn \char_set_lccode:nn #1#2
2950 {
2951   \tex_lccode:D \__int_eval:w #1 \__int_eval_end:
2952   = \__int_eval:w #2 \__int_eval_end:
2953 }
2954 \cs_new:Npn \char_value_lccode:n #1
2955 { \tex_the:D \tex_lccode:D \__int_eval:w #1 \__int_eval_end: }
2956 \cs_new_protected:Npn \char_show_value_lccode:n #1
2957 { \__msg_show_wrap:n { > ~ \char_value_lccode:n {#1} } }
2958 \cs_new_protected:Npn \char_set_uccode:nn #1#2
2959 {

```

```

2960 \tex_uccode:D \__int_eval:w #1 \__int_eval_end:
2961 = \__int_eval:w #2 \__int_eval_end:
2962 }
2963 \cs_new:Npn \char_value_uccode:n #1
2964 { \tex_the:D \tex_uccode:D \__int_eval:w #1\__int_eval_end: }
2965 \cs_new_protected:Npn \char_show_value_uccode:n #1
2966 { \_msg_show_wrap:n { > ~ \char_value_uccode:n {#1} } }
2967 \cs_new_protected:Npn \char_set_sfcode:nn #1#2
2968 {
2969 \tex_sfcode:D \__int_eval:w #1 \__int_eval_end:
2970 = \__int_eval:w #2 \__int_eval_end:
2971 }
2972 \cs_new:Npn \char_value_sfcode:n #1
2973 { \tex_the:D \tex_sfcode:D \__int_eval:w #1\__int_eval_end: }
2974 \cs_new_protected:Npn \char_show_value_sfcode:n #1
2975 { \_msg_show_wrap:n { > ~ \char_value_sfcode:n {#1} } }

```

(End definition for `\char_set_mathcode:nn`. This function is documented on page 54.)

`\l_char_active_seq`
`\l_char_special_seq`

Two sequences for dealing with special characters. They must be defined before `\char_set_catcode:nn` can be used. The first is characters which may be active, and contains the active characters themselves to allow easy redefinition. The second longer list is for “special” characters more generally, and these 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`). The only complication is dealing with `_`, which requires the use of `\use:n` and `\use:nn`.

```

2976 \seq_new:N \l_char_special_seq
2977 \seq_set_split:Nnn \l_char_special_seq { }
2978 { \ \ " \# \$ \% \& \^ \_ \{ \} \~ }
2979 \seq_new:N \l_char_active_seq
2980 \use:n
2981 {
2982 \group_begin:
2983 \char_set_catcode_active:N \"
2984 \char_set_catcode_active:N \$
2985 \char_set_catcode_active:N \&
2986 \char_set_catcode_active:N \^
2987 \char_set_catcode_active:N \_
2988 \char_set_catcode_active:N \~
2989 \use:nn
2990 {
2991 \group_end:
2992 \seq_set_split:Nnn \l_char_active_seq { }
2993 }
2994 }
2995 { { " $ & ^ _ ~ } } %$

```

(End definition for `\l_char_active_seq` and `\l_char_special_seq`. These variables are documented on page 55.)

7.2 Generic tokens

2996 `<@@=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.

2997 `\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 55.)

`\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.

2998 `\group_begin:`

2999 `__chk_if_free_cs:N \c_group_begin_token`

3000 `\tex_global:D \tex_let:D \c_group_begin_token {`

3001 `__chk_if_free_cs:N \c_group_end_token`

3002 `\tex_global:D \tex_let:D \c_group_end_token }`

3003 `\char_set_catcode_math_toggle:N *`

3004 `\cs_new_eq:NN \c_math_toggle_token *`

3005 `\char_set_catcode_alignment:N *`

3006 `\cs_new_eq:NN \c_alignment_token *`

3007 `\cs_new_eq:NN \c_parameter_token #`

3008 `\cs_new_eq:NN \c_math_superscript_token ^`

3009 `\char_set_catcode_math_subscript:N *`

3010 `\cs_new_eq:NN \c_math_subscript_token *`

3011 `__chk_if_free_cs:N \c_space_token`

3012 `\use:n { \tex_global:D \tex_let:D \c_space_token = ~ } ~`

3013 `\cs_new_eq:NN \c_catcode_letter_token a`

3014 `\cs_new_eq:NN \c_catcode_other_token 1`

3015 `\group_end:`

(End definition for `\c_group_begin_token` and others. These functions are documented on page 56.)

`\c_catcode_active_tl` Not an implicit token!

3016 `\group_begin:`

3017 `\char_set_catcode_active:N *`

3018 `\tl_const:Nn \c_catcode_active_tl { \exp_not:N * }`

3019 `\group_end:`

(End definition for `\c_catcode_active_tl`. This variable is documented on page 56.)

7.3 Token conditionals

`\token_if_group_begin_p:N` Check if token is a begin group token. We use the constant `\c_group_begin_token` for this.
`\token_if_group_begin:NTF`

```

3020 \prg_new_conditional:Npnn \token_if_group_begin:N #1 { p , T , F , TF }
3021 {
3022   \if_catcode:w \exp_not:N #1 \c_group_begin_token
3023   \prg_return_true: \else: \prg_return_false: \fi:
3024 }

```

(End definition for `\token_if_group_begin:NTF`. This function is documented on page 56.)

`\token_if_group_end_p:N` Check if token is a end group token. We use the constant `\c_group_end_token` for this.
`\token_if_group_end:NTF`

```

3025 \prg_new_conditional:Npnn \token_if_group_end:N #1 { p , T , F , TF }
3026 {
3027   \if_catcode:w \exp_not:N #1 \c_group_end_token
3028   \prg_return_true: \else: \prg_return_false: \fi:
3029 }

```

(End definition for `\token_if_group_end:NTF`. This function is documented on page 57.)

`\token_if_math_toggle_p:N` Check if token is a math shift token. We use the constant `\c_math_toggle_token` for this.
`\token_if_math_toggle:NTF`

```

3030 \prg_new_conditional:Npnn \token_if_math_toggle:N #1 { p , T , F , TF }
3031 {
3032   \if_catcode:w \exp_not:N #1 \c_math_toggle_token
3033   \prg_return_true: \else: \prg_return_false: \fi:
3034 }

```

(End definition for `\token_if_math_toggle:NTF`. This function is documented on page 57.)

`\token_if_alignment_p:N` Check if token is an alignment tab token. We use the constant `\c_alignment_token` for this.
`\token_if_alignment:NTF`

```

3035 \prg_new_conditional:Npnn \token_if_alignment:N #1 { p , T , F , TF }
3036 {
3037   \if_catcode:w \exp_not:N #1 \c_alignment_token
3038   \prg_return_true: \else: \prg_return_false: \fi:
3039 }

```

(End definition for `\token_if_alignment:NTF`. This function is documented on page 57.)

`\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.

```

3040 \group_begin:
3041 \cs_set_eq:NN \c_parameter_token \scan_stop:
3042 \prg_new_conditional:Npnn \token_if_parameter:N #1 { p , T , F , TF }
3043 {
3044   \if_catcode:w \exp_not:N #1 \c_parameter_token

```

```

3045     \prg_return_true: \else: \prg_return_false: \fi:
3046   }
3047 \group_end:

```

(End definition for `\token_if_parameter:NTF`. This function is documented on page 57.)

`\token_if_math_superscript_p:N` Check if token is a math superscript token. We use the constant `\c_math_superscript_token`
`\token_if_math_superscript:NTF` token for this.

```

3048 \prg_new_conditional:Npnn \token_if_math_superscript:N #1
3049 { p , T , F , TF }
3050 {
3051   \if_catcode:w \exp_not:N #1 \c_math_superscript_token
3052   \prg_return_true: \else: \prg_return_false: \fi:
3053 }

```

(End definition for `\token_if_math_superscript:NTF`. This function is documented on page 57.)

`\token_if_math_subscript_p:N` Check if token is a math subscript token. We use the constant `\c_math_subscript_token`
`\token_if_math_subscript:NTF` token for this.

```

3054 \prg_new_conditional:Npnn \token_if_math_subscript:N #1 { p , T , F , TF }
3055 {
3056   \if_catcode:w \exp_not:N #1 \c_math_subscript_token
3057   \prg_return_true: \else: \prg_return_false: \fi:
3058 }

```

(End definition for `\token_if_math_subscript:NTF`. This function is documented on page 57.)

`\token_if_space_p:N` Check if token is a space token. We use the constant `\c_space_token` for this.

```

\token_if_space:NTF
3059 \prg_new_conditional:Npnn \token_if_space:N #1 { p , T , F , TF }
3060 {
3061   \if_catcode:w \exp_not:N #1 \c_space_token
3062   \prg_return_true: \else: \prg_return_false: \fi:
3063 }

```

(End definition for `\token_if_space:NTF`. This function is documented on page 57.)

`\token_if_letter_p:N` Check if token is a letter token. We use the constant `\c_catcode_letter_token` for this.

```

\token_if_letter:NTF
3064 \prg_new_conditional:Npnn \token_if_letter:N #1 { p , T , F , TF }
3065 {
3066   \if_catcode:w \exp_not:N #1 \c_catcode_letter_token
3067   \prg_return_true: \else: \prg_return_false: \fi:
3068 }

```

(End definition for `\token_if_letter:NTF`. This function is documented on page 57.)

`\token_if_other_p:N` Check if token is an other char token. We use the constant `\c_catcode_other_token`
`\token_if_other:NTF` for this.

```

3069 \prg_new_conditional:Npnn \token_if_other:N #1 { p , T , F , TF }
3070 {
3071   \if_catcode:w \exp_not:N #1 \c_catcode_other_token
3072   \prg_return_true: \else: \prg_return_false: \fi:
3073 }

```

(End definition for `\token_if_other:NTF`. This function is documented on page 57.)

`\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.

`\token_if_active:NTF`

```

3074 \prg_new_conditional:Npnn \token_if_active:N #1 { p , T , F , TF }
3075 {
3076   \if_catcode:w \exp_not:N #1 \c_catcode_active_tl
3077   \prg_return_true: \else: \prg_return_false: \fi:
3078 }

```

(End definition for `\token_if_active:NTF`. This function is documented on page 58.)

`\token_if_eq_meaning_p:NN` Check if the tokens #1 and #2 have same meaning.

`\token_if_eq_meaning:NNTF`

```

3079 \prg_new_conditional:Npnn \token_if_eq_meaning:NN #1#2 { p , T , F , TF }
3080 {
3081   \if_meaning:w #1 #2
3082   \prg_return_true: \else: \prg_return_false: \fi:
3083 }

```

(End definition for `\token_if_eq_meaning:NNTF`. This function is documented on page 58.)

`\token_if_eq_catcode_p:NN` Check if the tokens #1 and #2 have same category code.

`\token_if_eq_catcode:NNTF`

```

3084 \prg_new_conditional:Npnn \token_if_eq_catcode:NN #1#2 { p , T , F , TF }
3085 {
3086   \if_catcode:w \exp_not:N #1 \exp_not:N #2
3087   \prg_return_true: \else: \prg_return_false: \fi:
3088 }

```

(End definition for `\token_if_eq_catcode:NNTF`. This function is documented on page 58.)

`\token_if_eq_charcode_p:NN` Check if the tokens #1 and #2 have same character code.

`\token_if_eq_charcode:NNTF`

```

3089 \prg_new_conditional:Npnn \token_if_eq_charcode:NN #1#2 { p , T , F , TF }
3090 {
3091   \if_charcode:w \exp_not:N #1 \exp_not:N #2
3092   \prg_return_true: \else: \prg_return_false: \fi:
3093 }

```

(End definition for `\token_if_eq_charcode:NNTF`. This function is documented on page 58.)

`\token_if_macro_p:N` When a token is a macro, `\token_to_meaning:N` will always output something like

`\token_if_macro:NTF`

`_token_if_macro_p:w`

`\long macro:#1->#1` so we could naively check to see if the meaning contains `->`. 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.

```

3094 \use:x
3095 {
3096   \prg_new_conditional:Npnn \exp_not:N \token_if_macro:N #1
3097   { p , T , F , TF }
3098   {
3099     \exp_not:N \exp_after:wN \exp_not:N \__token_if_macro_p:w
3100     \exp_not:N \token_to_meaning:N ##1 \tl_to_str:n { ma : }
3101     \exp_not:N \q_stop
3102   }
3103   \cs_new:Npn \exp_not:N \__token_if_macro_p:w
3104     ##1 \tl_to_str:n { ma } ##2 \c_colon_str ##3 \exp_not:N \q_stop
3105 }
3106 {
3107   \if_int_compare:w \__str_if_eq_x:nn { #2 } { cro } = \c_zero
3108   \prg_return_true:
3109   \else:
3110     \prg_return_false:
3111   \fi:
3112 }

```

(End definition for `\token_if_macro:NTF`. This function is documented on page 58.)

`\token_if_cs_p:N` Check if token has same catcode as a control sequence. This follows the same pattern as
`\token_if_cs:NTF` for `\token_if_letter:N` etc. We use `\scan_stop:` for this.

```

3113 \prg_new_conditional:Npnn \token_if_cs:N #1 { p , T , F , TF }
3114 {
3115   \if_catcode:w \exp_not:N #1 \scan_stop:
3116   \prg_return_true: \else: \prg_return_false: \fi:
3117 }

```

(End definition for `\token_if_cs:NTF`. This function is documented on page 58.)

`\token_if_expandable_p:N` Check if token is expandable. We use the fact that T_EX will temporarily convert `\exp_`
`\token_if_expandable:NTF` `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 T_EX's conditional apparatus).

```

3118 \prg_new_conditional:Npnn \token_if_expandable:N #1 { p , T , F , TF }
3119 {
3120   \exp_after:wN \if_meaning:w \exp_not:N #1 #1
3121   \prg_return_false:
3122   \else:
3123     \if_cs_exist:N #1
3124     \prg_return_true:
3125   \else:

```

```

3126         \prg_return_false:
3127         \fi:
3128     \fi:
3129 }

```

(End definition for `\token_if_expandable:N`. This function is documented on page 58.)

`\token_if_chardef_p:N` Most of these functions have to check the meaning of the token in question so we need to do some checkups on which characters are output by `\token_to_meaning:N`. As usual, these characters have catcode 12 so we must do some serious substitutions in the code below...

```

\token_if_muskip_register_p:N
\token_if_skip_register_p:N
\token_if_toks_register_p:N
\token_if_long_macro_p:N
\token_if_protected_macro_p:N
\token_if_protected_long_macro_p:N
\token_if_chardef:NTF
\token_if_mathchardef:NTF
\token_if_dim_register:NTF
\token_if_int_register:NTF
\token_if_muskip_register:NTF
\token_if_skip_register:NTF
\token_if_toks_register:NTF
\token_if_long_macro:NTF
\token_if_protected_macro:NTF
\token_if_protected_long_macro:NTF
__token_if_chardef:w
__token_if_dim_register:w
__token_if_int_register:w
__token_if_muskip_register:w
__token_if_skip_register:w
__token_if_toks_register:w
__token_if_protected_macro:w
__token_if_long_macro:w

```

```

3130 \group_begin:
3131 \char_set_lccode:nn { 'T } { 'T }
3132 \char_set_lccode:nn { 'F } { 'F }
3133 \char_set_lccode:nn { 'X } { 'n }
3134 \char_set_lccode:nn { 'Y } { 't }
3135 \char_set_lccode:nn { 'Z } { 'd }
3136 \tl_map_inline:nn { A C E G H I K L M O P R S U X Y Z R " }
3137 { \char_set_catcode:nn { '#1 } \c_twelve }

```

We convert the token list to lower case and restore the catcode and lowercase code changes.

```

3138 \tex_lowercase:D
3139 {
3140 \group_end:

```

First up is checking if something has been defined with `\chardef` or `\mathchardef`. This is easy since \TeX thinks of such tokens as hexadecimal so it stores them as `\char"<hex number>` or `\mathchar"<hex number>`. Grab until the first occurrence of `char"`, and compare what precedes with `\` or `\math`. In fact, the escape character may not be a backslash, so we compare with the result of converting some other control sequence to a string, namely `\char` or `\mathchar` (the auxiliary adds the `char` back).

```

3141 \prg_new_conditional:Npnn \token_if_chardef:N #1 { p , T , F , TF }
3142 {
3143     \__str_if_eq_x_return:nn
3144     {
3145         \exp_after:wN \__token_if_chardef:w
3146         \token_to_meaning:N #1 CHAR" \q_stop
3147     }
3148     { \token_to_str:N \char }
3149 }
3150 \prg_new_conditional:Npnn \token_if_mathchardef:N #1 { p , T , F , TF }
3151 {
3152     \__str_if_eq_x_return:nn
3153     {
3154         \exp_after:wN \__token_if_chardef:w
3155         \token_to_meaning:N #1 CHAR" \q_stop
3156     }
3157     { \token_to_str:N \mathchar }
3158 }

```

```

3159 \cs_new:Npn \__token_if_chardef:w #1 CHAR" #2 \q_stop { #1 CHAR }

```

Dim registers are a bit more difficult since their `\meaning` has the form `\dimen⟨number⟩`, and we must take care of the two primitives `\dimen` and `\dimendef`.

```

3160 \prg_new_conditional:Npnn \token_if_dim_register:N #1 { p , T , F , TF }
3161 {
3162   \if_meaning:w \tex_dimen:D #1
3163   \prg_return_false:
3164   \else:
3165     \if_meaning:w \tex_dimendef:D #1
3166     \prg_return_false:
3167     \else:
3168       \__str_if_eq_x_return:nn
3169       {
3170         \exp_after:wN \__token_if_dim_register:w
3171         \token_to_meaning:N #1 ZIMEX \q_stop
3172       }
3173       { \token_to_str:N \ }
3174     \fi:
3175   \fi:
3176 }
3177 \cs_new:Npn \__token_if_dim_register:w #1 ZIMEX #2 \q_stop { #1 ~ }

```

Integer registers are one step harder since constants are implemented differently from variables, and we also have to take care of the primitives `\count` and `\countdef`.

```

3178 \prg_new_conditional:Npnn \token_if_int_register:N #1 { p , T , F , TF }
3179 {
3180   % \token_if_chardef:NTF #1 { \prg_return_true: }
3181   % {
3182   %   \token_if_mathchardef:NTF #1 { \prg_return_true: }
3183   %   {
3184   \if_meaning:w \tex_count:D #1
3185   \prg_return_false:
3186   \else:
3187     \if_meaning:w \tex_countdef:D #1
3188     \prg_return_false:
3189     \else:
3190       \__str_if_eq_x_return:nn
3191       {
3192         \exp_after:wN \__token_if_int_register:w
3193         \token_to_meaning:N #1 COUXY \q_stop
3194       }
3195       { \token_to_str:N \ }
3196     \fi:
3197   \fi:
3198   % }
3199   % }
3200 }
3201 \cs_new:Npn \__token_if_int_register:w #1 COUXY #2 \q_stop { #1 ~ }

```

Muskip registers are done the same way as the dimension registers.

```

3202 \prg_new_conditional:Npnn \token_if_muskip_register:N #1
3203 { p , T , F , TF }
3204 {
3205   \if_meaning:w \tex_muskip:D #1
3206   \prg_return_false:
3207   \else:
3208     \if_meaning:w \tex_muskipdef:D #1
3209     \prg_return_false:
3210     \else:
3211       \__str_if_eq_x_return:nn
3212       {
3213         \exp_after:wN \__token_if_muskip_register:w
3214         \token_to_meaning:N #1 MUSKIP \q_stop
3215       }
3216       { \token_to_str:N \ }
3217     \fi:
3218   \fi:
3219 }
3220 \cs_new:Npn \__token_if_muskip_register:w #1 MUSKIP #2 \q_stop { #1 ~ }

```

Skip registers.

```

3221 \prg_new_conditional:Npnn \token_if_skip_register:N #1
3222 { p , T , F , TF }
3223 {
3224   \if_meaning:w \tex_skip:D #1
3225   \prg_return_false:
3226   \else:
3227     \if_meaning:w \tex_skipdef:D #1
3228     \prg_return_false:
3229     \else:
3230       \__str_if_eq_x_return:nn
3231       {
3232         \exp_after:wN \__token_if_skip_register:w
3233         \token_to_meaning:N #1 SKIP \q_stop
3234       }
3235       { \token_to_str:N \ }
3236     \fi:
3237   \fi:
3238 }
3239 \cs_new:Npn \__token_if_skip_register:w #1 SKIP #2 \q_stop { #1 ~ }

```

Toks registers.

```

3240 \prg_new_conditional:Npnn \token_if_toks_register:N #1
3241 { p , T , F , TF }
3242 {
3243   \if_meaning:w \tex_toks:D #1
3244   \prg_return_false:
3245   \else:
3246     \if_meaning:w \tex_toksdef:D #1
3247     \prg_return_false:
3248   \else:

```

```

3249         \__str_if_eq_x_return:nn
3250         {
3251             \exp_after:wN \__token_if_toks_register:w
3252             \token_to_meaning:N #1 YOKS \q_stop
3253         }
3254         { \token_to_str:N \ }
3255     \fi:
3256 \fi:
3257 }
3258 \cs_new:Npn \__token_if_toks_register:w #1 YOKS #2 \q_stop { #1 ~ }

```

Protected macros.

```

3259 \prg_new_conditional:Npnn \token_if_protected_macro:N #1
3260 { p , T , F , TF }
3261 {
3262     \__str_if_eq_x_return:nn
3263     {
3264         \exp_after:wN \__token_if_protected_macro:w
3265         \token_to_meaning:N #1 PROYECYEEZ~MACRO \q_stop
3266     }
3267     { \token_to_str:N \ }
3268 }
3269 \cs_new:Npn \__token_if_protected_macro:w
3270 #1 PROYECYEEZ~MACRO #2 \q_stop { #1 ~ }

```

Long macros and protected long macros share an auxiliary.

```

3271 \prg_new_conditional:Npnn \token_if_long_macro:N #1 { p , T , F , TF }
3272 {
3273     \__str_if_eq_x_return:nn
3274     {
3275         \exp_after:wN \__token_if_long_macro:w
3276         \token_to_meaning:N #1 LOXG~MACRO \q_stop
3277     }
3278     { \token_to_str:N \ }
3279 }
3280 \prg_new_conditional:Npnn \token_if_protected_long_macro:N #1
3281 { p , T , F , TF }
3282 {
3283     \__str_if_eq_x_return:nn
3284     {
3285         \exp_after:wN \__token_if_long_macro:w
3286         \token_to_meaning:N #1 LOXG~MACRO \q_stop
3287     }
3288     { \token_to_str:N \protected \token_to_str:N \ }
3289 }
3290 \cs_new:Npn \__token_if_long_macro:w #1 LOXG~MACRO #2 \q_stop { #1 ~ }

```

Finally the `\tex_lowercase:D` ends!

```

3291 }

```

(End definition for `\token_if_chardef:NTF` and others. These functions are documented on page 59.)

```

\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 inexistent 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.

```

3292 \tex_chardef:D \c__token_A_int = 'A ~ %
3293 \use:x
3294 {
3295   \prg_new_conditional:Npnn \exp_not:N \token_if_primitive:N ##1
3296   { p , T , F , TF }
3297   {
3298     \exp_not:N \token_if_macro:NTF ##1
3299     \exp_not:N \prg_return_false:
3300     {
3301       \exp_not:N \exp_after:wN \exp_not:N \__token_if_primitive:NNw
3302       \exp_not:N \token_to_meaning:N ##1
3303       \tl_to_str:n { : : : } \exp_not:N \q_stop ##1
3304     }
3305   }
3306   \cs_new:Npn \exp_not:N \__token_if_primitive:NNw
3307   ##1##2 ##3 \c_colon_str ##4 \exp_not:N \q_stop
3308   {
3309     \exp_not:N \tl_if_empty:oTF
3310     { \exp_not:N \__token_if_primitive_space:w ##3 ~ }
3311     {
3312       \exp_not:N \__token_if_primitive_loop:N ##3
3313       \c_colon_str \exp_not:N \q_stop
3314     }

```

```

3315         { \exp_not:N \__token_if_primitive_nullfont:N }
3316     }
3317 }
3318 \cs_new:Npn \__token_if_primitive_space:w #1 ~ { }
3319 \cs_new:Npn \__token_if_primitive_nullfont:N #1
3320 {
3321     \if_meaning:w \tex_nullfont:D #1
3322     \prg_return_true:
3323 }else:
3324     \prg_return_false:
3325 \fi:
3326 }
3327 \cs_new:Npn \__token_if_primitive_loop:N #1
3328 {
3329     \if_int_compare:w '#1 < \c__token_A_int %
3330     \exp_after:wN \__token_if_primitive:Nw
3331     \exp_after:wN #1
3332 }else:
3333     \exp_after:wN \__token_if_primitive_loop:N
3334 \fi:
3335 }
3336 \cs_new:Npn \__token_if_primitive:Nw #1 #2 \q_stop
3337 {
3338     \if:w : #1
3339     \exp_after:wN \__token_if_primitive_undefined:N
3340 }else:
3341     \prg_return_false:
3342     \exp_after:wN \use_none:n
3343 \fi:
3344 }
3345 \cs_new:Npn \__token_if_primitive_undefined:N #1
3346 {
3347     \if_cs_exist:N #1
3348     \prg_return_true:
3349 }else:
3350     \prg_return_false:
3351 \fi:
3352 }

```

(End definition for `\token_if_primitive:NTF`. This function is documented on page 60.)

7.4 Peeking ahead at the next token

```

3353 <@@=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` 3354 `\cs_new_eq:NN \l_peek_token ?`
 3355 `\cs_new_eq:NN \g_peek_token ?`

(End definition for \l_peek_token. This variable is documented on page 60.)

`\l__peek_search_token` The token to search for as an implicit token: cf. `\l__peek_search_tl`.

3356 `\cs_new_eq:NN \l__peek_search_token ?`

(End definition for \l__peek_search_token. This variable is documented on page ??.)

`\l__peek_search_tl` The token to search for as an explicit token: cf. `\l__peek_search_token`.

3357 `\tl_new:N \l__peek_search_tl`

(End definition for \l__peek_search_tl. This variable is documented on page ??.)

`__peek_true:w` Functions used by the branching and space-stripping code.

`__peek_true_aux:w` 3358 `\cs_new_nopar:Npn __peek_true:w { }`
`__peek_false:w` 3359 `\cs_new_nopar:Npn __peek_true_aux:w { }`
`__peek_tmp:w` 3360 `\cs_new_nopar:Npn __peek_false:w { }`
 3361 `\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` 3362 `\cs_new_protected_nopar:Npn \peek_after:Nw`
 3363 `{ \tex_futurelet:D \l_peek_token }`
 3364 `\cs_new_protected_nopar:Npn \peek_gafter:Nw`
 3365 `{ \tex_global:D \tex_futurelet:D \g_peek_token }`

(End definition for \peek_after:Nw. This function is documented on page 60.)

`__peek_true_remove:w` A function to remove the next token and then regain control.

3366 `\cs_new_protected:Npn __peek_true_remove:w`
 3367 `{`
 3368 `\group_align_safe_end:`
 3369 `\tex_afterassignment:D __peek_true_aux:w`
 3370 `\cs_set_eq:NN __peek_tmp:w`
 3371 `}`

(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.

```

3372 \cs_new_protected:Npn \__peek_token_generic:NNTF #1#2#3#4
3373 {
3374   \cs_set_eq:NN \l__peek_search_token #2
3375   \tl_set:Nn \l__peek_search_tl {#2}
3376   \cs_set_nopar:Npx \__peek_true:w
3377   {
3378     \exp_not:N \group_align_safe_end:
3379     \exp_not:n {#3}
3380   }
3381   \cs_set_nopar:Npx \__peek_false:w
3382   {
3383     \exp_not:N \group_align_safe_end:
3384     \exp_not:n {#4}
3385   }
3386   \group_align_safe_begin:
3387   \peek_after:Nw #1
3388 }
3389 \cs_new_protected:Npn \__peek_token_generic:NNT #1#2#3
3390 { \__peek_token_generic:NNTF #1 #2 {#3} { } }
3391 \cs_new_protected:Npn \__peek_token_generic:NNF #1#2#3
3392 { \__peek_token_generic:NNTF #1 #2 { } {#3} }

```

(End definition for `__peek_token_generic:NNTF`. This function is documented on page ??.)

`__peek_token_remove_generic:NNTF` For token removal there needs to be a call to the auxiliary function which does the work.

```

3393 \cs_new_protected:Npn \__peek_token_remove_generic:NNTF #1#2#3#4
3394 {
3395   \cs_set_eq:NN \l__peek_search_token #2
3396   \tl_set:Nn \l__peek_search_tl {#2}
3397   \cs_set_eq:NN \__peek_true:w \__peek_true_remove:w
3398   \cs_set_nopar:Npx \__peek_true_aux:w { \exp_not:n {#3} }
3399   \cs_set_nopar:Npx \__peek_false:w
3400   {
3401     \exp_not:N \group_align_safe_end:
3402     \exp_not:n {#4}
3403   }
3404   \group_align_safe_begin:
3405   \peek_after:Nw #1
3406 }
3407 \cs_new_protected:Npn \__peek_token_remove_generic:NNT #1#2#3
3408 { \__peek_token_remove_generic:NNTF #1 #2 {#3} { } }
3409 \cs_new_protected:Npn \__peek_token_remove_generic:NNF #1#2#3
3410 { \__peek_token_remove_generic:NNTF #1 #2 { } {#3} }

```

(End definition for `__peek_token_remove_generic:NNTF`. This function is documented on page ??.)

`__peek_execute_branches_meaning:` The meaning test is straight forward.

```

3411 \cs_new_nopar:Npn \__peek_execute_branches_meaning:
3412 {
3413   \if_meaning:w \l_peek_token \l__peek_search_token
3414   \exp_after:wN \__peek_true:w
3415   \else:
3416     \exp_after:wN \__peek_false:w
3417   \fi:
3418 }
```

(End definition for __peek_execute_branches_meaning:. This function is documented on page ??.)

`__peek_execute_branches_catcode:` The catcode and charcode tests are very similar, and in order to use the same auxiliaries we do something a little bit odd, firing `\if_catcode:w` and `\if_charcode:w` before finding the operands for those tests, which 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 T_EX's `\futurelet`, because we can only access the `\meaning` of tokens in that way. In those cases, detected thanks to a comparison with `\scan_stop:`, we grab the following token, and compare it explicitly with the explicit search token stored in `\l__peek_search_tl`. The `\exp_not:N` prevents outer macros (coming from non-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.

```

3419 \cs_new_nopar:Npn \__peek_execute_branches_catcode:
3420 { \if_catcode:w \__peek_execute_branches_catcode_aux: }
3421 \cs_new_nopar:Npn \__peek_execute_branches_charcode:
3422 { \if_charcode:w \__peek_execute_branches_catcode_aux: }
3423 \cs_new_nopar:Npn \__peek_execute_branches_catcode_aux:
3424 {
3425   \if_catcode:w \exp_not:N \l_peek_token \scan_stop:
3426   \exp_after:wN \exp_after:wN
3427   \exp_after:wN \__peek_execute_branches_catcode_auxii:N
3428   \exp_after:wN \exp_not:N
3429   \else:
3430     \exp_after:wN \__peek_execute_branches_catcode_auxiii:
3431   \fi:
3432 }
3433 \cs_new:Npn \__peek_execute_branches_catcode_auxii:N #1
```

```

3434 {
3435     \exp_not:N #1
3436     \exp_after:wN \exp_not:N \l__peek_search_tl
3437     \exp_after:wN \__peek_true:w
3438     \else:
3439     \exp_after:wN \__peek_false:w
3440     \fi:
3441     #1
3442 }
3443 \cs_new_nopar:Npn \__peek_execute_branches_catcode_auxiii:
3444 {
3445     \exp_not:N \l_peek_token
3446     \exp_after:wN \exp_not:N \l__peek_search_tl
3447     \exp_after:wN \__peek_true:w
3448     \else:
3449     \exp_after:wN \__peek_false:w
3450     \fi:
3451 }

```

(End definition for `__peek_execute_branches_catcode:` and `__peek_execute_branches_charcode:`. These functions are documented on page ??.)

`__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.

```

3452 \cs_new_protected_nopar:Npn \__peek_ignore_spaces_execute_branches:
3453 {
3454     \if_meaning:w \l_peek_token \c_space_token
3455     \exp_after:wN \peek_after:Nw
3456     \exp_after:wN \__peek_ignore_spaces_execute_branches:
3457     \exp:w \exp_end_continue_f:w
3458     \else:
3459     \exp_after:wN \__peek_execute_branches:
3460     \fi:
3461 }

```

(End definition for `__peek_ignore_spaces_execute_branches:`. This function is documented on page ??.)

`__peek_def:nnnn` The public functions themselves cannot be defined using `\prg_new_conditional:Npnn`
`__peek_def:nnnnn` 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.

```

3462 \group_begin:
3463 \cs_set:Npn \__peek_def:nnnn #1#2#3#4
3464 {
3465     \__peek_def:nnnnn {#1} {#2} {#3} {#4} { TF }
3466     \__peek_def:nnnnn {#1} {#2} {#3} {#4} { T }
3467     \__peek_def:nnnnn {#1} {#2} {#3} {#4} { F }

```

```

3468     }
3469 \cs_set:Npn \__peek_def:nnnnn #1#2#3#4#5
3470 {
3471     \cs_new_protected_nopar:cpx { #1 #5 }
3472     {
3473         \tl_if_empty:nF {#2}
3474         { \exp_not:n { \cs_set_eq:NN \__peek_execute_branches: #2 } }
3475         \exp_not:c { #3 #5 }
3476         \exp_not:n {#4}
3477     }
3478 }

```

(End definition for __peek_def:nnnn.)

\peek_catcode:NTF With everything in place the definitions can take place. First for category codes.
\peek_catcode_ignore_spaces:NTF
\peek_catcode_remove:NTF
\peek_catcode_remove_ignore_spaces:NTF

```

3479 \__peek_def:nnnn { peek_catcode:N }
3480 { }
3481 { __peek_token_generic:NN }
3482 { \__peek_execute_branches_catcode: }
3483 \__peek_def:nnnn { peek_catcode_ignore_spaces:N }
3484 { \__peek_execute_branches_catcode: }
3485 { __peek_token_generic:NN }
3486 { \__peek_ignore_spaces_execute_branches: }
3487 \__peek_def:nnnn { peek_catcode_remove:N }
3488 { }
3489 { __peek_token_remove_generic:NN }
3490 { \__peek_execute_branches_catcode: }
3491 \__peek_def:nnnn { peek_catcode_remove_ignore_spaces:N }
3492 { \__peek_execute_branches_catcode: }
3493 { __peek_token_remove_generic:NN }
3494 { \__peek_ignore_spaces_execute_branches: }

```

(End definition for \peek_catcode:NTF and others. These functions are documented on page 60.)

\peek_charcode:NTF Then for character codes.
\peek_charcode_ignore_spaces:NTF
\peek_charcode_remove:NTF
\peek_charcode_remove_ignore_spaces:NTF

```

3495 \__peek_def:nnnn { peek_charcode:N }
3496 { }
3497 { __peek_token_generic:NN }
3498 { \__peek_execute_branches_charcode: }
3499 \__peek_def:nnnn { peek_charcode_ignore_spaces:N }
3500 { \__peek_execute_branches_charcode: }
3501 { __peek_token_generic:NN }
3502 { \__peek_ignore_spaces_execute_branches: }
3503 \__peek_def:nnnn { peek_charcode_remove:N }
3504 { }
3505 { __peek_token_remove_generic:NN }
3506 { \__peek_execute_branches_charcode: }
3507 \__peek_def:nnnn { peek_charcode_remove_ignore_spaces:N }
3508 { \__peek_execute_branches_charcode: }
3509 { __peek_token_remove_generic:NN }
3510 { \__peek_ignore_spaces_execute_branches: }

```

(End definition for `\peek_charcode:N` and others. These functions are documented on page 61.)

`\peek_meaning:N`TF Finally for meaning, with the group closed to remove the temporary definition functions.

```

\peek_meaning_ignore_spaces:N
```

```

\peek_meaning_remove:N
```

```

\peek_meaning_remove_ignore_spaces:N
```

```

3511 \__peek_def:nnnn { peek_meaning:N }
3512 { }
3513 { \__peek_token_generic:NN }
3514 { \__peek_execute_branches_meaning: }
3515 \__peek_def:nnnn { peek_meaning_ignore_spaces:N }
3516 { \__peek_execute_branches_meaning: }
3517 { \__peek_token_generic:NN }
3518 { \__peek_ignore_spaces_execute_branches: }
3519 \__peek_def:nnnn { peek_meaning_remove:N }
3520 { }
3521 { \__peek_token_remove_generic:NN }
3522 { \__peek_execute_branches_meaning: }
3523 \__peek_def:nnnn { peek_meaning_remove_ignore_spaces:N }
3524 { \__peek_execute_branches_meaning: }
3525 { \__peek_token_remove_generic:NN }
3526 { \__peek_ignore_spaces_execute_branches: }
3527 \group_end:
```

(End definition for `\peek_meaning:N` and others. These functions are documented on page 62.)

7.5 Decomposing a macro definition

`\token_get_prefix_spec:N` We sometimes want to test if a control sequence can be expanded to reveal a hidden value.
`\token_get_arg_spec:N` However, we cannot just expand the macro blindly as it may have arguments and none
`\token_get_replacement_spec:N` might be present. Therefore we define these functions to pick either the prefix(es), the
`__peek_get_prefix_arg_replacement:wN` 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.

```

3528 \exp_args:Nno \use:nn
3529 { \cs_new:Npn \__peek_get_prefix_arg_replacement:wN #1 }
3530 { \tl_to_str:n { macro : } #2 -> #3 \q_stop #4 }
3531 { #4 {#1} {#2} {#3} }
3532 \cs_new:Npn \token_get_prefix_spec:N #1
3533 {
3534   \token_if_macro:NTF #1
3535   {
3536     \exp_after:wN \__peek_get_prefix_arg_replacement:wN
3537     \token_to_meaning:N #1 \q_stop \use_i:nnn
3538   }
3539   { \scan_stop: }
3540 }
3541 \cs_new:Npn \token_get_arg_spec:N #1
3542 {
3543   \token_if_macro:NTF #1
3544   {
3545     \exp_after:wN \__peek_get_prefix_arg_replacement:wN
```

```

3546         \token_to_meaning:N #1 \q_stop \use_ii:nnn
3547     }
3548     { \scan_stop: }
3549 }
3550 \cs_new:Npn \token_get_replacement_spec:N #1
3551 {
3552     \token_if_macro:NTF #1
3553     {
3554         \exp_after:wN \__peek_get_prefix_arg_replacement:wN
3555         \token_to_meaning:N #1 \q_stop \use_iii:nnn
3556     }
3557     { \scan_stop: }
3558 }

```

(End definition for \token_get_prefix_spec:N. This function is documented on page 64.)

```

3559 </initex | package>

```

8 l3int implementation

```

3560 <*initex | package>
3561 <@@=int>

```

The following test files are used for this code: m3int001,m3int002,m3int03.

\c_max_register_int Done in l3basics.

(End definition for \c_max_register_int. This variable is documented on page 76.)

__int_to_roman:w Done in l3basics.

\if_int_compare:w (End definition for __int_to_roman:w. This function is documented on page 77.)

\or: Done in l3basics.

(End definition for \or:. This function is documented on page 77.)

__int_value:w Here are the remaining primitives for number comparisons and expressions.

__int_eval:w	3562 \cs_new_eq:NN __int_value:w	\tex_number:D
__int_eval_end:	3563 \cs_new_eq:NN __int_eval:w	\etex_numexpr:D
\if_int_odd:w	3564 \cs_new_eq:NN __int_eval_end:	\tex_relax:D
\if_case:w	3565 \cs_new_eq:NN \if_int_odd:w	\tex_ifodd:D
	3566 \cs_new_eq:NN \if_case:w	\tex_ifcase:D

(End definition for __int_value:w. This function is documented on page 78.)

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. In format mode, there is already a definition in `l3alloc` for bootstrapping, which is therefore corrected to the “real” version here.

```

3567 <*initex>
3568 \cs_set:Npn \int_eval:n #1
3569 { \__int_value:w \__int_eval:w #1 \__int_eval_end: }
3570 </initex>
3571 <*package>
3572 \cs_new:Npn \int_eval:n #1
3573 { \__int_value:w \__int_eval:w #1 \__int_eval_end: }
3574 </package>

```

(End definition for `\int_eval:n`. This function is documented on page 65.)

\int_abs:n Functions for min, max, and absolute value with only one evaluation. The absolute value is obtained by removing a leading sign if any. All three functions expand in two steps.

```

\__int_abs:N
\int_max:nn
\int_min:nn
\__int_maxmin:wwN
3575 \cs_new:Npn \int_abs:n #1
3576 {
3577   \__int_value:w \exp_after:wN \__int_abs:N
3578   \int_use:N \__int_eval:w #1 \__int_eval_end:
3579   \exp_stop_f:
3580 }
3581 \cs_new:Npn \__int_abs:N #1
3582 { \if_meaning:w - #1 \else: \exp_after:wN #1 \fi: }
3583 \cs_set:Npn \int_max:nn #1#2
3584 {
3585   \__int_value:w \exp_after:wN \__int_maxmin:wwN
3586   \int_use:N \__int_eval:w #1 \exp_after:wN ;
3587   \int_use:N \__int_eval:w #2 ;
3588   >
3589   \exp_stop_f:
3590 }
3591 \cs_set:Npn \int_min:nn #1#2
3592 {
3593   \__int_value:w \exp_after:wN \__int_maxmin:wwN
3594   \int_use:N \__int_eval:w #1 \exp_after:wN ;
3595   \int_use:N \__int_eval:w #2 ;
3596   <
3597   \exp_stop_f:
3598 }
3599 \cs_new:Npn \__int_maxmin:wwN #1 ; #2 ; #3
3600 {
3601   \if_int_compare:w #1 #3 #2 ~
3602   #1
3603   \else:
3604   #2
3605   \fi:
3606 }

```

(End definition for `\int_abs:n`. This function is documented on page 65.)

`\int_div_truncate:nn` As `__int_eval:w` rounds the result of a division we also provide a version that truncates the result. We use an auxiliary to make sure numerator and denominator are only evaluated once: this comes in handy when those are more expressions are expensive to evaluate (e.g., `\tl_count:n`). If the numerator `#1#2` is 0, then we divide 0 by the denominator (this ensures that 0/0 is correctly reported as an error). Otherwise, shift the numerator `#1#2` towards 0 by $(|\#3\#4| - 1)/2$, which we round away from zero. It turns out that this quantity exactly compensates the difference between ε -TeX's rounding and the truncating behaviour that we want. The details are thanks to Heiko Oberdiek: getting things right in all cases is not so easy.

```

3607 \cs_new:Npn \int_div_truncate:nn #1#2
3608 {
3609   \int_use:N \__int_eval:w
3610   \exp_after:wN \__int_div_truncate:NwNw
3611   \int_use:N \__int_eval:w #1 \exp_after:wN ;
3612   \int_use:N \__int_eval:w #2 ;
3613   \__int_eval_end:
3614 }
3615 \cs_new:Npn \__int_div_truncate:NwNw #1#2; #3#4;
3616 {
3617   \if_meaning:w 0 #1
3618   \c_zero
3619   \else:
3620   (
3621     #1#2
3622     \if_meaning:w - #1 + \else: - \fi:
3623     ( \if_meaning:w - #3 - \fi: #3#4 - \c_one ) / \c_two
3624   )
3625   \fi:
3626   / #3#4
3627 }
```

For the sake of completeness:

```

3628 \cs_new:Npn \int_div_round:nn #1#2
3629 { \__int_value:w \__int_eval:w ( #1 ) / ( #2 ) \__int_eval_end: }
```

Finally there's the modulus operation.

```

3630 \cs_new:Npn \int_mod:nn #1#2
3631 {
3632   \__int_value:w \__int_eval:w \exp_after:wN \__int_mod:ww
3633   \__int_value:w \__int_eval:w #1 \exp_after:wN ;
3634   \__int_value:w \__int_eval:w #2 ;
3635   \__int_eval_end:
3636 }
3637 \cs_new:Npn \__int_mod:ww #1; #2;
3638 { #1 - ( \__int_div_truncate:NwNw #1 ; #2 ; ) * #2 }
```

(End definition for `\int_div_truncate:nn`. This function is documented on page 66.)

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, `\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.)

```

3639 <*package>
3640 \cs_new_protected:Npn \int_new:N #1
3641 {
3642   \__chk_if_free_cs:N #1
3643   \cs:w newcount \cs_end: #1
3644 }
3645 </package>
3646 \cs_generate_variant:Nn \int_new:N { c }

```

(End definition for `\int_new:N` and `\int_new:c`. These functions are documented on page 66.)

`\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.

```

\int_const:cn
\__int_constdef:Nw
\c__max_constdef_int
3647 \cs_new_protected:Npn \int_const:Nn #1#2
3648 {
3649   \int_compare:nNnTF {#2} > \c_minus_one
3650   {
3651     \int_compare:nNnTF {#2} > \c__max_constdef_int
3652     {
3653       \int_new:N #1
3654       \int_gset:Nn #1 {#2}
3655     }
3656     {
3657       \__chk_if_free_cs:N #1
3658       \tex_global:D \__int_constdef:Nw #1 =
3659       \__int_eval:w #2 \__int_eval_end:
3660     }
3661   }
3662   {
3663     \int_new:N #1
3664     \int_gset:Nn #1 {#2}
3665   }
3666 }
3667 \cs_generate_variant:Nn \int_const:Nn { c }
3668 \if_int_odd:w 0
3669   \cs_if_exist:NT \luatex luatexversion:D { 1 }
3670   \cs_if_exist:NT \uptex_disablecjktoken:D
3671   { \if_int_compare:w \ptex_jis:D "2121 = "3000 ~ 1 \fi: }
3672   \cs_if_exist:NT \xetex_XeTeXversion:D { 1 } ~
3673   \cs_if_exist:NTF \uptex_disablecjktoken:D
3674   { \cs_new_eq:NN \__int_constdef:Nw \uptex_kchardef:D }
3675   { \cs_new_eq:NN \__int_constdef:Nw \tex_chardef:D }
3676   \__int_constdef:Nw \c__max_constdef_int 1114111 ~

```

```

3677 \else:
3678   \cs_new_eq:NN \__int_constdef:Nw \tex_mathchardef:D
3679   \tex_mathchardef:D \c__max_constdef_int 32767 ~
3680 \fi:

```

(End definition for `\int_const:Nn` and `\int_const:cn`. These functions are documented on page 66.)

```

\int_zero:N Functions that reset an integer register to zero.
\int_zero:c 3681 \cs_new_protected:Npn \int_zero:N #1 { #1 = \c_zero }
\int_gzero:N 3682 \cs_new_protected:Npn \int_gzero:N #1 { \tex_global:D #1 = \c_zero }
\int_gzero:c 3683 \cs_generate_variant:Nn \int_zero:N { c }
3684 \cs_generate_variant:Nn \int_gzero:N { c }

```

(End definition for `\int_zero:N` and `\int_zero:c`. These functions are documented on page 66.)

```

\int_zero_new:N Create a register if needed, otherwise clear it.
\int_zero_new:c 3685 \cs_new_protected:Npn \int_zero_new:N #1
\int_gzero_new:N 3686 { \int_if_exist:NTF #1 { \int_zero:N #1 } { \int_new:N #1 } }
\int_gzero_new:c 3687 \cs_new_protected:Npn \int_gzero_new:N #1
3688 { \int_if_exist:NTF #1 { \int_gzero:N #1 } { \int_new:N #1 } }
3689 \cs_generate_variant:Nn \int_zero_new:N { c }
3690 \cs_generate_variant:Nn \int_gzero_new:N { c }

```

(End definition for `\int_zero_new:N` and others. These functions are documented on page 67.)

```

\int_set_eq:NN Setting equal means using one integer inside the set function of another.
\int_set_eq:cN 3691 \cs_new_protected:Npn \int_set_eq:NN #1#2 { #1 = #2 }
\int_set_eq:Nc 3692 \cs_generate_variant:Nn \int_set_eq:NN { c }
\int_set_eq:cc 3693 \cs_generate_variant:Nn \int_set_eq:NN { Nc , cc }
\int_gset_eq:NN 3694 \cs_new_protected:Npn \int_gset_eq:NN #1#2 { \tex_global:D #1 = #2 }
\int_gset_eq:cN 3695 \cs_generate_variant:Nn \int_gset_eq:NN { c }
\int_gset_eq:Nc 3696 \cs_generate_variant:Nn \int_gset_eq:NN { Nc , cc }
\int_gset_eq:cc

```

(End definition for `\int_set_eq:NN` and others. These functions are documented on page 67.)

```

\int_if_exist_p:N Copies of the cs functions defined in l3basics.
\int_if_exist_p:c 3697 \prg_new_eq_conditional:NNn \int_if_exist:N \cs_if_exist:N
\int_if_exist:NTF 3698 { TF , T , F , p }
\int_if_exist:cTF 3699 \prg_new_eq_conditional:NNn \int_if_exist:c \cs_if_exist:c
3700 { TF , T , F , p }

```

(End definition for `\int_if_exist:NTF` and `\int_if_exist:cTF`. These functions are documented on page 67.)

8.3 Setting and incrementing integers

`\int_add:Nn` Adding and subtracting to and from a counter ...

```

\int_add:cn 3701 \cs_new_protected:Npn \int_add:Nn #1#2
\int_gadd:Nn 3702 { \tex_advance:D #1 by \__int_eval:w #2 \__int_eval_end: }
\int_gadd:cn 3703 \cs_new_protected:Npn \int_sub:Nn #1#2
\int_sub:Nn 3704 { \tex_advance:D #1 by - \__int_eval:w #2 \__int_eval_end: }
\int_sub:cn 3705 \cs_new_protected_nopar:Npn \int_gadd:Nn
\int_gsub:Nn 3706 { \tex_global:D \int_add:Nn }
\int_gsub:cn 3707 \cs_new_protected_nopar:Npn \int_gsub:Nn
3708 { \tex_global:D \int_sub:Nn }
3709 \cs_generate_variant:Nn \int_add:Nn { c }
3710 \cs_generate_variant:Nn \int_gadd:Nn { c }
3711 \cs_generate_variant:Nn \int_sub:Nn { c }
3712 \cs_generate_variant:Nn \int_gsub:Nn { c }

```

(End definition for `\int_add:Nn` and `\int_add:cn`. These functions are documented on page 67.)

`\int_incr:N` Incrementing and decrementing of integer registers is done with the following functions.

```

\int_incr:c 3713 \cs_new_protected:Npn \int_incr:N #1
\int_gincr:N 3714 { \tex_advance:D #1 \c_one }
\int_gincr:c 3715 \cs_new_protected:Npn \int_decr:N #1
\int_decr:N 3716 { \tex_advance:D #1 \c_minus_one }
\int_decr:c 3717 \cs_new_protected_nopar:Npn \int_gincr:N
\int_gdecr:N 3718 { \tex_global:D \int_incr:N }
\int_gdecr:c 3719 \cs_new_protected_nopar:Npn \int_gdecr:N
3720 { \tex_global:D \int_decr:N }
3721 \cs_generate_variant:Nn \int_incr:N { c }
3722 \cs_generate_variant:Nn \int_decr:N { c }
3723 \cs_generate_variant:Nn \int_gincr:N { c }
3724 \cs_generate_variant:Nn \int_gdecr:N { c }

```

(End definition for `\int_incr:N` and `\int_incr:c`. These functions are documented on page 67.)

`\int_set:Nn` As integers are register-based TeX will issue an error if they are not defined. Thus there is no need for the checking code seen with token list variables.

```

\int_set:cn 3725 \cs_new_protected:Npn \int_set:Nn #1#2
\int_gset:Nn 3726 { #1 ~ \__int_eval:w #2\__int_eval_end: }
\int_gset:cn 3727 \cs_new_protected_nopar:Npn \int_gset:Nn { \tex_global:D \int_set:Nn }
3728 \cs_generate_variant:Nn \int_set:Nn { c }
3729 \cs_generate_variant:Nn \int_gset:Nn { c }

```

(End definition for `\int_set:Nn` and `\int_set:cn`. These functions are documented on page 67.)

8.4 Using integers

`\int_use:N` Here is how counters are accessed:

```

\int_use:c 3730 \cs_new_eq:NN \int_use:N \tex_the:D

```

We hand-code this for some speed gain:

```
3731 %\cs_generate_variant:Nn \int_use:N { c }
3732 \cs_new:Npn \int_use:c #1 { \tex_the:D \cs:w #1 \cs_end: }
```

(End definition for `\int_use:N` and `\int_use:c`. These functions are documented on page 68.)

8.5 Integer expression conditionals

`__prg_compare_error:` Those functions are used for comparison tests which use a simple syntax where only one set of braces is required and additional operators such as `!=` and `>=` are supported. `__prg_compare_error:Nw` 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 TeX error. If the first token which appears after evaluating and removing the left-hand side is not a known relation symbol, then a judiciously placed `__prg_compare_error:Nw` gets expanded, cleaning up the end of the test and telling the user what the problem was.

```
3733 \cs_new_protected_nopar:Npn \__prg_compare_error:
3734 {
3735   \if_int_compare:w \c_zero \c_zero \fi:
3736   =
3737   \__prg_compare_error:
3738 }
3739 \cs_new:Npn \__prg_compare_error:Nw
3740 #1#2 \q_stop
3741 {
3742   { }
3743   \c_zero \fi:
3744   \__msg_kernel_expandable_error:nnn
3745   { kernel } { unknown-comparison } {#1}
3746   \prg_return_false:
3747 }
```

(End definition for `__prg_compare_error:` and `__prg_compare_error:Nw`.)

<code>\int_compare_p:n</code>	Comparison tests using a simple syntax where only one set of braces is required, additional
<code>\int_compare:nTF</code>	operators such as <code>!=</code> and <code>>=</code> are supported, and multiple comparisons can be performed
<code>__int_compare:w</code>	at once, for instance <code>0 < 5 <= 1</code> . The idea is to loop through the argument, finding one
<code>__int_compare:Nw</code>	operand at a time, and comparing it to the previous one. The looping auxiliary <code>__-</code>
<code>__int_compare:NNw</code>	<code>int_compare:Nw</code> reads one <i>operand</i> and one <i>comparison</i> symbol, and leaves roughly
<code>__int_compare:nnN</code>	
<code>__int_compare_end=:NNw</code>	<code><operand> \prg_return_false: \fi:</code>
<code>__int_compare_=:NNw</code>	<code>\reverse_if:N \if_int_compare:w <operand> <comparison></code>
<code>__int_compare_<:NNw</code>	<code>__int_compare:Nw</code>
<code>__int_compare_>:NNw</code>	
<code>__int_compare_==:NNw</code>	in the input stream. Each call to this auxiliary provides the second operand of the last
<code>__int_compare_!=:NNw</code>	call's <code>\if_int_compare:w</code> . If one of the <i>comparisons</i> is <code>false</code> , the <code>true</code> branch of the
<code>__int_compare_<=:NNw</code>	TeX conditional is taken (because of <code>\reverse_if:N</code>), immediately returning <code>false</code> as
<code>__int_compare_>=:NNw</code>	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.

```

3748 \prg_new_conditional:Npnn \int_compare:n #1 { p , T , F , TF }
3749 {
3750   \exp_after:wN \__int_compare:w
3751   \int_use:N \__int_eval:w #1 \__prg_compare_error:
3752 }
3753 \cs_new:Npn \__int_compare:w #1 \__prg_compare_error:
3754 {
3755   \exp_after:wN \if_false: \__int_value:w
3756   \__int_compare:Nw #1 e { = nd_ } \q_stop
3757 }

```

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.

```

3758 \cs_new:Npn \__int_compare:Nw #1#2 \q_stop
3759 {
3760   \exp_after:wN \__int_compare:NNw
3761   \__int_to_roman:w - 0 #2 \q_mark
3762   #1#2 \q_stop
3763 }
3764 \cs_new:Npn \__int_compare:NNw #1#2#3 \q_mark
3765 {
3766   \etex_unexpanded:D
3767   \use:c
3768   {
3769     \__int_compare_ \token_to_str:N #1
3770     \if_meaning:w = #2 = \fi:
3771     :NNw
3772   }
3773   \__prg_compare_error:Nw #1
3774 }

```

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.

```

3775 \cs_new:cpn { __int_compare_end=:NNw } #1#2#3 e #4 \q_stop
3776 {
3777     {#3} \exp_stop_f:
3778     \prg_return_false: \else: \prg_return_true: \fi:
3779 }
3780 \cs_new:Npn \__int_compare:nnN #1#2#3
3781 {
3782     {#2} \exp_stop_f:
3783     \prg_return_false: \exp_after:wN \use_none_delimit_by_q_stop:w
3784     \fi:
3785     #1 #2 #3 \exp_after:wN \__int_compare:Nw \__int_value:w \__int_eval:w
3786 }
```

The actual comparisons are then simple function calls, using the relation as delimiter for a delimited argument and discarding `__prg_compare_error:Nw` $\langle token \rangle$ responsible for error detection.

```

3787 \cs_new:cpn { __int_compare=:NNw } #1#2#3 =
3788 { \__int_compare:nnN { \reverse_if:N \if_int_compare:w } {#3} = }
3789 \cs_new:cpn { __int_compare:<:NNw } #1#2#3 <
3790 { \__int_compare:nnN { \reverse_if:N \if_int_compare:w } {#3} < }
3791 \cs_new:cpn { __int_compare:>:NNw } #1#2#3 >
3792 { \__int_compare:nnN { \reverse_if:N \if_int_compare:w } {#3} > }
3793 \cs_new:cpn { __int_compare==:NNw } #1#2#3 ==
3794 { \__int_compare:nnN { \reverse_if:N \if_int_compare:w } {#3} = }
3795 \cs_new:cpn { __int_compare!=:NNw } #1#2#3 !=
3796 { \__int_compare:nnN { \if_int_compare:w } {#3} = }
3797 \cs_new:cpn { __int_compare<=:NNw } #1#2#3 <=
3798 { \__int_compare:nnN { \if_int_compare:w } {#3} > }
3799 \cs_new:cpn { __int_compare>=:NNw } #1#2#3 >=
3800 { \__int_compare:nnN { \if_int_compare:w } {#3} < }
```

(End definition for `\int_compare:nTF`. This function is documented on page 69.)

`\int_compare_p:nNn` More efficient but less natural in typing.

```

\int_compare:nNnTF
3801 \prg_new_conditional:Npnn \int_compare:nNn #1#2#3 { p , T , F , TF }
3802 {
3803     \if_int_compare:w \__int_eval:w #1 #2 \__int_eval:w #3 \__int_eval_end:
3804     \prg_return_true:
3805     \else:

```

```

3806     \prg_return_false:
3807     \fi:
3808 }

```

(End definition for \int_compare:nNnTF. This function is documented on page 68.)

```

\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.
\__int_case:nnTF
\__int_case:nw
\__int_case_end:nw
3809 \cs_new:Npn \int_case:nnTF #1
3810 {
3811     \exp:w
3812     \exp_args:Nf \__int_case:nnTF { \int_eval:n {#1} }
3813 }
3814 \cs_new:Npn \int_case:nnT #1#2#3
3815 {
3816     \exp:w
3817     \exp_args:Nf \__int_case:nnTF { \int_eval:n {#1} } {#2} {#3} { }
3818 }
3819 \cs_new:Npn \int_case:nnF #1#2
3820 {
3821     \exp:w
3822     \exp_args:Nf \__int_case:nnTF { \int_eval:n {#1} } {#2} { }
3823 }
3824 \cs_new:Npn \int_case:nn #1#2
3825 {
3826     \exp:w
3827     \exp_args:Nf \__int_case:nnTF { \int_eval:n {#1} } {#2} { } { }
3828 }
3829 \cs_new:Npn \__int_case:nnTF #1#2#3#4
3830 { \__int_case:nw {#1} #2 {#1} { } \q_mark {#3} \q_mark {#4} \q_stop }
3831 \cs_new:Npn \__int_case:nw #1#2#3
3832 {
3833     \int_compare:nNnTF {#1} = {#2}
3834     { \__int_case_end:nw {#3} }
3835     { \__int_case:nw {#1} }
3836 }
3837 \cs_new_eq:NN \__int_case_end:nw \__prg_case_end:nw

```

(End definition for \int_case:nn. This function is documented on page ??.)

```

\int_if_odd_p:n A predicate function.
\int_if_odd:nTF
\int_if_even_p:n
\int_if_even:nTF
3838 \prg_new_conditional:Npnn \int_if_odd:n #1 { p , T , F , TF}
3839 {
3840     \if_int_odd:w \__int_eval:w #1 \__int_eval_end:
3841     \prg_return_true:
3842     \else:
3843     \prg_return_false:
3844     \fi:
3845 }
3846 \prg_new_conditional:Npnn \int_if_even:n #1 { p , T , F , TF}

```

```

3847 {
3848   \if_int_odd:w \__int_eval:w #1 \__int_eval_end:
3849   \prg_return_false:
3850   \else:
3851     \prg_return_true:
3852   \fi:
3853 }

```

(End definition for `\int_if_odd:nTF`. This function is documented on page 70.)

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.

```

\int_until_do:nn
\int_do_while:nn
\int_do_until:nn
3854 \cs_new:Npn \int_while_do:nn #1#2
3855 {
3856   \int_compare:nT {#1}
3857   {
3858     #2
3859     \int_while_do:nn {#1} {#2}
3860   }
3861 }
3862 \cs_new:Npn \int_until_do:nn #1#2
3863 {
3864   \int_compare:nF {#1}
3865   {
3866     #2
3867     \int_until_do:nn {#1} {#2}
3868   }
3869 }
3870 \cs_new:Npn \int_do_while:nn #1#2
3871 {
3872   #2
3873   \int_compare:nT {#1}
3874   { \int_do_while:nn {#1} {#2} }
3875 }
3876 \cs_new:Npn \int_do_until:nn #1#2
3877 {
3878   #2
3879   \int_compare:nF {#1}
3880   { \int_do_until:nn {#1} {#2} }
3881 }

```

(End definition for `\int_while_do:nn`. This function is documented on page 71.)

`\int_while_do:nNnn` As above but not using the more natural syntax.

```

\int_until_do:nNnn
\int_do_while:nNnn
\int_do_until:nNnn
3882 \cs_new:Npn \int_while_do:nNnn #1#2#3#4
3883 {
3884   \int_compare:nNnT {#1} #2 {#3}
3885   {

```

```

3886         #4
3887         \int_while_do:nNnn {#1} #2 {#3} {#4}
3888     }
3889 }
3890 \cs_new:Npn \int_until_do:nNnn #1#2#3#4
3891 {
3892     \int_compare:nNnF {#1} #2 {#3}
3893     {
3894         #4
3895         \int_until_do:nNnn {#1} #2 {#3} {#4}
3896     }
3897 }
3898 \cs_new:Npn \int_do_while:nNnn #1#2#3#4
3899 {
3900     #4
3901     \int_compare:nNnT {#1} #2 {#3}
3902     { \int_do_while:nNnn {#1} #2 {#3} {#4} }
3903 }
3904 \cs_new:Npn \int_do_until:nNnn #1#2#3#4
3905 {
3906     #4
3907     \int_compare:nNnF {#1} #2 {#3}
3908     { \int_do_until:nNnn {#1} #2 {#3} {#4} }
3909 }

```

(End definition for `\int_while_do:nNnn`. This function is documented on page 71.)

8.7 Integer step functions

\int_step_function:nnnN Before all else, evaluate the initial value, step, and final value. Repeating a function by steps first needs a check on the direction of the steps. After that, do the function for the start value then step and loop around. It would be more symmetrical to test for a step size of zero before checking the sign, but we optimize for the most frequent case (positive step).

```

3910 \cs_new:Npn \int_step_function:nnnN #1#2#3
3911 {
3912     \exp_after:wN \__int_step:wwwN
3913     \int_use:N \__int_eval:w #1 \exp_after:wN ;
3914     \int_use:N \__int_eval:w #2 \exp_after:wN ;
3915     \int_use:N \__int_eval:w #3 ;
3916 }
3917 \cs_new:Npn \__int_step:wwwN #1; #2; #3; #4
3918 {
3919     \int_compare:nNnTF {#2} > \c_zero
3920     { \__int_step:NnnnN > }
3921     {
3922         \int_compare:nNnTF {#2} = \c_zero
3923         {
3924             \__msg_kernel_expandable_error:nnn { kernel } { zero-step } {#4}

```

```

3925         \use_none:nnnn
3926     }
3927     { \__int_step:NnnnN < }
3928 }
3929 {#1} {#2} {#3} #4
3930 }
3931 \cs_new:Npn \__int_step:NnnnN #1#2#3#4#5
3932 {
3933     \int_compare:nNf {#2} #1 {#4}
3934     {
3935         #5 {#2}
3936         \exp_args:NNf \__int_step:NnnnN
3937         #1 { \int_eval:n { #2 + #3 } } {#3} {#4} #5
3938     }
3939 }

```

(End definition for \int_step_function:nnnN. This function is documented on page 72.)

```

\int_step_inline:nnnn
\int_step_variable:nnnNn
\__int_step:NNnnnn

```

The approach here is to build a function, with a global integer required to make the nesting safe (as seen in other in line functions), and map that function using \int_step_function:nnnN. We put a __prg_break_point:Nn so that map_break functions from other modules correctly decrement \g__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.

```

3940 \cs_new_protected_nopar:Npn \int_step_inline:nnnn
3941 {
3942     \int_gincr:N \g__prg_map_int
3943     \exp_args:NNc \__int_step:NNnnnn
3944     \cs_gset_nopar:Npn
3945     { __prg_map_ \int_use:N \g__prg_map_int :w }
3946 }
3947 \cs_new_protected:Npn \int_step_variable:nnnNn #1#2#3#4#5
3948 {
3949     \int_gincr:N \g__prg_map_int
3950     \exp_args:NNc \__int_step:NNnnnn
3951     \cs_gset_nopar:Npx
3952     { __prg_map_ \int_use:N \g__prg_map_int :w }
3953     {#1}{#2}{#3}
3954     {
3955         \tl_set:Nn \exp_not:N #4 {##1}
3956         \exp_not:n {#5}
3957     }
3958 }
3959 \cs_new_protected:Npn \__int_step:NNnnnn #1#2#3#4#5#6
3960 {
3961     #1 #2 ##1 {#6}
3962     \int_step_function:nnnN {#3} {#4} {#5} #2
3963     \__prg_break_point:Nn \scan_stop: { \int_gdecr:N \g__prg_map_int }
3964 }

```

(End definition for \int_step_inline:nnnn. This function is documented on page 72.)

8.8 Formatting integers

`\int_to_arabic:n` Nothing exciting here.

```
3965 \cs_new_eq:NN \int_to_arabic:n \int_eval:n
```

(End definition for `\int_to_arabic:n`. This function is documented on page 72.)

`\int_to_symbols:nnn` For conversion of integers to arbitrary symbols the method is in general as follows. The input number (#1) is compared to the total number of symbols available at each place (#2). If the input is larger than the total number of symbols available then the modulus is needed, with one added so that the positions don't have to number from zero. Using an f-type expansion, this is done so that the system is recursive. The actual conversion function therefore gets a 'nice' number at each stage. Of course, if the initial input was small enough then there is no problem and everything is easy.

`__int_to_symbols:nnnn`

```
3966 \cs_new:Npn \int_to_symbols:nnn #1#2#3
3967 {
3968   \int_compare:nNnTF {#1} > {#2}
3969   {
3970     \exp_args:NNo \exp_args:No \__int_to_symbols:nnnn
3971     {
3972       \int_case:nn
3973       { 1 + \int_mod:nn { #1 - 1 } {#2} }
3974       {#3}
3975     }
3976     {#1} {#2} {#3}
3977   }
3978   { \int_case:nn {#1} {#3} }
3979 }
3980 \cs_new:Npn \__int_to_symbols:nnnn #1#2#3#4
3981 {
3982   \exp_args:Nf \int_to_symbols:nnn
3983   { \int_div_truncate:nn { #2 - 1 } {#3} } {#3} {#4}
3984   #1
3985 }
```

(End definition for `\int_to_symbols:nnn`. This function is documented on page 73.)

`\int_to_alph:n` These both use the above function with input functions that make sense for the alphabet in English.

`\int_to_Alph:n`

```
3986 \cs_new:Npn \int_to_alph:n #1
3987 {
3988   \int_to_symbols:nnn {#1} { 26 }
3989   {
3990     { 1 } { a }
3991     { 2 } { b }
3992     { 3 } { c }
3993     { 4 } { d }
3994     { 5 } { e }
3995     { 6 } { f }
3996     { 7 } { g }
```

```

3997         { 8 } { h }
3998         { 9 } { i }
3999         { 10 } { j }
4000         { 11 } { k }
4001         { 12 } { l }
4002         { 13 } { m }
4003         { 14 } { n }
4004         { 15 } { o }
4005         { 16 } { p }
4006         { 17 } { q }
4007         { 18 } { r }
4008         { 19 } { s }
4009         { 20 } { t }
4010         { 21 } { u }
4011         { 22 } { v }
4012         { 23 } { w }
4013         { 24 } { x }
4014         { 25 } { y }
4015         { 26 } { z }
4016     }
4017 }
4018 \cs_new:Npn \int_to_Alph:n #1
4019 {
4020     \int_to_symbols:nnn {#1} { 26 }
4021     {
4022         { 1 } { A }
4023         { 2 } { B }
4024         { 3 } { C }
4025         { 4 } { D }
4026         { 5 } { E }
4027         { 6 } { F }
4028         { 7 } { G }
4029         { 8 } { H }
4030         { 9 } { I }
4031         { 10 } { J }
4032         { 11 } { K }
4033         { 12 } { L }
4034         { 13 } { M }
4035         { 14 } { N }
4036         { 15 } { O }
4037         { 16 } { P }
4038         { 17 } { Q }
4039         { 18 } { R }
4040         { 19 } { S }
4041         { 20 } { T }
4042         { 21 } { U }
4043         { 22 } { V }
4044         { 23 } { W }
4045         { 24 } { X }
4046         { 25 } { Y }

```

```

4047         { 26 } { Z }
4048     }
4049 }

```

(End definition for `\int_to_alph:n` and `\int_to_Alph:n`. These functions are documented on page 73.)

```

\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_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.

```

4050 \cs_new:Npn \int_to_base:nn #1
4051 { \exp_args:Nf \__int_to_base:nn { \int_eval:n {#1} } }
4052 \cs_new:Npn \int_to_Base:nn #1
4053 { \exp_args:Nf \__int_to_Base:nn { \int_eval:n {#1} } }
4054 \cs_new:Npn \__int_to_base:nn #1#2
4055 {
4056   \int_compare:nNnTF {#1} < \c_zero
4057   { \exp_args:No \__int_to_base:nnN { \use_none:n #1 } {#2} - }
4058   { \__int_to_base:nnN {#1} {#2} \c_empty_tl }
4059 }
4060 \cs_new:Npn \__int_to_Base:nn #1#2
4061 {
4062   \int_compare:nNnTF {#1} < \c_zero
4063   { \exp_args:No \__int_to_Base:nnN { \use_none:n #1 } {#2} - }
4064   { \__int_to_Base:nnN {#1} {#2} \c_empty_tl }
4065 }

```

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.

```

4066 \cs_new:Npn \__int_to_base:nnN #1#2#3
4067 {
4068   \int_compare:nNnTF {#1} < {#2}
4069   { \exp_last_unbraced:Nf #3 { \__int_to_letter:n {#1} } }
4070   {
4071     \exp_args:Nf \__int_to_base:nnnN
4072     { \__int_to_letter:n { \int_mod:nn {#1} {#2} } }
4073     {#1}
4074     {#2}
4075     #3
4076   }
4077 }
4078 \cs_new:Npn \__int_to_base:nnnN #1#2#3#4
4079 {
4080   \exp_args:Nf \__int_to_base:nnN
4081   { \int_div_truncate:nn {#2} {#3} }
4082   {#3}
4083   #4

```

```

4084     #1
4085   }
4086 \cs_new:Npn \__int_to_Base:nnN #1#2#3
4087 {
4088   \int_compare:nNnTF {#1} < {#2}
4089   { \exp_last_unbraced:Nf #3 { \__int_to_Letter:n {#1} } }
4090   {
4091     \exp_args:Nf \__int_to_Base:nnnN
4092     { \__int_to_Letter:n { \int_mod:nn {#1} {#2} } }
4093     {#1}
4094     {#2}
4095     #3
4096   }
4097 }
4098 \cs_new:Npn \__int_to_Base:nnnN #1#2#3#4
4099 {
4100   \exp_args:Nf \__int_to_Base:nnN
4101   { \int_div_truncate:nn {#2} {#3} }
4102   {#3}
4103   #4
4104   #1
4105 }

```

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:`.

```

4106 \cs_new:Npn \__int_to_letter:n #1
4107 {
4108   \exp_after:wN \exp_after:wN
4109   \if_case:w \__int_eval:w #1 - \c_ten \__int_eval_end:
4110     a
4111   \or: b
4112   \or: c
4113   \or: d
4114   \or: e
4115   \or: f
4116   \or: g
4117   \or: h
4118   \or: i
4119   \or: j
4120   \or: k
4121   \or: l
4122   \or: m
4123   \or: n
4124   \or: o
4125   \or: p
4126   \or: q

```

```

4127     \or: r
4128     \or: s
4129     \or: t
4130     \or: u
4131     \or: v
4132     \or: w
4133     \or: x
4134     \or: y
4135     \or: z
4136     \else: \__int_value:w \__int_eval:w #1 \exp_after:wN \__int_eval_end:
4137     \fi:
4138 }
4139 \cs_new:Npn \__int_to_Letter:n #1
4140 {
4141     \exp_after:wN \exp_after:wN
4142     \if_case:w \__int_eval:w #1 - \c_ten \__int_eval_end:
4143         A
4144     \or: B
4145     \or: C
4146     \or: D
4147     \or: E
4148     \or: F
4149     \or: G
4150     \or: H
4151     \or: I
4152     \or: J
4153     \or: K
4154     \or: L
4155     \or: M
4156     \or: N
4157     \or: O
4158     \or: P
4159     \or: Q
4160     \or: R
4161     \or: S
4162     \or: T
4163     \or: U
4164     \or: V
4165     \or: W
4166     \or: X
4167     \or: Y
4168     \or: Z
4169     \else: \__int_value:w \__int_eval:w #1 \exp_after:wN \__int_eval_end:
4170     \fi:
4171 }

```

(End definition for `\int_to_base:nn` and `\int_to_Base:nn`. These functions are documented on page 74.)

`\int_to_bin:n` Wrappers around the generic function.
`\int_to_hex:n`
`\int_to_Hex:n`
`\int_to_oct:n`

```

4172 \cs_new:Npn \int_to_bin:n #1
4173   { \int_to_base:nn {#1} { 2 } }
4174 \cs_new:Npn \int_to_hex:n #1
4175   { \int_to_base:nn {#1} { 16 } }
4176 \cs_new:Npn \int_to_Hex:n #1
4177   { \int_to_Base:nn {#1} { 16 } }
4178 \cs_new:Npn \int_to_oct:n #1
4179   { \int_to_base:nn {#1} { 8 } }

```

(End definition for `\int_to_bin:n` and others. These functions are documented on page 73.)

```

\int_to_roman:n The \__int_to_roman:w primitive creates tokens of category code 12 (other). Usually,
\int_to_Roman:n what is actually wanted is letters. The approach here is to convert the output of the
                  primitive into letters using appropriate control sequence names. That keeps everything
                  expandable. The loop will be terminated by the conversion of the Q.
__int_to_roman:N
__int_to_roman:N
__int_to_roman_i:w 4180 \cs_new:Npn \int_to_roman:n #1
__int_to_roman_v:w 4181   {
__int_to_roman_x:w 4182     \exp_after:wN \__int_to_roman:N
__int_to_roman_l:w 4183     \__int_to_roman:w \int_eval:n {#1} Q
__int_to_roman_c:w 4184   }
__int_to_roman_d:w 4185 \cs_new:Npn \__int_to_roman:N #1
__int_to_roman_m:w 4186   {
__int_to_roman_Q:w 4187     \use:c { __int_to_roman_ #1 :w }
__int_to_Roman_i:w 4188     \__int_to_roman:N
__int_to_Roman_v:w 4189   }
__int_to_Roman_x:w 4190 \cs_new:Npn \int_to_Roman:n #1
__int_to_Roman_l:w 4191   {
__int_to_Roman_c:w 4192     \exp_after:wN \__int_to_Roman_aux:N
__int_to_Roman_d:w 4193     \__int_to_roman:w \int_eval:n {#1} Q
__int_to_Roman_m:w 4194   }
__int_to_Roman_Q:w 4195 \cs_new:Npn \__int_to_Roman_aux:N #1
                  4196   {
                  4197     \use:c { __int_to_Roman_ #1 :w }
                  4198     \__int_to_Roman_aux:N
                  4199   }
4200 \cs_new_nopar:Npn \__int_to_roman_i:w { i }
4201 \cs_new_nopar:Npn \__int_to_roman_v:w { v }
4202 \cs_new_nopar:Npn \__int_to_roman_x:w { x }
4203 \cs_new_nopar:Npn \__int_to_roman_l:w { l }
4204 \cs_new_nopar:Npn \__int_to_roman_c:w { c }
4205 \cs_new_nopar:Npn \__int_to_roman_d:w { d }
4206 \cs_new_nopar:Npn \__int_to_roman_m:w { m }
4207 \cs_new_nopar:Npn \__int_to_roman_Q:w #1 { }
4208 \cs_new_nopar:Npn \__int_to_Roman_i:w { I }
4209 \cs_new_nopar:Npn \__int_to_Roman_v:w { V }
4210 \cs_new_nopar:Npn \__int_to_Roman_x:w { X }
4211 \cs_new_nopar:Npn \__int_to_Roman_l:w { L }
4212 \cs_new_nopar:Npn \__int_to_Roman_c:w { C }
4213 \cs_new_nopar:Npn \__int_to_Roman_d:w { D }
4214 \cs_new_nopar:Npn \__int_to_Roman_m:w { M }

```

```
4215 \cs_new:Npn \__int_to_Roman_Q:w #1 { }
```

(End definition for `\int_to_roman:n` and `\int_to_Roman:n`. These functions are documented on page 74.)

8.9 Converting from other formats to integers

```
\__int_pass_signs:wn
\__int_pass_signs_end:wn
```

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.

```
4216 \cs_new:Npn \__int_pass_signs:wn #1
4217 {
4218   \if:w + \if:w - \exp_not:N #1 + \fi: \exp_not:N #1
4219   \exp_after:wN \__int_pass_signs:wn
4220   \else:
4221     \exp_after:wN \__int_pass_signs_end:wn
4222     \exp_after:wN #1
4223   \fi:
4224 }
4225 \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_alph:n
\__int_from_alph:nN
\__int_from_alph:N
```

First take care of signs then loop through the input using the recursion quarks. The `__int_from_alph:nN` auxiliary collects in its first argument the value obtained so far, and the auxiliary `__int_from_alph:N` converts one letter to an expression which evaluates to the correct number.

```
4226 \cs_new:Npn \int_from_alph:n #1
4227 {
4228   \int_eval:n
4229   {
4230     \exp_after:wN \__int_pass_signs:wn \tl_to_str:n {#1}
4231     \q_stop { \__int_from_alph:nN { 0 } }
4232     \q_recursion_tail \q_recursion_stop
4233   }
4234 }
4235 \cs_new:Npn \__int_from_alph:nN #1#2
4236 {
4237   \quark_if_recursion_tail_stop_do:Nn #2 {#1}
4238   \exp_args:Nf \__int_from_alph:nN
4239   { \int_eval:n { #1 * 26 + \__int_from_alph:N #2 } }
4240 }
4241 \cs_new:Npn \__int_from_alph:N #1
4242 { '#1 - \int_compare:nNnTF { '#1 } < { 91 } { 64 } { 96 } }
```

(End definition for `\int_from_alph:n`. This function is documented on page 74.)

`\int_from_base:nn` Leave the signs into the integer expression, then loop through characters, collecting the value found so far in the first argument of `__int_from_base:nnN`. To convert a single character, `__int_from_base:N` checks first for digits, then distinguishes lower from upper case letters, turning them into the appropriate number. Note that this auxiliary does not use `\int_eval:n`, hence is not safe for general use.

```

4243 \cs_new:Npn \int_from_base:nn #1#2
4244 {
4245   \int_eval:n
4246   {
4247     \exp_after:wN \__int_pass_signs:wN \tl_to_str:n {#1}
4248     \q_stop { \__int_from_base:nnN { 0 } {#2} }
4249     \q_recursion_tail \q_recursion_stop
4250   }
4251 }
4252 \cs_new:Npn \__int_from_base:nnN #1#2#3
4253 {
4254   \quark_if_recursion_tail_stop_do:Nn #3 {#1}
4255   \exp_args:Nf \__int_from_base:nnN
4256   { \int_eval:n { #1 * #2 + \__int_from_base:N #3 } }
4257   {#2}
4258 }
4259 \cs_new:Npn \__int_from_base:N #1
4260 {
4261   \int_compare:nNnTF { '#1 } < { 58 }
4262   {#1}
4263   { '#1 - \int_compare:nNnTF { '#1 } < { 91 } { 55 } { 87 } }
4264 }

```

(End definition for `\int_from_base:nn`. This function is documented on page 75.)

`\int_from_bin:n` Wrappers around the generic function.

```

4265 \cs_new:Npn \int_from_bin:n #1
4266 { \int_from_base:nn {#1} \c_two }
4267 \cs_new:Npn \int_from_hex:n #1
4268 { \int_from_base:nn {#1} \c_sixteen }
4269 \cs_new:Npn \int_from_oct:n #1
4270 { \int_from_base:nn {#1} \c_eight }

```

(End definition for `\int_from_bin:n`, `\int_from_hex:n`, and `\int_from_oct:n`. These functions are documented on page 74.)

`\c__int_from_roman_i_int` Constants used to convert from Roman numerals to integers.

```

4271 \int_const:cn { c__int_from_roman_i_int } { 1 }
4272 \int_const:cn { c__int_from_roman_v_int } { 5 }
4273 \int_const:cn { c__int_from_roman_x_int } { 10 }
4274 \int_const:cn { c__int_from_roman_l_int } { 50 }
4275 \int_const:cn { c__int_from_roman_c_int } { 100 }
4276 \int_const:cn { c__int_from_roman_d_int } { 500 }
4277 \int_const:cn { c__int_from_roman_m_int } { 1000 }
4278 \int_const:cn { c__int_from_roman_I_int } { 1 }

```

```

\c__int_from_roman_X_int
\c__int_from_roman_L_int
\c__int_from_roman_C_int
\c__int_from_roman_D_int
\c__int_from_roman_M_int

```

```

4279 \int_const:cn { c__int_from_roman_V_int } { 5 }
4280 \int_const:cn { c__int_from_roman_X_int } { 10 }
4281 \int_const:cn { c__int_from_roman_L_int } { 50 }
4282 \int_const:cn { c__int_from_roman_C_int } { 100 }
4283 \int_const:cn { c__int_from_roman_D_int } { 500 }
4284 \int_const:cn { c__int_from_roman_M_int } { 1000 }

```

(End definition for `\c__int_from_roman_i_int` and others. These variables are documented on page ??.)

`\int_from_roman:n` The method here is to iterate through the input, finding the appropriate value for each letter and building up a sum. This is then evaluated by \TeX . If any unknown letter is found, skip to the closing parenthesis and insert `*0-1` afterwards, to replace the value by -1 .

```

4285 \cs_new:Npn \int_from_roman:n #1
4286 {
4287   \int_eval:n
4288   {
4289     (
4290       \c_zero
4291       \exp_after:wN \__int_from_roman:NN \tl_to_str:n {#1}
4292       \q_recursion_tail \q_recursion_tail \q_recursion_stop
4293     )
4294   }
4295 }
4296 \cs_new:Npn \__int_from_roman:NN #1#2
4297 {
4298   \quark_if_recursion_tail_stop:N #1
4299   \int_if_exist:cF { c__int_from_roman_ #1 _int }
4300   { \__int_from_roman_error:w }
4301   \quark_if_recursion_tail_stop_do:Nn #2
4302   { + \use:c { c__int_from_roman_ #1 _int } }
4303   \int_if_exist:cF { c__int_from_roman_ #2 _int }
4304   { \__int_from_roman_error:w }
4305   \int_compare:nNnTF
4306   { \use:c { c__int_from_roman_ #1 _int } }
4307   <
4308   { \use:c { c__int_from_roman_ #2 _int } }
4309   {
4310     + \use:c { c__int_from_roman_ #2 _int }
4311     - \use:c { c__int_from_roman_ #1 _int }
4312     \__int_from_roman:NN
4313   }
4314   {
4315     + \use:c { c__int_from_roman_ #1 _int }
4316     \__int_from_roman:NN #2
4317   }
4318 }
4319 \cs_new:Npn \__int_from_roman_error:w #1 \q_recursion_stop #2
4320 { #2 * \c_zero - \c_one }

```

(End definition for `\int_from_roman:n`. This function is documented on page 75.)

8.10 Viewing integer

`\int_show:N` This is very similar to other registers done using `__kernel_register_show:N`, but differs because the variable #1 may be `\currentgrouplevel` or `\currentgrouptype`, in which case the value must be expanded in the current scope rather than when processing `\iow_wrap:nnnN`.

```

4321 \cs_new_protected:Npn \int_show:N #1
4322 {
4323   \use:x
4324   {
4325     \exp_not:n
4326     { \__msg_show_variable:NNNnn #1 \cs_if_exist:NTF ? { } }
4327     { > ~ \token_to_str:N #1 = \tex_the:D #1 }
4328   }
4329 }
4330 \cs_generate_variant:Nn \int_show:N { c }

```

(End definition for `\int_show:N` and `\int_show:c`. These functions are documented on page 75.)

`\int_show:n` We don't use the TeX primitive `\showthe` to show integer expressions: this gives a more unified output.

```

4331 \cs_new_protected_nopar:Npn \int_show:n
4332 { \__msg_show_wrap:Nn \int_eval:n }

```

(End definition for `\int_show:n`. This function is documented on page 75.)

8.11 Constant integers

`\c_minus_one` This is needed early, and so is in `l3basics`

(End definition for `\c_minus_one`. This variable is documented on page 76.)

`\c_zero` Again, in `l3basics`

`\c_sixteen` (End definition for `\c_zero` and `\c_sixteen`. These variables are documented on page 76.)

`\c_one` Low-number values not previously defined.

```

\c_two 4333 \int_const:Nn \c_one { 1 }
\c_three 4334 \int_const:Nn \c_two { 2 }
\c_four 4335 \int_const:Nn \c_three { 3 }
\c_five 4336 \int_const:Nn \c_four { 4 }
\c_six 4337 \int_const:Nn \c_five { 5 }
\c_seven 4338 \int_const:Nn \c_six { 6 }
\c_eight 4339 \int_const:Nn \c_seven { 7 }
\c_nine 4340 \int_const:Nn \c_eight { 8 }
\c_ten 4341 \int_const:Nn \c_nine { 9 }
\c_eleven 4342 \int_const:Nn \c_ten { 10 }
\c_twelve 4343 \int_const:Nn \c_eleven { 11 }
\c_thirteen
\c_fourteen
\c_fifteen

```

```

4344 \int_const:Nn \c_twelve { 12 }
4345 \int_const:Nn \c_thirteen { 13 }
4346 \int_const:Nn \c_fourteen { 14 }
4347 \int_const:Nn \c_fifteen { 15 }

```

(End definition for `\c_one` and others. These variables are documented on page 76.)

`\c_thirty_two` One middling value.

```

4348 \int_const:Nn \c_thirty_two { 32 }

```

(End definition for `\c_thirty_two`. This variable is documented on page 76.)

`\c_two_hundred_fifty_five` Two classic mid-range integer constants.

```

\c_two_hundred_fifty_six 4349 \int_const:Nn \c_two_hundred_fifty_five { 255 }
4350 \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 76.)

`\c_one_hundred` Simple runs of powers of ten.

```

\c_one_thousand 4351 \int_const:Nn \c_one_hundred { 100 }
\c_ten_thousand 4352 \int_const:Nn \c_one_thousand { 1000 }
4353 \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 76.)

`\c_max_int` The largest number allowed is $2^{31} - 1$

```

4354 \int_const:Nn \c_max_int { 2 147 483 647 }

```

(End definition for `\c_max_int`. This variable is documented on page 76.)

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 4355 \int_new:N \l_tmpa_int
\g_tmpa_int 4356 \int_new:N \l_tmpb_int
\g_tmpb_int 4357 \int_new:N \g_tmpa_int
4358 \int_new:N \g_tmpb_int

```

(End definition for `\l_tmpa_int` and `\l_tmpb_int`. These variables are documented on page 76.)

8.13 Deprecated functions

```

\int_to_binary:n    Deprecated 2014-02-11.
\int_from_binary:n  4359 \cs_new_eq:NN \int_to_binary:n    \int_to_bin:n
\int_to_hexadecimal:n 4360 \cs_new_eq:NN \int_to_hexadecimal:n \int_to_Hex:n
\int_from_hexadecimal:n 4361 \cs_new_eq:NN \int_to_octal:n    \int_to_oct:n
\int_to_octal:n      4362 \cs_new_eq:NN \int_from_binary:n    \int_from_bin:n
\int_from_octal:n    4363 \cs_new_eq:NN \int_from_hexadecimal:n \int_from_hex:n
                     4364 \cs_new_eq:NN \int_from_octal:n    \int_from_oct:n

```

(End definition for `\int_to_binary:n` and `\int_from_binary:n`. These functions are documented on page ??.)

```
4365 </initex | package>
```

9 l3skip implementation

```
4366 <*initex | package>
```

```
4367 <@@=dim>
```

9.1 Length primitives renamed

```

\if_dim:w    Primitives renamed.
\__dim_eval:w 4368 \cs_new_eq:NN \if_dim:w    \tex_ifdim:D
\__dim_eval_end: 4369 \cs_new_eq:NN \__dim_eval:w    \etex_dimexpr:D
                4370 \cs_new_eq:NN \__dim_eval_end:    \tex_relax:D

```

(End definition for `\if_dim:w`. This function is documented on page 93.)

9.2 Creating and initialising dim variables

```

\dim_new:N    Allocating <dim> registers ...
\dim_new:c    4371 <*package>
               4372 \cs_new_protected:Npn \dim_new:N #1
               4373 {
               4374     \__chk_if_free_cs:N #1
               4375     \cs:w newdimen \cs_end: #1
               4376 }
               4377 </package>
               4378 \cs_generate_variant:Nn \dim_new:N { c }

```

(End definition for `\dim_new:N` and `\dim_new:c`. These functions are documented on page 79.)

```

\dim_const:Nn Contrarily to integer constants, we cannot avoid using a register, even for constants.
\dim_const:cn 4379 \cs_new_protected:Npn \dim_const:Nn #1
               4380 {
               4381     \dim_new:N #1
               4382     \dim_gset:Nn #1
               4383 }
               4384 \cs_generate_variant:Nn \dim_const:Nn { c }

```

(End definition for `\dim_const:Nn` and `\dim_const:cn`. These functions are documented on page 79.)

```

\dim_zero:N Reset the register to zero.
\dim_zero:c 4385 \cs_new_protected:Npn \dim_zero:N #1 { #1 \c_zero_dim }
\dim_gzero:N 4386 \cs_new_protected:Npn \dim_gzero:N { \tex_global:D \dim_zero:N }
\dim_gzero:c 4387 \cs_generate_variant:Nn \dim_zero:N { c }
4388 \cs_generate_variant:Nn \dim_gzero:N { c }

```

(End definition for `\dim_zero:N` and `\dim_gzero:c`. These functions are documented on page 79.)

```

\dim_zero_new:N Create a register if needed, otherwise clear it.
\dim_zero_new:c 4389 \cs_new_protected:Npn \dim_zero_new:N #1
\dim_gzero_new:N 4390 { \dim_if_exist:NTF #1 { \dim_zero:N #1 } { \dim_new:N #1 } }
\dim_gzero_new:c 4391 \cs_new_protected:Npn \dim_gzero_new:N #1
4392 { \dim_if_exist:NTF #1 { \dim_gzero:N #1 } { \dim_new:N #1 } }
4393 \cs_generate_variant:Nn \dim_zero_new:N { c }
4394 \cs_generate_variant:Nn \dim_gzero_new:N { c }

```

(End definition for `\dim_zero_new:N` and others. These functions are documented on page 79.)

```

\dim_if_exist_p:N Copies of the cs functions defined in l3basics.
\dim_if_exist_p:c 4395 \prg_new_eq_conditional:NNn \dim_if_exist:N \cs_if_exist:N
\dim_if_exist:NTF 4396 { TF , T , F , p }
\dim_if_exist:cTF 4397 \prg_new_eq_conditional:NNn \dim_if_exist:c \cs_if_exist:c
4398 { TF , T , F , p }

```

(End definition for `\dim_if_exist:NTF` and `\dim_if_exist:cTF`. These functions are documented on page 79.)

9.3 Setting dim variables

```

\dim_set:Nn Setting dimensions is easy enough.
\dim_set:cn 4399 \cs_new_protected:Npn \dim_set:Nn #1#2
\dim_gset:Nn 4400 { #1 ~ \_dim_eval:w #2 \_dim_eval_end: }
\dim_gset:cn 4401 \cs_new_protected:Npn \dim_gset:Nn { \tex_global:D \dim_set:Nn }
4402 \cs_generate_variant:Nn \dim_set:Nn { c }
4403 \cs_generate_variant:Nn \dim_gset:Nn { c }

```

(End definition for `\dim_set:Nn` and `\dim_gset:cn`. These functions are documented on page 80.)

```

\dim_set_eq:NN All straightforward.
\dim_set_eq:cN 4404 \cs_new_protected:Npn \dim_set_eq:NN #1#2 { #1 = #2 }
\dim_set_eq:Nc 4405 \cs_generate_variant:Nn \dim_set_eq:NN { c }
\dim_set_eq:cc 4406 \cs_generate_variant:Nn \dim_set_eq:NN { Nc , cc }
\dim_gset_eq:NN 4407 \cs_new_protected:Npn \dim_gset_eq:NN #1#2 { \tex_global:D #1 = #2 }
\dim_gset_eq:cN 4408 \cs_generate_variant:Nn \dim_gset_eq:NN { c }
\dim_gset_eq:Nc 4409 \cs_generate_variant:Nn \dim_gset_eq:NN { Nc , cc }
\dim_gset_eq:cc

```

(End definition for `\dim_set_eq:NN` and others. These functions are documented on page 80.)

```

\dim_add:Nn Using by here deals with the (incorrect) case \dimen123.
\dim_add:cn 4410 \cs_new_protected:Npn \dim_add:Nn #1#2
\dim_gadd:Nn 4411 { \tex_advance:D #1 by \__dim_eval:w #2 \__dim_eval_end: }
\dim_gadd:cn 4412 \cs_new_protected:Npn \dim_gadd:Nn { \tex_global:D \dim_add:Nn }
\dim_sub:Nn 4413 \cs_generate_variant:Nn \dim_add:Nn { c }
\dim_sub:cn 4414 \cs_generate_variant:Nn \dim_gadd:Nn { c }
\dim_gsub:Nn 4415 \cs_new_protected:Npn \dim_sub:Nn #1#2
\dim_gsub:cn 4416 { \tex_advance:D #1 by - \__dim_eval:w #2 \__dim_eval_end: }
4417 \cs_new_protected:Npn \dim_gsub:Nn { \tex_global:D \dim_sub:Nn }
4418 \cs_generate_variant:Nn \dim_sub:Nn { c }
4419 \cs_generate_variant:Nn \dim_gsub:Nn { c }

```

(End definition for `\dim_add:Nn` and `\dim_add:cn`. These functions are documented on page 80.)

9.4 Utilities for dimension calculations

```

\dim_abs:n Functions for min, max, and absolute value with only one evaluation. The absolute value
\__dim_abs:N is evaluated by removing a leading - if present.
\dim_max:nn 4420 \cs_new:Npn \dim_abs:n #1
\dim_min:nn 4421 {
\__dim_maxmin:wwN 4422 \exp_after:wN \__dim_abs:N
4423 \dim_use:N \__dim_eval:w #1 \__dim_eval_end:
4424 }
4425 \cs_new:Npn \__dim_abs:N #1
4426 { \if_meaning:w - #1 \else: \exp_after:wN #1 \fi: }
4427 \cs_set:Npn \dim_max:nn #1#2
4428 {
4429 \dim_use:N \__dim_eval:w \exp_after:wN \__dim_maxmin:wwN
4430 \dim_use:N \__dim_eval:w #1 \exp_after:wN ;
4431 \dim_use:N \__dim_eval:w #2 ;
4432 >
4433 \__dim_eval_end:
4434 }
4435 \cs_set:Npn \dim_min:nn #1#2
4436 {
4437 \dim_use:N \__dim_eval:w \exp_after:wN \__dim_maxmin:wwN
4438 \dim_use:N \__dim_eval:w #1 \exp_after:wN ;
4439 \dim_use:N \__dim_eval:w #2 ;
4440 <
4441 \__dim_eval_end:
4442 }
4443 \cs_new:Npn \__dim_maxmin:wwN #1 ; #2 ; #3
4444 {
4445 \if_dim:w #1 #3 #2 ~
4446 #1
4447 \else:
4448 #2
4449 \fi:
4450 }

```

(End definition for `\dim_abs:n`. This function is documented on page 80.)

`\dim_ratio:nn` With dimension expressions, something like `10 pt * (5 pt / 10 pt)` will not work. Instead, the ratio part needs to be converted to an integer expression. Using `__int_value:w` forces everything into `sp`, avoiding any decimal parts.

```

4451 \cs_new:Npn \dim_ratio:nn #1#2
4452   { \__dim_ratio:n {#1} / \__dim_ratio:n {#2} }
4453 \cs_new:Npn \__dim_ratio:n #1
4454   { \__int_value:w \__dim_eval:w #1 \__dim_eval_end: }

```

(End definition for `\dim_ratio:nn`. This function is documented on page 81.)

9.5 Dimension expression conditionals

`\dim_compare_p:nNn` Simple comparison.

```

\dim_compare:nNnTF
4455 \prg_new_conditional:Npnn \dim_compare:nNn #1#2#3 { p , T , F , TF }
4456 {
4457   \if_dim:w \__dim_eval:w #1 #2 \__dim_eval:w #3 \__dim_eval_end:
4458   \prg_return_true: \else: \prg_return_false: \fi:
4459 }

```

(End definition for `\dim_compare:nNnTF`. This function is documented on page 81.)

`\dim_compare_p:n` This code is adapted from the `\int_compare:nTF` function. First make sure that there is at least one relation operator, by evaluating a dimension expression with a trailing `__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 `>`.

```

\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
4460 \prg_new_conditional:Npnn \dim_compare:n #1 { p , T , F , TF }
4461 {
4462   \exp_after:wN \__dim_compare:w
4463   \dim_use:N \__dim_eval:w #1 \__prg_compare_error:
4464 }
4465 \cs_new:Npn \__dim_compare:w #1 \__prg_compare_error:
4466 {
4467   \exp_after:wN \if_false: \exp:w \exp_end_continue_f:w
4468   \__dim_compare:wNN #1 ? { = \__dim_compare_end:w \else: } \q_stop
4469 }
4470 \exp_args:Nno \use:nn
4471 { \cs_new:Npn \__dim_compare:wNN #1 }
4472 { \tl_to_str:n {pt} }
4473 #2#3
4474 {
4475   \if_meaning:w = #3
4476   \use:c { __dim_compare_#2:w }
4477   \fi:
4478   #1 pt \exp_stop_f:

```

```

4479     \prg_return_false:
4480     \exp_after:wN \use_none_delimit_by_q_stop:w
4481     \fi:
4482     \reverse_if:N \if_dim:w #1 pt #2
4483     \exp_after:wN \__dim_compare:wNN
4484     \dim_use:N \__dim_eval:w #3
4485   }
4486   \cs_new:cpn { __dim_compare_ ! :w }
4487     #1 \reverse_if:N #2 ! #3 = { #1 #2 = #3 }
4488   \cs_new:cpn { __dim_compare_ = :w }
4489     #1 \__dim_eval:w = { #1 \__dim_eval:w }
4490   \cs_new:cpn { __dim_compare_ < :w }
4491     #1 \reverse_if:N #2 < #3 = { #1 #2 > #3 }
4492   \cs_new:cpn { __dim_compare_ > :w }
4493     #1 \reverse_if:N #2 > #3 = { #1 #2 < #3 }
4494   \cs_new:Npn \__dim_compare_end:w #1 \prg_return_false: #2 \q_stop
4495     { #1 \prg_return_false: \else: \prg_return_true: \fi: }

```

(End definition for \dim_compare:nnTF. This function is documented on page 82.)

\dim_case:nn 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.

```

\dim_case:nnTF
\__dim_case:nnTF
\__dim_case:nw
\__dim_case_end:nw
4496   \cs_new:Npn \dim_case:nnTF #1
4497   {
4498     \exp:w
4499     \exp_args:Nf \__dim_case:nnTF { \dim_eval:n {#1} }
4500   }
4501   \cs_new:Npn \dim_case:nnT #1#2#3
4502   {
4503     \exp:w
4504     \exp_args:Nf \__dim_case:nnTF { \dim_eval:n {#1} } {#2} {#3} { }
4505   }
4506   \cs_new:Npn \dim_case:nnF #1#2
4507   {
4508     \exp:w
4509     \exp_args:Nf \__dim_case:nnTF { \dim_eval:n {#1} } {#2} { }
4510   }
4511   \cs_new:Npn \dim_case:nn #1#2
4512   {
4513     \exp:w
4514     \exp_args:Nf \__dim_case:nnTF { \dim_eval:n {#1} } {#2} { } { }
4515   }
4516   \cs_new:Npn \__dim_case:nnTF #1#2#3#4
4517   { \__dim_case:nw {#1} #2 {#1} { } \q_mark {#3} \q_mark {#4} \q_stop }
4518   \cs_new:Npn \__dim_case:nw #1#2#3
4519   {
4520     \dim_compare:nNnTF {#1} = {#2}
4521     { \__dim_case_end:nw {#3} }
4522     { \__dim_case:nw {#1} }
4523   }

```

```
4524 \cs_new_eq:NN \__dim_case_end:nw \__prg_case_end:nw
```

(End definition for `\dim_case:nn`. This function is documented on page ??.)

9.6 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_until_do:nn
\dim_do_while:nn
\dim_do_until:nn
4525 \cs_set:Npn \dim_while_do:nn #1#2
4526 {
4527   \dim_compare:nT {#1}
4528   {
4529     #2
4530     \dim_while_do:nn {#1} {#2}
4531   }
4532 }
4533 \cs_set:Npn \dim_until_do:nn #1#2
4534 {
4535   \dim_compare:nF {#1}
4536   {
4537     #2
4538     \dim_until_do:nn {#1} {#2}
4539   }
4540 }
4541 \cs_set:Npn \dim_do_while:nn #1#2
4542 {
4543   #2
4544   \dim_compare:nT {#1}
4545   { \dim_do_while:nn {#1} {#2} }
4546 }
4547 \cs_set:Npn \dim_do_until:nn #1#2
4548 {
4549   #2
4550   \dim_compare:nF {#1}
4551   { \dim_do_until:nn {#1} {#2} }
4552 }
```

(End definition for `\dim_while_do:nn`. This function is documented on page 84.)

`\dim_while_do:nNnn` `while_do` and `do_while` functions for dimensions. Same as for the `int` type only the names have changed.

```
\dim_until_do:nNnn
\dim_do_while:nNnn
\dim_do_until:nNnn
4553 \cs_set:Npn \dim_while_do:nNnn #1#2#3#4
4554 {
4555   \dim_compare:nNnT {#1} #2 {#3}
4556   {
4557     #4
4558     \dim_while_do:nNnn {#1} #2 {#3} {#4}
4559   }
4560 }
4561 \cs_set:Npn \dim_until_do:nNnn #1#2#3#4
```

```

4562 {
4563   \dim_compare:nNnF {#1} #2 {#3}
4564   {
4565     #4
4566     \dim_until_do:nNnn {#1} #2 {#3} {#4}
4567   }
4568 }
4569 \cs_set:Npn \dim_do_while:nNnn #1#2#3#4
4570 {
4571   #4
4572   \dim_compare:nNnT {#1} #2 {#3}
4573   { \dim_do_while:nNnn {#1} #2 {#3} {#4} }
4574 }
4575 \cs_set:Npn \dim_do_until:nNnn #1#2#3#4
4576 {
4577   #4
4578   \dim_compare:nNnF {#1} #2 {#3}
4579   { \dim_do_until:nNnn {#1} #2 {#3} {#4} }
4580 }

```

(End definition for `\dim_while_do:nNnn`. This function is documented on page 84.)

9.7 Using dim expressions and variables

`\dim_eval:n` Evaluating a dimension expression expandably.

```

4581 \cs_new:Npn \dim_eval:n #1
4582 { \dim_use:N \__dim_eval:w #1 \__dim_eval_end: }

```

(End definition for `\dim_eval:n`. This function is documented on page 84.)

`\dim_use:N` Accessing a $\langle dim \rangle$.

`\dim_use:c` 4583 \cs_new_eq:NN \dim_use:N \tex_the:D

We hand-code this for some speed gain:

```

4584 %\cs_generate_variant:Nn \dim_use:N { c }
4585 \cs_new:Npn \dim_use:c #1 { \tex_the:D \cs:w #1 \cs_end: }

```

(End definition for `\dim_use:N` and `\dim_use:c`. These functions are documented on page 85.)

`\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.

```

4586 \cs_new:Npn \dim_to_decimal:n #1
4587 {
4588   \exp_after:wN
4589   \__dim_to_decimal:w \dim_use:N \__dim_eval:w (#1) \__dim_eval_end:
4590 }
4591 \use:x
4592 {

```

```

4593 \cs_new:Npn \exp_not:N \__dim_to_decimal:w
4594   ##1 . ##2 \tl_to_str:n { pt }
4595 }
4596 {
4597   \int_compare:nNnTF {#2} > \c_zero
4598     { #1 . #2 }
4599     { #1 }
4600 }

```

(End definition for `\dim_to_decimal:n`. This function is documented on page 85.)

`\dim_to_decimal_in_bp:n` Conversion to big points is done using a scaling inside `__dim_eval:w` as ε -TeX does that using 64-bit precision. Here, 800/803 is the integer fraction for 72/72.27. This is a common case so is hand-coded for accuracy (and speed).

```

4601 \cs_new:Npn \dim_to_decimal_in_bp:n #1
4602   { \dim_to_decimal:n { ( #1 ) * 800 / 803 } }

```

(End definition for `\dim_to_decimal_in_bp:n`. This function is documented on page 85.)

`\dim_to_decimal_in_sp:n` Another hard-coded conversion: this one is necessary to avoid things going off-scale.

```

4603 \cs_new:Npn \dim_to_decimal_in_sp:n #1
4604   { \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 85.)

`\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.

```

4605 \cs_new:Npn \dim_to_decimal_in_unit:nn #1#2
4606   {
4607     \dim_to_decimal:n
4608     {
4609       1pt *
4610       \dim_ratio:nn {#1} {#2}
4611     }
4612   }

```

(End definition for `\dim_to_decimal_in_unit:nn`. This function is documented on page 86.)

`\dim_to_fp:n` Defined in `l3fp-convert`, documented here.

(End definition for `\dim_to_fp:n`. This function is documented on page 86.)

9.8 Viewing dim variables

`\dim_show:N` Diagnostics.

```

\dim_show:c 4613 \cs_new_eq:NN \dim_show:N \__kernel_register_show:N
4614 \cs_generate_variant:Nn \dim_show:N { c }

```

(End definition for `\dim_show:N` and `\dim_show:c`. These functions are documented on page 86.)

\dim_show:n Diagnostics. We don't use the TeX primitive `\showthe` to show dimension expressions: this gives a more unified output.

```
4615 \cs_new_protected_nopar:Npn \dim_show:n
4616 { \_msg_show_wrap:Nn \dim_eval:n }
```

(End definition for `\dim_show:n`. This function is documented on page 86.)

9.9 Constant dimensions

\c_zero_dim Constant dimensions: in package mode, a couple of registers can be saved.

```
\c_max_dim 4617 \dim_const:Nn \c_zero_dim { 0 pt }
4618 \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 86.)

9.10 Scratch dimensions

\l_tmpa_dim We provide two local and two global scratch registers, maybe we need more or less.

```
\l_tmpb_dim 4619 \dim_new:N \l_tmpa_dim
\g_tmpa_dim 4620 \dim_new:N \l_tmpb_dim
\g_tmpb_dim 4621 \dim_new:N \g_tmpa_dim
4622 \dim_new:N \g_tmpb_dim
```

(End definition for `\l_tmpa_dim` and `\l_tmpb_dim`. These variables are documented on page 87.)

9.11 Creating and initialising skip variables

\skip_new:N Allocation of a new internal registers.

```
\skip_new:c 4623 \<package>
4624 \cs_new_protected:Npn \skip_new:N #1
4625 {
4626   \_chk_if_free_cs:N #1
4627   \cs:w newskip \cs_end: #1
4628 }
4629 \</package>
4630 \cs_generate_variant:Nn \skip_new:N { c }
```

(End definition for `\skip_new:N` and `\skip_new:c`. These functions are documented on page 87.)

\skip_const:Nn Contrarily to integer constants, we cannot avoid using a register, even for constants.

```
\skip_const:cn 4631 \cs_new_protected:Npn \skip_const:Nn #1
4632 {
4633   \skip_new:N #1
4634   \skip_gset:Nn #1
4635 }
4636 \cs_generate_variant:Nn \skip_const:Nn { c }
```

(End definition for `\skip_const:Nn` and `\skip_const:cn`. These functions are documented on page 87.)

`\skip_zero:N` Reset the register to zero.

```

\skip_zero:c 4637 \cs_new_protected:Npn \skip_zero:N #1 { #1 \c_zero_skip }
\skip_gzero:N 4638 \cs_new_protected:Npn \skip_gzero:N { \tex_global:D \skip_zero:N }
\skip_gzero:c 4639 \cs_generate_variant:Nn \skip_zero:N { c }
4640 \cs_generate_variant:Nn \skip_gzero:N { c }

```

(End definition for `\skip_zero:N` and `\skip_zero:c`. These functions are documented on page 87.)

`\skip_zero_new:N` Create a register if needed, otherwise clear it.

```

\skip_zero_new:c 4641 \cs_new_protected:Npn \skip_zero_new:N #1
\skip_gzero_new:N 4642 { \skip_if_exist:NNTF #1 { \skip_zero:N #1 } { \skip_new:N #1 } }
\skip_gzero_new:c 4643 \cs_new_protected:Npn \skip_gzero_new:N #1
4644 { \skip_if_exist:NNTF #1 { \skip_gzero:N #1 } { \skip_new:N #1 } }
4645 \cs_generate_variant:Nn \skip_zero_new:N { c }
4646 \cs_generate_variant:Nn \skip_gzero_new:N { c }

```

(End definition for `\skip_zero_new:N` and others. These functions are documented on page 87.)

`\skip_if_exist_p:N` Copies of the `cs` functions defined in `l3basics`.

```

\skip_if_exist_p:c 4647 \prg_new_eq_conditional:NNn \skip_if_exist:N \cs_if_exist:N
\skip_if_exist:NNTF 4648 { TF , T , F , p }
\skip_if_exist:cTF 4649 \prg_new_eq_conditional:NNn \skip_if_exist:c \cs_if_exist:c
4650 { TF , T , F , p }

```

(End definition for `\skip_if_exist:NNTF` and `\skip_if_exist:cTF`. These functions are documented on page 87.)

9.12 Setting skip variables

`\skip_set:Nn` Much the same as for dimensions.

```

\skip_set:cn 4651 \cs_new_protected:Npn \skip_set:Nn #1#2
\skip_gset:Nn 4652 { #1 ~ \etex_glueexpr:D #2 \scan_stop: }
\skip_gset:cn 4653 \cs_new_protected:Npn \skip_gset:Nn { \tex_global:D \skip_set:Nn }
4654 \cs_generate_variant:Nn \skip_set:Nn { c }
4655 \cs_generate_variant:Nn \skip_gset:Nn { c }

```

(End definition for `\skip_set:Nn` and `\skip_set:cn`. These functions are documented on page 88.)

`\skip_set_eq:NN` All straightforward.

```

\skip_set_eq:cN 4656 \cs_new_protected:Npn \skip_set_eq:NN #1#2 { #1 = #2 }
\skip_set_eq:Nc 4657 \cs_generate_variant:Nn \skip_set_eq:NN { c }
\skip_set_eq:cc 4658 \cs_generate_variant:Nn \skip_set_eq:NN { Nc , cc }
\skip_gset_eq:NN 4659 \cs_new_protected:Npn \skip_gset_eq:NN #1#2 { \tex_global:D #1 = #2 }
\skip_gset_eq:cN 4660 \cs_generate_variant:Nn \skip_gset_eq:NN { c }
\skip_gset_eq:Nc 4661 \cs_generate_variant:Nn \skip_gset_eq:NN { Nc , cc }
\skip_gset_eq:cc

```

(End definition for `\skip_set_eq:NN` and others. These functions are documented on page 88.)

```

\skip_add:Nn Using by here deals with the (incorrect) case \skip123.
\skip_add:cn 4662 \cs_new_protected:Npn \skip_add:Nn #1#2
\skip_gadd:Nn 4663 { \tex_advance:D #1 by \etex_glueexpr:D #2 \scan_stop: }
\skip_gadd:cn 4664 \cs_new_protected:Npn \skip_gadd:Nn { \tex_global:D \skip_add:Nn }
\skip_sub:Nn 4665 \cs_generate_variant:Nn \skip_add:Nn { c }
\skip_sub:cn 4666 \cs_generate_variant:Nn \skip_gadd:Nn { c }
\skip_gsub:Nn 4667 \cs_new_protected:Npn \skip_sub:Nn #1#2
\skip_gsub:cn 4668 { \tex_advance:D #1 by - \etex_glueexpr:D #2 \scan_stop: }
4669 \cs_new_protected:Npn \skip_gsub:Nn { \tex_global:D \skip_sub:Nn }
4670 \cs_generate_variant:Nn \skip_sub:Nn { c }
4671 \cs_generate_variant:Nn \skip_gsub:Nn { c }

```

(End definition for `\skip_add:Nn` and `\skip_add:cn`. These functions are documented on page 88.)

9.13 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.

```

4672 \prg_new_conditional:Npnn \skip_if_eq:nn #1#2 { p , T , F , TF }
4673 {
4674   \if_int_compare:w
4675     \__str_if_eq_x:nn { \skip_eval:n { #1 } } { \skip_eval:n { #2 } }
4676     = \c_zero
4677     \prg_return_true:
4678   \else:
4679     \prg_return_false:
4680   \fi:
4681 }

```

(End definition for `\skip_if_eq:nnTF`. This function is documented on page 88.)

`\skip_if_finite_p:n` With ϵ -TeX, we have an easy access to the order of infinities of the stretch and shrink components of a skip. However, to access both, we either need to evaluate the expression twice, or evaluate it, then call an auxiliary to extract both pieces of information from the result. Since we are going to need an auxiliary anyways, it is quicker to make it search for the string `fil` which characterizes infinite glue.
`\skip_if_finite:nTF`
`__skip_if_finite:wwNw`

```

4682 \cs_set_protected:Npn \__cs_tmp:w #1
4683 {
4684   \prg_new_conditional:Npnn \skip_if_finite:n ##1 { p , T , F , TF }
4685   {
4686     \exp_after:wN \__skip_if_finite:wwNw
4687     \skip_use:N \etex_glueexpr:D ##1 ; \prg_return_false:
4688     #1 ; \prg_return_true: \q_stop
4689   }
4690   \cs_new:Npn \__skip_if_finite:wwNw ##1 #1 ##2 ; ##3 ##4 \q_stop {##3}
4691 }
4692 \exp_args:No \__cs_tmp:w { \tl_to_str:n { fil } }

```

(End definition for `\skip_if_finite:nTF`. This function is documented on page 88.)

9.14 Using skip expressions and variables

\skip_eval:n Evaluating a skip expression expandably.

```
4693 \cs_new:Npn \skip_eval:n #1
4694 { \skip_use:N \etex_glueexpr:D #1 \scan_stop: }
```

(End definition for \skip_eval:n. This function is documented on page 89.)

\skip_use:N Accessing a $\langle skip \rangle$.

```
\skip_use:c 4695 \cs_new_eq:NN \skip_use:N \tex_the:D
4696 %\cs_generate_variant:Nn \skip_use:N { c }
4697 \cs_new:Npn \skip_use:c #1 { \tex_the:D \cs:w #1 \cs_end: }
```

(End definition for \skip_use:N and \skip_use:c. These functions are documented on page 89.)

9.15 Inserting skips into the output

\skip_horizontal:N Inserting skips.

```
\skip_horizontal:c 4698 \cs_new_eq:NN \skip_horizontal:N \tex_hskip:D
\skip_horizontal:n 4699 \cs_new:Npn \skip_horizontal:n #1
\skip_vertical:N 4700 { \skip_horizontal:N \etex_glueexpr:D #1 \scan_stop: }
\skip_vertical:c 4701 \cs_new_eq:NN \skip_vertical:N \tex_vskip:D
\skip_vertical:n 4702 \cs_new:Npn \skip_vertical:n #1
4703 { \skip_vertical:N \etex_glueexpr:D #1 \scan_stop: }
4704 \cs_generate_variant:Nn \skip_horizontal:N { c }
4705 \cs_generate_variant:Nn \skip_vertical:N { c }
```

(End definition for \skip_horizontal:N, \skip_horizontal:c, and \skip_horizontal:n. These functions are documented on page 90.)

9.16 Viewing skip variables

\skip_show:N Diagnostics.

```
\skip_show:c 4706 \cs_new_eq:NN \skip_show:N \__kernel_register_show:N
4707 \cs_generate_variant:Nn \skip_show:N { c }
```

(End definition for \skip_show:N and \skip_show:c. These functions are documented on page 89.)

\skip_show:n Diagnostics. We don't use the T_EX primitive `\showthe` to show skip expressions: this gives a more unified output.

```
4708 \cs_new_protected_nopar:Npn \skip_show:n
4709 { \__msg_show_wrap:Nn \skip_eval:n }
```

(End definition for \skip_show:n. This function is documented on page 89.)

9.17 Constant skips

\c_zero_skip Skips with no rubber component are just dimensions but need to terminate correctly.

```
\c_max_skip 4710 \skip_const:Nn \c_zero_skip { \c_zero_dim }
4711 \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 89.)

9.18 Scratch skips

We provide two local and two global scratch registers, maybe we need more or less.

```

\l_tmpa_skip 4712 \skip_new:N \l_tmpa_skip
\l_tmpb_skip 4713 \skip_new:N \l_tmpb_skip
\g_tmpa_skip 4714 \skip_new:N \g_tmpa_skip
\g_tmpb_skip 4715 \skip_new:N \g_tmpb_skip

```

(End definition for `\l_tmpa_skip` and `\l_tmpb_skip`. These variables are documented on page 90.)

9.19 Creating and initialising muskip variables

`\muskip_new:N` And then we add muskips.

```

\muskip_new:c 4716 <*package>
4717 \cs_new_protected:Npn \muskip_new:N #1
4718 {
4719     \__chk_if_free_cs:N #1
4720     \cs:w newmuskip \cs_end: #1
4721 }
4722 </package>
4723 \cs_generate_variant:Nn \muskip_new:N { c }

```

(End definition for `\muskip_new:N` and `\muskip_new:c`. These functions are documented on page 90.)

`\muskip_const:Nn` Contrarily to integer constants, we cannot avoid using a register, even for constants.

```

\muskip_const:cn 4724 \cs_new_protected:Npn \muskip_const:Nn #1
4725 {
4726     \muskip_new:N #1
4727     \muskip_gset:Nn #1
4728 }
4729 \cs_generate_variant:Nn \muskip_const:Nn { c }

```

(End definition for `\muskip_const:Nn` and `\muskip_const:cn`. These functions are documented on page 90.)

`\muskip_zero:N` Reset the register to zero.

```

\muskip_zero:c 4730 \cs_new_protected:Npn \muskip_zero:N #1
\muskip_gzero:N 4731 { #1 \c_zero_muskip }
\muskip_gzero:c 4732 \cs_new_protected:Npn \muskip_gzero:N { \tex_global:D \muskip_zero:N }
4733 \cs_generate_variant:Nn \muskip_zero:N { c }
4734 \cs_generate_variant:Nn \muskip_gzero:N { c }

```

(End definition for `\muskip_zero:N` and `\muskip_zero:c`. These functions are documented on page 90.)

`\muskip_zero_new:N` Create a register if needed, otherwise clear it.

```

\muskip_zero_new:c 4735 \cs_new_protected:Npn \muskip_zero_new:N #1
\muskip_gzero_new:N 4736 { \muskip_if_exist:NTF #1 { \muskip_zero:N #1 } { \muskip_new:N #1 } }
\muskip_gzero_new:c 4737 \cs_new_protected:Npn \muskip_gzero_new:N #1
4738 { \muskip_if_exist:NTF #1 { \muskip_gzero:N #1 } { \muskip_new:N #1 } }
4739 \cs_generate_variant:Nn \muskip_zero_new:N { c }
4740 \cs_generate_variant:Nn \muskip_gzero_new:N { c }

```

(End definition for `\muskip_zero_new:N` and others. These functions are documented on page 91.)

`\muskip_if_exist_p:N` Copies of the `cs` functions defined in `l3basics`.
`\muskip_if_exist_p:c` 4741 `\prg_new_eq_conditional:NnN \muskip_if_exist:N \cs_if_exist:N`
`\muskip_if_exist:NTF` 4742 `{ TF , T , F , p }`
`\muskip_if_exist:cTF` 4743 `\prg_new_eq_conditional:NnN \muskip_if_exist:c \cs_if_exist:c`
4744 `{ TF , T , F , p }`

(End definition for `\muskip_if_exist:NTF` and `\muskip_if_exist:cTF`. These functions are documented on page 91.)

9.20 Setting muskip variables

`\muskip_set:Nn` This should be pretty familiar.
`\muskip_set:cn` 4745 `\cs_new_protected:Npn \muskip_set:Nn #1#2`
`\muskip_gset:Nn` 4746 `{ #1 ~ \etex_muexpr:D #2 \scan_stop: }`
`\muskip_gset:cn` 4747 `\cs_new_protected:Npn \muskip_gset:Nn { \tex_global:D \muskip_set:Nn }`
4748 `\cs_generate_variant:Nn \muskip_set:Nn { c }`
4749 `\cs_generate_variant:Nn \muskip_gset:Nn { c }`

(End definition for `\muskip_set:Nn` and `\muskip_set:cn`. These functions are documented on page 91.)

`\muskip_set_eq:NN` All straightforward.
`\muskip_set_eq:cN` 4750 `\cs_new_protected:Npn \muskip_set_eq:NN #1#2 { #1 = #2 }`
`\muskip_set_eq:Nc` 4751 `\cs_generate_variant:Nn \muskip_set_eq:NN { c }`
`\muskip_set_eq:cc` 4752 `\cs_generate_variant:Nn \muskip_set_eq:NN { Nc , cc }`
`\muskip_gset_eq:NN` 4753 `\cs_new_protected:Npn \muskip_gset_eq:NN #1#2 { \tex_global:D #1 = #2 }`
`\muskip_gset_eq:cN` 4754 `\cs_generate_variant:Nn \muskip_gset_eq:NN { c }`
`\muskip_gset_eq:Nc` 4755 `\cs_generate_variant:Nn \muskip_gset_eq:NN { Nc , cc }`
`\muskip_gset_eq:cc`

(End definition for `\muskip_set_eq:NN` and others. These functions are documented on page 91.)

`\muskip_add:Nn` Using `by` here deals with the (incorrect) case `\muskip123`.
`\muskip_add:cn` 4756 `\cs_new_protected:Npn \muskip_add:Nn #1#2`
`\muskip_gadd:Nn` 4757 `{ \tex_advance:D #1 by \etex_muexpr:D #2 \scan_stop: }`
`\muskip_gadd:cn` 4758 `\cs_new_protected:Npn \muskip_gadd:Nn { \tex_global:D \muskip_add:Nn }`
`\muskip_sub:Nn` 4759 `\cs_generate_variant:Nn \muskip_add:Nn { c }`
`\muskip_sub:cn` 4760 `\cs_generate_variant:Nn \muskip_gadd:Nn { c }`
`\muskip_gsub:Nn` 4761 `\cs_new_protected:Npn \muskip_sub:Nn #1#2`
`\muskip_gsub:cn` 4762 `{ \tex_advance:D #1 by - \etex_muexpr:D #2 \scan_stop: }`
4763 `\cs_new_protected:Npn \muskip_gsub:Nn { \tex_global:D \muskip_sub:Nn }`
4764 `\cs_generate_variant:Nn \muskip_sub:Nn { c }`
4765 `\cs_generate_variant:Nn \muskip_gsub:Nn { c }`

(End definition for `\muskip_add:Nn` and `\muskip_add:cn`. These functions are documented on page 91.)

9.21 Using muskip expressions and variables

\muskip_eval:n Evaluating a muskip expression expandably.

```
4766 \cs_new:Npn \muskip_eval:n #1
4767 { \muskip_use:N \etex_muexpr:D #1 \scan_stop: }
```

(End definition for \muskip_eval:n. This function is documented on page 92.)

\muskip_use:N Accessing a $\langle muskip \rangle$.

```
\muskip_use:c 4768 \cs_new_eq:NN \muskip_use:N \tex_the:D
4769 \cs_generate_variant:Nn \muskip_use:N { c }
```

(End definition for \muskip_use:N and \muskip_use:c. These functions are documented on page 92.)

9.22 Viewing muskip variables

\muskip_show:N Diagnostics.

```
\muskip_show:c 4770 \cs_new_eq:NN \muskip_show:N \__kernel_register_show:N
4771 \cs_generate_variant:Nn \muskip_show:N { c }
```

(End definition for \muskip_show:N and \muskip_show:c. These functions are documented on page 92.)

\muskip_show:n Diagnostics. We don't use the T_EX primitive \showthe to show muskip expressions: this gives a more unified output.

```
4772 \cs_new_protected_nopar:Npn \muskip_show:n
4773 { \__msg_show_wrap:Nn \muskip_eval:n }
```

(End definition for \muskip_show:n. This function is documented on page 92.)

9.23 Constant muskips

\c_zero_muskip Constant muskips given by their value.

```
\c_max_muskip 4774 \muskip_const:Nn \c_zero_muskip { 0 mu }
4775 \muskip_const:Nn \c_max_muskip { 16383.99999 mu }
```

(End definition for \c_zero_muskip. This function is documented on page 92.)

9.24 Scratch muskips

\l_tmpa_muskip We provide two local and two global scratch registers, maybe we need more or less.

```
\l_tmpb_muskip 4776 \muskip_new:N \l_tmpa_muskip
\g_tmpa_muskip 4777 \muskip_new:N \l_tmpb_muskip
\g_tmpb_muskip 4778 \muskip_new:N \g_tmpa_muskip
4779 \muskip_new:N \g_tmpb_muskip
```

(End definition for \l_tmpa_muskip and \l_tmpb_muskip. These variables are documented on page 93.)

9.25 Deprecated functions

`_dim_strip_bp:n` Deprecated 2014-07-15.

`_dim_strip_pt:n` 4780 \cs_new_eq:NN _dim_strip_bp:n \dim_to_decimal_in_bp:n
4781 \cs_new_eq:NN _dim_strip_pt:n \dim_to_decimal:n

(End definition for _dim_strip_bp:n and _dim_strip_pt:n. These functions are documented on page ??.)

4782 \</initex | package>

10 l3tl implementation

4783 \<*initex | package>

4784 \<@@=tl>

A token list variable is a \TeX macro that holds tokens. By using the $\varepsilon\text{-TeX}$ primitive `\unexpanded` inside a \TeX `\edef` it is possible to store any tokens, including `#`, in this way.

10.1 Functions

`\tl_new:N` Creating new token list variables is a case of checking for an existing definition and doing the definition.

`\tl_new:c`

4785 \cs_new_protected:Npn \tl_new:N #1
4786 {
4787 _chk_if_free_cs:N #1
4788 \cs_gset_eq:NN #1 \c_empty_tl
4789 }
4790 \cs_generate_variant:Nn \tl_new:N { c }

(End definition for \tl_new:N and \tl_new:c. These functions are documented on page 95.)

`\tl_const:Nn` Constants are also easy to generate.

`\tl_const:Nx`

`\tl_const:cn`

`\tl_const:cx`

4791 \cs_new_protected:Npn \tl_const:Nn #1#2
4792 {
4793 _chk_if_free_cs:N #1
4794 \cs_gset_nopar:Npx #1 { \exp_not:n {#2} }
4795 }
4796 \cs_new_protected:Npn \tl_const:Nx #1#2
4797 {
4798 _chk_if_free_cs:N #1
4799 \cs_gset_nopar:Npx #1 {#2}
4800 }
4801 \cs_generate_variant:Nn \tl_const:Nn { c }
4802 \cs_generate_variant:Nn \tl_const:Nx { c }

(End definition for \tl_const:Nn and others. These functions are documented on page 95.)

\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.

\tl_clear:c

\tl_gclear:N 4803 \cs_new_protected:Npn \tl_clear:N #1
4804 { \tl_set_eq:NN #1 \c_empty_tl }

\tl_gclear:c 4805 \cs_new_protected:Npn \tl_gclear:N #1
4806 { \tl_gset_eq:NN #1 \c_empty_tl }

4807 \cs_generate_variant:Nn \tl_clear:N { c }

4808 \cs_generate_variant:Nn \tl_gclear:N { c }

(End definition for \tl_clear:N and \tl_clear:c. These functions are documented on page 95.)

\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.

\tl_clear_new:c

\tl_gclear_new:N 4809 \cs_new_protected:Npn \tl_clear_new:N #1
4810 { \tl_if_exist:NTF #1 { \tl_clear:N #1 } { \tl_new:N #1 } }

\tl_gclear_new:c 4811 \cs_new_protected:Npn \tl_gclear_new:N #1
4812 { \tl_if_exist:NTF #1 { \tl_gclear:N #1 } { \tl_new:N #1 } }

4813 \cs_generate_variant:Nn \tl_clear_new:N { c }

4814 \cs_generate_variant:Nn \tl_gclear_new:N { c }

(End definition for \tl_clear_new:N and \tl_clear_new:c. These functions are documented on page 95.)

\tl_set_eq:NN For setting token list variables equal to each other.

\tl_set_eq:Nc 4815 \cs_new_eq:NN \tl_set_eq:NN \cs_set_eq:NN

\tl_set_eq:cN 4816 \cs_new_eq:NN \tl_set_eq:cN \cs_set_eq:cN

\tl_set_eq:cc 4817 \cs_new_eq:NN \tl_set_eq:Nc \cs_set_eq:Nc

\tl_gset_eq:NN 4818 \cs_new_eq:NN \tl_set_eq:cc \cs_set_eq:cc

\tl_gset_eq:Nc 4819 \cs_new_eq:NN \tl_gset_eq:NN \cs_gset_eq:NN

\tl_gset_eq:cN 4820 \cs_new_eq:NN \tl_gset_eq:cN \cs_gset_eq:cN

\tl_gset_eq:cc 4821 \cs_new_eq:NN \tl_gset_eq:Nc \cs_gset_eq:Nc

4822 \cs_new_eq:NN \tl_gset_eq:cc \cs_gset_eq:cc

(End definition for \tl_set_eq:NN and others. These functions are documented on page 95.)

\tl_concat:NNN Concatenating token lists is easy.

\tl_concat:ccc 4823 \cs_new_protected:Npn \tl_concat:NNN #1#2#3

\tl_gconcat:NNN 4824 { \tl_set:Nx #1 { \exp_not:o {#2} \exp_not:o {#3} } }

\tl_gconcat:ccc 4825 \cs_new_protected:Npn \tl_gconcat:NNN #1#2#3

4826 { \tl_gset:Nx #1 { \exp_not:o {#2} \exp_not:o {#3} } }

4827 \cs_generate_variant:Nn \tl_concat:NNN { ccc }

4828 \cs_generate_variant:Nn \tl_gconcat:NNN { ccc }

(End definition for \tl_concat:NNN and \tl_concat:ccc. These functions are documented on page 95.)

\tl_if_exist_p:N Copies of the cs functions defined in l3basics.

\tl_if_exist_p:c 4829 \prg_new_eq_conditional:NNn \tl_if_exist:N \cs_if_exist:N { TF , T , F , p }

\tl_if_exist:NTF 4830 \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 and \tl_if_exist:cTF. These functions are documented on page 95.)

10.2 Constant token lists

`\c_empty_tl` Never full. We need to define that constant before using `\tl_new:N`.

```
4831 \tl_const:Nn \c_empty_tl { }
```

(End definition for `\c_empty_tl`. This variable is documented on page 108.)

`\c_space_tl` A space as a token list (as opposed to as a character).

```
4832 \tl_const:Nn \c_space_tl { ~ }
```

(End definition for `\c_space_tl`. This variable is documented on page 108.)

10.3 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 T_EX more or less redundant. The `\tl_set:No` version is done “by hand” as it is used quite a lot.

```
4833 \cs_new_protected:Npn \tl_set:Nn #1#2
4834 { \cs_set_nopar:Npx #1 { \exp_not:n {#2} } }
4835 \cs_new_protected:Npn \tl_set:No #1#2
4836 { \cs_set_nopar:Npx #1 { \exp_not:o {#2} } }
4837 \cs_new_protected:Npn \tl_set:Nx #1#2
4838 { \cs_set_nopar:Npx #1 {#2} }
4839 \cs_new_protected:Npn \tl_gset:Nn #1#2
4840 { \cs_gset_nopar:Npx #1 { \exp_not:n {#2} } }
4841 \cs_new_protected:Npn \tl_gset:No #1#2
4842 { \cs_gset_nopar:Npx #1 { \exp_not:o {#2} } }
4843 \cs_new_protected:Npn \tl_gset:Nx #1#2
4844 { \cs_gset_nopar:Npx #1 {#2} }
4845 \cs_generate_variant:Nn \tl_set:Nn { NV , Nv , Nf }
4846 \cs_generate_variant:Nn \tl_set:Nx { c }
4847 \cs_generate_variant:Nn \tl_set:Nn { c , co , cV , cv , cf }
4848 \cs_generate_variant:Nn \tl_gset:Nn { NV , Nv , Nf }
4849 \cs_generate_variant:Nn \tl_gset:Nx { c }
4850 \cs_generate_variant:Nn \tl_gset:Nn { c , co , cV , cv , cf }
\tl_gset:cv
\tl_gset:co
```

(End definition for `\tl_set:Nn` and others. These functions are documented on page 96.)

`\tl_put_gset:cn` Adding to the left is done directly to gain a little performance.

```
4851 \cs_new_protected:Npn \tl_put_left:Nn #1#2
4852 { \cs_set_nopar:Npx #1 { \exp_not:n {#2} \exp_not:o #1 } }
4853 \cs_new_protected:Npn \tl_put_left:Nv #1#2
4854 { \cs_set_nopar:Npx #1 { \exp_not:V #2 \exp_not:o #1 } }
4855 \cs_new_protected:Npn \tl_put_left:No #1#2
4856 { \cs_set_nopar:Npx #1 { \exp_not:o {#2} \exp_not:o #1 } }
4857 \cs_new_protected:Npn \tl_put_left:Nx #1#2
4858 { \cs_set_nopar:Npx #1 { #2 \exp_not:o #1 } }
4859 \cs_new_protected:Npn \tl_gput_left:Nn #1#2
4860 { \cs_gset_nopar:Npx #1 { \exp_not:n {#2} \exp_not:o #1 } }
4861 \cs_new_protected:Npn \tl_gput_left:Nv #1#2
\tl_gput_left:cn
\tl_gput_left:cV
\tl_gput_left:co
\tl_gput_left:cx
```

```

4862 { \cs_gset_nopar:Npx #1 { \exp_not:V #2 \exp_not:o #1 } }
4863 \cs_new_protected:Npn \tl_gput_left:No #1#2
4864 { \cs_gset_nopar:Npx #1 { \exp_not:o {#2} \exp_not:o #1 } }
4865 \cs_new_protected:Npn \tl_gput_left:Nx #1#2
4866 { \cs_gset_nopar:Npx #1 { #2 \exp_not:o {#1} } }
4867 \cs_generate_variant:Nn \tl_put_left:Nn { c }
4868 \cs_generate_variant:Nn \tl_put_left:NV { c }
4869 \cs_generate_variant:Nn \tl_put_left:No { c }
4870 \cs_generate_variant:Nn \tl_put_left:Nx { c }
4871 \cs_generate_variant:Nn \tl_gput_left:Nn { c }
4872 \cs_generate_variant:Nn \tl_gput_left:NV { c }
4873 \cs_generate_variant:Nn \tl_gput_left:No { c }
4874 \cs_generate_variant:Nn \tl_gput_left:Nx { c }

```

(End definition for `\tl_put_left:Nn` and others. These functions are documented on page 96.)

```

\tl_put_right:Nn The same on the right.
\tl_put_right:NV 4875 \cs_new_protected:Npn \tl_put_right:Nn #1#2
\tl_put_right:No 4876 { \cs_set_nopar:Npx #1 { \exp_not:o #1 \exp_not:n {#2} } }
\tl_put_right:Nx 4877 \cs_new_protected:Npn \tl_put_right:NV #1#2
\tl_put_right:cn 4878 { \cs_set_nopar:Npx #1 { \exp_not:o #1 \exp_not:V #2 } }
\tl_put_right:cV 4879 \cs_new_protected:Npn \tl_put_right:No #1#2
\tl_put_right:co 4880 { \cs_set_nopar:Npx #1 { \exp_not:o #1 \exp_not:o {#2} } }
\tl_put_right:cx 4881 \cs_new_protected:Npn \tl_put_right:Nx #1#2
\tl_gput_right:Nn 4882 { \cs_set_nopar:Npx #1 { \exp_not:o #1 #2 } }
\tl_gput_right:NV 4883 \cs_new_protected:Npn \tl_gput_right:Nn #1#2
\tl_gput_right:No 4884 { \cs_gset_nopar:Npx #1 { \exp_not:o #1 \exp_not:n {#2} } }
\tl_gput_right:Nx 4885 \cs_new_protected:Npn \tl_gput_right:NV #1#2
\tl_gput_right:cn 4886 { \cs_gset_nopar:Npx #1 { \exp_not:o #1 \exp_not:V #2 } }
\tl_gput_right:cV 4887 \cs_new_protected:Npn \tl_gput_right:No #1#2
\tl_gput_right:co 4888 { \cs_gset_nopar:Npx #1 { \exp_not:o #1 \exp_not:o {#2} } }
\tl_gput_right:cx 4889 \cs_new_protected:Npn \tl_gput_right:Nx #1#2
4890 { \cs_gset_nopar:Npx #1 { \exp_not:o {#1} #2 } }
4891 \cs_generate_variant:Nn \tl_put_right:Nn { c }
4892 \cs_generate_variant:Nn \tl_put_right:NV { c }
4893 \cs_generate_variant:Nn \tl_put_right:No { c }
4894 \cs_generate_variant:Nn \tl_put_right:Nx { c }
4895 \cs_generate_variant:Nn \tl_gput_right:Nn { c }
4896 \cs_generate_variant:Nn \tl_gput_right:NV { c }
4897 \cs_generate_variant:Nn \tl_gput_right:No { c }
4898 \cs_generate_variant:Nn \tl_gput_right:Nx { c }

```

(End definition for `\tl_put_right:Nn` and others. These functions are documented on page 96.)

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`.

```

4899 <*package>
4900 \tex_ifodd:D \l@expl@check@declarations@bool

```

```

4901 \cs_set_protected:Npn \__cs_tmp:w #1
4902 {
4903   \if_meaning:w \q_recursion_tail #1
4904   \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
4905   \fi:
4906   \use:x
4907   {
4908     \cs_set_protected:Npn #1 \exp_not:n { ##1 ##2 }
4909     {
4910       \__chk_if_exist_var:N \exp_not:n {##1}
4911       \exp_not:o { #1 {##1} {##2} }
4912     }
4913   }
4914   \__cs_tmp:w
4915 }
4916 \__cs_tmp:w
4917 \tl_set:Nn \tl_set:No \tl_set:Nx
4918 \tl_gset:Nn \tl_gset:No \tl_gset:Nx
4919 \tl_put_left:Nn \tl_put_left:NV
4920 \tl_put_left:No \tl_put_left:Nx
4921 \tl_gput_left:Nn \tl_gput_left:NV
4922 \tl_gput_left:No \tl_gput_left:Nx
4923 \tl_put_right:Nn \tl_put_right:NV
4924 \tl_put_right:No \tl_put_right:Nx
4925 \tl_gput_right:Nn \tl_gput_right:NV
4926 \tl_gput_right:No \tl_gput_right:Nx
4927 \q_recursion_tail \q_recursion_stop
4928 </package>

```

The two `set_eq` functions are done by hand as the internals there are a bit different.

```

4929 <*package>
4930 \cs_set_protected:Npn \tl_set_eq:NN #1#2
4931 {
4932   \__chk_if_exist_var:N #1
4933   \__chk_if_exist_var:N #2
4934   \cs_set_eq:NN #1 #2
4935 }
4936 \cs_set_protected:Npn \tl_gset_eq:NN #1#2
4937 {
4938   \__chk_if_exist_var:N #1
4939   \__chk_if_exist_var:N #2
4940   \cs_gset_eq:NN #1 #2
4941 }
4942 </package>

```

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.

```

4943 <*package>
4944 \cs_set_protected:Npn \tl_concat:NNN #1#2#3
4945 {

```

```

4946     \_chk_if_exist_var:N #1
4947     \_chk_if_exist_var:N #2
4948     \_chk_if_exist_var:N #3
4949     \tl_set:Nx #1 { \exp_not:o {#2} \exp_not:o {#3} }
4950   }
4951   \cs_set_protected:Npn \tl_gconcat:NNN #1#2#3
4952   {
4953     \_chk_if_exist_var:N #1
4954     \_chk_if_exist_var:N #2
4955     \_chk_if_exist_var:N #3
4956     \tl_gset:Nx #1 { \exp_not:o {#2} \exp_not:o {#3} }
4957   }
4958   \tex_fi:D
4959   \</package>

```

10.4 Reassigning token list category codes

`\c_tl_rescan_marker_tl` The rescanning code needs a special token list containing the same character (chosen here to be a colon) with two different category codes: it cannot appear in the tokens being rescanned since all colons have the same category code.

```

4960 \tl_const:Nx \c_tl_rescan_marker_tl { : \token_to_str:N : }

```

(End definition for `\c_tl_rescan_marker_tl`. This variable is documented on page ??.)

```

\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 \_
\tl_set_rescan:Nnx _tl_set_rescan:n and stored into \l_tl_internal_a_tl, then passed to #1#2 outside
\tl_set_rescan:cnn the group after expansion. The auxiliary \_tl_set_rescan:n is defined later: in the
\tl_set_rescan:cno simplest case, this auxiliary calls \_tl_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
\tl_gset_rescan:cnn ! File ended while scanning definition of ...
\tl_gset_rescan:cno
\tl_gset_rescan:cnx

\tl_rescan:nn The standard solution is to use an x-expanding assignment and set \everyeof to \exp_
\_tl_set_rescan:NNnn not:N to suppress the error at the end of the file. Since the rescanned tokens should not
\_tl_set_rescan_multi:n be expanded, they will be taken as a delimited argument of an auxiliary which wraps
\_tl_rescan:w 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.

```

4961 \cs_new_protected_nopar:Npn \tl_set_rescan:Nnn
4962 { \_tl_set_rescan:NNnn \tl_set:Nn }
4963 \cs_new_protected_nopar:Npn \tl_gset_rescan:Nnn
4964 { \_tl_set_rescan:NNnn \tl_gset:Nn }
4965 \cs_new_protected_nopar:Npn \tl_rescan:nn

```

```

4966 { \_tl\_set\_rescan:NNnn \prg\_do\_nothing: \use:n }
4967 \cs\_new\_protected:Npn \_tl\_set\_rescan:NNnn #1#2#3#4
4968 {
4969   \tl\_if\_empty:nTF {#4}
4970   {
4971     \group\_begin:
4972       #3
4973     \group\_end:
4974     #1 #2 { }
4975   }
4976   {
4977     \group\_begin:
4978     \exp\_args:No \etex\_everyeof:D { \c\_tl\_rescan\_marker\_tl \exp\_not:N }
4979     \int\_compare:nNnT \tex\_endlinechar:D = { 32 }
4980     { \tex\_endlinechar:D \c\_minus\_one }
4981     \tex\_newlinechar:D \tex\_endlinechar:D
4982     #3 \scan\_stop:
4983     \exp\_args:No \_tl\_set\_rescan:n { \tl\_to\_str:n {#4} }
4984     \exp\_args:NNNo
4985     \group\_end:
4986     #1 #2 \l\_tl\_internal\_a\_tl
4987   }
4988 }
4989 \cs\_new\_protected:Npn \_tl\_set\_rescan\_multi:n #1
4990 {
4991   \tl\_set:Nx \l\_tl\_internal\_a\_tl
4992   {
4993     \exp\_after:wN \_tl\_rescan:w
4994     \exp\_after:wN \prg\_do\_nothing:
4995     \etex\_scantokens:D {#1}
4996   }
4997 }
4998 \exp\_args:Nno \use:nn
4999 { \cs\_new:Npn \_tl\_rescan:w #1 } \c\_tl\_rescan\_marker\_tl
5000 { \exp\_not:o {#1} }
5001 \cs\_generate\_variant:Nn \tl\_set\_rescan:NNn { Nno , Nnx }
5002 \cs\_generate\_variant:Nn \tl\_set\_rescan:NNn { c , cno , cnx }
5003 \cs\_generate\_variant:Nn \tl\_gset\_rescan:NNn { Nno , Nnx }
5004 \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 97.)

`_tl_set_rescan:n` This function calls `_tl_set_rescan_multiple:n` or `_tl_set_rescan_single:nn`
`_tl_set_rescan:NnTF` { ' } depending on whether its argument is a single-line fragment of code/data or
`_tl_set_rescan_single:nn` is made of multiple lines by testing for the presence of a `\newlinechar` character. If
`_tl_set_rescan_single_aux:nn` `\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 `__tl_set_rescan_multi:n`, except that `__tl_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 `__tl_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__tl_rescan_marker_tl`, 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.

```

5005 \group_begin:
5006   \tex_catcode:D '\^~@ = 12 \scan_stop:
5007   \cs_new_protected:Npn \__tl_set_rescan:n #1
5008     {
5009       \int_compare:nNnTF \tex_newlinechar:D < \c_zero
5010         { \use_ii:nn }
5011         {
5012           \char_set_lccode:nn { 0 } { \tex_newlinechar:D }
5013           \tex_lowercase:D { \__tl_set_rescan:NnTF ^~@ } {#1}
5014         }
5015         { \__tl_set_rescan_multi:n }
5016         { \__tl_set_rescan_single:nn { ' } }
5017       {#1}
5018     }
5019   \cs_new_protected:Npn \__tl_set_rescan:NnTF #1#2
5020     { \tl_if_in:nnTF {#2} {#1} }
5021   \cs_new_protected:Npn \__tl_set_rescan_single:nn #1

```

```

5022 {
5023   \int_compare:nNnTF
5024     { \char_value_catcode:n { '#1 } / \c_three } = \c_four
5025     { \__tl_set_rescan_single_aux:nn {#1} }
5026     {
5027       \int_compare:nNnTF { '#1 } < { '\~ }
5028       {
5029         \char_set_lccode:nn { 0 } { '#1 + 1 }
5030         \tex_lowercase:D { \__tl_set_rescan_single:nn { ^^@ } }
5031       }
5032       { \__tl_set_rescan_single_aux:nn { } }
5033     }
5034   }
5035   \cs_new_protected:Npn \__tl_set_rescan_single_aux:nn #1#2
5036   {
5037     \tex_endlinechar:D \c_minus_one
5038     \use:x
5039     {
5040       \exp_not:N \use:n
5041       {
5042         \exp_not:n { \cs_set:Npn \__tl_rescan:w ##1 }
5043         \exp_after:wN \__tl_rescan:w
5044         \exp_after:wN \prg_do_nothing:
5045         \etex_scantokens:D {#1}
5046       }
5047       \c__tl_rescan_marker_tl
5048     }
5049     { \exp_not:o {##1} }
5050   \tl_set:Nx \l__tl_internal_a_tl
5051   {
5052     \int_compare:nNnT
5053     {
5054       \char_value_catcode:n
5055       { \exp_last_unbraced:Nf ' \str_head:n {#2} ~ }
5056     }
5057     = \c_ten { ~ }
5058     \exp_after:wN \__tl_rescan:w
5059     \exp_after:wN \prg_do_nothing:
5060     \etex_scantokens:D { #2 #1 }
5061   }
5062 }
5063 \group_end:

```

(End definition for `__tl_set_rescan:n` and `__tl_set_rescan:NnTF`.)

10.5 Reassigning token list character codes

`\tl_to_lowercase:n` Just some names for a few primitives: we take care of wrapping the argument in braces.
`\tl_to_uppercase:n`

```

5064 \cs_new_protected:Npn \tl_to_lowercase:n #1
5065 { \tex_lowercase:D {#1} }

```

```

5066 \cs_new_protected:Npn \tl_to_uppercase:n #1
5067 { \tex_uppercase:D {#1} }

```

(End definition for `\tl_to_lowercase:n`. This function is documented on page 98.)

10.6 Modifying token list variables

`\tl_replace_all:Nnn` All of the `replace` functions call `__tl_replace:NnnNnn` with appropriate arguments. `\tl_replace_all:cnn` The first two arguments are explained later. The next controls whether the replacement 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 $\langle tl\ var \rangle \{ \langle pattern \rangle \} \{ \langle replacement \rangle \}$ provided by the user. When describing the auxiliary functions below, we denote the contents of the $\langle tl\ var \rangle$ by $\langle token\ list \rangle$.

```

5068 \cs_new_protected_nopar:Npn \tl_replace_once:Nnn
5069 { \__tl_replace:NnnNnn \q_mark ? \__tl_replace_wrap:w \tl_set:Nx }
5070 \cs_new_protected_nopar:Npn \tl_greplace_once:Nnn
5071 { \__tl_replace:NnnNnn \q_mark ? \__tl_replace_wrap:w \tl_gset:Nx }
5072 \cs_new_protected_nopar:Npn \tl_replace_all:Nnn
5073 { \__tl_replace:NnnNnn \q_mark ? \__tl_replace_next:w \tl_set:Nx }
5074 \cs_new_protected_nopar:Npn \tl_greplace_all:Nnn
5075 { \__tl_replace:NnnNnn \q_mark ? \__tl_replace_next:w \tl_gset:Nx }
5076 \cs_generate_variant:Nn \tl_replace_once:Nnn { c }
5077 \cs_generate_variant:Nn \tl_greplace_once:Nnn { c }
5078 \cs_generate_variant:Nn \tl_replace_all:Nnn { c }
5079 \cs_generate_variant:Nn \tl_greplace_all:Nnn { c }

```

(End definition for `\tl_replace_all:Nnn` and `\tl_replace_all:cnn`. These functions are documented on page 96.)

```

\__tl_replace:NnnNnn
\__tl_replace_auxi:NnnNnnNnn
\__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 $\langle delimiter \rangle$ with the following properties:

- all occurrences of the $\langle pattern \rangle$ #6 in “ $\langle token\ list \rangle \langle delimiter \rangle$ ” belong to the $\langle token\ list \rangle$ and have no overlap with the $\langle delimiter \rangle$,
- the first occurrence of the $\langle delimiter \rangle$ in “ $\langle token\ list \rangle \langle delimiter \rangle$ ” is the trailing $\langle delimiter \rangle$.

We first find the building blocks for the $\langle delimiter \rangle$, namely two tokens $\langle A \rangle$ and $\langle B \rangle$ such that $\langle A \rangle$ does not appear in #6 and #6 is not $\langle B \rangle$ (this condition is trivial if #6 has more than one token). Then we consider the delimiters “ $\langle A \rangle$ ” and “ $\langle A \rangle \langle A \rangle^n \langle B \rangle \langle A \rangle^n \langle B \rangle$ ”, for $n \geq 1$, where $\langle A \rangle^n$ denotes n copies of $\langle A \rangle$, and we choose as our $\langle delimiter \rangle$ the first one which is not in the $\langle token\ list \rangle$.

Every delimiter in the set obeys the first condition: #6 does not contain $\langle A \rangle$ hence cannot be overlapping with the $\langle token\ list \rangle$ and the $\langle delimiter \rangle$, and it cannot be within the $\langle delimiter \rangle$ since it would have to be in one of the two $\langle B \rangle$ hence be equal to this single token (or empty, but this is an error case filtered separately). Given the particular form of these delimiters, for which no prefix is also a suffix, the second condition is actually a

consequence of the weaker condition that the $\langle \text{delimiter} \rangle$ we choose does not appear in the $\langle \text{token list} \rangle$. Additionally, the set of delimiters is such that a $\langle \text{token list} \rangle$ of n tokens can contain at most $O(n^{1/2})$ of them, hence we find a $\langle \text{delimiter} \rangle$ with at most $O(n^{1/2})$ tokens in a time at most $O(n^{3/2})$. Bear in mind that these upper bounds are reached only in very contrived scenarios: we include the case “ $\langle A \rangle$ ” in the list of delimiters to try, so that the $\langle \text{delimiter} \rangle$ will simply be $\backslash \text{q_mark}$ in the most common situation where neither the $\langle \text{token list} \rangle$ nor the $\langle \text{pattern} \rangle$ contains $\backslash \text{q_mark}$.

Let us now ahead, optimizing for this most common case. First, two special cases: an empty $\langle \text{pattern} \rangle$ #6 is an error, and if #1 is absent from both the $\langle \text{token list} \rangle$ #5 and the $\langle \text{pattern} \rangle$ #6 then we can use it as the $\langle \text{delimiter} \rangle$ through $\backslash _ \text{tl_replace_auxii:nNNNnn} \{ \#1 \}$. Otherwise, we end up calling $\backslash _ \text{tl_replace:NnnNNnn}$ repeatedly with the first two arguments $\backslash \text{q_mark} \{ ? \}$, $\backslash ? \{ ?? \}$, $\backslash ?? \{ ??? \}$, and so on, until #6 does not contain the control sequence #1, which we take as our $\langle A \rangle$. The argument #2 only serves to collect ? characters for #1. Note that the order of the tests means that the first two are done every time, which is wasteful (for instance, we repeatedly test for the emptiness of #6). However, this is rare enough not to matter. Finally, choose $\langle B \rangle$ to be $\backslash \text{q_nil}$ or $\backslash \text{q_stop}$ such that it is not equal to #6.

The $\backslash _ \text{tl_replace_auxi:NnnNNNnn}$ auxiliary receives $\{ \langle A \rangle \}$ and $\{ \langle A \rangle^n \langle B \rangle \}$ as its arguments, initially with $n = 1$. If “ $\langle A \rangle \langle A \rangle^n \langle B \rangle \langle A \rangle^n \langle B \rangle$ ” is in the $\langle \text{token list} \rangle$ then increase n and try again. Once it is not anymore in the $\langle \text{token list} \rangle$ we take it as our $\langle \text{delimiter} \rangle$ and pass this to the auxii auxiliary.

```

5080 \cs_new_protected:Npn \_tl_replace:NnnNNnn #1#2#3#4#5#6#7
5081 {
5082   \tl_if_empty:nTF {#6}
5083   {
5084     \_msg_kernel_error:nmx { kernel } { empty-search-pattern }
5085     { \tl_to_str:n {#7} }
5086   }
5087   {
5088     \tl_if_in:ontF { #5 #6 } {#1}
5089     {
5090       \tl_if_in:nnTF {#6} {#1}
5091       { \exp_args:Nc \_tl_replace:NnnNNnn {#2} {#2?} }
5092       {
5093         \quark_if_nil:nTF {#6}
5094         { \_tl_replace_auxi:NnnNNNnn #5 {#1} { #1 \q_stop } }
5095         { \_tl_replace_auxi:NnnNNNnn #5 {#1} { #1 \q_nil } }
5096       }
5097     }
5098     { \_tl_replace_auxii:nNNNnn {#1} }
5099     #3#4#5 {#6} {#7}
5100   }
5101 }
5102 \cs_new_protected:Npn \_tl_replace_auxi:NnnNNNnn #1#2#3
5103 {
5104   \tl_if_in:NnTF #1 { #2 #3 #3 }
5105   { \_tl_replace_auxi:NnnNNNnn #1 { #2 #3 } {#2} }
5106   { \_tl_replace_auxii:nNNNnn { #2 #3 #3 } }

```

5107 }

The auxiliary `__tl_replace_auxii:nNNNnn` receives the following arguments: $\{\langle delimiter \rangle\}$ $\langle function \rangle$ $\langle assignment \rangle$ $\langle tl var \rangle$ $\{\langle pattern \rangle\}$ $\{\langle replacement \rangle\}$. All of its work is done between `\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 $\langle assignment \rangle$ `#3` to the $\langle tl var \rangle$ `#4`. The auxiliary `__tl_replace_next:w` is called, followed by the $\langle token list \rangle$, some tokens including the $\langle delimiter \rangle$ `#1`, followed by the $\langle pattern \rangle$ `#5`. This auxiliary finds an argument delimited by `#5` (the presence of a trailing `#5` avoids runaway arguments) and calls `__tl_replace_wrap:w` to test whether this `#5` is found within the $\langle token list \rangle$ or is the trailing one.

If on the one hand it is found within the $\langle token list \rangle$, then `##1` cannot contain the $\langle delimiter \rangle$ `#1` that we worked so hard to obtain, thus `__tl_replace_wrap:w` gets `##1` as its own argument `##1`, and wraps it using `\exp_not:o` for consumption by the x-expanding assignment. It also finds `\exp_not:n` as `##2` and does nothing to it, thus letting through `\exp_not:n {\langle replacement \rangle}` into the assignment. (Note that `__tl_replace_next:w` and `__tl_replace_wrap:w` are always called followed by `\prg_do_nothing:` to avoid losing braces when grabbing delimited arguments, hence the use of `\exp_not:o` rather than `\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 $\langle remaining tokens \rangle$ in the $\langle token list \rangle$ and `##2` is some $\langle ending code \rangle$ 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 $\langle pattern \rangle$ `#5`, then `##1` is “`\prg_do_nothing: \langle token list \rangle \langle delimiter \rangle {\langle ending code \rangle}`”, hence `__tl_replace_wrap:w` finds “`\prg_do_nothing: \langle token list \rangle`” as `##1` and the $\langle ending code \rangle$ as `##2`. It leaves the $\langle token list \rangle$ into the assignment and unbraces the $\langle ending code \rangle$ which removes what remains (essentially the $\langle delimiter \rangle$ and $\langle replacement \rangle$).

```

5108 \cs_new_protected:Npn \__tl_replace_auxii:nNNNnn #1#2#3#4#5#6
5109 {
5110   \group_align_safe_begin:
5111   \cs_set:Npn \__tl_replace_wrap:w ##1 #1 ##2 { \exp_not:o {##1} ##2 }
5112   \cs_set:Npx \__tl_replace_next:w ##1 #5
5113   {
5114     \exp_not:N \__tl_replace_wrap:w ##1
5115     \exp_not:n { #1 }
5116     \exp_not:n { \exp_not:n {#6} }
5117     \exp_not:n { #2 \prg_do_nothing: }
5118   }
5119   #3 #4
5120   {
5121     \exp_after:wN \__tl_replace_next:w
5122     \exp_after:wN \prg_do_nothing: #4
5123     #1
5124     {
5125       \if_false: { \fi: }
5126       \exp_after:wN \use_none:n \exp_after:wN { \if_false: } \fi:

```

```

5127     }
5128     #5
5129     \q_recursion_stop
5130   }
5131   \group_align_safe_end:
5132 }
5133 \cs_new_eq:NN \__tl_replace_wrap:w ?
5134 \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.
\tl_remove_once:cn 5135 \cs_new_protected:Npn \tl_remove_once:Nn #1#2
\tl_gremove_once:Nn 5136 { \tl_replace_once:Nnn #1 {#2} { } }
\tl_gremove_once:cn 5137 \cs_new_protected:Npn \tl_gremove_once:Nn #1#2
5138 { \tl_greplace_once:Nnn #1 {#2} { } }
5139 \cs_generate_variant:Nn \tl_remove_once:Nn { c }
5140 \cs_generate_variant:Nn \tl_gremove_once:Nn { c }

```

(End definition for `\tl_remove_once:Nn` and `\tl_remove_once:cn`. These functions are documented on page 96.)

```

\tl_remove_all:Nn Removal is just a special case of replacement.
\tl_remove_all:cn 5141 \cs_new_protected:Npn \tl_remove_all:Nn #1#2
\tl_gremove_all:Nn 5142 { \tl_replace_all:Nnn #1 {#2} { } }
\tl_gremove_all:cn 5143 \cs_new_protected:Npn \tl_gremove_all:Nn #1#2
5144 { \tl_greplace_all:Nnn #1 {#2} { } }
5145 \cs_generate_variant:Nn \tl_remove_all:Nn { c }
5146 \cs_generate_variant:Nn \tl_gremove_all:Nn { c }

```

10.7 Token list conditionals

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`.

```

\tl_if_blank_p:n \__tl_if_empty_return:o
\tl_if_blank_p:V 5147 \prg_new_conditional:Npnn \tl_if_blank:n #1 { p , T , F , TF }
\tl_if_blank_p:o 5148 { \__tl_if_empty_return:o { \use_none:n #1 ? } }
\tl_if_blank:nTF 5149 \cs_generate_variant:Nn \tl_if_blank_p:n { V }
\tl_if_blank:VTF 5150 \cs_generate_variant:Nn \tl_if_blank:nT { V }
\tl_if_blank:oTF 5151 \cs_generate_variant:Nn \tl_if_blank:nF { V }
\__tl_if_blank_p:NNw 5152 \cs_generate_variant:Nn \tl_if_blank:nTF { V }
5153 \cs_generate_variant:Nn \tl_if_blank_p:n { o }
5154 \cs_generate_variant:Nn \tl_if_blank:nT { o }
5155 \cs_generate_variant:Nn \tl_if_blank:nF { o }
5156 \cs_generate_variant:Nn \tl_if_blank:nTF { o }

```

(End definition for `\tl_remove_all:Nn` and `\tl_remove_all:cn`. These functions are documented on page 97.)

`\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).

`\tl_if_empty_p:c`

`\tl_if_empty:NTF`

`\tl_if_empty:cTF`

```

5157 \prg_new_conditional:Npnn \tl_if_empty:N #1 { p , T , F , TF }
5158 {
5159   \if_meaning:w #1 \c_empty_tl
5160   \prg_return_true:
5161   \else:
5162     \prg_return_false:
5163   \fi:
5164 }
5165 \cs_generate_variant:Nn \tl_if_empty_p:N { c }
5166 \cs_generate_variant:Nn \tl_if_empty:N { NT }
5167 \cs_generate_variant:Nn \tl_if_empty:N { NF }
5168 \cs_generate_variant:Nn \tl_if_empty:N { NTF }

```

(End definition for `\tl_if_empty:NTF` and `\tl_if_empty:cTF`. These functions are documented on page 99.)

`\tl_if_empty_p:n` Convert the argument to a string: this will be empty if and only if the argument is. Then

`\tl_if_empty_p:V` `\if_meaning:w \q_nil ... \q_nil` is true if and only if the string ... is empty. It

`\tl_if_empty:nTF` could be tempting to use `\if_meaning:w \q_nil #1 \q_nil` directly. This fails on a

`\tl_if_empty:VTF` 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...

```

5169 \prg_new_conditional:Npnn \tl_if_empty:n #1 { p , TF , T , F }
5170 {
5171   \exp_after:wN \if_meaning:w \exp_after:wN \q_nil
5172   \tl_to_str:n {#1} \q_nil
5173   \prg_return_true:
5174   \else:
5175     \prg_return_false:
5176   \fi:
5177 }
5178 \cs_generate_variant:Nn \tl_if_empty_p:n { V }
5179 \cs_generate_variant:Nn \tl_if_empty:n { nTF }
5180 \cs_generate_variant:Nn \tl_if_empty:n { nT }
5181 \cs_generate_variant:Nn \tl_if_empty:n { nF }

```

(End definition for `\tl_if_empty:nTF` and `\tl_if_empty:VTF`. These functions are documented on page 99.)

`\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

`\tl_if_empty:oTF` function to it. The test for emptiness is based on `\tl_if_empty:n(TF)`, but the expansion is hard-coded for efficiency, as this auxiliary function is used in many places. Note

`__tl_if_empty_return:o` that this works because `\etex_detokenize:D` expands tokens that follow until reading a catcode 1 (begin-group) token.

```

5182 \cs_new:Npn \__tl_if_empty_return:o #1

```

```

5183 {
5184   \exp_after:wN \if_meaning:w \exp_after:wN \q_nil
5185   \etex_detokenize:D \exp_after:wN {#1} \q_nil
5186   \prg_return_true:
5187   \else:
5188     \prg_return_false:
5189   \fi:
5190 }
5191 \prg_new_conditional:Npnn \tl_if_empty:o #1 { p , TF , T , F }
5192 { \_tl_if_empty_return:o {#1} }

```

(End definition for `\tl_if_empty:oTF`. This function is documented on page ??.)

`\tl_if_eq_p:NN` Returns `\c_true_bool` if and only if the two token list variables are equal.

```

\l__tl_internal_a_tl 5193 \prg_new_conditional:Npnn \tl_if_eq:NN #1#2 { p , T , F , TF }
\l__tl_internal_b_tl 5194 {
5195   \if_meaning:w #1 #2
5196   \prg_return_true:
5197   \else:
5198     \prg_return_false:
5199   \fi:
5200 }
5201 \cs_generate_variant:Nn \tl_if_eq_p:NN { Nc , c , cc }
5202 \cs_generate_variant:Nn \tl_if_eq:NNTF { Nc , c , cc }
5203 \cs_generate_variant:Nn \tl_if_eq:NNT { Nc , c , cc }
5204 \cs_generate_variant:Nn \tl_if_eq:NNF { Nc , c , cc }

```

(End definition for `\tl_if_eq:NNTF` and others. These functions are documented on page 99.)

`\tl_if_eq:nnTF` A simple store and compare routine.

```

\l__tl_internal_a_tl 5205 \prg_new_protected_conditional:Npnn \tl_if_eq:nn #1#2 { T , F , TF }
\l__tl_internal_b_tl 5206 {
5207   \group_begin:
5208     \tl_set:Nn \l__tl_internal_a_tl {#1}
5209     \tl_set:Nn \l__tl_internal_b_tl {#2}
5210     \if_meaning:w \l__tl_internal_a_tl \l__tl_internal_b_tl
5211     \group_end:
5212     \prg_return_true:
5213   \else:
5214     \group_end:
5215     \prg_return_false:
5216   \fi:
5217 }
5218 \tl_new:N \l__tl_internal_a_tl
5219 \tl_new:N \l__tl_internal_b_tl

```

(End definition for `\tl_if_eq:nnTF`. This function is documented on page 99.)

`\tl_if_in:NnTF` See `\tl_if_in:nn(TF)` for further comments. Here we simply expand the token list variable and pass it to `\tl_if_in:nn(TF)`.
`\tl_if_in:cnTF`

```

5220 \cs_new_protected_nopar:Npn \tl_if_in:NnT { \exp_args:No \tl_if_in:nnT }
5221 \cs_new_protected_nopar:Npn \tl_if_in:NnF { \exp_args:No \tl_if_in:nnF }
5222 \cs_new_protected_nopar:Npn \tl_if_in:NnTF { \exp_args:No \tl_if_in:nnTF }
5223 \cs_generate_variant:Nn \tl_if_in:NnT { c }
5224 \cs_generate_variant:Nn \tl_if_in:NnF { c }
5225 \cs_generate_variant:Nn \tl_if_in:NnTF { c }

```

(End definition for `\tl_if_in:NnTF` and `\tl_if_in:cnTF`. These functions are documented on page 99.)

`\tl_if_in:nnTF` Once more, the test relies on the emptiness test for robustness. The function `__tl_tmp:w` removes tokens until the first occurrence of `#2`. If this does not appear in `#1`, then the final `#2` is removed, leaving an empty token list. Otherwise some tokens remain, and the test is false. See `\tl_if_empty:n(TF)` 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:`.

```

5226 \prg_new_protected_conditional:Npnn \tl_if_in:nn #1#2 { T , F , TF }
5227 {
5228   \if_false: { \fi:
5229     \cs_set:Npn \__tl_tmp:w ##1 #2 { }
5230     \tl_if_empty:oTF { \__tl_tmp:w #1 {} {} #2 }
5231     { \prg_return_false: } { \prg_return_true: }
5232     \if_false: } \fi:
5233 }
5234 \cs_generate_variant:Nn \tl_if_in:nnT { V , o , no }
5235 \cs_generate_variant:Nn \tl_if_in:nnF { V , o , no }
5236 \cs_generate_variant:Nn \tl_if_in:nnTF { V , o , no }

```

(End definition for `\tl_if_in:nnTF` and others. These functions are documented on page 99.)

`\tl_if_single_p:N` Expand the token list and feed it to `\tl_if_single:n`.

```

\red\tl_if_single:N\TF
5237 \cs_new:Npn \tl_if_single_p:N { \exp_args:No \tl_if_single_p:n }
5238 \cs_new:Npn \tl_if_single:NNT { \exp_args:No \tl_if_single:nT }
5239 \cs_new:Npn \tl_if_single:NNF { \exp_args:No \tl_if_single:nF }
5240 \cs_new:Npn \tl_if_single:NNTF { \exp_args:No \tl_if_single:nTF }

```

(End definition for `\tl_if_single:NNTF`. This function is documented on page 100.)

`\tl_if_single_p:n` This test is similar to `\tl_if_empty:nTF`. Expanding `\use_none:nn #1 ??` once yields an empty result if `#1` is blank, a single `?` if `#1` has a single item, and otherwise yields some tokens ending with `??`. Then, `\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 Lua \TeX . Instead, `__tl_if_single:nnw` 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.

```

5241 \prg_new_conditional:Npnn \tl_if_single:n #1 { p , T , F , TF }
5242 {
5243   \if_catcode:w ^ \exp_after:wN \__tl_if_single:nnw
5244     \tl_to_str:n \exp_after:wN { \use_none:nn #1 ?? } ^ ? \q_stop
5245     \prg_return_true:
5246   \else:
5247     \prg_return_false:
5248   \fi:
5249 }
5250 \cs_new:Npn \__tl_if_single:nnw #1#2#3 \q_stop {#2}

```

(End definition for `\tl_if_single:nTF`. This function is documented on page 100.)

<pre> \tl_case:Nn \tl_case:cn \tl_case:NnTF \tl_case:cnTF __tl_case:nnTF __tl_case:Nw __prg_case_end:nw __tl_case_end:nw </pre>	<p>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.</p> <pre> 5251 \cs_new:Npn \tl_case:Nn #1#2 5252 { 5253 \exp:w 5254 __tl_case:NnTF #1 {#2} { } { } 5255 } 5256 \cs_new:Npn \tl_case:NnT #1#2#3 5257 { 5258 \exp:w 5259 __tl_case:NnTF #1 {#2} {#3} { } 5260 } 5261 \cs_new:Npn \tl_case:NnF #1#2#3 5262 { 5263 \exp:w 5264 __tl_case:NnTF #1 {#2} { } {#3} 5265 } 5266 \cs_new:Npn \tl_case:NnTF #1#2 5267 { 5268 \exp:w 5269 __tl_case:NnTF #1 {#2} 5270 } 5271 \cs_new:Npn __tl_case:NnTF #1#2#3#4 5272 { __tl_case:Nw #1 #2 #1 { } \q_mark {#3} \q_mark {#4} \q_stop } 5273 \cs_new:Npn __tl_case:Nw #1#2#3 5274 { 5275 \tl_if_eq:NNTF #1 #2 5276 { __tl_case_end:nw {#3} } 5277 { __tl_case:Nw #1 } 5278 } </pre>
---	--

```

5279 \cs_generate_variant:Nn \tl_case:Nn { c }
5280 \cs_generate_variant:Nn \tl_case:NnT { c }
5281 \cs_generate_variant:Nn \tl_case:NnF { c }
5282 \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).

```

5283 \cs_new:Npn \__prg_case_end:nw #1#2#3 \q_mark #4#5 \q_stop
5284 { \exp_end: #1 #4 }
5285 \cs_new_eq:NN \__tl_case_end:nw \__prg_case_end:nw

```

(End definition for \tl_case:Nn and \tl_case:cn. These functions are documented on page ??.)

10.8 Mapping to token lists

\tl_map_function:nN 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
5286 \cs_new:Npn \tl_map_function:nN #1#2
5287 {
5288   \__tl_map_function:Nn #2 #1
5289   \q_recursion_tail
5290   \__prg_break_point:Nn \tl_map_break: { }
5291 }
5292 \cs_new_nopar:Npn \tl_map_function:NN
5293 { \exp_args:No \tl_map_function:nN }
5294 \cs_new:Npn \__tl_map_function:Nn #1#2
5295 {
5296   \__quark_if_recursion_tail_break:nN {#2} \tl_map_break:
5297   #1 {#2} \__tl_map_function:Nn #1
5298 }
5299 \cs_generate_variant:Nn \tl_map_function:NN { c }

```

(End definition for \tl_map_function:nN. This function is documented on page 100.)

\tl_map_inline:nn 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.

```

5300 \cs_new_protected:Npn \tl_map_inline:nn #1#2
5301 {
5302   \int_gincr:N \g__prg_map_int
5303   \cs_gset:cpn { __prg_map_ \int_use:N \g__prg_map_int :w } ##1 {#2}
5304   \exp_args:Nc \__tl_map_function:Nn
5305   { __prg_map_ \int_use:N \g__prg_map_int :w }

```

```

5306     #1 \q_recursion_tail
5307     \_prg_break_point:Nn \tl_map_break: { \int_gdecr:N \g__prg_map_int }
5308   }
5309   \cs_new_protected:Npn \tl_map_inline:Nn
5310     { \exp_args:No \tl_map_inline:nn }
5311   \cs_generate_variant:Nn \tl_map_inline:Nn { c }

```

(End definition for `\tl_map_inline:nn`. This function is documented on page 101.)

```

\tl_map_variable:nNn \tl_map_variable:nNn <token list> <temp> <action> assigns <temp> to each element and
\tl_map_variable:NNn executes <action>.
\tl_map_variable:cNn
\_tl_map_variable:Nnn
5312   \cs_new_protected:Npn \tl_map_variable:nNn #1#2#3
5313   {
5314     \_tl_map_variable:Nnn #2 {#3} #1
5315     \q_recursion_tail
5316     \_prg_break_point:Nn \tl_map_break: { }
5317   }
5318   \cs_new_protected_nopar:Npn \tl_map_variable:NNn
5319     { \exp_args:No \tl_map_variable:nNn }
5320   \cs_new_protected:Npn \_tl_map_variable:Nnn #1#2#3
5321   {
5322     \tl_set:Nn #1 {#3}
5323     \_quark_if_recursion_tail_break:NN #1 \tl_map_break:
5324     \use:n {#2}
5325     \_tl_map_variable:Nnn #1 {#2}
5326   }
5327   \cs_generate_variant:Nn \tl_map_variable:NNn { c }

```

(End definition for `\tl_map_variable:nNn`. This function is documented on page 101.)

```

\tl_map_break: The break statements use the general \_prg_map_break:Nn.
\tl_map_break:n
5328   \cs_new_nopar:Npn \tl_map_break:
5329     { \_prg_map_break:Nn \tl_map_break: { } }
5330   \cs_new_nopar:Npn \tl_map_break:n
5331     { \_prg_map_break:Nn \tl_map_break: }

```

(End definition for `\tl_map_break:.` This function is documented on page 101.)

10.9 Using token lists

`\tl_to_str:n` Another name for a primitive: defined in `l3basics`.

(End definition for `\tl_to_str:n`. This function is documented on page 102.)

`\tl_to_str:N` These functions return the replacement text of a token list as a string.

```

\tl_to_str:c
5332   \cs_new:Npn \tl_to_str:N #1 { \etex_detokenize:D \exp_after:wN {#1} }
5333   \cs_generate_variant:Nn \tl_to_str:N { c }

```

(End definition for `\tl_to_str:N` and `\tl_to_str:c`. These functions are documented on page 103.)

\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.

```

5334 \cs_new:Npn \tl_use:N #1
5335 {
5336   \tl_if_exist:NTF #1 {#1}
5337   {
5338     \__msg_kernel_expandable_error:nnn
5339     { kernel } { bad-variable } {#1}
5340   }
5341 }
5342 \cs_generate_variant:Nn \tl_use:N { c }

```

(End definition for **\tl_use:N** and **\tl_use:c**. These functions are documented on page 103.)

10.10 Working with the contents of token lists

\tl_count:n Count number of elements within a token list or token list variable. Brace groups within the list are read as a single element. Spaces are ignored. **__tl_count:n** grabs the element and replaces it by +1. The 0 ensures that it works on an empty list.

```

5343 \cs_new:Npn \tl_count:n #1
5344 {
5345   \int_eval:n
5346   { 0 \tl_map_function:nN {#1} \__tl_count:n }
5347 }
5348 \cs_new:Npn \tl_count:N #1
5349 {
5350   \int_eval:n
5351   { 0 \tl_map_function:NN #1 \__tl_count:n }
5352 }
5353 \cs_new:Npn \__tl_count:n #1 { + \c_one }
5354 \cs_generate_variant:Nn \tl_count:n { V , o }
5355 \cs_generate_variant:Nn \tl_count:N { c }

```

(End definition for **\tl_count:n**, **\tl_count:V**, and **\tl_count:o**. These functions are documented on page 103.)

\tl_reverse_items:n Reversal of a token list is done by taking one item at a time and putting it after **\q_stop**.

```

5356 \cs_new:Npn \tl_reverse_items:n #1
5357 {
5358   \__tl_reverse_items:nwNwn #1 ?
5359   \q_mark \__tl_reverse_items:nwNwn
5360   \q_mark \__tl_reverse_items:wn
5361   \q_stop { }
5362 }
5363 \cs_new:Npn \__tl_reverse_items:nwNwn #1 #2 \q_mark #3 #4 \q_stop #5
5364 {
5365   #3 #2
5366   \q_mark \__tl_reverse_items:nwNwn

```

```

5367     \q_mark \_tl_reverse_items:wn
5368     \q_stop { {#1} #5 }
5369   }
5370 \cs_new:Npn \_tl_reverse_items:wn #1 \q_stop #2
5371 { \exp_not:o { \use_none:nn #2 } }

```

(End definition for `\tl_reverse_items:n`. This function is documented on page 104.)

`\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.

```

5372 \cs_new:Npn \tl_trim_spaces:n #1
5373 { \_tl_trim_spaces:nn { \q_mark #1 } \exp_not:o }
5374 \cs_new_protected:Npn \tl_trim_spaces:N #1
5375 { \tl_set:Nx #1 { \exp_args:No \tl_trim_spaces:n {#1} } }
5376 \cs_new_protected:Npn \tl_gtrim_spaces:N #1
5377 { \tl_gset:Nx #1 { \exp_args:No \tl_trim_spaces:n {#1} } }
5378 \cs_generate_variant:Nn \tl_trim_spaces:N { c }
5379 \cs_generate_variant:Nn \tl_gtrim_spaces:N { c }

```

(End definition for `\tl_trim_spaces:n`. This function is documented on page 104.)

`_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 `␣`. 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_spaces: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*.

```

5380 \cs_set:Npn \_tl_tmp:w #1
5381 {
5382   \cs_new:Npn \_tl_trim_spaces:nn ##1
5383   {
5384     \_tl_trim_spaces_auxi:w
5385     ##1
5386     \q_nil
5387     \q_mark #1 { }
5388     \q_mark \_tl_trim_spaces_auxii:w
5389     \_tl_trim_spaces_auxiii:w
5390     #1 \q_nil
5391     \_tl_trim_spaces_auxiv:w
5392     \q_stop
5393   }
5394   \cs_new:Npn \_tl_trim_spaces_auxi:w ##1 \q_mark #1 ##2 \q_mark ##3

```

```

5395     {
5396         ##3
5397         \__tl_trim_spaces_auxi:w
5398         \q_mark
5399         ##2
5400         \q_mark #1 {##1}
5401     }
5402     \cs_new:Npn \__tl_trim_spaces_auxii:w
5403         \__tl_trim_spaces_auxi:w \q_mark \q_mark ##1
5404     {
5405         \__tl_trim_spaces_auxiii:w
5406         ##1
5407     }
5408     \cs_new:Npn \__tl_trim_spaces_auxiii:w ##1 #1 \q_nil ##2
5409     {
5410         ##2
5411         ##1 \q_nil
5412         \__tl_trim_spaces_auxiii:w
5413     }
5414     \cs_new:Npn \__tl_trim_spaces_auxiv:w ##1 \q_nil ##2 \q_stop ##3
5415     { ##3 { \use_none:n ##1 } }
5416 }
5417 \__tl_tmp:w { ~ }

```

(End definition for __tl_trim_spaces:nn.)

10.11 Token by token changes

\q__tl_act_mark 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. These variables are documented on page ??.)

__tl_act:NNNnn To help control the expansion, __tl_act:NNNnn should always be preceded 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.

```

5418 \cs_new:Npn \__tl_act:NNNnn #1#2#3#4#5
5419 {
5420     \group_align_safe_begin:
5421     \__tl_act_loop:w #5 \q__tl_act_mark \q__tl_act_stop
5422     {#4} #1 #2 #3
5423     \__tl_act_result:n { }
5424 }

```

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:wnnn` gobble the space.

```

5425 \cs_new:Npn \__tl_act_loop:w #1 \q__tl_act_stop
5426 {
5427   \tl_if_head_is_N_type:nTF {#1}
5428   { \__tl_act_normal:Nwnnn }
5429   {
5430     \tl_if_head_is_group:nTF {#1}
5431     { \__tl_act_group:wnnn }
5432     { \__tl_act_space:wnnn }
5433   }
5434   #1 \q__tl_act_stop
5435 }
5436 \cs_new:Npn \__tl_act_normal:Nwnnn #1 #2 \q__tl_act_stop #3#4
5437 {
5438   \if_meaning:w \q__tl_act_mark #1
5439   \exp_after:wN \__tl_act_end:wn
5440   \fi:
5441   #4 {#3} #1
5442   \__tl_act_loop:w #2 \q__tl_act_stop
5443   {#3} #4
5444 }
5445 \cs_new:Npn \__tl_act_end:wn #1 \__tl_act_result:n #2
5446 { \group_align_safe_end: \exp_end: #2 }
5447 \cs_new:Npn \__tl_act_group:wnnn #1 #2 \q__tl_act_stop #3#4#5
5448 {
5449   #5 {#3} {#1}
5450   \__tl_act_loop:w #2 \q__tl_act_stop
5451   {#3} #4 #5
5452 }
5453 \exp_last_unbraced:NNo
5454 \cs_new:Npn \__tl_act_space:wnnn \c_space_tl #1 \q__tl_act_stop #2#3#4#5
5455 {
5456   #5 {#2}
5457   \__tl_act_loop:w #1 \q__tl_act_stop
5458   {#2} #3 #4 #5
5459 }

```

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.

```

5460 \cs_new:Npn \__tl_act_output:n #1 #2 \__tl_act_result:n #3
5461 { #2 \__tl_act_result:n { #3 #1 } }
5462 \cs_new:Npn \__tl_act_reverse_output:n #1 #2 \__tl_act_result:n #3
5463 { #2 \__tl_act_result:n { #1 #3 } }

```

(End definition for `__tl_act:NNnn`.)

`\tl_reverse:n` The goal here is to reverse without losing spaces nor braces. This is done using the
`\tl_reverse:o` general internal function `__tl_act:NNNnn`. Spaces and “normal” tokens are output on
`\tl_reverse:V` the left of the current output. Grouped tokens are output to the left but without any
`__tl_reverse_normal:nN` reversal within the group. All of the internal functions here drop one argument: this is
`_tl_reverse_group_preserve:nn` needed by `__tl_act:NNNnn` when changing case (to record which direction the change
`__tl_reverse_space:n` is in), but not when reversing the tokens.

```

5464 \cs_new:Npn \tl_reverse:n #1
5465 {
5466   \etex_unexpanded:D \exp_after:wN
5467   {
5468     \exp:w
5469     \__tl_act:NNNnn
5470     \__tl_reverse_normal:nN
5471     \__tl_reverse_group_preserve:nn
5472     \__tl_reverse_space:n
5473     { }
5474     {#1}
5475   }
5476 }
5477 \cs_generate_variant:Nn \tl_reverse:n { o , V }
5478 \cs_new:Npn \__tl_reverse_normal:nN #1#2
5479 { \_tl_act_reverse_output:n {#2} }
5480 \cs_new:Npn \__tl_reverse_group_preserve:nn #1#2
5481 { \_tl_act_reverse_output:n { {#2} } }
5482 \cs_new:Npn \__tl_reverse_space:n #1
5483 { \_tl_act_reverse_output:n { ~ } }

```

(End definition for `\tl_reverse:n`, `\tl_reverse:o`, and `\tl_reverse:V`. These functions are documented on page 103.)

`\tl_reverse:N` This reverses the list, leaving `\exp_stop_f:` in front, which stops the `f`-expansion.

```

\__tl_reverse:c
\__tl_greverse:N
\__tl_greverse:c
5484 \cs_new_protected:Npn \tl_reverse:N #1
5485 { \tl_set:Nx #1 { \exp_args:No \tl_reverse:n { #1 } } }
5486 \cs_new_protected:Npn \tl_greverse:N #1
5487 { \tl_gset:Nx #1 { \exp_args:No \tl_reverse:n { #1 } } }
5488 \cs_generate_variant:Nn \tl_reverse:N { c }
5489 \cs_generate_variant:Nn \tl_greverse:N { c }

```

(End definition for `\tl_reverse:N` and others. These functions are documented on page 103.)

10.12 The first token from a token list

`\tl_head:N` Finding the head of a token list expandably will always strip braces, which is fine as
`\tl_head:n` this is consistent with for example mapping to a list. The empty brace groups in `\tl_-`
`\tl_head:V` `head:n` ensure that a blank argument gives an empty result. The result is returned
`\tl_head:v` within the `\unexpanded` primitive. The approach here is to use `\if_false:` to allow
`\tl_head:f` us to use `}` as the closing delimiter: this is the only safe choice, as any other token
`__tl_head_auxi:nw` would not be able to parse it’s own code. Using a marker, we can see if what we are
`__tl_head_auxii:n` grabbing is exactly the marker, or there is anything else to deal with. Is there is, there
`\tl_head:w`
`\tl_tail:N`
`\tl_tail:n`
`\tl_tail:V`
`\tl_tail:v`
`\tl_tail:f`

is a loop. If not, tidy up and leave the item in the output stream. More detail in <http://tex.stackexchange.com/a/70168>.

```

5490 \cs_new:Npn \tl_head:n #1
5491 {
5492   \etex_unexpanded:D
5493   \if_false: { \fi: \tl_head_auxi:nw #1 { } \q_stop }
5494 }
5495 \cs_new:Npn \tl_head_auxi:nw #1#2 \q_stop
5496 {
5497   \exp_after:wN \tl_head_auxii:n \exp_after:wN {
5498     \if_false: } \fi: {#1}
5499 }
5500 \cs_new:Npn \tl_head_auxii:n #1
5501 {
5502   \exp_after:wN \if_meaning:w \exp_after:wN \q_nil
5503   \tl_to_str:n \exp_after:wN { \use_none:n #1 } \q_nil
5504   \exp_after:wN \use_i:nn
5505   \else:
5506     \exp_after:wN \use_ii:nn
5507   \fi:
5508   {#1}
5509   { \if_false: { \fi: \tl_head_auxi:nw #1 } }
5510 }
5511 \cs_generate_variant:Nn \tl_head:n { V , v , f }
5512 \cs_new:Npn \tl_head:w #1#2 \q_stop {#1}
5513 \cs_new_nopar:Npn \tl_head:N { \exp_args:No \tl_head:n }

```

To corrected leave the tail of a token list, it's important *not* to absorb any of the tail part as an argument. For example, the simple definition

```

\cs_new:Npn \tl_tail:n #1 { \tl_tail:w #1 \q_stop }
\cs_new:Npn \tl_tail:w #1#2 \q_stop

```

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.

```

5514 \cs_new:Npn \tl_tail:n #1
5515 {
5516   \etex_unexpanded:D
5517   \tl_if_blank:nTF {#1}
5518     { { } }
5519     { \exp_after:wN { \use_none:n #1 } }
5520 }
5521 \cs_generate_variant:Nn \tl_tail:n { V , v , f }
5522 \cs_new_nopar:Npn \tl_tail:N { \exp_args:No \tl_tail:n }

```

(End definition for `\tl_head:N` and others. These functions are documented on page 105.)

```

\tl_if_head_eq_meaning_p:nN
\tl_if_head_eq_meaning:nNTF
\tl_if_head_eq_charcode_p:nN
\tl_if_head_eq_charcode:nNTF
\tl_if_head_eq_charcode_p:fN
\tl_if_head_eq_charcode:fNTF
\tl_if_head_eq_catcode_p:nN
\tl_if_head_eq_catcode:nNTF

```

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`.

```

\tl_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.

```

5523 \prg_new_conditional:Npnn \tl_if_head_eq_charcode:nN #1#2 { p , T , F , TF }
5524 {
5525   \if_charcode:w
5526     \exp_not:N #2
5527     \tl_if_head_is_N_type:nTF { #1 ? }
5528     {
5529       \exp_after:wN \exp_not:N
5530       \tl_head:w #1 { ? \use_none:nn } \q_stop
5531     }
5532     { \str_head:n {#1} }
5533     \prg_return_true:
5534   \else:
5535     \prg_return_false:
5536   \fi:
5537 }
5538 \cs_generate_variant:Nn \tl_if_head_eq_charcode_p:nN { f }
5539 \cs_generate_variant:Nn \tl_if_head_eq_charcode:nNTF { f }
5540 \cs_generate_variant:Nn \tl_if_head_eq_charcode:nNT { f }
5541 \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.

```

5542 \prg_new_conditional:Npnn \tl_if_head_eq_catcode:nN #1 #2 { p , T , F , TF }
5543 {
5544   \if_catcode:w
5545     \exp_not:N #2
5546     \tl_if_head_is_N_type:nTF { #1 ? }

```

```

5547         {
5548             \exp_after:wN \exp_not:N
5549             \tl_head:w #1 { ? \use_none:nn } \q_stop
5550         }
5551         {
5552             \tl_if_head_is_group:nTF {#1}
5553             { \c_group_begin_token }
5554             { \c_space_token }
5555         }
5556         \prg_return_true:
5557     \else:
5558         \prg_return_false:
5559     \fi:
5560 }

```

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.

```

5561 \prg_new_conditional:Npnn \tl_if_head_eq_meaning:nN #1#2 { p , T , F , TF }
5562 {
5563     \tl_if_head_is_N_type:nTF { #1 ? }
5564     { \__tl_if_head_eq_meaning_normal:nN }
5565     { \__tl_if_head_eq_meaning_special:nN }
5566     {#1} #2
5567 }
5568 \cs_new:Npn \__tl_if_head_eq_meaning_normal:nN #1 #2
5569 {
5570     \exp_after:wN \if_meaning:w
5571     \tl_head:w #1 { ?? \use_none:nnn } \q_stop #2
5572     \prg_return_true:
5573     \else:
5574         \prg_return_false:
5575     \fi:
5576 }
5577 \cs_new:Npn \__tl_if_head_eq_meaning_special:nN #1 #2
5578 {
5579     \if_charcode:w \str_head:n {#1} \exp_not:N #2
5580     \exp_after:wN \use:n
5581     \else:
5582         \prg_return_false:
5583     \exp_after:wN \use_none:n
5584     \fi:
5585     {
5586         \if_catcode:w \exp_not:N #2
5587         \tl_if_head_is_group:nTF {#1}
5588         { \c_group_begin_token }

```

```

5589             { \c_space_token }
5590         \prg_return_true:
5591     \else:
5592         \prg_return_false:
5593     \fi:
5594 }
5595 }

```

(End definition for `\tl_if_head_eq_meaning:nNTF`. This function is documented on page 106.)

`\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.

```

5596 \prg_new_conditional:Npnn \tl_if_head_is_N_type:n #1 { p , T , F , TF }
5597 {
5598     \if_catcode:w
5599         \if_false: { \fi: \__tl_if_head_is_N_type:w ? #1 ~ }
5600         \exp_after:wN \use_none:n
5601         \exp_after:wN { \exp_after:wN { \token_to_str:N #1 ? } }
5602         * *
5603     \prg_return_true:
5604 \else:
5605     \prg_return_false:
5606 \fi:
5607 }
5608 \cs_new:Npn \__tl_if_head_is_N_type:w #1 ~
5609 {
5610     \tl_if_empty:oTF { \use_none:n #1 } { ^ } { }
5611     \exp_after:wN \use_none:n \exp_after:wN { \if_false: } \fi:
5612 }

```

(End definition for `\tl_if_head_is_N_type:nTF`. This function is documented on page 107.)

`\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.⁷

```

5613 \prg_new_conditional:Npnn \tl_if_head_is_group:n #1 { p , T , F , TF }
5614 {
5615     \if_catcode:w
5616         \exp_after:wN \use_none:n
5617         \exp_after:wN { \exp_after:wN { \token_to_str:N #1 ? } }
5618         * *

```

⁷Bruno: 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.

```

5619     \prg_return_false:
5620   \else:
5621     \prg_return_true:
5622   \fi:
5623 }

```

(End definition for `\tl_if_head_is_group:nTF`. This function is documented on page 106.)

`\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).

`\tl_if_head_is_space:nTF`
`__tl_if_head_is_space:w`

```

5624 \prg_new_conditional:Npnn \tl_if_head_is_space:n #1 { p , T , F , TF }
5625 {
5626   \exp:w \if_false: { \fi:
5627     \__tl_if_head_is_space:w ? #1 ? ~ }
5628 }
5629 \cs_new:Npn \__tl_if_head_is_space:w #1 ~
5630 {
5631   \tl_if_empty:oTF { \use_none:n #1 }
5632     { \exp_after:wN \exp_end: \exp_after:wN \prg_return_true: }
5633     { \exp_after:wN \exp_end: \exp_after:wN \prg_return_false: }
5634   \exp_after:wN \use_none:n \exp_after:wN { \if_false: } \fi:
5635 }

```

(End definition for `\tl_if_head_is_space:nTF`. This function is documented on page 107.)

10.13 Using a single item

`\tl_item:nn` The idea here is to find the offset of the item from the left, then use a loop to grab the correct item. If the resulting offset is too large, then `\quark_if_recursion_tail_stop:n` terminates the loop, and returns nothing at all.

`\tl_item:Nn`
`\tl_item:cn`
`__tl_item:nn`

```

5636 \cs_new:Npn \tl_item:nn #1#2
5637 {
5638   \exp_args:Nf \__tl_item:nn
5639   {
5640     \int_eval:n
5641     {
5642       \int_compare:nNnT {#2} < \c_zero
5643       { \tl_count:n {#1} + \c_one + }
5644       #2
5645     }
5646   }
5647   #1
5648   \q_recursion_tail
5649   \__prg_break_point:

```

```

5650 }
5651 \cs_new:Npn \__tl_item:nn #1#2
5652 {
5653   \__quark_if_recursion_tail_break:nN {#2} \__prg_break:
5654   \int_compare:nNnTF {#1} = \c_one
5655     { \__prg_break:n { \exp_not:n {#2} } }
5656     { \exp_args:Nf \__tl_item:nn { \int_eval:n { #1 - 1 } } }
5657 }
5658 \cs_new_nopar:Npn \tl_item:Nn { \exp_args:No \tl_item:nn }
5659 \cs_generate_variant:Nn \tl_item:Nn { c }

```

(End definition for `\tl_item:nn`, `\tl_item:Nn`, and `\tl_item:cn`. These functions are documented on page 107.)

10.14 Viewing token lists

`\tl_show:N` Showing token list variables is done after checking that the variable is defined (see `__kernel_register_show:N`).

```

5660 \cs_new_protected:Npn \tl_show:N #1
5661 {
5662   \__msg_show_variable:NNNnn #1 \tl_if_exist:NTF ? { }
5663   { > ~ \token_to_str:N #1 = \tl_to_str:N #1 }
5664 }
5665 \cs_generate_variant:Nn \tl_show:N { c }

```

(End definition for `\tl_show:N` and `\tl_show:c`. These functions are documented on page 107.)

`\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.

```

5666 \cs_new_protected:Npn \tl_show:n #1
5667 { \__msg_show_wrap:n { > ~ \tl_to_str:n {#1} } }

```

(End definition for `\tl_show:n`. This function is documented on page 108.)

10.15 Scratch token lists

`\g_tmpa_tl` **`\g_tmpb_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.

```

5668 \tl_new:N \g_tmpa_tl
5669 \tl_new:N \g_tmpb_tl

```

(End definition for `\g_tmpa_tl` and `\g_tmpb_tl`. These variables are documented on page 108.)

`\l_tmpa_tl` **`\l_tmpb_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.

```

5670 \tl_new:N \l_tmpa_tl
5671 \tl_new:N \l_tmpb_tl

```

(End definition for `\l_tmpa_tl` and `\l_tmpb_tl`. These variables are documented on page 108.)

```

5672 </initex | package>

```

11 l3str implementation

```
5673 <*initex | package>
5674 <@@=str>
```

11.1 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 hand.

\str_new:c

\str_use:N 5675 \group_begin:

\str_use:c 5676 \cs_set_protected:Npn __str_tmp:n #1

\str_clear:N 5677 {

\str_clear:c 5678 \tl_if_blank:nF {#1}

\str_gclear:N 5679 {

\str_gclear:c 5680 \cs_new_eq:cc { str_ #1 :N } { tl_ #1 :N }

\str_clear_new:N 5681 \exp_args:Nc \cs_generate_variant:Nn { str_ #1 :N } { c }

\str_clear_new:c 5682 __str_tmp:n

\str_gclear_new:N 5683 }

\str_gclear_new:c 5684 }

\str_set_eq:NN 5685 __str_tmp:n

\str_set_eq:cN 5686 { new }

\str_set_eq:Nc 5687 { use }

\str_set_eq:cc 5688 { clear }

\str_gset_eq:NN 5689 { gclear }

\str_gset_eq:cN 5690 { clear_new }

\str_gset_eq:Nc 5691 { gclear_new }

\str_gset_eq:cc 5692 { }

5693 \group_end:

5694 \cs_new_eq:NN \str_set_eq:NN \tl_set_eq:NN

5695 \cs_new_eq:NN \str_gset_eq:NN \tl_gset_eq:NN

5696 \cs_generate_variant:Nn \str_set_eq:NN { c , Nc , cc }

5697 \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 109.)

\str_set:Nn Simply convert the token list inputs to <strings>.

\str_set:Nx 5698 \group_begin:

\str_set:cn 5699 \cs_set_protected:Npn __str_tmp:n #1

\str_set:cx 5700 {

\str_gset:Nn 5701 \tl_if_blank:nF {#1}

\str_gset:Nx 5702 {

\str_gset:cn 5703 \cs_new_protected:cpx { str_ #1 :Nn } ##1##2

\str_gset:cx 5704 { \exp_not:c { tl_ #1 :Nx } ##1 { \exp_not:N \tl_to_str:n {##2} } }

\str_const:Nn 5705 \exp_args:Nc \cs_generate_variant:Nn { str_ #1 :Nn } { Nx , cn , cx }

\str_const:Nx 5706 __str_tmp:n

\str_const:cn 5707 }

\str_const:cx 5708 }

\str_put_left:Nn 5709 __str_tmp:n

\str_put_left:Nx 5710 { set }

\str_put_left:cn 5711 { gset }

\str_put_left:cx

\str_gput_left:Nn

\str_gput_left:Nx

\str_gput_left:cn

\str_gput_left:cx

\str_put_right:Nn

\str_put_right:Nx

\str_put_right:cn

\str_put_right:cx

```

5712     { const }
5713     { put_left }
5714     { gput_left }
5715     { put_right }
5716     { gput_right }
5717     { }
5718 \group_end:

```

(End definition for `\str_set:Nn` and others. These functions are documented on page 110.)

11.2 String comparisons

`\str_if_empty_p:N` More copy-paste!

```

\str_if_empty_p:c 5719 \prg_new_eq_conditional:NNn \str_if_exist:N \tl_if_exist:N { p , T , F , TF }
\str_if_exist_p:N 5720 \prg_new_eq_conditional:NNn \str_if_exist:c \tl_if_exist:c { p , T , F , TF }
\str_if_exist_p:c 5721 \prg_new_eq_conditional:NNn \str_if_empty:N \tl_if_empty:N { p , T , F , TF }
\str_if_empty:NTF 5722 \prg_new_eq_conditional:NNn \str_if_empty:c \tl_if_empty:c { p , T , F , TF }
\str_if_empty:cTF
\str_if_exist:NTF
\str_if_exist:cTF
\__str_if_eq_x:n
\__str_escape_x:n

```

(End definition for `\str_if_empty:NTF` and others. These functions are documented on page 111.)

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.

```

5723 \cs_new:Npn \__str_if_eq_x:nn #1#2 { \pdfstrcmp:D {#1} {#2} }
5724 \cs_if_exist:NT \luatex_luaexversion:D
5725 {
5726     \cs_set:Npn \__str_if_eq_x:nn #1#2
5727     {
5728         \luatex_directlua:D
5729         {
5730             l3kernel_strcmp
5731             (
5732                 " \__str_escape_x:n {#1} " ,
5733                 " \__str_escape_x:n {#2} "
5734             )
5735         }
5736     }
5737 \cs_new:Npn \__str_escape_x:n #1
5738 {
5739     \luatex_luaescapestring:D
5740     {
5741         \etex_detokenize:D \exp_after:wN { \luatex_expanded:D {#1} }
5742     }

```

```

5743     }
5744 }

```

(End definition for `_str_if_eq_x:nn`.)

`_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.

```

5745 \cs_new:Npn \_str_if_eq_x_return:nn #1 #2
5746 {
5747   \if_int_compare:w \_str_if_eq_x:nn {#1} {#2} = \c_zero
5748   \prg_return_true:
5749   \else:
5750     \prg_return_false:
5751   \fi:
5752 }

```

(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.

```

5753 \prg_new_conditional:Npnn \str_if_eq:nn #1#2 { p , T , F , TF }
5754 {
5755   \if_int_compare:w
5756     \_str_if_eq_x:nn { \exp_not:n {#1} } { \exp_not:n {#2} }
5757     = \c_zero
5758     \prg_return_true: \else: \prg_return_false: \fi:
5759 }
5760 \cs_generate_variant:Nn \str_if_eq_p:nn { V , o }
5761 \cs_generate_variant:Nn \str_if_eq_p:nn { nV , no , VV }
5762 \cs_generate_variant:Nn \str_if_eq:nnT { V , o }
5763 \cs_generate_variant:Nn \str_if_eq:nnT { nV , no , VV }
5764 \cs_generate_variant:Nn \str_if_eq:nnF { V , o }
5765 \cs_generate_variant:Nn \str_if_eq:nnF { nV , no , VV }
5766 \cs_generate_variant:Nn \str_if_eq:nnTF { V , o }
5767 \cs_generate_variant:Nn \str_if_eq:nnTF { nV , no , VV }
5768 \prg_new_conditional:Npnn \str_if_eq_x:nn #1#2 { p , T , F , TF }
5769 {
5770   \if_int_compare:w \_str_if_eq_x:nn {#1} {#2} = \c_zero
5771   \prg_return_true: \else: \prg_return_false: \fi:
5772 }

```

(End definition for `\str_if_eq:nnTF` and others. These functions are documented on page [111](#).)

`\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.

```

5773 \prg_new_conditional:Npnn \str_if_eq:NN #1#2 { p , TF , T , F }
5774 {
5775   \if_int_compare:w \_str_if_eq_x:nn { \tl_to_str:N #1 } { \tl_to_str:N #2 }

```

`\str_if_eq:NcTF`
`\str_if_eq:cNTF`
`\str_if_eq:ccTF`

```

5776         = \c_zero \prg_return_true: \else: \prg_return_false: \fi:
5777     }
5778 \cs_generate_variant:Nn \str_if_eq:NNT { c , Nc , cc }
5779 \cs_generate_variant:Nn \str_if_eq:NNTF { c , Nc , cc }
5780 \cs_generate_variant:Nn \str_if_eq:NNTF { c , Nc , cc }
5781 \cs_generate_variant:Nn \str_if_eq_p:NN { c , Nc , cc }

```

(End definition for \str_if_eq:NNTF and others. These functions are documented on page 111.)

```

\str_case:nn Much the same as \tl_case:nn(TF) here: just a change in the internal comparison.
\str_case:on 5782 \cs_new:Npn \str_case:nn #1#2
\str_case:nV 5783 {
\str_case:nv 5784     \exp:w
\str_case_x:nn 5785     \__str_case:nnTF {#1} {#2} { } { }
\str_case:nnTF 5786 }
\str_case:onTF 5787 \cs_new:Npn \str_case:nnT #1#2#3
\str_case:nVTF 5788 {
\str_case:nvTF 5789     \exp:w
\str_case_x:nnTF 5790     \__str_case:nnTF {#1} {#2} {#3} { }
5791 }
5792 \cs_new:Npn \str_case:nnF #1#2
5793 {
5794     \exp:w
5795     \__str_case:nnTF {#1} {#2} { }
5796 }
5797 \cs_new:Npn \str_case:nnTF #1#2
5798 {
5799     \exp:w
5800     \__str_case:nnTF {#1} {#2}
5801 }
5802 \cs_new:Npn \__str_case:nnTF #1#2#3#4
5803 { \__str_case:nw {#1} #2 {#1} { } \q_mark {#3} \q_mark {#4} \q_stop }
5804 \cs_generate_variant:Nn \str_case:nn { o , nV , nv }
5805 \cs_generate_variant:Nn \str_case:nnT { o , nV , nv }
5806 \cs_generate_variant:Nn \str_case:nnF { o , nV , nv }
5807 \cs_generate_variant:Nn \str_case:nnTF { o , nV , nv }
5808 \cs_new:Npn \__str_case:nw #1#2#3
5809 {
5810     \str_if_eq:nnTF {#1} {#2}
5811     { \__str_case_end:nw {#3} }
5812     { \__str_case:nw {#1} }
5813 }
5814 \cs_new:Npn \str_case_x:nn #1#2
5815 {
5816     \exp:w
5817     \__str_case_x:nnTF {#1} {#2} { } { }
5818 }
5819 \cs_new:Npn \str_case_x:nnT #1#2#3
5820 {
5821     \exp:w

```

```

5822     \__str_case_x:nnTF {#1} {#2} {#3} { }
5823   }
5824   \cs_new:Npn \str_case_x:nnF #1#2
5825   {
5826     \exp:w
5827     \__str_case_x:nnTF {#1} {#2} { }
5828   }
5829   \cs_new:Npn \str_case_x:nnTF #1#2
5830   {
5831     \exp:w
5832     \__str_case_x:nnTF {#1} {#2}
5833   }
5834   \cs_new:Npn \__str_case_x:nnTF #1#2#3#4
5835   { \__str_case_x:nw {#1} #2 {#1} { } \q_mark {#3} \q_mark {#4} \q_stop }
5836   \cs_new:Npn \__str_case_x:nw #1#2#3
5837   {
5838     \str_if_eq_x:nnTF {#1} {#2}
5839     { \__str_case_end:nw {#3} }
5840     { \__str_case_x:nw {#1} }
5841   }
5842   \cs_new_eq:NN \__str_case_end:nw \__prg_case_end:nw

```

(End definition for `\str_case:nn` and others. These functions are documented on page ??.)

11.3 Accessing specific characters in a string

`__str_to_other:n` First apply `\tl_to_str:n`, then replace all spaces by “other” spaces, 8 at a time, storing the converted part of the string between the `\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).

```

5843   \cs_new:Npn \__str_to_other:n #1
5844   {
5845     \exp_after:wN \__str_to_other_loop:w
5846     \tl_to_str:n {#1} ~ A ~ A ~ A ~ A ~ A ~ A ~ A ~ \q_mark \q_stop
5847   }
5848   \group_begin:
5849   \tex_lccode:D '\* = '\ %
5850   \tex_lccode:D '\A = '\A
5851   \tex_lowercase:D
5852   {
5853     \group_end:
5854     \cs_new:Npn \__str_to_other_loop:w
5855     #1 ~ #2 ~ #3 ~ #4 ~ #5 ~ #6 ~ #7 ~ #8 ~ #9 \q_stop
5856     {
5857       \if_meaning:w A #8
5858       \__str_to_other_end:w
5859       \fi:

```

```

5860     \_str_to_other_loop:w
5861     #9 #1 * #2 * #3 * #4 * #5 * #6 * #7 * #8 * \q_stop
5862 }
5863 \cs_new:Npn \_str_to_other_end:w \fi: #1 \q_mark #2 * A #3 \q_stop
5864 { \fi: #2 }
5865 }

```

(End definition for _str_to_other:n.)

\str_item:Nn The **\str_item:nn** hands its argument with spaces escaped to **_str_item:nn**, and **\str_item:cn** 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 *<index>* argument **#2** and count characters in the string, passing those two numbers to **_str_item:w** for further analysis. If the *<index>* is negative, shift it by the *<count>* to know the how many character to discard, and if that is still negative give an empty result. If the *<index>* is larger than the *<count>*, give an empty result, and otherwise discard *<index>* − 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 *<index>* is zero.

```

5866 \cs_new_nopar:Npn \str_item:Nn { \exp_args:No \str_item:nn }
5867 \cs_generate_variant:Nn \str_item:Nn { c }
5868 \cs_new:Npn \str_item:nn #1#2
5869 {
5870     \exp_args:Nf \tl_to_str:n
5871     {
5872         \exp_args:Nf \_str_item:nn
5873         { \_str_to_other:n {#1} } {#2}
5874     }
5875 }
5876 \cs_new:Npn \str_item_ignore_spaces:nn #1
5877 { \exp_args:No \_str_item:nn { \tl_to_str:n {#1} } }
5878 \cs_new:Npn \_str_item:nn #1#2
5879 {
5880     \exp_after:wN \_str_item:w
5881     \int_use:N \_int_eval:w #2 \exp_after:wN ;
5882     \_int_value:w \_str_count:n {#1} ;
5883     #1 \q_stop
5884 }
5885 \cs_new:Npn \_str_item:w #1; #2;
5886 {
5887     \int_compare:nNnTF {#1} < \c_zero
5888     {
5889         \int_compare:nNnTF {#1} < {-#2}
5890         { \use_none_delimit_by_q_stop:w }
5891         {
5892             \exp_after:wN \use_i_delimit_by_q_stop:nw
5893             \exp:w \exp_after:wN \_str_skip_exp_end:w
5894             \int_use:N \_int_eval:w #1 + #2 ;

```

```

5895     }
5896   }
5897   {
5898     \int_compare:nNnTF {#1} > {#2}
5899     { \use_none_delimit_by_q_stop:w }
5900     {
5901       \exp_after:wN \use_i_delimit_by_q_stop:nw
5902       \exp:w \__str_skip_exp_end:w #1 ; { }
5903     }
5904   }
5905 }

```

(End definition for `\str_item:Nn` and others. These functions are documented on page 114.)

```

\__str_skip_exp_end:w
\__str_skip_loop:wNNNNNNNN
\__str_skip_end:w
\__str_skip_end:NNNNNNNN

```

Removes `max(#1,0)` characters from the input stream, and then leaves `\exp_end:.` This should be expanded using `\exp:w`. We remove characters 8 at a time until there are at most 8 to remove. Then we do a dirty trick: the `\if_case:w` construction leaves between 0 and 8 times the `\or:` control sequence, and those `\or:` become arguments of `__str_skip_end:NNNNNNNN`. If the number of characters to remove is 6, say, then there are two `\or:` left, and the 8 arguments of `__str_skip_end:NNNNNNNN` are the two `\or:`, and 6 characters from the input stream, exactly what we wanted to remove. Then close the `\if_case:w` conditional with `\fi:`, and stop the initial expansion with `\exp_end:` (see places where `__str_skip_exp_end:w` is called).

```

5906 \cs_new:Npn \__str_skip_exp_end:w #1;
5907 {
5908   \if_int_compare:w #1 > \c_eight
5909   \exp_after:wN \__str_skip_loop:wNNNNNNNN
5910   \else:
5911     \exp_after:wN \__str_skip_end:w
5912     \int_use:N \__int_eval:w
5913   \fi:
5914   #1 ;
5915 }
5916 \cs_new:Npn \__str_skip_loop:wNNNNNNNN #1; #2#3#4#5#6#7#8#9
5917 { \exp_after:wN \__str_skip_exp_end:w \int_use:N \__int_eval:w #1 - \c_eight ; }
5918 \cs_new:Npn \__str_skip_end:w #1 ;
5919 {
5920   \exp_after:wN \__str_skip_end:NNNNNNNN
5921   \if_case:w #1 \exp_stop_f: \or: \or: \or: \or: \or: \or: \or: \or:
5922 }
5923 \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`.)

```

\str_range:Nnn
\str_range:nnn
\str_range_ignore_spaces:nnn
\__str_range:nnn
\__str_range:w
\__str_range:nw

```

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.

```

5924 \cs_new_nopar:Npn \str_range:Nnn { \exp_args:No \str_range:nnn }
5925 \cs_generate_variant:Nn \str_range:Nnn { c }
5926 \cs_new:Npn \str_range:nnn #1#2#3
5927 {
5928   \exp_args:Nf \tl_to_str:n
5929   {
5930     \exp_args:Nf \__str_range:nnn
5931     { \__str_to_other:n {#1} } {#2} {#3}
5932   }
5933 }
5934 \cs_new:Npn \str_range_ignore_spaces:nnn #1
5935 { \exp_args:No \__str_range:nnn { \tl_to_str:n {#1} } }
5936 \cs_new:Npn \__str_range:nnn #1#2#3
5937 {
5938   \exp_after:wN \__str_range:w
5939   \__int_value:w \__str_count:n {#1} \exp_after:wN ;
5940   \int_use:N \__int_eval:w #2 - \c_one \exp_after:wN ;
5941   \int_use:N \__int_eval:w #3 ;
5942   #1 \q_stop
5943 }
5944 \cs_new:Npn \__str_range:w #1; #2; #3;
5945 {
5946   \exp_args:Nf \__str_range:nnw
5947   { \__str_range_normalize:nn {#2} {#1} }
5948   { \__str_range_normalize:nn {#3} {#1} }
5949 }
5950 \cs_new:Npn \__str_range:nnw #1#2
5951 {
5952   \exp_after:wN \__str_collect_delimit_by_q_stop:w
5953   \int_use:N \__int_eval:w #2 - #1 \exp_after:wN ;
5954   \exp:w \__str_skip_exp_end:w #1 ;
5955 }

```

(End definition for `\str_range:Nnn`, `\str_range:nnn`, and `\str_range_ignore_spaces:nnn`. These functions are documented on page 114.)

`__str_range_normalize:nn` This function converts an *index* argument into an explicit position in the string (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].

```

5956 \cs_new:Npn \__str_range_normalize:nn #1#2
5957 {
5958   \int_eval:n
5959   {
5960     \if_int_compare:w #1 < \c_zero
5961       \if_int_compare:w #1 < -#2 \exp_stop_f:
5962       \c_zero

```

```

5963         \else:
5964             #1 + #2 + \c_one
5965         \fi:
5966     \else:
5967         \if_int_compare:w #1 < #2 \exp_stop_f:
5968             #1
5969         \else:
5970             #2
5971         \fi:
5972     \fi:
5973 }
5974 }

```

(End definition for `__str_range_normalize:nn`.)

`__str_collect_delimit_by_q_stop:w` 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.

```

5975 \cs_new:Npn \__str_collect_delimit_by_q_stop:w #1;
5976 { \__str_collect_loop:wn #1 ; { } }
5977 \cs_new:Npn \__str_collect_loop:wn #1 ;
5978 {
5979     \if_int_compare:w #1 > \c_seven
5980     \exp_after:wN \__str_collect_loop:wnNNNNNNN
5981     \else:
5982     \exp_after:wN \__str_collect_end:wn
5983     \fi:
5984     #1 ;
5985 }
5986 \cs_new:Npn \__str_collect_loop:wnNNNNNNN #1; #2 #3#4#5#6#7#8#9
5987 {
5988     \exp_after:wN \__str_collect_loop:wn
5989     \int_use:N \__int_eval:w #1 - \c_seven ;
5990     { #2 #3#4#5#6#7#8#9 }
5991 }
5992 \cs_new:Npn \__str_collect_end:wn #1 ;
5993 {
5994     \exp_after:wN \__str_collect_end:nnnnnnnnw
5995     \if_case:w \if_int_compare:w #1 > \c_zero #1 \else: 0 \fi: \exp_stop_f:
5996     \or: \or: \or: \or: \or: \or: \or: \fi:
5997 }
5998 \cs_new:Npn \__str_collect_end:nnnnnnnnw #1#2#3#4#5#6#7#8 #9 \q_stop
5999 { #1#2#3#4#5#6#7#8 }

```

(End definition for `__str_collect_delimit_by_q_stop:w`.)

11.4 Counting characters

`\str_count_spaces:N` To speed up this function, we grab and discard 9 space-delimited arguments in each iteration of the loop. The loop stops when the last argument is one of the trailing $X\langle number \rangle$, and that $\langle number \rangle$ is added to the sum of 9 that precedes, to adjust the result.

```

6000 \cs_new_nopar:Npn \str_count_spaces:N
6001   { \exp_args:No \str_count_spaces:n }
6002 \cs_generate_variant:Nn \str_count_spaces:N { c }
6003 \cs_new:Npn \str_count_spaces:n #1
6004   {
6005     \int_eval:n
6006     {
6007       \exp_after:wN \__str_count_spaces_loop:w
6008       \tl_to_str:n {#1} ~
6009       X 7 ~ X 6 ~ X 5 ~ X 4 ~ X 3 ~ X 2 ~ X 1 ~ X 0 ~ X -1 ~
6010       \q_stop
6011     }
6012   }
6013 \cs_new:Npn \__str_count_spaces_loop:w #1~#2~#3~#4~#5~#6~#7~#8~#9~
6014   {
6015     \if_meaning:w X #9
6016       \use_i_delimit_by_q_stop:nw
6017     \fi:
6018     \c_nine + \__str_count_spaces_loop:w
6019   }

```

(End definition for `\str_count_spaces:N`, `\str_count_spaces:c`, and `\str_count_spaces:n`. These functions are documented on page 113.)

`\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.

`\str_count:c` 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.

`\str_count:n`

`__str_count_aux:n`

`__str_count_loop:NNNNNNNN`

```

6020 \cs_new_nopar:Npn \str_count:N { \exp_args:No \str_count:n }
6021 \cs_generate_variant:Nn \str_count:N { c }
6022 \cs_new:Npn \str_count:n #1
6023   {
6024     \__str_count_aux:n
6025     {
6026       \str_count_spaces:n {#1}
6027       + \exp_after:wN \__str_count_loop:NNNNNNNN \tl_to_str:n {#1}
6028     }

```

```

6029 }
6030 \cs_new:Npn \__str_count:n #1
6031 {
6032   \__str_count_aux:n
6033   { \__str_count_loop:NNNNNNNN #1 }
6034 }
6035 \cs_new:Npn \str_count_ignore_spaces:n #1
6036 {
6037   \__str_count_aux:n
6038   { \exp_after:wN \__str_count_loop:NNNNNNNN \tl_to_str:n {#1} }
6039 }
6040 \cs_new:Npn \__str_count_aux:n #1
6041 {
6042   \int_eval:n
6043   {
6044     #1
6045     { X \c_eight } { X \c_seven } { X \c_six }
6046     { X \c_five } { X \c_four } { X \c_three }
6047     { X \c_two } { X \c_one } { X \c_zero }
6048     \q_stop
6049   }
6050 }
6051 \cs_new:Npn \__str_count_loop:NNNNNNNN #1#2#3#4#5#6#7#8#9
6052 {
6053   \if_meaning:w X #9
6054   \exp_after:wN \use_none_delimit_by_q_stop:w
6055   \fi:
6056   \c_nine + \__str_count_loop:NNNNNNNN
6057 }

```

(End definition for `\str_count:N` and others. These functions are documented on page 113.)

11.5 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`.
`\str_head:c`
`\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`.

```

6058 \cs_new_nopar:Npn \str_head:N { \exp_args:No \str_head:n }
6059 \cs_generate_variant:Nn \str_head:N { c }
6060 \cs_set:Npn \str_head:n #1
6061 {
6062   \exp_after:wN \__str_head:w

```

```

6063     \tl_to_str:n {#1}
6064     { { } } ~ \q_stop
6065   }
6066   \cs_set:Npn \__str_head:w #1 ~ %
6067   { \use_i_delimit_by_q_stop:nw #1 { ~ } }
6068   \cs_new:Npn \str_head_ignore_spaces:n #1
6069   {
6070     \exp_after:wN \use_i_delimit_by_q_stop:nw
6071     \tl_to_str:n {#1} { } \q_stop
6072   }

```

(End definition for `\str_head:N` and others. These functions are documented on page 113.)

`\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.

```

6073   \cs_new_nopar:Npn \str_tail:N { \exp_args:No \str_tail:n }
6074   \cs_generate_variant:Nn \str_tail:N { c }
6075   \cs_set:Npn \str_tail:n #1
6076   {
6077     \exp_after:wN \__str_tail_auxi:w
6078     \reverse_if:N \if_charcode:w
6079     \scan_stop: \tl_to_str:n {#1} X X \q_stop
6080   }
6081   \cs_set:Npn \__str_tail_auxi:w #1 X #2 \q_stop { \fi: #1 }
6082   \cs_new:Npn \str_tail_ignore_spaces:n #1
6083   {
6084     \exp_after:wN \__str_tail_auxii:w
6085     \tl_to_str:n {#1} \q_mark \q_mark \q_stop
6086   }
6087   \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 113.)

11.6 String manipulation

`\str_fold_case:n` Case changing for programmatic reasons is done by first detokenizing input then doing a simple loop that only has to worry about spaces and everything else. The output is detokenized to allow data sharing with text-based case changing. The key idea here of splitting up the data files based on the character code of the current character is shared with the text case changer too.

`\str_fold_case:V`

`\str_lower_case:n`

`\str_lower_case:f`

`\str_upper_case:n`

`\str_upper_case:f`

```

6088   \cs_new:Npn \str_fold_case:n #1 { \__str_change_case:nn {#1} { fold } }

```

`__str_change_case:nn`

`__str_change_case_aux:nn`

`__str_change_case_loop:nw`

`__str_change_case_space:n`

`__str_change_case_char:nN`

`__str_change_case_char:NNNNNNNNn`

```

6089 \cs_new:Npn \str_lower_case:n #1 { \__str_change_case:nn {#1} { lower } }
6090 \cs_new:Npn \str_upper_case:n #1 { \__str_change_case:nn {#1} { upper } }
6091 \cs_generate_variant:Nn \str_fold_case:n { V }
6092 \cs_generate_variant:Nn \str_lower_case:n { f }
6093 \cs_generate_variant:Nn \str_upper_case:n { f }
6094 \cs_new:Npn \__str_change_case:nn #1
6095 {
6096   \exp_after:wN \__str_change_case_aux:nn \exp_after:wN
6097   { \tl_to_str:n {#1} }
6098 }
6099 \cs_new:Npn \__str_change_case_aux:nn #1#2
6100 {
6101   \__str_change_case_loop:nw {#2} #1 \q_recursion_tail \q_recursion_stop
6102 }
6103 \cs_new:Npn \__str_change_case_loop:nw #1#2 \q_recursion_stop
6104 {
6105   \tl_if_head_is_space:nTF {#2}
6106   { \__str_change_case_space:n }
6107   { \__str_change_case_char:nN }
6108   {#1} #2 \q_recursion_stop
6109 }
6110 \use:x
6111 { \cs_new:Npn \exp_not:N \__str_change_case_space:n ##1 \c_space_tl }
6112 {
6113   \c_space_tl
6114   \__str_change_case_loop:nw {#1}
6115 }
6116 \cs_new:Npn \__str_change_case_char:nN #1#2
6117 {
6118   \quark_if_recursion_tail_stop:N #2
6119   \exp_args:Nf \tl_to_str:n
6120   {
6121     \exp_after:wN \__str_change_case_char:NNNNNNNNn
6122     \int_use:N \__int_eval:w 1000000 + '#2 \__int_eval_end: #2 {#1}
6123   }
6124   \__str_change_case_loop:nw {#1}
6125 }
6126 \cs_new:Npn \__str_change_case_char:NNNNNNNNn #1#2#3#4#5#6#7#8#9
6127 {
6128   \str_case:nvF #8
6129   { c__unicode_ #9 _ #6 _X_ #7 _tl }
6130   { #8 }
6131 }

```

(End definition for `\str_fold_case:n` and others. These functions are documented on page 116.)

<pre> \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 </pre>	<pre> For all of those strings, use \cs_to_str:N to get characters with the correct category code without worries 6132 \str_const:Nx \c_ampersand_str { \cs_to_str:N \& } 6133 \str_const:Nx \c_atsign_str { \cs_to_str:N \@ } </pre>
--	---

```

6134 \str_const:Nx \c_backslash_str { \cs_to_str:N \\\ }
6135 \str_const:Nx \c_left_brace_str { \cs_to_str:N \{ }
6136 \str_const:Nx \c_right_brace_str { \cs_to_str:N \} }
6137 \str_const:Nx \c_circumflex_str { \cs_to_str:N \^ }
6138 \str_const:Nx \c_colon_str { \cs_to_str:N \: }
6139 \str_const:Nx \c_dollar_str { \cs_to_str:N \$ }
6140 \str_const:Nx \c_hash_str { \cs_to_str:N \# }
6141 \str_const:Nx \c_percent_str { \cs_to_str:N \% }
6142 \str_const:Nx \c_tilde_str { \cs_to_str:N \~ }
6143 \str_const:Nx \c_underscore_str { \cs_to_str:N \_ }

```

(End definition for `\c_underscore_str` and others. These variables are documented on page 117.)

```

\l_tmpa_str Scratch strings.
\l_tmpb_str
\g_tmpa_str
\g_tmpb_str
6144 \str_new:N \l_tmpa_str
6145 \str_new:N \l_tmpb_str
6146 \str_new:N \g_tmpa_str
6147 \str_new:N \g_tmpb_str

```

(End definition for `\l_tmpa_str` and others. These variables are documented on page 117.)

11.7 Viewing strings

```

\str_show:n Displays a string on the terminal.
\str_show:N
\str_show:c
6148 \cs_new_eq:NN \str_show:n \tl_show:n
6149 \cs_new_eq:NN \str_show:N \tl_show:N
6150 \cs_generate_variant:Nn \str_show:N { c }

```

(End definition for `\str_show:n`, `\str_show:N`, and `\str_show:c`. These functions are documented on page 116.)

```

6151 </initex | package>

```

12 l3seq implementation

The following test files are used for this code: `m3seq002`, `m3seq003`.

```

6152 <*initex | package>
6153 <@@=seq>

```

A sequence is a control sequence whose top-level expansion is of the form “`\s__seq __seq_item:n {⟨item1⟩} ... __seq_item:n {⟨itemn⟩}`”, with a leading scan mark followed by n items of the same form. An earlier implementation used the structure “`\seq_elt:w ⟨item1⟩ \seq_elt_end: ... \seq_elt:w ⟨itemn⟩ \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`. This variable is documented on page 130.)

__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.

```

6154 \cs_new:Npn \__seq_item:n
6155 {
6156   \_msg_kernel_expandable_error:nn { kernel } { misused-sequence }
6157   \use_none:n
6158 }

```

(End definition for __seq_item:n.)

\l__seq_internal_a_tl Scratch space for various internal uses.

```

\l__seq_internal_b_tl
6159 \tl_new:N \l__seq_internal_a_tl
6160 \tl_new:N \l__seq_internal_b_tl

```

(End definition for \l__seq_internal_a_tl and \l__seq_internal_b_tl. These variables are documented on page ??.)

__seq_tmp:w Scratch function for internal use.

```

6161 \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.

```

6162 \tl_const:Nn \c_empty_seq { \s__seq }

```

(End definition for \c_empty_seq. This variable is documented on page 129.)

12.1 Allocation and initialisation

\seq_new:N Sequences are initialized to \c_empty_seq.

```

\seq_new:c
6163 \cs_new_protected:Npn \seq_new:N #1
6164 {
6165   \__chk_if_free_cs:N #1
6166   \cs_gset_eq:NN #1 \c_empty_seq
6167 }
6168 \cs_generate_variant:Nn \seq_new:N { c }

```

(End definition for \seq_new:N and \seq_new:c. These functions are documented on page 119.)

\seq_clear:N Clearing a sequence is similar to setting it equal to the empty one.

```

\seq_clear:c
\seq_gclear:N
\seq_gclear:c
6169 \cs_new_protected:Npn \seq_clear:N #1
6170 { \seq_set_eq:NN #1 \c_empty_seq }
6171 \cs_generate_variant:Nn \seq_clear:N { c }
6172 \cs_new_protected:Npn \seq_gclear:N #1
6173 { \seq_gset_eq:NN #1 \c_empty_seq }
6174 \cs_generate_variant:Nn \seq_gclear:N { c }

```

(End definition for \seq_clear:N and \seq_clear:c. These functions are documented on page 119.)

`\seq_clear_new:N` Once again we copy code from the token list functions.

```

\seq_clear_new:c 6175 \cs_new_protected:Npn \seq_clear_new:N #1
\seq_gclear_new:N 6176 { \seq_if_exist:NTF #1 { \seq_clear:N #1 } { \seq_new:N #1 } }
\seq_gclear_new:c 6177 \cs_generate_variant:Nn \seq_clear_new:N { c }
6178 \cs_new_protected:Npn \seq_gclear_new:N #1
6179 { \seq_if_exist:NTF #1 { \seq_gclear:N #1 } { \seq_new:N #1 } }
6180 \cs_generate_variant:Nn \seq_gclear_new:N { c }

```

(End definition for `\seq_clear_new:N` and `\seq_clear_new:c`. These functions are documented on page 119.)

`\seq_set_eq:NN` Copying a sequence is the same as copying the underlying token list.

```

\seq_set_eq:cN 6181 \cs_new_eq:NN \seq_set_eq:NN \tl_set_eq:NN
\seq_set_eq:Nc 6182 \cs_new_eq:NN \seq_set_eq:Nc \tl_set_eq:Nc
\seq_set_eq:cc 6183 \cs_new_eq:NN \seq_set_eq:cN \tl_set_eq:cN
\seq_gset_eq:NN 6184 \cs_new_eq:NN \seq_set_eq:cc \tl_set_eq:cc
\seq_gset_eq:cN 6185 \cs_new_eq:NN \seq_gset_eq:NN \tl_gset_eq:NN
\seq_gset_eq:Nc 6186 \cs_new_eq:NN \seq_gset_eq:Nc \tl_gset_eq:Nc
\seq_gset_eq:cN 6187 \cs_new_eq:NN \seq_gset_eq:cN \tl_gset_eq:cN
\seq_gset_eq:cc 6188 \cs_new_eq:NN \seq_gset_eq:cc \tl_gset_eq:cc

```

(End definition for `\seq_set_eq:NN` and others. These functions are documented on page 119.)

`\seq_set_from_clist:NN` Setting a sequence from a comma-separated list is done using a simple mapping.

```

\seq_set_from_clist:cN 6189 \cs_new_protected:Npn \seq_set_from_clist:NN #1#2
\seq_set_from_clist:Nc 6190 {
\seq_set_from_clist:cc 6191   \tl_set:Nx #1
\seq_set_from_clist:Nn 6192   { \s__seq \clist_map_function:NN #2 \__seq_wrap_item:n }
\seq_set_from_clist:cn 6193 }
\seq_gset_from_clist:NN 6194 \cs_new_protected:Npn \seq_set_from_clist:Nn #1#2
\seq_gset_from_clist:cN 6195 {
\seq_gset_from_clist:Nc 6196   \tl_set:Nx #1
\seq_gset_from_clist:cc 6197   { \s__seq \clist_map_function:nN {#2} \__seq_wrap_item:n }
\seq_gset_from_clist:Nn 6198 }
\seq_gset_from_clist:Nn 6199 \cs_new_protected:Npn \seq_gset_from_clist:NN #1#2
\seq_gset_from_clist:cn 6200 {
6201   \tl_gset:Nx #1
6202   { \s__seq \clist_map_function:NN #2 \__seq_wrap_item:n }
6203 }
6204 \cs_new_protected:Npn \seq_gset_from_clist:Nn #1#2
6205 {
6206   \tl_gset:Nx #1
6207   { \s__seq \clist_map_function:nN {#2} \__seq_wrap_item:n }
6208 }
6209 \cs_generate_variant:Nn \seq_set_from_clist:NN { Nc }
6210 \cs_generate_variant:Nn \seq_set_from_clist:NN { c , cc }
6211 \cs_generate_variant:Nn \seq_set_from_clist:Nn { c }
6212 \cs_generate_variant:Nn \seq_gset_from_clist:NN { Nc }
6213 \cs_generate_variant:Nn \seq_gset_from_clist:NN { c , cc }
6214 \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 119.)

```

\seq_set_split:Nnn When the separator is empty, everything is very simple, just map \__seq_wrap_item:n
\seq_set_split:NnV through the items of the last argument. For non-trivial separators, the goal is to split
\seq_gset_split:Nnn a given token list at the marker, strip spaces from each item, and remove one set of
\seq_gset_split:NnV outer braces if after removing leading and trailing spaces the item is enclosed within
\__seq_set_split:NNnn braces. After \tl_replace_all:Nnn, the token list \l__seq_internal_a_tl is a repe-
\__seq_set_split_auxi:w tion of the pattern \__seq_set_split_auxi:w \prg_do_nothing: <item with spaces>
\__seq_set_split_auxii:w \__seq_set_split_end:. Then, x-expansion causes \__seq_set_split_auxi:w to trim
\__seq_set_split_end: 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.

6215 \cs_new_protected_nopar:Npn \seq_set_split:Nnn
6216 { \__seq_set_split:NNnn \tl_set:Nx }
6217 \cs_new_protected_nopar:Npn \seq_gset_split:Nnn
6218 { \__seq_set_split:NNnn \tl_gset:Nx }
6219 \cs_new_protected:Npn \__seq_set_split:NNnn #1#2#3#4
6220 {
6221   \tl_if_empty:nTF {#3}
6222   {
6223     \tl_set:Nn \l__seq_internal_a_tl
6224     { \tl_map_function:nN {#4} \__seq_wrap_item:n }
6225   }
6226   {
6227     \tl_set:Nn \l__seq_internal_a_tl
6228     {
6229       \__seq_set_split_auxi:w \prg_do_nothing:
6230       #4
6231       \__seq_set_split_end:
6232     }
6233     \tl_replace_all:Nnn \l__seq_internal_a_tl { #3 }
6234     {
6235       \__seq_set_split_end:
6236       \__seq_set_split_auxi:w \prg_do_nothing:
6237     }
6238     \tl_set:Nx \l__seq_internal_a_tl { \l__seq_internal_a_tl }
6239   }
6240   #1 #2 { \s__seq \l__seq_internal_a_tl }
6241 }
6242 \cs_new:Npn \__seq_set_split_auxi:w #1 \__seq_set_split_end:
6243 {
6244   \exp_not:N \__seq_set_split_auxii:w
6245   \exp_args:No \tl_trim_spaces:n {#1}
6246   \exp_not:N \__seq_set_split_end:
6247 }
6248 \cs_new:Npn \__seq_set_split_auxii:w #1 \__seq_set_split_end:
6249 { \__seq_wrap_item:n {#1} }

```

```

6250 \cs_generate_variant:Nn \seq_set_split:Nnn { NnV }
6251 \cs_generate_variant:Nn \seq_gset_split:Nnn { NnV }

```

(End definition for `\seq_set_split:Nnn` and others. These functions are documented on page 120.)

`\seq_concat:NNN` 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.

```

\seq_concat:ccc
\seq_gconcat:NNN
\seq_gconcat:ccc
6252 \cs_new_protected:Npn \seq_concat:NNN #1#2#3
6253 { \tl_set:Nf #1 { \exp_after:wN \use_i:nn \exp_after:wN #2 #3 } }
6254 \cs_new_protected:Npn \seq_gconcat:NNN #1#2#3
6255 { \tl_gset:Nf #1 { \exp_after:wN \use_i:nn \exp_after:wN #2 #3 } }
6256 \cs_generate_variant:Nn \seq_concat:NNN { ccc }
6257 \cs_generate_variant:Nn \seq_gconcat:NNN { ccc }

```

(End definition for `\seq_concat:NNN` and `\seq_concat:ccc`. These functions are documented on page 120.)

`\seq_if_exist_p:N` Copies of the `cs` functions defined in `l3basics`.

```

\seq_if_exist_p:c
\seq_if_exist:NTF
\seq_if_exist:cTF
6258 \prg_new_eq_conditional:NNn \seq_if_exist:N \cs_if_exist:N
6259 { TF , T , F , p }
6260 \prg_new_eq_conditional:NNn \seq_if_exist:c \cs_if_exist:c
6261 { TF , T , F , p }

```

(End definition for `\seq_if_exist:NTF` and `\seq_if_exist:cTF`. These functions are documented on page 120.)

12.2 Appending data to either end

`\seq_put_left:Nn` When adding to the left of a sequence, remove `\s__seq`. This is done by `__seq_put_left_aux:w`, which also stops f-expansion.

```

\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
6262 \cs_new_protected:Npn \seq_put_left:Nn #1#2
6263 {
6264   \tl_set:Nx #1
6265   {
6266     \exp_not:n { \s__seq \__seq_item:n {#2} }
6267     \exp_not:f { \exp_after:wN \__seq_put_left_aux:w #1 }
6268   }
6269 }
6270 \cs_new_protected:Npn \seq_gput_left:Nn #1#2
6271 {
6272   \tl_gset:Nx #1
6273   {
6274     \exp_not:n { \s__seq \__seq_item:n {#2} }
6275     \exp_not:f { \exp_after:wN \__seq_put_left_aux:w #1 }
6276   }
6277 }
6278 \cs_new:Npn \__seq_put_left_aux:w \s__seq { \exp_stop_f: }
6279 \cs_generate_variant:Nn \seq_put_left:Nn { NV , Nv , No , Nx }
6280 \cs_generate_variant:Nn \seq_put_left:Nn { c , cV , cv , co , cx }
6281 \cs_generate_variant:Nn \seq_gput_left:Nn { NV , Nv , No , Nx }
6282 \cs_generate_variant:Nn \seq_gput_left:Nn { c , cV , cv , co , cx }

```

(End definition for `\seq_put_left:Nn` and others. These functions are documented on page 120.)

`\seq_put_right:Nn` Since there is no trailing marker, adding an item to the right of a sequence simply means wrapping it in `__seq_item:n`.

```

\seq_put_right:NV
\seq_put_right:Nv
\seq_put_right:No
\seq_put_right:Nx
\seq_put_right:cn
\seq_put_right:cV
\seq_put_right:cv
\seq_put_right:co
\seq_put_right:cx

```

```

6283 \cs_new_protected:Npn \seq_put_right:Nn #1#2
6284 { \tl_put_right:Nn #1 { \__seq_item:n {#2} } }
6285 \cs_new_protected:Npn \seq_gput_right:Nn #1#2
6286 { \tl_gput_right:Nn #1 { \__seq_item:n {#2} } }
6287 \cs_generate_variant:Nn \seq_gput_right:Nn { NV , Nv , No , Nx }
6288 \cs_generate_variant:Nn \seq_gput_right:Nn { c , cV , cv , co , cx }
6289 \cs_generate_variant:Nn \seq_put_right:Nn { NV , Nv , No , Nx }
6290 \cs_generate_variant:Nn \seq_put_right:Nn { c , cV , cv , co , cx }

```

(End definition for `\seq_put_right:Nn` and others. These functions are documented on page 120.)

```

\seq_gput_right:NV
\seq_gput_right:Nv
\seq_gput_right:No
\seq_gput_right:Nx
\seq_gput_right:cn
\seq_gput_right:cV
\seq_gput_right:cv
\seq_gput_right:co
\l__seq_remove_seq
\seq_gput_right:cx

```

12.3 Modifying sequences

This function converts its argument to a proper sequence item in an x-expansion context.

```

6291 \cs_new:Npn \__seq_wrap_item:n #1 { \exp_not:n { \__seq_item:n {#1} } }

```

(End definition for `__seq_wrap_item:n`.)

An internal sequence for the removal routines.

```

6292 \seq_new:N \l__seq_remove_seq

```

(End definition for `\l__seq_remove_seq`. This variable is documented on page ??.)

Removing duplicates means making a new list then copying it.

```

\seq_remove_duplicates:N
\seq_remove_duplicates:c
\seq_gremove_duplicates:N
\seq_gremove_duplicates:c
\__seq_remove_duplicates:NN

```

```

6293 \cs_new_protected:Npn \seq_remove_duplicates:N
6294 { \__seq_remove_duplicates:NN \seq_set_eq:NN }
6295 \cs_new_protected:Npn \seq_gremove_duplicates:N
6296 { \__seq_remove_duplicates:NN \seq_gset_eq:NN }
6297 \cs_new_protected:Npn \__seq_remove_duplicates:NN #1#2
6298 {
6299   \seq_clear:N \l__seq_remove_seq
6300   \seq_map_inline:Nn #2
6301   {
6302     \seq_if_in:NnF \l__seq_remove_seq {##1}
6303     { \seq_put_right:Nn \l__seq_remove_seq {##1} }
6304   }
6305   #1 #2 \l__seq_remove_seq
6306 }
6307 \cs_generate_variant:Nn \seq_remove_duplicates:N { c }
6308 \cs_generate_variant:Nn \seq_gremove_duplicates:N { c }

```

(End definition for `\seq_remove_duplicates:N` and `\seq_remove_duplicates:c`. These functions are documented on page 123.)

`\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 and intermediate scratch list: the lead-off x-type expansion (`#1 #2 {#2}`) will ensure that nothing is lost.

```

6309 \cs_new_protected:Npn \seq_remove_all:Nn
6310 { \__seq_remove_all_aux:NNn \tl_set:Nx }
6311 \cs_new_protected:Npn \seq_gremove_all:Nn
6312 { \__seq_remove_all_aux:NNn \tl_gset:Nx }
6313 \cs_new_protected:Npn \__seq_remove_all_aux:NNn #1#2#3
6314 {
6315   \__seq_push_item_def:n
6316   {
6317     \str_if_eq:nnT {##1} {#3}
6318     {
6319       \if_false: { \fi: }
6320       \tl_set:Nn \l__seq_internal_b_tl {##1}
6321       #1 #2
6322       { \if_false: } \fi:
6323       \exp_not:o {#2}
6324       \tl_if_eq:NNT \l__seq_internal_a_tl \l__seq_internal_b_tl
6325       { \use_none:nn }
6326     }
6327     \__seq_wrap_item:n {##1}
6328   }
6329   \tl_set:Nn \l__seq_internal_a_tl {#3}
6330   #1 #2 {#2}
6331   \__seq_pop_item_def:
6332 }
6333 \cs_generate_variant:Nn \seq_remove_all:Nn { c }
6334 \cs_generate_variant:Nn \seq_gremove_all:Nn { c }

```

(End definition for `\seq_remove_all:Nn` and `\seq_remove_all:cn`. These functions are documented on page 123.)

`\seq_reverse:N` Previously, `\seq_reverse:N` was coded by collecting the items in reverse order after an `\exp_stop_f:` marker.

```

\seq_reverse:c
\seq_greverse:N
\seq_greverse:c
\__seq_reverse:NN
\__seq_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_f:

```

```

{
  #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.

```

6335 \cs_new_protected_nopar:Npn \seq_reverse:N
6336 { \_seq_reverse:NN \tl_set:Nx }
6337 \cs_new_protected_nopar:Npn \seq_greverse:N
6338 { \_seq_reverse:NN \tl_gset:Nx }
6339 \cs_new_protected:Npn \_seq_reverse:NN #1 #2
6340 {
6341   \cs_set_eq:NN \_seq_tmp:w \_seq_item:n
6342   \cs_set_eq:NN \_seq_item:n \_seq_reverse_item:nwn
6343   #1 #2 { #2 \exp_not:n { } }
6344   \cs_set_eq:NN \_seq_item:n \_seq_tmp:w
6345 }
6346 \cs_new:Npn \_seq_reverse_item:nwn #1 #2 \exp_not:n #3
6347 {
6348   #2
6349   \exp_not:n { \_seq_item:n {#1} #3 }
6350 }
6351 \cs_generate_variant:Nn \seq_reverse:N { c }
6352 \cs_generate_variant:Nn \seq_greverse:N { c }

```

(End definition for `\seq_reverse:N` and others. These functions are documented on page [123](#).)

12.4 Sequence conditionals

<code>\seq_if_empty_p:N</code>	Similar to token lists, we compare with the empty sequence.
<code>\seq_if_empty_p:c</code>	
<code>\seq_if_empty:NTF</code>	
<code>\seq_if_empty:cTF</code>	

```

6353 \prg_new_conditional:Npnn \seq_if_empty:N #1 { p , T , F , TF }
6354 {
6355   \if_meaning:w #1 \c_empty_seq
6356   \prg_return_true:
6357   \else:
6358   \prg_return_false:
6359   \fi:
6360 }

```

```

6361 \cs_generate_variant:Nn \seq_if_empty_p:N { c }
6362 \cs_generate_variant:Nn \seq_if_empty:NT { c }
6363 \cs_generate_variant:Nn \seq_if_empty:NF { c }
6364 \cs_generate_variant:Nn \seq_if_empty:NTF { c }

```

(End definition for `\seq_if_empty:NTF` and `\seq_if_empty:cTF`. These functions are documented on page 124.)

`\seq_if_in:NnTF` The approach here is to define `__seq_item:n` to compare its argument with the test sequence. If the two items are equal, the mapping is terminated and `\group_end: \prg_return_true:` is inserted after skipping over the rest of the recursion. On the other hand, if there is no match then the loop will break returning `\prg_return_false:`. Everything is inside a group so that `__seq_item:n` is preserved in nested situations.

```

6365 \prg_new_protected_conditional:Npnn \seq_if_in:Nn #1#2
6366 { T , F , TF }
6367 {
6368   \group_begin:
6369     \tl_set:Nn \l__seq_internal_a_tl {#2}
6370     \cs_set_protected:Npn \__seq_item:n ##1
6371     {
6372       \tl_set:Nn \l__seq_internal_b_tl {##1}
6373       \if_meaning:w \l__seq_internal_a_tl \l__seq_internal_b_tl
6374       \exp_after:wN \__seq_if_in:
6375       \fi:
6376     }
6377     #1
6378   \group_end:
6379   \prg_return_false:
6380   \__prg_break_point:
6381 }
6382 \cs_new_nopar:Npn \__seq_if_in:
6383 { \__prg_break:n { \group_end: \prg_return_true: } }
6384 \cs_generate_variant:Nn \seq_if_in:NnT { NV , Nv , No , Nx }
6385 \cs_generate_variant:Nn \seq_if_in:NnT { c , cV , cv , co , cx }
6386 \cs_generate_variant:Nn \seq_if_in:NnF { NV , Nv , No , Nx }
6387 \cs_generate_variant:Nn \seq_if_in:NnF { c , cV , cv , co , cx }
6388 \cs_generate_variant:Nn \seq_if_in:NnTF { NV , Nv , No , Nx }
6389 \cs_generate_variant:Nn \seq_if_in:NnTF { c , cV , cv , co , cx }

```

(End definition for `\seq_if_in:NnTF` and others. These functions are documented on page 124.)

12.5 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.

```

6390 \cs_new_protected:Npn \__seq_pop:NnNN #1#2#3#4
6391 {
6392   \if_meaning:w #3 \c_empty_seq
6393   \tl_set:Nn #4 { \q_no_value }
6394   \else:

```

```

6395         #1#2#3#4
6396     \fi:
6397 }
6398 \cs_new_protected:Npn \__seq_pop_TF:NNNN #1#2#3#4
6399 {
6400     \if_meaning:w #3 \c_empty_seq
6401     % \tl_set:Nn #4 { \q_no_value }
6402     \prg_return_false:
6403 }else:
6404     #1#2#3#4
6405     \prg_return_true:
6406 \fi:
6407 }

```

(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
\seq_get_left:cN after __seq_item:n at the start. We append a \q_no_value item to cover the case of
__seq_get_left:wnw an empty sequence

```

6408 \cs_new_protected:Npn \seq_get_left:NN #1#2
6409 {
6410     \tl_set:Nx #2
6411     {
6412         \exp_after:wN \__seq_get_left:wnw
6413         #1 \__seq_item:n { \q_no_value } \q_stop
6414     }
6415 }
6416 \cs_new:Npn \__seq_get_left:wnw #1 \__seq_item:n #2#3 \q_stop
6417 { \exp_not:n {#2} }
6418 \cs_generate_variant:Nn \seq_get_left:NN { c }

```

(End definition for \seq_get_left:NN and \seq_get_left:cN. These functions are documented on page 121.)

\seq_pop_left:NN The approach to popping an item is pretty similar to that to get an item, with the only
\seq_pop_left:cN difference being that the sequence itself has to be redefined. This makes it more sensible
\seq_gpop_left:NN to use an auxiliary function for the local and global cases.

```

6419 \cs_new_protected_nopar:Npn \seq_pop_left:NN
6420 { \__seq_pop:NNNN \__seq_pop_left:NNN \tl_set:Nn }
6421 \cs_new_protected_nopar:Npn \seq_gpop_left:NN
6422 { \__seq_pop:NNNN \__seq_pop_left:NNN \tl_gset:Nn }
6423 \cs_new_protected:Npn \__seq_pop_left:NNN #1#2#3
6424 { \exp_after:wN \__seq_pop_left:wnwNNN #2 \q_stop #1#2#3 }
6425 \cs_new_protected:Npn \__seq_pop_left:wnwNNN
6426 #1 \__seq_item:n #2#3 \q_stop #4#5#6
6427 {
6428     #4 #5 { #1 #3 }
6429     \tl_set:Nn #6 {#2}
6430 }
6431 \cs_generate_variant:Nn \seq_pop_left:NN { c }
6432 \cs_generate_variant:Nn \seq_gpop_left:NN { c }

```

(End definition for `\seq_pop_left:NN` and `\seq_pop_left:cN`. These functions are documented on page 121.)

`\seq_get_right:NN` First remove `\s__seq` and prepend `\q_no_value`, then take two arguments at a time.
`\seq_get_right:cN` 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.

```

6433 \cs_new_protected:Npn \seq_get_right:NN #1#2
6434 {
6435   \exp_after:wN \use_i_ii:nnn
6436   \exp_after:wN \__seq_get_right_loop:nn
6437   \exp_after:wN \q_no_value
6438   #1
6439   { ?? \tl_set:Nn #2 }
6440   { } { }
6441 }
6442 \cs_new_protected:Npn \__seq_get_right_loop:nn #1#2
6443 {
6444   \use_none:nn #2 {#1}
6445   \__seq_get_right_loop:nn
6446 }
6447 \cs_generate_variant:Nn \seq_get_right:NN { c }

```

(End definition for `\seq_get_right:NN` and `\seq_get_right:cN`. These functions are documented on page 121.)

`\seq_pop_right:NN` The approach to popping from the right is a bit more involved, but does use some
`\seq_pop_right:cN` of the same ideas as getting from the right. What is needed is a “flexible length”
`\seq_gpop_right:NN` way to set a token list variable. This is supplied by the `{ \if_false: } \fi:`
`\seq_gpop_right:cN` `... \if_false: { \fi: }` construct. Using an x-type expansion and a “non-expanding”
`__seq_pop_right:NNN` definition for `__seq_item:n`, the left-most $n - 1$ entries in a sequence of n items will
`__seq_pop_right_loop:nn` 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.

```

6448 \cs_new_protected_nopar:Npn \seq_pop_right:NN
6449 { \__seq_pop:NNNN \__seq_pop_right:NNN \tl_set:Nx }
6450 \cs_new_protected_nopar:Npn \seq_gpop_right:NN
6451 { \__seq_pop:NNNN \__seq_pop_right:NNN \tl_gset:Nx }
6452 \cs_new_protected:Npn \__seq_pop_right:NNN #1#2#3
6453 {
6454   \cs_set_eq:NN \__seq_tmp:w \__seq_item:n
6455   \cs_set_eq:NN \__seq_item:n \scan_stop:
6456   #1 #2
6457   { \if_false: } \fi: \s__seq
6458   \exp_after:wN \use_i:nnn

```

```

6459         \exp_after:wN \__seq_pop_right_loop:nn
6460         #2
6461         {
6462             \if_false: { \fi: }
6463             \tl_set:Nx #3
6464         }
6465         { } \use_none:nn
6466         \cs_set_eq:NN \__seq_item:n \__seq_tmp:w
6467     }
6468     \cs_new:Npn \__seq_pop_right_loop:nn #1#2
6469     {
6470         #2 { \exp_not:n {#1} }
6471         \__seq_pop_right_loop:nn
6472     }
6473     \cs_generate_variant:Nn \seq_pop_right:NN { c }
6474     \cs_generate_variant:Nn \seq_gpop_right:NN { c }

```

(End definition for `\seq_pop_right:NN` and `\seq_pop_right:cN`. These functions are documented on page 121.)

`\seq_get_left:NNTF` Getting from the left or right with a check on the results. The first argument to `__seq_pop_TF:NNNN` is left unused.

```

\seq_get_left:cNTF
\seq_get_right:NNTF
\seq_get_right:cNTF
6475 \prg_new_protected_conditional:Npnn \seq_get_left:NN #1#2 { T , F , TF }
6476 { \__seq_pop_TF:NNNN \prg_do_nothing: \seq_get_left:NN #1#2 }
6477 \prg_new_protected_conditional:Npnn \seq_get_right:NN #1#2 { T , F , TF }
6478 { \__seq_pop_TF:NNNN \prg_do_nothing: \seq_get_right:NN #1#2 }
6479 \cs_generate_variant:Nn \seq_get_left:NN { c }
6480 \cs_generate_variant:Nn \seq_get_left:NNF { c }
6481 \cs_generate_variant:Nn \seq_get_left:NNTF { c }
6482 \cs_generate_variant:Nn \seq_get_right:NN { c }
6483 \cs_generate_variant:Nn \seq_get_right:NNF { c }
6484 \cs_generate_variant:Nn \seq_get_right:NNTF { c }

```

(End definition for `\seq_get_left:NNTF` and `\seq_get_left:cNTF`. These functions are documented on page 122.)

`\seq_pop_left:NNTF` More or less the same for popping.

```

\seq_pop_left:cNTF
\seq_gpop_left:NNTF
\seq_gpop_left:cNTF
\seq_pop_right:NNTF
\seq_pop_right:cNTF
\seq_gpop_right:NNTF
\seq_gpop_right:cNTF
6485 \prg_new_protected_conditional:Npnn \seq_pop_left:NN #1#2 { T , F , TF }
6486 { \__seq_pop_TF:NNNN \__seq_pop_left:NNN \tl_set:Nn #1 #2 }
6487 \prg_new_protected_conditional:Npnn \seq_gpop_left:NN #1#2 { T , F , TF }
6488 { \__seq_pop_TF:NNNN \__seq_pop_left:NNN \tl_gset:Nn #1 #2 }
6489 \prg_new_protected_conditional:Npnn \seq_pop_right:NN #1#2 { T , F , TF }
6490 { \__seq_pop_TF:NNNN \__seq_pop_right:NNN \tl_set:Nx #1 #2 }
6491 \prg_new_protected_conditional:Npnn \seq_gpop_right:NN #1#2 { T , F , TF }
6492 { \__seq_pop_TF:NNNN \__seq_pop_right:NNN \tl_gset:Nx #1 #2 }
6493 \cs_generate_variant:Nn \seq_pop_left:NN { c }
6494 \cs_generate_variant:Nn \seq_pop_left:NNF { c }
6495 \cs_generate_variant:Nn \seq_pop_left:NNTF { c }
6496 \cs_generate_variant:Nn \seq_gpop_left:NN { c }
6497 \cs_generate_variant:Nn \seq_gpop_left:NNF { c }

```

```

6498 \cs_generate_variant:Nn \seq_gpop_left:NNTF { c }
6499 \cs_generate_variant:Nn \seq_pop_right:NNT { c }
6500 \cs_generate_variant:Nn \seq_pop_right:NNF { c }
6501 \cs_generate_variant:Nn \seq_pop_right:NNTF { c }
6502 \cs_generate_variant:Nn \seq_gpop_right:NNT { c }
6503 \cs_generate_variant:Nn \seq_gpop_right:NNF { c }
6504 \cs_generate_variant:Nn \seq_gpop_right:NNTF { c }

```

(End definition for `\seq_pop_left:NNTF` and `\seq_pop_left:CNF`. These functions are documented on page 122.)

`\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.

`\seq_item:cn`

`__seq_item:wNn`

`__seq_item:nnn`

```

6505 \cs_new:Npn \seq_item:Nn #1
6506 { \exp_after:wN \__seq_item:wNn #1 \q_stop #1 }
6507 \cs_new:Npn \__seq_item:wNn \s__seq #1 \q_stop #2#3
6508 {
6509   \exp_args:Nf \__seq_item:nnn
6510   {
6511     \int_eval:n
6512     {
6513       \int_compare:nNnT {#3} < \c_zero
6514       { \seq_count:N #2 + \c_one + }
6515       #3
6516     }
6517   }
6518   #1
6519   { ? \__prg_break: } { }
6520   \__prg_break_point:
6521 }
6522 \cs_new:Npn \__seq_item:nnn #1#2#3
6523 {
6524   \use_none:n #2
6525   \int_compare:nNnTF {#1} = \c_one
6526   { \__prg_break:n { \exp_not:n {#3} } }
6527   { \exp_args:Nf \__seq_item:nnn { \int_eval:n { #1 - 1 } } }
6528 }
6529 \cs_generate_variant:Nn \seq_item:Nn { c }

```

(End definition for `\seq_item:Nn` and `\seq_item:cn`. These functions are documented on page 122.)

12.6 Mapping to sequences

`\seq_map_break:` To break a function, the special token `__prg_break_point:Nn` is used to find the end of the code. Any ending code is then inserted before the return value of `\seq_map_break:n` is inserted.

`\seq_map_break:n`

```

6530 \cs_new_nopar:Npn \seq_map_break:
6531 { \__prg_break_point:Nn \seq_map_break: { } }

```

```

6532 \cs_new_nopar:Npn \seq_map_break:n
6533 { \__prg_map_break:Nn \seq_map_break: }

```

(End definition for \seq_map_break:. This function is documented on page 125.)

\seq_map_function:NN The idea here is to apply the code of #2 to each item in the sequence without altering the definition of __seq_item:n. 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.

\seq_map_function:cN
__seq_map_function:NNn

```

6534 \cs_new:Npn \seq_map_function:NN #1#2
6535 {
6536   \exp_after:wN \use_i_ii:nnn
6537   \exp_after:wN \__seq_map_function:NNn
6538   \exp_after:wN #2
6539   #1
6540   { ? \seq_map_break: } { }
6541   \__prg_break_point:Nn \seq_map_break: { }
6542 }
6543 \cs_new:Npn \__seq_map_function:NNn #1#2#3
6544 {
6545   \use_none:n #2
6546   #1 {#3}
6547   \__seq_map_function:NNn #1
6548 }
6549 \cs_generate_variant:Nn \seq_map_function:NN { c }

```

(End definition for \seq_map_function:NN and \seq_map_function:cN. These functions are documented on page 124.)

__seq_push_item_def:n The definition of __seq_item:n needs to be saved and restored at various points within the mapping and manipulation code. That is handled here: as always, this approach uses global assignments.

__seq_push_item_def:x
__seq_push_item_def:
__seq_pop_item_def:

```

6550 \cs_new_protected:Npn \__seq_push_item_def:n
6551 {
6552   \__seq_push_item_def:
6553   \cs_gset:Npn \__seq_item:n ##1
6554 }
6555 \cs_new_protected:Npn \__seq_push_item_def:x
6556 {
6557   \__seq_push_item_def:
6558   \cs_gset:Npx \__seq_item:n ##1
6559 }
6560 \cs_new_protected:Npn \__seq_push_item_def:
6561 {
6562   \int_gincr:N \g__prg_map_int
6563   \cs_gset_eq:cN { __prg_map_ \int_use:N \g__prg_map_int :w }
6564   \__seq_item:n
6565 }
6566 \cs_new_protected_nopar:Npn \__seq_pop_item_def:

```

```

6567 {
6568   \cs_gset_eq:Nc \__seq_item:n
6569   { \prg_map_ \int_use:N \g__prg_map_int :w }
6570   \int_gdecr:N \g__prg_map_int
6571 }

```

(End definition for __seq_push_item_def:n and __seq_push_item_def:x.)

\seq_map_inline:Nn The idea here is that __seq_item:n is already “applied” to each item in a sequence,
\seq_map_inline:cn and so an in-line mapping is just a case of redefining __seq_item:n.

```

6572 \cs_new_protected:Npn \seq_map_inline:Nn #1#2
6573 {
6574   \__seq_push_item_def:n {#2}
6575   #1
6576   \prg_break_point:Nn \seq_map_break: { \__seq_pop_item_def: }
6577 }
6578 \cs_generate_variant:Nn \seq_map_inline:Nn { c }

```

(End definition for \seq_map_inline:Nn and \seq_map_inline:cn. These functions are documented on page 124.)

\seq_map_variable:NNn 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.
\seq_map_variable:Ncn
\seq_map_variable:cNn
\seq_map_variable:ccn

```

6579 \cs_new_protected:Npn \seq_map_variable:NNn #1#2#3
6580 {
6581   \__seq_push_item_def:x
6582   {
6583     \tl_set:Nn \exp_not:N #2 {##1}
6584     \exp_not:n {#3}
6585   }
6586   #1
6587   \prg_break_point:Nn \seq_map_break: { \__seq_pop_item_def: }
6588 }
6589 \cs_generate_variant:Nn \seq_map_variable:NNn { Nc }
6590 \cs_generate_variant:Nn \seq_map_variable:NNn { c , cc }

```

(End definition for \seq_map_variable:NNn and others. These functions are documented on page 124.)

\seq_count:N Counting the items in a sequence is done using the same approach as for other count
\seq_count:c functions: turn each entry into a +1 then use integer evaluation to actually do the math-
__seq_count:n ematics.

```

6591 \cs_new:Npn \seq_count:N #1
6592 {
6593   \int_eval:n
6594   {
6595     0
6596     \seq_map_function:NN #1 \__seq_count:n
6597   }
6598 }
6599 \cs_new:Npn \__seq_count:n #1 { + \c_one }
6600 \cs_generate_variant:Nn \seq_count:N { c }

```

(End definition for `\seq_count:N` and `\seq_count:c`. These functions are documented on page 125.)

12.7 Using sequences

`\seq_use:Nnnn` 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`.

`\seq_use:cnnn`

`_seq_use:NNnNnn`

`_seq_use_setup:w`

`_seq_use:nwwwnwn`

`_seq_use:nwn`

`\seq_use:Nn`

`\seq_use:cn`

```

6601 \cs_new:Npn \seq_use:Nnnn #1#2#3#4
6602 {
6603   \seq_if_exist:NTF #1
6604   {
6605     \int_case:nnF { \seq_count:N #1 }
6606     {
6607       { 0 } { }
6608       { 1 } { \exp_after:wN \_seq_use:NNnNnn #1 ? { } { } }
6609       { 2 } { \exp_after:wN \_seq_use:NNnNnn #1 {#2} }
6610     }
6611     {
6612       \exp_after:wN \_seq_use_setup:w #1 \_seq_item:n
6613       \q_mark { \_seq_use:nwwwnwn {#3} }
6614       \q_mark { \_seq_use:nwn {#4} }
6615       \q_stop { }
6616     }
6617   }
6618   {
6619     \_msg_kernel_expandable_error:nnn
6620     { kernel } { bad-variable } {#1}
6621   }
6622 }
6623 \cs_generate_variant:Nn \seq_use:Nnnn { c }
6624 \cs_new:Npn \_seq_use:NNnNnn #1#2#3#4#5#6 { \exp_not:n { #3 #6 #5 } }
6625 \cs_new:Npn \_seq_use_setup:w \s__seq { \_seq_use:nwwwnwn { } }
6626 \cs_new:Npn \_seq_use:nwwwnwn
6627   #1 \_seq_item:n #2 \_seq_item:n #3 \_seq_item:n #4#5
6628   \q_mark #6#7 \q_stop #8
6629   {
6630     #6 \_seq_item:n {#3} \_seq_item:n {#4} #5
6631     \q_mark {#6} #7 \q_stop { #8 #1 #2 }
6632   }
6633 \cs_new:Npn \_seq_use:nwn #1 \_seq_item:n #2 #3 \q_stop #4
6634   { \exp_not:n { #4 #1 #2 } }
6635 \cs_new:Npn \seq_use:Nn #1#2
6636   { \seq_use:Nnnn #1 {#2} {#2} {#2} }
6637 \cs_generate_variant:Nn \seq_use:Nn { c }

```

(End definition for `\seq_use:Nnnn` and `\seq_use:cnnn`. These functions are documented on page 126.)

12.8 Sequence stacks

The same functions as for sequences, but with the correct naming.

\seq_push:Nn Pushing to a sequence is the same as adding on the left.

```

\seq_push:NV 6638 \cs_new_eq:NN \seq_push:Nn \seq_put_left:Nn
\seq_push:Nv 6639 \cs_new_eq:NN \seq_push:Nv \seq_put_left:Nv
\seq_push:No 6640 \cs_new_eq:NN \seq_push:Nv \seq_put_left:Nv
\seq_push:Nx 6641 \cs_new_eq:NN \seq_push:No \seq_put_left:No
\seq_push:cn 6642 \cs_new_eq:NN \seq_push:Nx \seq_put_left:Nx
\seq_push:cV 6643 \cs_new_eq:NN \seq_push:cn \seq_put_left:cn
\seq_push:cV 6644 \cs_new_eq:NN \seq_push:cV \seq_put_left:cV
\seq_push:cv 6645 \cs_new_eq:NN \seq_push:cv \seq_put_left:cv
\seq_push:co 6646 \cs_new_eq:NN \seq_push:co \seq_put_left:co
\seq_push:cx 6647 \cs_new_eq:NN \seq_push:cx \seq_put_left:cx
\seq_gpush:Nn 6648 \cs_new_eq:NN \seq_gpush:Nn \seq_gput_left:Nn
\seq_gpush:NV 6649 \cs_new_eq:NN \seq_gpush:Nv \seq_gput_left:Nv
\seq_gpush:Nv 6650 \cs_new_eq:NN \seq_gpush:Nv \seq_gput_left:Nv
\seq_gpush:No 6651 \cs_new_eq:NN \seq_gpush:No \seq_gput_left:No
\seq_gpush:Nx 6652 \cs_new_eq:NN \seq_gpush:Nx \seq_gput_left:Nx
\seq_gpush:cn 6653 \cs_new_eq:NN \seq_gpush:cn \seq_gput_left:cn
\seq_gpush:cV 6654 \cs_new_eq:NN \seq_gpush:cV \seq_gput_left:cV
\seq_gpush:cv 6655 \cs_new_eq:NN \seq_gpush:cv \seq_gput_left:cv
\seq_gpush:co 6656 \cs_new_eq:NN \seq_gpush:co \seq_gput_left:co
\seq_gpush:cx 6657 \cs_new_eq:NN \seq_gpush:cx \seq_gput_left:cx

```

(End definition for \seq_push:Nn and others. These functions are documented on page 127.)

\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 6658 \cs_new_eq:NN \seq_get:NN \seq_get_left:NN
\seq_pop:cN 6659 \cs_new_eq:NN \seq_get:cN \seq_get_left:cN
\seq_gpop:NN 6660 \cs_new_eq:NN \seq_pop:NN \seq_pop_left:NN
\seq_gpop:cN 6661 \cs_new_eq:NN \seq_pop:cN \seq_pop_left:cN
6662 \cs_new_eq:NN \seq_gpop:NN \seq_gpop_left:NN
6663 \cs_new_eq:NN \seq_gpop:cN \seq_gpop_left:cN

```

(End definition for \seq_get:NN and \seq_get:cN. These functions are documented on page 127.)

\seq_get:NNTF More copies.

```

\seq_get:cNTF 6664 \prg_new_eq_conditional:NNn \seq_get:NN \seq_get_left:NN { T , F , TF }
\seq_pop:NNTF 6665 \prg_new_eq_conditional:NNn \seq_get:cN \seq_get_left:cN { T , F , TF }
\seq_pop:cNTF 6666 \prg_new_eq_conditional:NNn \seq_pop:NN \seq_pop_left:NN { T , F , TF }
\seq_gpop:NNTF 6667 \prg_new_eq_conditional:NNn \seq_pop:cN \seq_pop_left:cN { T , F , TF }
\seq_gpop:cNTF 6668 \prg_new_eq_conditional:NNn \seq_gpop:NN \seq_gpop_left:NN { T , F , TF }
6669 \prg_new_eq_conditional:NNn \seq_gpop:cN \seq_gpop_left:cN { T , F , TF }

```

(End definition for \seq_get:NNTF and \seq_get:cNTF. These functions are documented on page 127.)

12.9 Viewing sequences

\seq_show:N Apply the general `_msg_show_variable:NNNnn`.
\seq_show:c

```

6670 \cs_new_protected:Npn \seq_show:N #1
6671 {
6672   \_msg_show_variable:NNNnn #1
6673   \seq_if_exist:NTF \seq_if_empty:NTF { seq }
6674   { \seq_map_function:NN #1 \_msg_show_item:n }
6675 }
6676 \cs_generate_variant:Nn \seq_show:N { c }

```

(End definition for `\seq_show:N` and `\seq_show:c`. These functions are documented on page 130.)

12.10 Scratch sequences

\l_tmpa_seq Temporary comma list variables.
\l_tmpb_seq
\g_tmpa_seq
\g_tmpb_seq

```

6677 \seq_new:N \l_tmpa_seq
6678 \seq_new:N \l_tmpb_seq
6679 \seq_new:N \g_tmpa_seq
6680 \seq_new:N \g_tmpb_seq

```

(End definition for `\l_tmpa_seq` and others. These variables are documented on page 129.)

```

6681 </initex | package>

```

13 l3clist implementation

The following test files are used for this code: `m3clist002`.

```

6682 <*initex | package>
6683 <@@=clist>

```

\c_empty_clist An empty comma list is simply an empty token list.

```

6684 \cs_new_eq:NN \c_empty_clist \c_empty_tl

```

(End definition for `\c_empty_clist`. This variable is documented on page 139.)

\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`

```

6685 \tl_new:N \l__clist_internal_clist

```

(End definition for `\l__clist_internal_clist`. This variable is documented on page ??.)

__clist_tmp:w A temporary function for various purposes.

```

6686 \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      6687 \cs_new_eq:NN \clist_new:N \tl_new:N
                  6688 \cs_new_eq:NN \clist_new:c \tl_new:c
```

(End definition for \clist_new:N and \clist_new:c. These functions are documented on page 131.)

\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
\clist_const:Nx      6689 \cs_new_protected:Npn \clist_const:Nn #1#2
\clist_const:cx      6690 { \tl_const:Nx #1 { \__clist_trim_spaces:n {#2} } }
                  6691 \cs_generate_variant:Nn \clist_const:Nn { c , Nx , cx }
```

(End definition for \clist_const:Nn and others. These functions are documented on page 131.)

\clist_clear:N Clearing comma lists is just the same as clearing token lists.

```
\clist_clear:c      6692 \cs_new_eq:NN \clist_clear:N \tl_clear:N
\clist_gclear:N      6693 \cs_new_eq:NN \clist_clear:c \tl_clear:c
\clist_gclear:c      6694 \cs_new_eq:NN \clist_gclear:N \tl_gclear:N
                  6695 \cs_new_eq:NN \clist_gclear:c \tl_gclear:c
```

(End definition for \clist_clear:N and \clist_clear:c. These functions are documented on page 131.)

\clist_clear_new:N Once again a copy from the token list functions.

```
\clist_clear_new:c  6696 \cs_new_eq:NN \clist_clear_new:N \tl_clear_new:N
\clist_gclear_new:N 6697 \cs_new_eq:NN \clist_clear_new:c \tl_clear_new:c
\clist_gclear_new:c 6698 \cs_new_eq:NN \clist_gclear_new:N \tl_gclear_new:N
                  6699 \cs_new_eq:NN \clist_gclear_new:c \tl_gclear_new:c
```

(End definition for \clist_clear_new:N and \clist_clear_new:c. These functions are documented on page 132.)

\clist_set_eq:NN Once again, these are simple copies from the token list functions.

```
\clist_set_eq:cN    6700 \cs_new_eq:NN \clist_set_eq:NN \tl_set_eq:NN
\clist_set_eq:Nc    6701 \cs_new_eq:NN \clist_set_eq:Nc \tl_set_eq:Nc
\clist_set_eq:cN    6702 \cs_new_eq:NN \clist_set_eq:cN \tl_set_eq:cN
\clist_gset_eq:NN   6703 \cs_new_eq:NN \clist_set_eq:cc \tl_set_eq:cc
\clist_gset_eq:cN   6704 \cs_new_eq:NN \clist_gset_eq:NN \tl_gset_eq:NN
\clist_gset_eq:Nc   6705 \cs_new_eq:NN \clist_gset_eq:Nc \tl_gset_eq:Nc
\clist_gset_eq:cN   6706 \cs_new_eq:NN \clist_gset_eq:cN \tl_gset_eq:cN
\clist_gset_eq:cc   6707 \cs_new_eq:NN \clist_gset_eq:cc \tl_gset_eq:cc
```

(End definition for \clist_set_eq:NN and others. These functions are documented on page 132.)

\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 6708 \cs_new_protected:Npn \clist_set_from_seq:NN
\clist_gset_from_seq:cN 6709 { \__clist_set_from_seq:NnNNN \clist_clear:N \tl_set:Nx }
\clist_gset_from_seq:Nc
\clist_gset_from_seq:cc
```

```
\__clist_set_from_seq:NnNNN
  \__clist_wrap_item:n
  \__clist_set_from_seq:w
```

```

6710 \cs_new_protected:Npn \clist_gset_from_seq:NN
6711 { \__clist_set_from_seq:NNNN \clist_gclear:N \tl_gset:Nx }
6712 \cs_new_protected:Npn \__clist_set_from_seq:NNNN #1#2#3#4
6713 {
6714   \seq_if_empty:NTF #4
6715   { #1 #3 }
6716   {
6717     #2 #3
6718     {
6719       \exp_last_unbraced:Nf \use_none:n
6720       { \seq_map_function:NN #4 \__clist_wrap_item:n }
6721     }
6722   }
6723 }
6724 \cs_new:Npn \__clist_wrap_item:n #1
6725 {
6726   ,
6727   \tl_if_empty:oTF { \__clist_set_from_seq:w #1 ~ , #1 ~ }
6728   { \exp_not:n {#1} }
6729   { \exp_not:n { {#1} } }
6730 }
6731 \cs_new:Npn \__clist_set_from_seq:w #1 , #2 ~ { }
6732 \cs_generate_variant:Nn \clist_set_from_seq:NN { Nc }
6733 \cs_generate_variant:Nn \clist_set_from_seq:NN { c , cc }
6734 \cs_generate_variant:Nn \clist_gset_from_seq:NN { Nc }
6735 \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 132.)

```

\clist_concat:NNN Concatenating comma lists is not quite as easy as it seems, as there needs to be the
\clist_concat:ccc correct addition of a comma to the output. So a little work to do.
\clist_gconcat:NNN
\clist_gconcat:ccc
\__clist_concat:NNNN
6736 \cs_new_protected_nopar:Npn \clist_concat:NNN
6737 { \__clist_concat:NNNN \tl_set:Nx }
6738 \cs_new_protected_nopar:Npn \clist_gconcat:NNN
6739 { \__clist_concat:NNNN \tl_gset:Nx }
6740 \cs_new_protected:Npn \__clist_concat:NNNN #1#2#3#4
6741 {
6742   #1 #2
6743   {
6744     \exp_not:o #3
6745     \clist_if_empty:NF #3 { \clist_if_empty:NF #4 { , } }
6746     \exp_not:o #4
6747   }
6748 }
6749 \cs_generate_variant:Nn \clist_concat:NNN { ccc }
6750 \cs_generate_variant:Nn \clist_gconcat:NNN { ccc }

```

(End definition for `\clist_concat:NNN` and `\clist_concat:ccc`. These functions are documented on page 132.)

```

\clist_if_exist_p:N Copies of the cs functions defined in l3basics.
\clist_if_exist_p:c 6751 \prg_new_eq_conditional:NNn \clist_if_exist:N \cs_if_exist:N
\clist_if_exist:NTF 6752 { TF , T , F , p }
\clist_if_exist:cTF 6753 \prg_new_eq_conditional:NNn \clist_if_exist:c \cs_if_exist:c
6754 { TF , T , F , p }

```

(End definition for `\clist_if_exist:N` and `\clist_if_exist:c`. These functions are documented on page 132.)

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 `l3tl` 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>`.

```

6755 \cs_new:Npn \_clist_trim_spaces_generic:nw #1#2 ,
6756 {
6757   \_tl_trim_spaces:nn {#2}
6758   { \exp_args:No \_clist_trim_spaces_generic:nn } {#1}
6759 }
6760 \cs_new:Npn \_clist_trim_spaces_generic:nn #1#2 { #2 {#1} }

```

(End definition for `_clist_trim_spaces_generic:nw`.)

`_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.

```

6761 \cs_new:Npn \_clist_trim_spaces:n #1
6762 {
6763   \_clist_trim_spaces_generic:nw
6764   { \_clist_trim_spaces:nn { } }
6765   \q_mark #1 ,
6766   \q_recursion_tail, \q_recursion_stop
6767 }
6768 \cs_new:Npn \_clist_trim_spaces:nn #1 #2
6769 {
6770   \quark_if_recursion_tail_stop:n {#2}
6771   \tl_if_empty:nTF {#2}
6772   {
6773     \_clist_trim_spaces_generic:nw
6774     { \_clist_trim_spaces:nn {#1} } \q_mark
6775   }
6776   {
6777     #1 \exp_not:n {#2}
6778     \_clist_trim_spaces_generic:nw
6779     { \_clist_trim_spaces:nn { , } } \q_mark

```

```

6780     }
6781 }
(End definition for \_clist_trim_spaces:n.)

```

13.3 Adding data to comma lists

```

\clist_set:Nn
\clist_set:NV 6782 \cs_new_protected:Npn \clist_set:Nn #1#2
\clist_set:No 6783 { \tl_set:Nx #1 { \_clist_trim_spaces:n {#2} } }
\clist_set:Nx 6784 \cs_new_protected:Npn \clist_gset:Nn #1#2
\clist_set:cn 6785 { \tl_gset:Nx #1 { \_clist_trim_spaces:n {#2} } }
\clist_set:cV 6786 \cs_generate_variant:Nn \clist_set:Nn { NV , No , Nx , c , cV , co , cx }
\clist_set:co 6787 \cs_generate_variant:Nn \clist_gset:Nn { NV , No , Nx , c , cV , co , cx }
\clist_set:cx
(End definition for \clist_set:Nn and others. These functions are documented on page 132.)

```

Comma lists cannot hold empty values: there are therefore a couple of sanity checks to avoid accumulating commas.

```

\clist_gset:Nn
\clist_put_left:Nn
\clist_put_left:NV 6788 \cs_new_protected_nopar:Npn \clist_put_left:Nn
\clist_put_left:No 6789 { \_clist_put_left:NNNn \clist_concat:NNN \clist_set:Nn }
\clist_put_left:Nx 6790 \cs_new_protected_nopar:Npn \clist_gput_left:Nn
\clist_put_left:cn 6791 { \_clist_put_left:NNNn \clist_gconcat:NNN \clist_set:Nn }
\clist_put_left:cV 6792 \cs_new_protected:Npn \_clist_put_left:NNNn #1#2#3#4
\clist_put_left:co 6793 {
\clist_put_left:cx 6794     #2 \l__clist_internal_clist {#4}
\clist_gput_left:Nn 6795     #1 #3 \l__clist_internal_clist #3
\clist_gput_left:NV 6796 }
\clist_gput_left:No 6797 \cs_generate_variant:Nn \clist_put_left:Nn { NV , No , Nx }
\clist_gput_left:Nx 6798 \cs_generate_variant:Nn \clist_put_left:Nn { c , cV , co , cx }
\clist_gput_left:cn 6799 \cs_generate_variant:Nn \clist_gput_left:Nn { NV , No , Nx }
\clist_gput_left:cV 6800 \cs_generate_variant:Nn \clist_gput_left:Nn { c , cV , co , cx }
\clist_gput_left:co
(End definition for \clist_put_left:Nn and others. These functions are documented on page 133.)
\__clist_put_left:NNNn
\__\clist_put_right:Nn
\clist_put_right:NV 6801 \cs_new_protected_nopar:Npn \clist_put_right:Nn
\clist_put_right:No 6802 { \_clist_put_right:NNNn \clist_concat:NNN \clist_set:Nn }
\clist_put_right:Nx 6803 \cs_new_protected_nopar:Npn \clist_gput_right:Nn
\clist_put_right:cn 6804 { \_clist_put_right:NNNn \clist_gconcat:NNN \clist_set:Nn }
\clist_put_right:cV 6805 \cs_new_protected:Npn \_clist_put_right:NNNn #1#2#3#4
\clist_put_right:co 6806 {
\clist_put_right:cx 6807     #2 \l__clist_internal_clist {#4}
\clist_gput_right:Nn 6808     #1 #3 #3 \l__clist_internal_clist
\clist_gput_right:NV 6809 }
\clist_gput_right:Nx 6810 \cs_generate_variant:Nn \clist_put_right:Nn { NV , No , Nx }
\clist_gput_right:No 6811 \cs_generate_variant:Nn \clist_put_right:Nn { c , cV , co , cx }
\clist_gput_right:Nx 6812 \cs_generate_variant:Nn \clist_gput_right:Nn { NV , No , Nx }
\clist_gput_right:cn 6813 \cs_generate_variant:Nn \clist_gput_right:Nn { c , cV , co , cx }
\clist_gput_right:cV
(End definition for \clist_put_right:Nn and others. These functions are documented on page 133.)
\clist_gput_right:co
\clist_gput_right:cx
\__clist_put_right:NNNn

```

13.4 Comma lists as stacks

\clist_get:NN Getting an item from the left of a comma list is pretty easy: just trim off the first item
\clist_get:cN using the comma.
__clist_get:wN

```

6814 \cs_new_protected:Npn \clist_get:NN #1#2
6815 {
6816   \if_meaning:w #1 \c_empty_clist
6817     \tl_set:Nn #2 { \q_no_value }
6818   \else:
6819     \exp_after:wN \__clist_get:wN #1 , \q_stop #2
6820   \fi:
6821 }
6822 \cs_new_protected:Npn \__clist_get:wN #1 , #2 \q_stop #3
6823 { \tl_set:Nn #3 {#1} }
6824 \cs_generate_variant:Nn \clist_get:NN { c }

```

(End definition for `\clist_get:NN` and `\clist_get:cN`. These functions are documented on page 138.)

\clist_pop:NN An empty clist leads to `\q_no_value`, otherwise grab until the first comma and assign
\clist_pop:cN to the variable. The second argument of `__clist_pop:wwNNN` is a comma list ending
\clist_gpop:NN in a comma and `\q_mark`, unless the original clist contained exactly one item: then the
\clist_gpop:cN argument is just `\q_mark`. The next auxiliary picks either `\exp_not:n` or `\use_none:n`
__clist_pop:NNN as #2, ensuring that the result can safely be an empty comma list.
__clist_pop:wwNNN
__clist_pop:wN

```

6825 \cs_new_protected_nopar:Npn \clist_pop:NN
6826 { \__clist_pop:NNN \tl_set:Nx }
6827 \cs_new_protected_nopar:Npn \clist_gpop:NN
6828 { \__clist_pop:NNN \tl_gset:Nx }
6829 \cs_new_protected:Npn \__clist_pop:NNN #1#2#3
6830 {
6831   \if_meaning:w #2 \c_empty_clist
6832     \tl_set:Nn #3 { \q_no_value }
6833   \else:
6834     \exp_after:wN \__clist_pop:wwNNN #2 , \q_mark \q_stop #1#2#3
6835   \fi:
6836 }
6837 \cs_new_protected:Npn \__clist_pop:wwNNN #1 , #2 \q_stop #3#4#5
6838 {
6839   \tl_set:Nn #5 {#1}
6840   #3 #4
6841   {
6842     \__clist_pop:wN \prg_do_nothing:
6843       #2 \exp_not:o
6844       , \q_mark \use_none:n
6845     \q_stop
6846   }
6847 }
6848 \cs_new:Npn \__clist_pop:wN #1 , \q_mark #2 #3 \q_stop { #2 {#1} }
6849 \cs_generate_variant:Nn \clist_pop:NN { c }
6850 \cs_generate_variant:Nn \clist_gpop:NN { c }

```

(End definition for \clist_pop:NN and \clist_pop:cN. These functions are documented on page 138.)

\clist_get:NNTF The same, as branching code: very similar to the above.

```

6851 \clist_get:cNTF \prg_new_protected_conditional:Npnn \clist_get:NN #1#2 { T , F , TF }
6852 {
6853   \if_meaning:w #1 \c_empty_clist
6854     \prg_return_false:
6855   \else:
6856     \exp_after:wN \__clist_get:wN #1 , \q_stop #2
6857     \prg_return_true:
6858   \fi:
6859 }
6860 \cs_generate_variant:Nn \clist_get:NNT { c }
6861 \cs_generate_variant:Nn \clist_get:NNF { c }
6862 \cs_generate_variant:Nn \clist_get:NNTF { c }
6863 \prg_new_protected_conditional:Npnn \clist_pop:NN #1#2 { T , F , TF }
6864 { \__clist_pop_TF:NNN \tl_set:Nx #1 #2 }
6865 \prg_new_protected_conditional:Npnn \clist_gpop:NN #1#2 { T , F , TF }
6866 { \__clist_pop_TF:NNN \tl_gset:Nx #1 #2 }
6867 \cs_new_protected:Npn \__clist_pop_TF:NNN #1#2#3
6868 {
6869   \if_meaning:w #2 \c_empty_clist
6870     \prg_return_false:
6871   \else:
6872     \exp_after:wN \__clist_pop:wwNNN #2 , \q_mark \q_stop #1#2#3
6873     \prg_return_true:
6874   \fi:
6875 }
6876 \cs_generate_variant:Nn \clist_pop:NNT { c }
6877 \cs_generate_variant:Nn \clist_pop:NNF { c }
6878 \cs_generate_variant:Nn \clist_pop:NNTF { c }
6879 \cs_generate_variant:Nn \clist_gpop:NNT { c }
6880 \cs_generate_variant:Nn \clist_gpop:NNF { c }
6881 \cs_generate_variant:Nn \clist_gpop:NNTF { c }

```

(End definition for \clist_get:NNTF and \clist_get:cNTF. These functions are documented on page 138.)

\clist_push:Nn Pushing to a comma list is the same as adding on the left.

```

6882 \cs_new_eq:NN \clist_push:Nn \clist_put_left:Nn
6883 \cs_new_eq:NN \clist_push:NV \clist_put_left:NV
6884 \cs_new_eq:NN \clist_push:No \clist_put_left:No
6885 \cs_new_eq:NN \clist_push:Nx \clist_put_left:Nx
6886 \cs_new_eq:NN \clist_push:cn \clist_put_left:cn
6887 \cs_new_eq:NN \clist_push:cV \clist_put_left:cV
6888 \cs_new_eq:NN \clist_push:co \clist_put_left:co
6889 \cs_new_eq:NN \clist_push:cx \clist_put_left:cx
6890 \cs_new_eq:NN \clist_gpush:Nn \clist_gput_left:Nn
6891 \cs_new_eq:NN \clist_gpush:NV \clist_gput_left:NV
6892 \cs_new_eq:NN \clist_gpush:No \clist_gput_left:No

```

\clist_gpush:Nn
\clist_gpush:NV
\clist_gpush:No
\clist_gpush:Nx
\clist_gpush:cn
\clist_gpush:cV
\clist_gpush:co
\clist_gpush:cx

```

6893 \cs_new_eq:NN \clist_gpush:Nx \clist_gput_left:Nx
6894 \cs_new_eq:NN \clist_gpush:cn \clist_gput_left:cn
6895 \cs_new_eq:NN \clist_gpush:cV \clist_gput_left:cV
6896 \cs_new_eq:NN \clist_gpush:co \clist_gput_left:co
6897 \cs_new_eq:NN \clist_gpush:cx \clist_gput_left:cx

```

(End definition for `\clist_push:Nn` and others. These functions are documented on page 139.)

13.5 Modifying comma lists

`\l__clist_internal_remove_clist` An internal comma list for the removal routines.

```

6898 \clist_new:N \l__clist_internal_remove_clist

```

(End definition for `\l__clist_internal_remove_clist`. This variable is documented on page ??.)

`\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
6899 \cs_new_protected:Npn \clist_remove_duplicates:N
6900 { \__clist_remove_duplicates:NN \clist_set_eq:NN }
6901 \cs_new_protected:Npn \clist_gremove_duplicates:N
6902 { \__clist_remove_duplicates:NN \clist_gset_eq:NN }
6903 \cs_new_protected:Npn \__clist_remove_duplicates:NN #1#2
6904 {
6905   \clist_clear:N \l__clist_internal_remove_clist
6906   \clist_map_inline:Nn #2
6907   {
6908     \clist_if_in:NnF \l__clist_internal_remove_clist {##1}
6909     { \clist_put_right:Nn \l__clist_internal_remove_clist {##1} }
6910   }
6911   #1 #2 \l__clist_internal_remove_clist
6912 }
6913 \cs_generate_variant:Nn \clist_remove_duplicates:N { c }
6914 \cs_generate_variant:Nn \clist_gremove_duplicates:N { c }

```

(End definition for `\clist_remove_duplicates:N` and `\clist_remove_duplicates:c`. These functions are documented on page 133.)

`\clist_remove_all:Nn` The method used here is very similar to `\tl_replace_all:Nnn`. Build a function delimited by the `<item>` that should be removed, surrounded with commas, and call that function followed by the expanded comma list, and another copy of the `<item>`. 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 `<item>` 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.

```

6915 \cs_new_protected:Npn \clist_remove_all:Nn
6916 { \__clist_remove_all:NNn \tl_set:Nx }
6917 \cs_new_protected:Npn \clist_gremove_all:Nn
6918 { \__clist_remove_all:NNn \tl_gset:Nx }
6919 \cs_new_protected:Npn \__clist_remove_all:NNn #1#2#3
6920 {
6921   \cs_set:Npn \__clist_tmp:w ##1 , #3 ,
6922   {
6923     ##1
6924     , \q_mark , \use_none_delimit_by_q_stop:w ,
6925     \__clist_remove_all:
6926   }
6927   #1 #2
6928   {
6929     \exp_after:wN \__clist_remove_all:
6930     #2 , \q_mark , #3 , \q_stop
6931   }
6932   \clist_if_empty:NF #2
6933   {
6934     #1 #2
6935     {
6936       \exp_args:No \exp_not:o
6937       { \exp_after:wN \use_none:n #2 }
6938     }
6939   }
6940 }
6941 \cs_new:Npn \__clist_remove_all:
6942 { \exp_after:wN \__clist_remove_all:w \__clist_tmp:w , }
6943 \cs_new:Npn \__clist_remove_all:w #1 , \q_mark , #2 , { \exp_not:n {#1} }
6944 \cs_generate_variant:Nn \clist_remove_all:Nn { c }
6945 \cs_generate_variant:Nn \clist_gremove_all:Nn { c }

```

(End definition for `\clist_remove_all:Nn` and `\clist_remove_all:cn`. These functions are documented on page 133.)

`\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`

```

6946 \cs_new_protected:Npn \clist_reverse:N #1
6947 { \tl_set:Nx #1 { \exp_args:No \clist_reverse:n {#1} } }
6948 \cs_new_protected:Npn \clist_greverse:N #1
6949 { \tl_gset:Nx #1 { \exp_args:No \clist_reverse:n {#1} } }
6950 \cs_generate_variant:Nn \clist_reverse:N { c }
6951 \cs_generate_variant:Nn \clist_greverse:N { c }

```

(End definition for `\clist_reverse:N` and others. These functions are documented on page 134.)

`\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 “ $\langle item \rangle$,”. We start from a comma list “ $\langle item_1 \rangle, \dots, \langle item_n \rangle$ ”. During the loop, the auxiliary `__clist_reverse:wwNww` receives “ $\langle item_i \rangle$ ” as #1, “ $\langle item_{i+1} \rangle, \dots, \langle item_n \rangle$ ” as #2, `__clist_reverse:wwNww` as #3, what remains until `\q_stop` as #4, and “ $\langle item_{i-1} \rangle, \dots, \langle item_1 \rangle$,” 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`.

```

6952 \cs_new:Npn \clist_reverse:n #1
6953 {
6954   \__clist_reverse:wwNww ? #1 ,
6955   \q_mark \__clist_reverse:wwNww ! ,
6956   \q_mark \__clist_reverse_end:ww
6957   \q_stop ? \q_mark
6958 }
6959 \cs_new:Npn \__clist_reverse:wwNww
6960   #1 , #2 \q_mark #3 #4 \q_stop ? #5 \q_mark
6961   { #3 ? #2 \q_mark #3 #4 \q_stop #1 , #5 \q_mark }
6962 \cs_new:Npn \__clist_reverse_end:ww #1 ! #2 , \q_mark
6963   { \exp_not:o { \use_none:n #2 } }

```

(End definition for `\clist_reverse:n`. This function is documented on page 134.)

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

```

```

6964 \prg_new_eq_conditional:NNn \clist_if_empty:N \tl_if_empty:N
6965   { p , T , F , TF }
6966 \prg_new_eq_conditional:NNn \clist_if_empty:c \tl_if_empty:c
6967   { p , T , F , TF }

```

(End definition for `\clist_if_empty:NTF` and `\clist_if_empty:cTF`. These functions are documented on page 134.)

`\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:oTF` 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.

```

6968 \prg_new_conditional:Npnn \clist_if_empty:n #1 { p , T , F , TF }
6969 {
6970   \__clist_if_empty_n:w ? #1

```

```

6971     , \q_mark \prg_return_false:
6972     , \q_mark \prg_return_true:
6973     \q_stop
6974   }
6975   \cs_new:Npn \__clist_if_empty_n:w #1 ,
6976   {
6977     \tl_if_empty:oTF { \use_none:nn #1 ? }
6978     { \__clist_if_empty_n:w ? }
6979     { \__clist_if_empty_n:wNw }
6980   }
6981   \cs_new:Npn \__clist_if_empty_n:wNw #1 \q_mark #2#3 \q_stop {#2}

```

(End definition for \clist_if_empty:nTF. This function is documented on page 134.)

```

\clist_if_in:NnTF See description of the \tl_if_in:Nn function for details. We simply surround the comma
\clist_if_in:NvTF list, and the item, with commas.
\clist_if_in:NoTF
\clist_if_in:cnTF 6982 \prg_new_protected_conditional:Npnn \clist_if_in:Nn #1#2 { T , F , TF }
\clist_if_in:cVTF 6983 {
\clist_if_in:coTF 6984   \exp_args:No \__clist_if_in_return:nn #1 {#2}
\clist_if_in:nnTF 6985 }
\clist_if_in:nvTF 6986 \prg_new_protected_conditional:Npnn \clist_if_in:nn #1#2 { T , F , TF }
\clist_if_in:noTF 6987 {
\__clist_if_in_return:nn 6988   \clist_set:Nn \l__clist_internal_clist {#1}
6989   \exp_args:No \__clist_if_in_return:nn \l__clist_internal_clist {#2}
6990 }
6991 \cs_new_protected:Npn \__clist_if_in_return:nn #1#2
6992 {
6993   \cs_set:Npn \__clist_tmp:w ##1 ,#2, { }
6994   \tl_if_empty:oTF
6995   { \__clist_tmp:w ,#1, {} {} ,#2, }
6996   { \prg_return_false: } { \prg_return_true: }
6997 }
6998 \cs_generate_variant:Nn \clist_if_in:NnT { NV , No }
6999 \cs_generate_variant:Nn \clist_if_in:NnT { c , cV , co }
7000 \cs_generate_variant:Nn \clist_if_in:NnF { NV , No }
7001 \cs_generate_variant:Nn \clist_if_in:NnF { c , cV , co }
7002 \cs_generate_variant:Nn \clist_if_in:NnTF { NV , No }
7003 \cs_generate_variant:Nn \clist_if_in:NnTF { c , cV , co }
7004 \cs_generate_variant:Nn \clist_if_in:nnT { nV , no }
7005 \cs_generate_variant:Nn \clist_if_in:nnF { nV , no }
7006 \cs_generate_variant:Nn \clist_if_in:nnTF { nV , no }

```

(End definition for \clist_if_in:NnTF and others. These functions are documented on page 134.)

13.7 Mapping to comma lists

```

\clist_map_function:NN If the variable is empty, the mapping is skipped (otherwise, that comma-list would be
\clist_map_function:cN seen as consisting of one empty item). Then loop over the comma-list, grabbing one
\__clist_map_function:Nw

```

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.

```

7007 \cs_new:Npn \clist_map_function:NN #1#2
7008 {
7009   \clist_if_empty:NF #1
7010   {
7011     \exp_last_unbraced:NNo \_clist_map_function:Nw #2 #1
7012     , \q_recursion_tail ,
7013     \_prg_break_point:Nn \clist_map_break: { }
7014   }
7015 }
7016 \cs_new:Npn \_clist_map_function:Nw #1#2 ,
7017 {
7018   \_quark_if_recursion_tail_break:nN {#2} \clist_map_break:
7019   #1 {#2}
7020   \_clist_map_function:Nw #1
7021 }
7022 \cs_generate_variant:Nn \clist_map_function:NN { c }

```

(End definition for `\clist_map_function:NN` and `\clist_map_function:cN`. These functions are documented on page 135.)

`\clist_map_function:nN` 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`. `_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`.

```

7023 \cs_new:Npn \clist_map_function:nN #1#2
7024 {
7025   \_clist_trim_spaces_generic:nw { \_clist_map_function_n:Nn #2 }
7026   \q_mark #1, \q_recursion_tail,
7027   \_prg_break_point:Nn \clist_map_break: { }
7028 }
7029 \cs_new:Npn \_clist_map_function_n:Nn #1 #2
7030 {
7031   \_quark_if_recursion_tail_break:nN {#2} \clist_map_break:
7032   \tl_if_empty:nF {#2} { \_clist_map_unbrace:Nw #1 #2, }
7033   \_clist_trim_spaces_generic:nw { \_clist_map_function_n:Nn #1 }
7034   \q_mark
7035 }
7036 \cs_new:Npn \_clist_map_unbrace:Nw #1 #2, { #1 {#2} }

```

(End definition for `\clist_map_function:nN`. This function is documented on page 135.)

`\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 \TeX ’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.

```

7037 \cs_new_protected:Npn \clist_map_inline:Nn #1#2
7038 {
7039   \clist_if_empty:NF #1
7040   {
7041     \int_gincr:N \g__prg_map_int
7042     \cs_gset:cpn { __prg_map_ \int_use:N \g__prg_map_int :w } ##1 {#2}
7043     \exp_last_unbraced:Nco \__clist_map_function:Nw
7044     { __prg_map_ \int_use:N \g__prg_map_int :w }
7045     #1 , \q_recursion_tail ,
7046     \__prg_break_point:Nn \clist_map_break:
7047     { \int_gdecr:N \g__prg_map_int }
7048   }
7049 }
7050 \cs_new_protected:Npn \clist_map_inline:nn #1
7051 {
7052   \clist_set:Nn \l__clist_internal_clist {#1}
7053   \clist_map_inline:Nn \l__clist_internal_clist
7054 }
7055 \cs_generate_variant:Nn \clist_map_inline:Nn { c }

```

(End definition for `\clist_map_inline:Nn` and `\clist_map_inline:cn`. These functions are documented on page 135.)

`\clist_map_variable:NNn` As for other comma-list mappings, filter out the case of an empty list. Same approach
`\clist_map_variable:cNn` as `\clist_map_function:Nn`, additionally we store each item in the given variable. As
`\clist_map_variable:nNn` for inline mappings, space trimming for the `n` variant is done by storing the comma list
`__clist_map_variable:Nnw` in a variable.

```

7056 \cs_new_protected:Npn \clist_map_variable:NNn #1#2#3
7057 {
7058   \clist_if_empty:NF #1
7059   {
7060     \exp_args:Nno \use:nn
7061     { \__clist_map_variable:Nnw #2 {#3} }
7062     #1
7063     , \q_recursion_tail , \q_recursion_stop
7064     \__prg_break_point:Nn \clist_map_break: { }
7065   }
7066 }
7067 \cs_new_protected:Npn \clist_map_variable:nNn #1
7068 {
7069   \clist_set:Nn \l__clist_internal_clist {#1}
7070   \clist_map_variable:NNn \l__clist_internal_clist
7071 }
7072 \cs_new_protected:Npn \__clist_map_variable:Nnw #1#2#3,
7073 {
7074   \tl_set:Nn #1 {#3}

```

```

7075 \quark_if_recursion_tail_stop:N #1
7076 \use:n {#2}
7077 \__clist_map_variable:Nnw #1 {#2}
7078 }
7079 \cs_generate_variant:Nn \clist_map_variable:NNn { c }

```

(End definition for `\clist_map_variable:NNn` and `\clist_map_variable:cNn`. These functions are documented on page 135.)

`\clist_map_break:` The break statements use the general `__prg_map_break:Nn` mechanism.
`\clist_map_break:n`

```

7080 \cs_new_nopar:Npn \clist_map_break:
7081 { \__prg_map_break:Nn \clist_map_break: { } }
7082 \cs_new_nopar:Npn \clist_map_break:n
7083 { \__prg_map_break:Nn \clist_map_break: }

```

(End definition for `\clist_map_break:` and `\clist_map_break:n`. These functions are documented on page 136.)

`\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).

```

7084 \cs_new:Npn \clist_count:N #1
7085 {
7086   \int_eval:n
7087   {
7088     0
7089     \clist_map_function:NN #1 \__clist_count:n
7090   }
7091 }
7092 \cs_generate_variant:Nn \clist_count:N { c }
7093 \cs_new:Npx \clist_count:n #1
7094 {
7095   \exp_not:N \int_eval:n
7096   {
7097     0
7098     \exp_not:N \__clist_count:w \c_space_tl
7099     #1 \exp_not:n { , \q_recursion_tail , \q_recursion_stop }
7100   }
7101 }
7102 \cs_new:Npn \__clist_count:n #1 { + \c_one }
7103 \cs_new:Npx \__clist_count:w #1 ,
7104 {
7105   \exp_not:n { \exp_args:Nf \quark_if_recursion_tail_stop:n } {#1}
7106   \exp_not:N \tl_if_blank:nF {#1} { + \c_one }
7107   \exp_not:N \__clist_count:w \c_space_tl
7108 }

```

(End definition for `\clist_count:N`, `\clist_count:c`, and `\clist_count:n`. These functions are documented on page 136.)

13.8 Using comma lists

```

\clist_use:Nnnn
\clist_use:cnnn
\__clist_use:wwn
\__clist_use:nwwwnwn
\__clist_use:nwnn
\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>*.

```

7109 \cs_new:Npn \clist_use:Nnnn #1#2#3#4
7110 {
7111   \clist_if_exist:NTF #1
7112   {
7113     \int_case:nnF { \clist_count:N #1 }
7114     {
7115       { 0 } { }
7116       { 1 } { \exp_after:wN \__clist_use:wwn #1 , , { } }
7117       { 2 } { \exp_after:wN \__clist_use:wwn #1 , {#2} }
7118     }
7119     {
7120       \exp_after:wN \__clist_use:nwwwnwn
7121       \exp_after:wN { \exp_after:wN } #1 ,
7122       \q_mark , { \__clist_use:nwwwnwn {#3} }
7123       \q_mark , { \__clist_use:nwnn {#4} }
7124       \q_stop { }
7125     }
7126   }
7127   {
7128     \_msg_kernel_expandable_error:nnn
7129     { kernel } { bad-variable } {#1}
7130   }
7131 }
7132 \cs_generate_variant:Nn \clist_use:Nnnn { c }
7133 \cs_new:Npn \__clist_use:wwn #1 , #2 , #3 { \exp_not:n { #1 #3 #2 } }
7134 \cs_new:Npn \__clist_use:nwwwnwn
7135   #1#2 , #3 , #4 , #5 \q_mark , #6#7 \q_stop #8
7136   { #6 {#3} , {#4} , #5 \q_mark , {#6} #7 \q_stop { #8 #1 #2 } }
7137 \cs_new:Npn \__clist_use:nwnn #1#2 , #3 \q_stop #4
7138   { \exp_not:n { #4 #1 #2 } }
7139 \cs_new:Npn \clist_use:Nn #1#2

```

```

7140 { \clist_use:Nnnn #1 {#2} {#2} {#2} }
7141 \cs_generate_variant:Nn \clist_use:Nn { c }

```

(End definition for `\clist_use:Nnnn` and `\clist_use:cnnn`. These functions are documented on page 137.)

13.9 Using a single item

`\clist_item:Nn` To avoid needing to test the end of the list at each step, we first compute the $\langle length \rangle$ of the list. If the item number is 0, less than $-\langle length \rangle$, or more than $\langle length \rangle$, the result is empty. If it is negative, but not less than $-\langle length \rangle$, add $\langle length \rangle + 1$ to the item number before performing the loop. The loop itself is very simple, return the item if the counter reached 1, otherwise, decrease the counter and repeat.

```

7142 \cs_new:Npn \clist_item:Nn #1#2
7143 {
7144   \exp_args:Nfo \__clist_item:nnNn
7145   { \clist_count:N #1 }
7146   #1
7147   \__clist_item_N_loop:nw
7148   {#2}
7149 }
7150 \cs_new:Npn \__clist_item:nnNn #1#2#3#4
7151 {
7152   \int_compare:nNnTF {#4} < \c_zero
7153   {
7154     \int_compare:nNnTF {#4} < { - #1 }
7155     { \use_none_delimit_by_q_stop:w }
7156     { \exp_args:Nf #3 { \int_eval:n { #4 + \c_one + #1 } } }
7157   }
7158   {
7159     \int_compare:nNnTF {#4} > {#1}
7160     { \use_none_delimit_by_q_stop:w }
7161     { #3 {#4} }
7162   }
7163   { } , #2 , \q_stop
7164 }
7165 \cs_new:Npn \__clist_item_N_loop:nw #1 #2,
7166 {
7167   \int_compare:nNnTF {#1} = \c_zero
7168   { \use_i_delimit_by_q_stop:nw { \exp_not:n {#2} } }
7169   { \exp_args:Nf \__clist_item_N_loop:nw { \int_eval:n { #1 - 1 } } }
7170 }
7171 \cs_generate_variant:Nn \clist_item:Nn { c }

```

(End definition for `\clist_item:Nn` and `\clist_item:cn`. These functions are documented on page 139.)

`\clist_item:nn` This starts in the same way as `\clist_item:Nn` by counting the items of the comma list. The final item should be space-trimmed before being brace-stripped, hence we insert a couple of odd-looking `\prg_do_nothing:` to avoid losing braces. Blank items are ignored.

```

7172 \cs_new:Npn \clist_item:nn #1#2
7173 {
7174   \exp_args:Nf \__clist_item:nnNn
7175     { \clist_count:n {#1} }
7176     {#1}
7177     \__clist_item_n:nw
7178     {#2}
7179 }
7180 \cs_new:Npn \__clist_item_n:nw #1
7181 { \__clist_item_n_loop:nw {#1} \prg_do_nothing: }
7182 \cs_new:Npn \__clist_item_n_loop:nw #1 #2,
7183 {
7184   \exp_args:No \tl_if_blank:nTF {#2}
7185     { \__clist_item_n_loop:nw {#1} \prg_do_nothing: }
7186     {
7187       \int_compare:nNnTF {#1} = \c_zero
7188         { \exp_args:No \__clist_item_n_end:n {#2} }
7189         {
7190           \exp_args:Nf \__clist_item_n_loop:nw
7191             { \int_eval:n { #1 - 1 } }
7192           \prg_do_nothing:
7193         }
7194     }
7195 }
7196 \cs_new:Npn \__clist_item_n_end:n #1 #2 \q_stop
7197 {
7198   \__tl_trim_spaces:nn { \q_mark #1 }
7199   { \exp_last_unbraced:No \__clist_item_n_strip:w } ,
7200 }
7201 \cs_new:Npn \__clist_item_n_strip:w #1 , { \exp_not:n {#1} }

```

(End definition for `\clist_item:nn`. This function is documented on page [139](#).)

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).

`\clist_show:c`

`\clist_show:n`

```

7202 \cs_new_protected:Npn \clist_show:N #1
7203 {
7204   \__msg_show_variable:NNNnn #1
7205   \clist_if_exist:NTF \clist_if_empty:NTF { clist }
7206   { \clist_map_function:NN #1 \__msg_show_item:n }
7207 }
7208 \cs_new_protected:Npn \clist_show:n #1
7209 {
7210   \__msg_show_pre:nnxxxx { LaTeX / kernel } { show-clist }
7211   { } { \clist_if_empty:nF {#1} { ? } } { } { }
7212   \__msg_show_wrap:n

```

```

7213     { \clist_map_function:nN {#1} \_msg_show_item:n }
7214   }
7215   \cs_generate_variant:Nn \clist_show:N { c }

```

(End definition for `\clist_show:N` and `\clist_show:c`. These functions are documented on page 139.)

13.11 Scratch comma lists

`\l_tmpa_clist` Temporary comma list variables.
`\l_tmpb_clist`
`\g_tmpa_clist`
`\g_tmpb_clist`

```

7216 \clist_new:N \l_tmpa_clist
7217 \clist_new:N \l_tmpb_clist
7218 \clist_new:N \g_tmpa_clist
7219 \clist_new:N \g_tmpb_clist

```

(End definition for `\l_tmpa_clist` and `\l_tmpb_clist`. These variables are documented on page 139.)

```

7220 \</initex | package>

```

14 l3prop implementation

The following test files are used for this code: `m3prop001`, `m3prop002`, `m3prop003`, `m3prop004`, `m3show001`.

```

7221 \*initex | package>
7222 \@@=prop>

```

A property list is a macro whose top-level expansion is of the form

```

\s__prop \__prop_pair:wn <key1> \s__prop {<value1>}
...
\__prop_pair:wn <keyn> \s__prop {<valuen>}

```

where `\s__prop` is a scan mark (equal to `\scan_stop:`), and `__prop_pair:wn` can be used to map through the property list.

`\s__prop` A private scan mark is used as a marker after each key, and at the very beginning of the property list.

```

7223 \__scan_new:N \s__prop

```

(End definition for `\s__prop`.)

`__prop_pair:wn` The delimiter is always defined, but when misused simply triggers an error and removes its argument.

```

7224 \cs_new:Npn \__prop_pair:wn #1 \s__prop #2
7225 { \_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.

```

7226 \tl_new:N \l__prop_internal_tl

```

(End definition for `\l__prop_internal_tl`. This variable is documented on page 146.)

`\c_empty_prop` An empty prop.

```
7227 \tl_const:Nn \c_empty_prop { \s__prop }
```

(End definition for `\c_empty_prop`. This variable is documented on page 146.)

14.1 Allocation and initialisation

`\prop_new:N` Property lists are initialized with the value `\c_empty_prop`.

```
\prop_new:c 7228 \cs_new_protected:Npn \prop_new:N #1
7229 {
7230   \__chk_if_free_cs:N #1
7231   \cs_gset_eq:NN #1 \c_empty_prop
7232 }
7233 \cs_generate_variant:Nn \prop_new:N { c }
```

(End definition for `\prop_new:N` and `\prop_new:c`. These functions are documented on page 141.)

`\prop_clear:N` The same idea for clearing.

```
\prop_clear:c 7234 \cs_new_protected:Npn \prop_clear:N #1
\prop_gclear:N 7235 { \prop_set_eq:NN #1 \c_empty_prop }
\prop_gclear:c 7236 \cs_generate_variant:Nn \prop_clear:N { c }
7237 \cs_new_protected:Npn \prop_gclear:N #1
7238 { \prop_gset_eq:NN #1 \c_empty_prop }
7239 \cs_generate_variant:Nn \prop_gclear:N { c }
```

(End definition for `\prop_clear:N` and `\prop_clear:c`. These functions are documented on page 141.)

`\prop_clear_new:N` Once again a simple variation of the token list functions.

```
\prop_clear_new:c 7240 \cs_new_protected:Npn \prop_clear_new:N #1
\prop_gclear_new:N 7241 { \prop_if_exist:NTF #1 { \prop_clear:N #1 } { \prop_new:N #1 } }
\prop_gclear_new:c 7242 \cs_generate_variant:Nn \prop_clear_new:N { c }
7243 \cs_new_protected:Npn \prop_gclear_new:N #1
7244 { \prop_if_exist:NTF #1 { \prop_gclear:N #1 } { \prop_new:N #1 } }
7245 \cs_generate_variant:Nn \prop_gclear_new:N { c }
```

(End definition for `\prop_clear_new:N` and `\prop_clear_new:c`. These functions are documented on page 141.)

`\prop_set_eq:NN` These are simply copies from the token list functions.

```
\prop_set_eq:cN 7246 \cs_new_eq:NN \prop_set_eq:NN \tl_set_eq:NN
\prop_set_eq:Nc 7247 \cs_new_eq:NN \prop_set_eq:Nc \tl_set_eq:Nc
\prop_set_eq:cc 7248 \cs_new_eq:NN \prop_set_eq:cN \tl_set_eq:cN
\prop_gset_eq:NN 7249 \cs_new_eq:NN \prop_set_eq:cc \tl_set_eq:cc
\prop_gset_eq:cN 7250 \cs_new_eq:NN \prop_gset_eq:NN \tl_gset_eq:NN
\prop_gset_eq:Nc 7251 \cs_new_eq:NN \prop_gset_eq:Nc \tl_gset_eq:Nc
\prop_gset_eq:cN 7252 \cs_new_eq:NN \prop_gset_eq:cN \tl_gset_eq:cN
7253 \cs_new_eq:NN \prop_gset_eq:cc \tl_gset_eq:cc
```

(End definition for `\prop_set_eq:NN` and others. These functions are documented on page 141.)


```

7273     { }
7274 }
7275 \cs_new_protected:Npn \prop_gremove:Nn #1#2
7276 {
7277     \__prop_split:NnTF #1 {#2}
7278     { \tl_gset:Nn #1 { ##1 ##3 } }
7279     { }
7280 }
7281 \cs_generate_variant:Nn \prop_remove:Nn { NV }
7282 \cs_generate_variant:Nn \prop_remove:Nn { c , cV }
7283 \cs_generate_variant:Nn \prop_gremove:Nn { NV }
7284 \cs_generate_variant:Nn \prop_gremove:Nn { c , cV }

```

(End definition for `\prop_remove:Nn` and others. These functions are documented on page 143.)

\prop_get:NnN Getting an item from a list is very easy: after splitting, if the key is in the property list, just set the token list variable to the return value, otherwise to `\q_no_value`.

```

\prop_get:NVN
\prop_get:NoN
\prop_get:cnN
\prop_get:cVN
\prop_get:coN
7285 \cs_new_protected:Npn \prop_get:NnN #1#2#3
7286 {
7287     \__prop_split:NnTF #1 {#2}
7288     { \tl_set:Nn #3 {##2} }
7289     { \tl_set:Nn #3 { \q_no_value } }
7290 }
7291 \cs_generate_variant:Nn \prop_get:NnN { NV , No }
7292 \cs_generate_variant:Nn \prop_get:NnN { c , cV , co }

```

(End definition for `\prop_get:NnN` and others. These functions are documented on page 142.)

\prop_pop:NnN Popping a value also starts by doing the split. If the key is present, save the value in the token list and update the property list as when deleting. If the key is missing, save `\q_no_value` in the token list.

```

\prop_pop:NoN
\prop_pop:cnN
\prop_pop:coN
7293 \cs_new_protected:Npn \prop_pop:NnN #1#2#3
7294 {
7295     \__prop_split:NnTF #1 {#2}
7296     {
7297         \tl_set:Nn #3 {##2}
7298         \tl_set:Nn #1 { ##1 ##3 }
7299     }
7300     { \tl_set:Nn #3 { \q_no_value } }
7301 }
7302 \cs_new_protected:Npn \prop_gpop:NnN #1#2#3
7303 {
7304     \__prop_split:NnTF #1 {#2}
7305     {
7306         \tl_set:Nn #3 {##2}
7307         \tl_gset:Nn #1 { ##1 ##3 }
7308     }
7309     { \tl_set:Nn #3 { \q_no_value } }
7310 }
7311 \cs_generate_variant:Nn \prop_pop:NnN { No }

```

```

7312 \cs_generate_variant:Nn \prop_pop:NnN { c , co }
7313 \cs_generate_variant:Nn \prop_gpop:NnN { No }
7314 \cs_generate_variant:Nn \prop_gpop:NnN { c , co }

```

(End definition for `\prop_pop:NnN` and others. These functions are documented on page 142.)

`\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.

```

7315 \cs_new:Npn \prop_item:Nn #1#2
7316 {
7317   \exp_last_unbraced:Noo \__prop_item_Nn:nwn { \tl_to_str:n {#2} } #1
7318   \__prop_pair:wn \tl_to_str:n {#2} \s__prop { }
7319   \__prg_break_point:
7320 }
7321 \cs_new:Npn \__prop_item_Nn:nwn #1#2 \__prop_pair:wn #3 \s__prop #4
7322 {
7323   \str_if_eq_x:nnTF {#1} {#3}
7324   { \__prg_break:n { \exp_not:n {#4} } }
7325   { \__prop_item_Nn:nwn {#1} }
7326 }
7327 \cs_generate_variant:Nn \prop_item:Nn { c }

```

(End definition for `\prop_item:Nn` and `\prop_item:cn`. These functions are documented on page 143.)

`\prop_pop:NnN` 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_pop:cnN`

`\prop_gpop:NnN`

`\prop_gpop:cnN`

```

7328 \prg_new_protected_conditional:Npnn \prop_pop:NnN #1#2#3 { T , F , TF }
7329 {
7330   \__prop_split:NnTF #1 {#2}
7331   {
7332     \tl_set:Nn #3 {##2}
7333     \tl_set:Nn #1 { ##1 ##3 }
7334     \prg_return_true:
7335   }
7336   { \prg_return_false: }
7337 }
7338 \prg_new_protected_conditional:Npnn \prop_gpop:NnN #1#2#3 { T , F , TF }
7339 {
7340   \__prop_split:NnTF #1 {#2}
7341   {
7342     \tl_set:Nn #3 {##2}
7343     \tl_gset:Nn #1 { ##1 ##3 }
7344     \prg_return_true:

```

```

7345     }
7346     { \prg_return_false: }
7347 }
7348 \cs_generate_variant:Nn \prop_pop:NnNT { c }
7349 \cs_generate_variant:Nn \prop_pop:NnNF { c }
7350 \cs_generate_variant:Nn \prop_pop:NnNTF { c }
7351 \cs_generate_variant:Nn \prop_gpop:NnNT { c }
7352 \cs_generate_variant:Nn \prop_gpop:NnNF { c }
7353 \cs_generate_variant:Nn \prop_gpop:NnNTF { c }

```

(End definition for `\prop_pop:NnNTF` and others. These functions are documented on page 144.)

`\prop_put:Nnn` Since the branches of `__prop_split:NnTF` are used as the replacement text of an internal macro, and since the *key* and new *value* may contain arbitrary tokens, it is not safe to include them in the argument of `__prop_split:NnTF`. We thus start by storing in `\l__prop_internal_tl` tokens which (after x-expansion) encode the key–value pair. This variable can safely be used in `__prop_split:NnTF`. If the *key* was absent, append the new key–value to the list. Otherwise concatenate the extracts `##1` and `##3` with the new key–value pair `\l__prop_internal_tl`. The updated entry is placed at the same spot as the original *key* in the property list, preserving the order of entries.

```

7354 \cs_new_protected_nopar:Npn \prop_put:Nnn { \__prop_put:NNnn \tl_set:Nx }
7355 \cs_new_protected_nopar:Npn \prop_gput:Nnn { \__prop_put:NNnn \tl_gset:Nx }
7356 \cs_new_protected:Npn \__prop_put:NNnn #1#2#3#4
7357 {
7358   \tl_set:Nn \l__prop_internal_tl
7359   {
7360     \exp_not:N \__prop_pair:wn \tl_to_str:n {#3}
7361     \s__prop { \exp_not:n {#4} }
7362   }
7363   \__prop_split:NnTF #2 {#3}
7364   { #1 #2 { \exp_not:n {##1} \l__prop_internal_tl \exp_not:n {##3} } }
7365   { #1 #2 { \exp_not:o {#2} \l__prop_internal_tl } }
7366 }
7367 \cs_generate_variant:Nn \prop_put:Nnn
7368 { NnV , Nno , Nnx , NV , NVV , No , Noo }
7369 \cs_generate_variant:Nn \prop_put:Nnn
7370 { c , cnV , cno , cnx , cV , cVV , co , coo }
7371 \cs_generate_variant:Nn \prop_gput:Nnn
7372 { NnV , Nno , Nnx , NV , NVV , No , Noo }
7373 \cs_generate_variant:Nn \prop_gput:Nnn
7374 { c , cnV , cno , cnx , cV , cVV , co , coo }

```

(End definition for `\prop_put:Nnn` and others. These functions are documented on page 142.)

`\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`.

```

7375 \cs_new_protected_nopar:Npn \prop_put_if_new:Nnn
7376 { \__prop_put_if_new:NNnn \tl_set:Nx }

```

```

7377 \cs_new_protected_nopar:Npn \prop_gput_if_new:Nnn
7378 { \__prop_put_if_new:NNnn \tl_gset:Nx }
7379 \cs_new_protected:Npn \__prop_put_if_new:NNnn #1#2#3#4
7380 {
7381   \tl_set:Nn \l__prop_internal_tl
7382   {
7383     \exp_not:N \__prop_pair:wn \tl_to_str:n {#3}
7384     \s__prop \exp_not:n { {#4} }
7385   }
7386   \__prop_split:NnTF #2 {#3}
7387   { }
7388   { #1 #2 { \exp_not:o {#2} \l__prop_internal_tl } }
7389 }
7390 \cs_generate_variant:Nn \prop_put_if_new:Nnn { c }
7391 \cs_generate_variant:Nn \prop_gput_if_new:Nnn { c }

```

(End definition for `\prop_put_if_new:Nnn` and `\prop_gput_if_new:cnn`. These functions are documented on page 142.)

14.3 Property list conditionals

`\prop_if_exist_p:N` Copies of the cs functions defined in `l3basics`.

```

\prop_if_exist_p:c 7392 \prg_new_eq_conditional:NNn \prop_if_exist:N \cs_if_exist:N
\prop_if_exist:NTF 7393 { TF , T , F , p }
\prop_if_exist:cTF 7394 \prg_new_eq_conditional:NNn \prop_if_exist:c \cs_if_exist:c
7395 { TF , T , F , p }

```

(End definition for `\prop_if_exist:NTF` and `\prop_if_exist:cTF`. These functions are documented on page 143.)

`\prop_if_empty_p:N` Same test as for token lists.

```

\prop_if_empty_p:c 7396 \prg_new_conditional:Npnn \prop_if_empty:N #1 { p , T , F , TF }
\prop_if_empty:NTF 7397 {
\prop_if_empty:cTF 7398   \tl_if_eq:NNTF #1 \c_empty_prop
7399   \prg_return_true: \prg_return_false:
7400 }
7401 \cs_generate_variant:Nn \prop_if_empty_p:N { c }
7402 \cs_generate_variant:Nn \prop_if_empty:NT { c }
7403 \cs_generate_variant:Nn \prop_if_empty:NF { c }
7404 \cs_generate_variant:Nn \prop_if_empty:NTF { c }

```

(End definition for `\prop_if_empty:NTF` and `\prop_if_empty:cTF`. These functions are documented on page 143.)

`\prop_if_in_p:Nn` Testing expandably if a key is in a property list requires to go through the key-value pairs one by one. This is rather slow, and a faster test would be

```

\prop_if_in_p:Nv \prop_if_in_p:No \prop_if_in_p:cn \prop_if_in_p:cV \prop_if_in_p:co
\prop_if_in:NnTF \prg_new_protected_conditional:Npnn \prop_if_in:Nn #1 #2
{
  \@@_split:NnTF #1 {#2}
  { \prg_return_true: }

```

```

\prop_if_in:NvTF
\prop_if_in:NoTF
\prop_if_in:cnTF
\prop_if_in:cVTF
\prop_if_in:coTF
\__prop_if_in:nwnn
\__prop_if_in:N

```

```

    { \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:nwn` is most often empty. When the $\langle key \rangle$ 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.

```

7405 \prg_new_conditional:Npnn \prop_if_in:Nn #1#2 { p , T , F , TF }
7406 {
7407   \exp_last_unbraced:Noo \__prop_if_in:nwn { \tl_to_str:n {#2} } #1
7408   \__prop_pair:wn \tl_to_str:n {#2} \s__prop { }
7409   \q_recursion_tail
7410   \__prg_break_point:
7411 }
7412 \cs_new:Npn \__prop_if_in:nwn #1#2 \__prop_pair:wn #3 \s__prop #4
7413 {
7414   \str_if_eq_x:nnTF {#1} {#3}
7415   { \__prop_if_in:N }
7416   { \__prop_if_in:nwn {#1} }
7417 }
7418 \cs_new:Npn \__prop_if_in:N #1
7419 {
7420   \if_meaning:w \q_recursion_tail #1
7421   \prg_return_false:
7422   \else:
7423     \prg_return_true:
7424   \fi:
7425   \__prg_break:
7426 }
7427 \cs_generate_variant:Nn \prop_if_in_p:Nn { NV , No }
7428 \cs_generate_variant:Nn \prop_if_in_p:Nn { c , cV , co }
7429 \cs_generate_variant:Nn \prop_if_in:NnT { NV , No }
7430 \cs_generate_variant:Nn \prop_if_in:NnT { c , cV , co }
7431 \cs_generate_variant:Nn \prop_if_in:NnF { NV , No }
7432 \cs_generate_variant:Nn \prop_if_in:NnF { c , cV , co }
7433 \cs_generate_variant:Nn \prop_if_in:NnTF { NV , No }
7434 \cs_generate_variant:Nn \prop_if_in:NnTF { c , cV , co }

```

(End definition for `\prop_if_in:NnTF` and others. These functions are documented on page 143.)

14.4 Recovering values from property lists with branching

`\prop_get:NnTF` Getting the value corresponding to a key, keeping track of whether the key was present or not, is implemented as a conditional (with side effects). If the key was absent, the token list is not altered.

```

\prop_get:NVNTF
\prop_get:NoNTF
\prop_get:cnNTF
\prop_get:cVNTF
\prop_get:coNTF
7435 \prg_new_protected_conditional:Npnn \prop_get:NnN #1#2#3 { T , F , TF }
7436 {
7437     \__prop_split:NnTF #1 {#2}
7438     {
7439         \tl_set:Nn #3 {##2}
7440         \prg_return_true:
7441     }
7442     { \prg_return_false: }
7443 }
7444 \cs_generate_variant:Nn \prop_get:NnNT { NV , No }
7445 \cs_generate_variant:Nn \prop_get:NnNF { NV , No }
7446 \cs_generate_variant:Nn \prop_get:NnNTF { NV , No }
7447 \cs_generate_variant:Nn \prop_get:NnNT { c , cV , co }
7448 \cs_generate_variant:Nn \prop_get:NnNF { c , cV , co }
7449 \cs_generate_variant:Nn \prop_get:NnNTF { c , cV , co }

```

(End definition for `\prop_get:NnNTF` and others. These functions are documented on page 144.)

14.5 Mapping to property lists

`\prop_map_function:NN` 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`.

```

\__prop_map_function:Nwwn
7450 \cs_new:Npn \prop_map_function:NN #1#2
7451 {
7452     \exp_last_unbraced:NNo \__prop_map_function:Nwwn #2 #1
7453     \__prop_pair:wn \q_recursion_tail \s__prop { }
7454     \__prg_break_point:Nn \prop_map_break: { }
7455 }
7456 \cs_new:Npn \__prop_map_function:Nwwn #1#2 \__prop_pair:wn #3 \s__prop #4
7457 {
7458     \if_meaning:w \q_recursion_tail #3
7459     \exp_after:wN \prop_map_break:
7460     \fi:
7461     #1 {#3} {#4}
7462     \__prop_map_function:Nwwn #1
7463 }
7464 \cs_generate_variant:Nn \prop_map_function:NN { Nc }
7465 \cs_generate_variant:Nn \prop_map_function:NN { c , cc }

```

(End definition for `\prop_map_function:NN` and others. These functions are documented on page 144.)

`\prop_map_inline:Nn` Mapping in line requires a nesting level counter. Store the current definition of `__prop_pair:wn`, and define it anew. At the end of the loop, revert to the earlier definition. Note

that besides pairs of the form `__prop_pair:wn <key> \s__prop {<value>}`, there are a leading and a trailing tokens, but both are equal to `\scan_stop:`, hence have no effect in such inline mapping.

```

7466 \cs_new_protected:Npn \prop_map_inline:Nn #1#2
7467 {
7468   \cs_gset_eq:cN
7469   { __prg_map_ \int_use:N \g__prg_map_int :wn } \__prop_pair:wn
7470   \int_gincr:N \g__prg_map_int
7471   \cs_gset:Npn \__prop_pair:wn ##1 \s__prop ##2 {#2}
7472   #1
7473   \__prg_break_point:Nn \prop_map_break:
7474   {
7475     \int_gdecr:N \g__prg_map_int
7476     \cs_gset_eq:Nc \__prop_pair:wn
7477     { __prg_map_ \int_use:N \g__prg_map_int :wn }
7478   }
7479 }
7480 \cs_generate_variant:Nn \prop_map_inline:Nn { c }

```

(End definition for `\prop_map_inline:Nn` and `\prop_map_inline:cn`. These functions are documented on page 145.)

`\prop_map_break:` The break statements are based on the general `__prg_map_break:Nn`.
`\prop_map_break:n`

```

7481 \cs_new_nopar:Npn \prop_map_break:
7482 { \__prg_map_break:Nn \prop_map_break: { } }
7483 \cs_new_nopar:Npn \prop_map_break:n
7484 { \__prg_map_break:Nn \prop_map_break: }

```

(End definition for `\prop_map_break:`. This function is documented on page 145.)

14.6 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.

```

7485 \cs_new_protected:Npn \prop_show:N #1
7486 {
7487   \__msg_show_variable:NNNnn #1
7488   \prop_if_exist:NTF \prop_if_empty:NTF { prop }
7489   { \prop_map_function:NN #1 \__msg_show_item:nn }
7490 }
7491 \cs_generate_variant:Nn \prop_show:N { c }

```

(End definition for `\prop_show:N` and `\prop_show:c`. These functions are documented on page 145.)

14.7 Deprecated functions

`\prop_get:Nn` Deprecated 2014-07-17.

```

7492 \cs_new_eq:NN \prop_get:Nn \prop_item:Nn
7493 \cs_new_eq:NN \prop_get:cn \prop_item:cn

```

(End definition for `\prop_get:Nn` and `\prop_get:cn`. These functions are documented on page ??.)

```
7494 \</initex | package>
```

15 l3box implementation

```
7495 \*initex | package>
```

```
7496 \@@=box>
```

The code in this module is very straight forward so I'm not going to comment it very extensively.

15.1 Creating and initialising boxes

The following test files are used for this code: `m3box001.lvt`.

\box_new:N Defining a new `\box` register: remember that box 255 is not generally available.

```
\box_new:c 7497 \*package>
7498 \cs_new_protected:Npn \box_new:N #1
7499 {
7500     \__chk_if_free_cs:N #1
7501     \cs:w newbox \cs_end: #1
7502 }
7503 \</package>
7504 \cs_generate_variant:Nn \box_new:N { c }
```

Clear a `\box` register.

```
7505 \cs_new_protected:Npn \box_clear:N #1
\box_clear:N 7506 { \box_set_eq:NN #1 \c_empty_box }
\box_clear:c 7507 \cs_new_protected:Npn \box_gclear:N #1
\box_gclear:N 7508 { \box_gset_eq:NN #1 \c_empty_box }
\box_gclear:c 7509 \cs_generate_variant:Nn \box_clear:N { c }
7510 \cs_generate_variant:Nn \box_gclear:N { c }
```

Clear or new.

```
7511 \cs_new_protected:Npn \box_clear_new:N #1
\box_clear_new:N 7512 { \box_if_exist:NTF #1 { \box_clear:N #1 } { \box_new:N #1 } }
\box_clear_new:c 7513 \cs_new_protected:Npn \box_gclear_new:N #1
\box_gclear_new:N 7514 { \box_if_exist:NTF #1 { \box_gclear:N #1 } { \box_new:N #1 } }
\box_gclear_new:c 7515 \cs_generate_variant:Nn \box_clear_new:N { c }
7516 \cs_generate_variant:Nn \box_gclear_new:N { c }
```

Assigning the contents of a box to be another box.

```
7517 \cs_new_protected:Npn \box_set_eq:NN #1#2
\box_set_eq:NN 7518 { \tex_setbox:D #1 \tex_copy:D #2 }
\box_set_eq:cN 7519 \cs_new_protected:Npn \box_gset_eq:NN
\box_set_eq:Nc 7520 { \tex_global:D \box_set_eq:NN }
\box_set_eq:cc 7521 \cs_generate_variant:Nn \box_set_eq:NN { c , Nc , cc }
\box_gset_eq:NN 7522 \cs_generate_variant:Nn \box_gset_eq:NN { c , Nc , cc }
\box_gset_eq:cN
\box_gset_eq:Nc
\box_gset_eq:cc
```

Assigning the contents of a box to be another box. This clears the second box globally (that's how TeX does it).

```

\box_set_eq_clear:NN 7523 \cs_new_protected:Npn \box_set_eq_clear:NN #1#2
\box_set_eq_clear:cN 7524 { \tex_setbox:D #1 \tex_box:D #2 }
\box_set_eq_clear:Nc 7525 \cs_new_protected:Npn \box_gset_eq_clear:NN
\box_set_eq_clear:cc 7526 { \tex_global:D \box_set_eq_clear:NN }
\box_gset_eq_clear:NN 7527 \cs_generate_variant:Nn \box_set_eq_clear:NN { c , Nc , cc }
\box_gset_eq_clear:cN 7528 \cs_generate_variant:Nn \box_gset_eq_clear:NN { c , Nc , cc }
\box_gset_eq_clear:Nc
\box_gset_eq_clear:cc

\box_if_exist_p:N
\box_if_exist_p:c
\box_if_exist:N $\underline{TF}$ 
\box_if_exist:c $\underline{TF}$ 

```

Copies of the cs functions defined in l3basics.

```

7529 \prg_new_eq_conditional:NNn \box_if_exist:N \cs_if_exist:N
7530 { TF , T , F , p }
7531 \prg_new_eq_conditional:NNn \box_if_exist:c \cs_if_exist:c
7532 { TF , T , F , p }

```

15.2 Measuring and setting box dimensions

Accessing the height, depth, and width of a $\langle box \rangle$ register.

```

\box_ht:N 7533 \cs_new_eq:NN \box_ht:N \tex_ht:D
\box_ht:c 7534 \cs_new_eq:NN \box_dp:N \tex_dp:D
\box_dp:N 7535 \cs_new_eq:NN \box_wd:N \tex_wd:D
\box_dp:c 7536 \cs_generate_variant:Nn \box_ht:N { c }
\box_wd:N 7537 \cs_generate_variant:Nn \box_dp:N { c }
\box_wd:c 7538 \cs_generate_variant:Nn \box_wd:N { c }

```

Measuring is easy: all primitive work. These primitives are not expandable, so the derived functions are not either.

```

\box_set_ht:Nn 7539 \cs_new_protected:Npn \box_set_dp:Nn #1#2
\box_set_ht:cn 7540 { \box_dp:N #1 \__dim_eval:w #2 \__dim_eval_end: }
\box_set_dp:Nn 7541 \cs_new_protected:Npn \box_set_ht:Nn #1#2
\box_set_dp:cn 7542 { \box_ht:N #1 \__dim_eval:w #2 \__dim_eval_end: }
\box_set_wd:Nn 7543 \cs_new_protected:Npn \box_set_wd:Nn #1#2
\box_set_wd:cn 7544 { \box_wd:N #1 \__dim_eval:w #2 \__dim_eval_end: }
7545 \cs_generate_variant:Nn \box_set_ht:Nn { c }
7546 \cs_generate_variant:Nn \box_set_dp:Nn { c }
7547 \cs_generate_variant:Nn \box_set_wd:Nn { c }

```

15.3 Using boxes

Using a $\langle box \rangle$. These are just TeX primitives with meaningful names.

```

\box_use_clear:N 7548 \cs_new_eq:NN \box_use_clear:N \tex_box:D
\box_use_clear:c 7549 \cs_new_eq:NN \box_use:N \tex_copy:D
\box_use:N 7550 \cs_generate_variant:Nn \box_use_clear:N { c }
\box_use:c 7551 \cs_generate_variant:Nn \box_use:N { c }

```

Move box material in different directions.

```

\box_move_left:nn 7552 \cs_new_protected:Npn \box_move_left:nn #1#2
\box_move_right:nn 7553 { \tex_moveleft:D \__dim_eval:w #1 \__dim_eval_end: #2 }
\box_move_up:nn
\box_move_down:nn

```

```

7554 \cs_new_protected:Npn \box_move_right:nn #1#2
7555 { \tex_moveright:D \__dim_eval:w #1 \__dim_eval_end: #2 }
7556 \cs_new_protected:Npn \box_move_up:nn #1#2
7557 { \tex_raise:D \__dim_eval:w #1 \__dim_eval_end: #2 }
7558 \cs_new_protected:Npn \box_move_down:nn #1#2
7559 { \tex_lower:D \__dim_eval:w #1 \__dim_eval_end: #2 }

```

15.4 Box conditionals

The primitives for testing if a $\langle box \rangle$ is empty/void or which type of box it is.

```

\if_hbox:N
\if_vbox:N
\if_box_empty:N

7560 \cs_new_eq:NN \if_hbox:N \tex_ifhbox:D
7561 \cs_new_eq:NN \if_vbox:N \tex_ifvbox:D
7562 \cs_new_eq:NN \if_box_empty:N \tex_ifvoid:D

7563 \prg_new_conditional:Npnn \box_if_horizontal:N #1 { p , T , F , TF }
7564 { \if_hbox:N #1 \prg_return_true: \else: \prg_return_false: \fi: }
7565 \prg_new_conditional:Npnn \box_if_vertical:N #1 { p , T , F , TF }
7566 { \if_vbox:N #1 \prg_return_true: \else: \prg_return_false: \fi: }
7567 \cs_generate_variant:Nn \box_if_horizontal_p:N { c }
7568 \cs_generate_variant:Nn \box_if_horizontal:NT { c }
7569 \cs_generate_variant:Nn \box_if_horizontal:NF { c }
7570 \cs_generate_variant:Nn \box_if_horizontal:NTF { c }
7571 \cs_generate_variant:Nn \box_if_vertical_p:N { c }
7572 \cs_generate_variant:Nn \box_if_vertical:NT { c }
7573 \cs_generate_variant:Nn \box_if_vertical:NF { c }
7574 \cs_generate_variant:Nn \box_if_vertical:NTF { c }

```

Testing if a $\langle box \rangle$ is empty/void.

```

7575 \prg_new_conditional:Npnn \box_if_empty:N #1 { p , T , F , TF }
7576 { \if_box_empty:N #1 \prg_return_true: \else: \prg_return_false: \fi: }
7577 \cs_generate_variant:Nn \box_if_empty_p:N { c }
7578 \cs_generate_variant:Nn \box_if_empty:NT { c }
7579 \cs_generate_variant:Nn \box_if_empty:NF { c }
7580 \cs_generate_variant:Nn \box_if_empty:NTF { c }

```

(End definition for $\backslash box_new:N$ and $\backslash box_new:c$. These functions are documented on page 147.)

15.5 The last box inserted

```

\box_set_to_last:N
\box_set_to_last:c
\box_gset_to_last:N
\box_gset_to_last:c

7581 \cs_new_protected:Npn \box_set_to_last:N #1
7582 { \tex_setbox:D #1 \tex_lastbox:D }
7583 \cs_new_protected:Npn \box_gset_to_last:N
7584 { \tex_global:D \box_set_to_last:N }
7585 \cs_generate_variant:Nn \box_set_to_last:N { c }
7586 \cs_generate_variant:Nn \box_gset_to_last:N { c }

```

(End definition for $\backslash box_set_to_last:N$ and $\backslash box_set_to_last:c$. These functions are documented on page 150.)

15.6 Constant boxes

`\c_empty_box` A box we never use.

```
7587 \box_new:N \c_empty_box
```

(End definition for `\c_empty_box`. This variable is documented on page 150.)

15.7 Scratch boxes

`\l_tmpa_box` Scratch boxes.

```
7588 \box_new:N \l_tmpa_box
```

```
7589 \box_new:N \l_tmpb_box
```

```
7590 \box_new:N \g_tmpa_box
```

```
7591 \box_new:N \g_tmpb_box
```

(End definition for `\l_tmpa_box` and others. These variables are documented on page 150.)

15.8 Viewing box contents

TeX's `\showbox` is not really that helpful in many cases, and it is also inconsistent with other L^AT_EX3 show functions as it does not actually shows material in the terminal. So we provide a richer set of functionality.

`\box_show:N` Essentially a wrapper around the internal function.

```
\box_show:c      7592 \cs_new_protected:Npn \box_show:N #1
\box_show:Nnn    7593 { \box_show:Nnn #1 \c_max_int \c_max_int }
\box_show:cnn    7594 \cs_generate_variant:Nn \box_show:N { c }
                  7595 \cs_new_protected_nopar:Npn \box_show:Nnn
                  7596 { \__box_show:NNnn \c_one }
                  7597 \cs_generate_variant:Nn \box_show:Nnn { c }
```

(End definition for `\box_show:N` and `\box_show:c`. These functions are documented on page 150.)

`\box_log:N` Getting TeX to write to the log without interruption the run is done by altering the interaction mode. For that, the ε -TeX extensions are needed.

```
\box_log:c      7598 \cs_new_protected:Npn \box_log:N #1
\box_log:Nnn    7599 { \box_log:Nnn #1 \c_max_int \c_max_int }
\box_log:cnn    7600 \cs_generate_variant:Nn \box_log:N { c }
                  7601 \cs_new_protected:Npn \box_log:Nnn #1#2#3
                  7602 {
                  7603   \use:x
                  7604   {
                  7605     \etex_interactionmode:D \c_zero
                  7606     \__box_show:NNnn \c_zero \exp_not:N #1
                  7607     { \int_eval:n {#2} } { \int_eval:n {#3} }
                  7608     \etex_interactionmode:D
                  7609     = \tex_the:D \etex_interactionmode:D \scan_stop:
                  7610   }
                  7611 }
                  7612 \cs_generate_variant:Nn \box_log:Nnn { c }
```

(End definition for `\box_log:N` and `\box_log:c`. These functions are documented on page 150.)

`__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 `\errorcontextlines` is used to control what appears in the terminal.

```

7613 \cs_new_protected:Npn \__box_show:NNnn #1#2#3#4
7614 {
7615   \group_begin:
7616     \int_set:Nn \tex_showboxbreadth:D {#3}
7617     \int_set:Nn \tex_showboxdepth:D {#4}
7618     \int_set_eq:NN \tex_tracingonline:D #1
7619     \int_set_eq:NN \tex_errorcontextlines:D \c_minus_one
7620     \box_if_exist:NTF #2
7621       { \tex_showbox:D \use:n {#2} }
7622       {
7623         \__msg_kernel_error:nxx { kernel } { variable-not-defined }
7624         { \token_to_str:N #2 }
7625       }
7626   \group_end:
7627 }
```

(End definition for `__box_show:NNnn`.)

15.9 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.

```

7628 \cs_new_protected:Npn \hbox:n #1 { \tex_hbox:D \scan_stop: {#1} }
```

(End definition for `\hbox:n`. This function is documented on page 151.)

```

\hbox_set:Nn
\hbox_set:cn
\hbox_gset:Nn
\hbox_gset:cn
7629 \cs_new_protected:Npn \hbox_set:Nn #1#2
7630 { \tex_setbox:D #1 \tex_hbox:D {#2} }
7631 \cs_new_protected:Npn \hbox_gset:Nn { \tex_global:D \hbox_set:Nn }
7632 \cs_generate_variant:Nn \hbox_set:Nn { c }
7633 \cs_generate_variant:Nn \hbox_gset:Nn { c }
```

(End definition for `\hbox_set:Nn` and `\hbox_set:cn`. These functions are documented on page 151.)

```

\hbox_set_to_wd:Nnn
\hbox_set_to_wd:cnn
\hbox_gset_to_wd:Nnn
\hbox_gset_to_wd:cnn
7634 \cs_new_protected:Npn \hbox_set_to_wd:Nnn #1#2#3
7635 { \tex_setbox:D #1 \tex_hbox:D to \__dim_eval:w #2 \__dim_eval_end: {#3} }
7636 \cs_new_protected:Npn \hbox_gset_to_wd:Nnn
7637 { \tex_global:D \hbox_set_to_wd:Nnn }
7638 \cs_generate_variant:Nn \hbox_set_to_wd:Nnn { c }
7639 \cs_generate_variant:Nn \hbox_gset_to_wd:Nnn { c }
```

(End definition for `\hbox_set_to_wd:Nnn` and `\hbox_set_to_wd:cnn`. These functions are documented on page 151.)

`\hbox_set:Nw` Storing material in a horizontal box. This type is useful in environment definitions.

```

\hbox_set:cw 7640 \cs_new_protected:Npn \hbox_set:Nw #1
\hbox_gset:Nw 7641 { \tex_setbox:D #1 \tex_hbox:D \c_group_begin_token }
\hbox_gset:cw 7642 \cs_new_protected:Npn \hbox_gset:Nw
\hbox_set_end: 7643 { \tex_global:D \hbox_set:Nw }
\hbox_gset_end: 7644 \cs_generate_variant:Nn \hbox_set:Nw { c }
7645 \cs_generate_variant:Nn \hbox_gset:Nw { c }
7646 \cs_new_eq:NN \hbox_set_end: \c_group_end_token
7647 \cs_new_eq:NN \hbox_gset_end: \c_group_end_token

```

(End definition for `\hbox_set:Nw` and `\hbox_set:cw`. These functions are documented on page 151.)

`\hbox_to_wd:nn` Put a horizontal box directly into the input stream.

```

\hbox_to_zero:n 7648 \cs_new_protected:Npn \hbox_to_wd:nn #1#2
7649 { \tex_hbox:D to \__dim_eval:w #1 \__dim_eval_end: {#2} }
7650 \cs_new_protected:Npn \hbox_to_zero:n #1 { \tex_hbox:D to \c_zero_dim {#1} }

```

(End definition for `\hbox_to_wd:nn`. This function is documented on page 151.)

`\hbox_overlap_left:n` Put a zero-sized box with the contents pushed against one side (which makes it stick out
`\hbox_overlap_right:n` on the other) directly into the input stream.

```

7651 \cs_new_protected:Npn \hbox_overlap_left:n #1
7652 { \hbox_to_zero:n { \tex_hss:D #1 } }
7653 \cs_new_protected:Npn \hbox_overlap_right:n #1
7654 { \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 151.)

`\hbox_unpack:N` Unpacking a box and if requested also clear it.

```

\hbox_unpack:c 7655 \cs_new_eq:NN \hbox_unpack:N \tex_unhcopy:D
\hbox_unpack_clear:N 7656 \cs_new_eq:NN \hbox_unpack_clear:N \tex_unhbox:D
\hbox_unpack_clear:c 7657 \cs_generate_variant:Nn \hbox_unpack:N { c }
7658 \cs_generate_variant:Nn \hbox_unpack_clear:N { c }

```

(End definition for `\hbox_unpack:N` and `\hbox_unpack:c`. These functions are documented on page 152.)

15.10 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.

```

7659 \cs_new_protected:Npn \vbox:n #1 { \tex_vbox:D { #1 \par } }
7660 \cs_new_protected:Npn \vbox_top:n #1 { \tex_vtop:D { #1 \par } }

```

(End definition for `\vbox:n`. This function is documented on page 152.)

`\vbox_to_ht:nn` Put a vertical box directly into the input stream.

`\vbox_to_zero:n` 7661 `\cs_new_protected:Npn \vbox_to_ht:nn #1#2`
`\vbox_to_ht:nn` 7662 `{ \tex_vbox:D to _dim_eval:w #1 _dim_eval_end: { #2 \par } }`
`\vbox_to_zero:n` 7663 `\cs_new_protected:Npn \vbox_to_zero:n #1`
7664 `{ \tex_vbox:D to \c_zero_dim { #1 \par } }`

(End definition for `\vbox_to_ht:nn` and `\vbox_to_zero:n`. These functions are documented on page 152.)

`\vbox_set:Nn` Storing material in a vertical box with a natural height.

`\vbox_set:cn` 7665 `\cs_new_protected:Npn \vbox_set:Nn #1#2`
`\vbox_gset:Nn` 7666 `{ \tex_setbox:D #1 \tex_vbox:D { #2 \par } }`
`\vbox_gset:cn` 7667 `\cs_new_protected:Npn \vbox_gset:Nn { \tex_global:D \vbox_set:Nn }`
7668 `\cs_generate_variant:Nn \vbox_set:Nn { c }`
7669 `\cs_generate_variant:Nn \vbox_gset:Nn { c }`

(End definition for `\vbox_set:Nn` and `\vbox_set:cn`. These functions are documented on page 153.)

`\vbox_set_top:Nn` Storing material in a vertical box with a natural height and reference point at the baseline of the first object in the box.

`\vbox_set_top:cn`

`\vbox_gset_top:Nn` 7670 `\cs_new_protected:Npn \vbox_set_top:Nn #1#2`
`\vbox_gset_top:cn` 7671 `{ \tex_setbox:D #1 \tex_vtop:D { #2 \par } }`
7672 `\cs_new_protected:Npn \vbox_gset_top:Nn`
7673 `{ \tex_global:D \vbox_set_top:Nn }`
7674 `\cs_generate_variant:Nn \vbox_set_top:Nn { c }`
7675 `\cs_generate_variant:Nn \vbox_gset_top:Nn { c }`

(End definition for `\vbox_set_top:Nn` and `\vbox_set_top:cn`. These functions are documented on page 153.)

`\vbox_set_to_ht:Nnn` Storing material in a vertical box with a specified height.

`\vbox_set_to_ht:cnn` 7676 `\cs_new_protected:Npn \vbox_set_to_ht:Nnn #1#2#3`
`\vbox_gset_to_ht:Nnn` 7677 `{`
`\vbox_gset_to_ht:cnn` 7678 `\tex_setbox:D #1 \tex_vbox:D to _dim_eval:w #2 _dim_eval_end:`
7679 `{ #3 \par }`
7680 `}`
7681 `\cs_new_protected:Npn \vbox_gset_to_ht:Nnn`
7682 `{ \tex_global:D \vbox_set_to_ht:Nnn }`
7683 `\cs_generate_variant:Nn \vbox_set_to_ht:Nnn { c }`
7684 `\cs_generate_variant:Nn \vbox_gset_to_ht:Nnn { c }`

(End definition for `\vbox_set_to_ht:Nnn` and `\vbox_set_to_ht:cnn`. These functions are documented on page 153.)

`\vbox_set:Nw` Storing material in a vertical box. This type is useful in environment definitions.

`\vbox_set:cw` 7685 `\cs_new_protected:Npn \vbox_set:Nw #1`
`\vbox_gset:Nw` 7686 `{ \tex_setbox:D #1 \tex_vbox:D \c_group_begin_token }`
`\vbox_gset:cw` 7687 `\cs_new_protected:Npn \vbox_gset:Nw`
`\vbox_set_end:` 7688 `{ \tex_global:D \vbox_set:Nw }`
`\vbox_gset_end:` 7689 `\cs_generate_variant:Nn \vbox_set:Nw { c }`
7690 `\cs_generate_variant:Nn \vbox_gset:Nw { c }`

```

7691 \cs_new_protected:Npn \vbox_set_end:
7692 {
7693     \par
7694     \c_group_end_token
7695 }
7696 \cs_new_eq:NN \vbox_gset_end: \vbox_set_end:

```

(End definition for `\vbox_set:Nw` and `\vbox_set:cw`. These functions are documented on page 153.)

```

\vbox_unpack:N Unpacking a box and if requested also clear it.
\vbox_unpack:c 7697 \cs_new_eq:NN \vbox_unpack:N \tex_unvcopy:D
\vbox_unpack_clear:N 7698 \cs_new_eq:NN \vbox_unpack_clear:N \tex_unvbox:D
\vbox_unpack_clear:c 7699 \cs_generate_variant:Nn \vbox_unpack:N { c }
7700 \cs_generate_variant:Nn \vbox_unpack_clear:N { c }

```

(End definition for `\vbox_unpack:N` and `\vbox_unpack:c`. These functions are documented on page 153.)

```

\vbox_set_split_to_ht:NNn Splitting a vertical box in two.
7701 \cs_new_protected:Npn \vbox_set_split_to_ht:NNn #1#2#3
7702 { \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 153.)

7703 </initex | package>

```

16 l3coffins Implementation

```

7704 <*initex | package>
7705 <@@=coffin>

```

16.1 Coffins: data structures and general variables

```

\l__coffin_internal_box Scratch variables.
\l__coffin_internal_dim 7706 \box_new:N \l__coffin_internal_box
\l__coffin_internal_tl 7707 \dim_new:N \l__coffin_internal_dim
7708 \tl_new:N \l__coffin_internal_tl

(End definition for \l__coffin_internal_box. This variable is documented on page ??.)

\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.
7709 \prop_new:N \c__coffin_corners_prop
7710 \prop_put:Nnn \c__coffin_corners_prop { tl } { { 0 pt } { 0 pt } }
7711 \prop_put:Nnn \c__coffin_corners_prop { tr } { { 0 pt } { 0 pt } }
7712 \prop_put:Nnn \c__coffin_corners_prop { bl } { { 0 pt } { 0 pt } }
7713 \prop_put:Nnn \c__coffin_corners_prop { br } { { 0 pt } { 0 pt } }

(End definition for \c__coffin_corners_prop. This variable is documented on page ??.)

```

`\c__coffin_poles_prop` Pole positions are given for horizontal, vertical and reference-point based values.

```

7714 \prop_new:N \c__coffin_poles_prop
7715 \tl_set:Nn \l__coffin_internal_tl { { 0 pt } { 0 pt } { 0 pt } { 1000 pt } }
7716 \prop_put:Nno \c__coffin_poles_prop { l } { \l__coffin_internal_tl }
7717 \prop_put:Nno \c__coffin_poles_prop { hc } { \l__coffin_internal_tl }
7718 \prop_put:Nno \c__coffin_poles_prop { r } { \l__coffin_internal_tl }
7719 \tl_set:Nn \l__coffin_internal_tl { { 0 pt } { 0 pt } { 1000 pt } { 0 pt } }
7720 \prop_put:Nno \c__coffin_poles_prop { b } { \l__coffin_internal_tl }
7721 \prop_put:Nno \c__coffin_poles_prop { vc } { \l__coffin_internal_tl }
7722 \prop_put:Nno \c__coffin_poles_prop { t } { \l__coffin_internal_tl }
7723 \prop_put:Nno \c__coffin_poles_prop { B } { \l__coffin_internal_tl }
7724 \prop_put:Nno \c__coffin_poles_prop { H } { \l__coffin_internal_tl }
7725 \prop_put:Nno \c__coffin_poles_prop { T } { \l__coffin_internal_tl }

```

(End definition for \c__coffin_poles_prop. This variable is documented on page ??.)

`\l__coffin_slope_x_fp` Used for calculations of intersections.
`\l__coffin_slope_y_fp`

```

7726 \fp_new:N \l__coffin_slope_x_fp
7727 \fp_new:N \l__coffin_slope_y_fp

```

(End definition for \l__coffin_slope_x_fp. This variable is documented on page ??.)

`\l__coffin_error_bool` For propagating errors so that parts of the code can work around them.

```

7728 \bool_new:N \l__coffin_error_bool

```

(End definition for \l__coffin_error_bool. This variable is documented on page ??.)

`\l__coffin_offset_x_dim` The offset between two sets of coffin handles when typesetting. These values are corrected
`\l__coffin_offset_y_dim` from those requested in an alignment for the positions of the handles.

```

7729 \dim_new:N \l__coffin_offset_x_dim
7730 \dim_new:N \l__coffin_offset_y_dim

```

(End definition for \l__coffin_offset_x_dim. This variable is documented on page ??.)

`\l__coffin_pole_a_tl` Needed for finding the intersection of two poles.
`\l__coffin_pole_b_tl`

```

7731 \tl_new:N \l__coffin_pole_a_tl
7732 \tl_new:N \l__coffin_pole_b_tl

```

(End definition for \l__coffin_pole_a_tl. This variable is documented on page ??.)

`\l__coffin_x_dim` For calculating intersections and so forth.
`\l__coffin_y_dim`

```

7733 \dim_new:N \l__coffin_x_dim
7734 \dim_new:N \l__coffin_y_dim
7735 \dim_new:N \l__coffin_x_prime_dim
7736 \dim_new:N \l__coffin_y_prime_dim

```

(End definition for \l__coffin_x_dim. This variable is documented on page ??.)

16.2 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 exist. A cleaner way to handle this is provided here: both the box and the coffin structure are checked.

```

\coffin_if_exist_p:c
\coffin_if_exist:NTF
\coffin_if_exist:cTF
7737 \prg_new_conditional:Npnn \coffin_if_exist:N #1 { p , T , F , TF }
7738 {
7739   \cs_if_exist:NTF #1
7740   {
7741     \cs_if_exist:cTF { l__coffin_poles_ \__int_value:w #1 _prop }
7742     { \prg_return_true: }
7743     { \prg_return_false: }
7744   }
7745   { \prg_return_false: }
7746 }
7747 \cs_generate_variant:Nn \coffin_if_exist_p:N { c }
7748 \cs_generate_variant:Nn \coffin_if_exist:NT { c }
7749 \cs_generate_variant:Nn \coffin_if_exist:NF { c }
7750 \cs_generate_variant:Nn \coffin_if_exist:NTF { c }

```

(End definition for `\coffin_if_exist:NTF` and `\coffin_if_exist:cTF`. These functions are documented on page 155.)

`__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.

```

7751 \cs_new_protected:Npn \__coffin_if_exist:NT #1#2
7752 {
7753   \coffin_if_exist:NTF #1
7754   { #2 }
7755   {
7756     \__msg_kernel_error:nxx { kernel } { unknown-coffin }
7757     { \token_to_str:N #1 }
7758   }
7759 }

```

(End definition for `__coffin_if_exist:NT`. This function is documented on page ??.)

`\coffin_clear:N` Clearing coffins means emptying the box and resetting all of the structures.

```

\coffin_clear:c
7760 \cs_new_protected:Npn \coffin_clear:N #1
7761 {
7762   \__coffin_if_exist:NT #1
7763   {
7764     \box_clear:N #1
7765     \__coffin_reset_structure:N #1
7766   }
7767 }
7768 \cs_generate_variant:Nn \coffin_clear:N { c }

```

(End definition for `\coffin_clear:N` and `\coffin_clear:c`. These functions are documented on page 155.)

`\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.

```

7769 \cs_new_protected:Npn \coffin_new:N #1
7770 {
7771   \box_new:N #1
7772   \__chk_suspend_log:
7773   \prop_clear_new:c { l__coffin_corners_ \__int_value:w #1 _prop }
7774   \prop_clear_new:c { l__coffin_poles_ \__int_value:w #1 _prop }
7775   \prop_gset_eq:cN { l__coffin_corners_ \__int_value:w #1 _prop }
7776   \c__coffin_corners_prop
7777   \prop_gset_eq:cN { l__coffin_poles_ \__int_value:w #1 _prop }
7778   \c__coffin_poles_prop
7779   \__chk_resume_log:
7780 }
7781 \cs_generate_variant:Nn \coffin_new:N { c }

```

(End definition for `\coffin_new:N` and `\coffin_new:c`. These functions are documented on page 155.)

`\hcoffin_set:Nn` Horizontal coffins are relatively easy: set the appropriate box, reset the structures then
`\hcoffin_set:cn` update the handle positions.

```

7782 \cs_new_protected:Npn \hcoffin_set:Nn #1#2
7783 {
7784   \__coffin_if_exist:NT #1
7785   {
7786     \hbox_set:Nn #1
7787     {
7788       \color_group_begin:
7789       \color_ensure_current:
7790       #2
7791       \color_group_end:
7792     }
7793     \__coffin_reset_structure:N #1
7794     \__coffin_update_poles:N #1
7795     \__coffin_update_corners:N #1
7796   }
7797 }
7798 \cs_generate_variant:Nn \hcoffin_set:Nn { c }

```

(End definition for `\hcoffin_set:Nn` and `\hcoffin_set:cn`. These functions are documented on page 155.)

`\vcoffin_set:Nnn` Setting vertical coffins is more complex. First, the material is typeset with a given width.
`\vcoffin_set:cnn` The default handles and poles are set as for a horizontal coffin, before finding the top 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).

```

7799 \cs_new_protected:Npn \vcoffin_set:Nnn #1#2#3
7800 {
7801   \__coffin_if_exist:NT #1
7802   {
7803     \vbox_set:Nn #1
7804     {
7805       \dim_set:Nn \tex_hsize:D {#2}
7806     }
7807     \dim_set_eq:NN \linewidth \tex_hsize:D
7808     \dim_set_eq:NN \columnwidth \tex_hsize:D
7809   }
7810   \color_group_begin:
7811   #3
7812   \color_group_end:
7813   }
7814   \__coffin_reset_structure:N #1
7815   \__coffin_update_poles:N #1
7816   \__coffin_update_corners:N #1
7817   \vbox_set_top:Nn \l__coffin_internal_box { \vbox_unpack:N #1 }
7818   \__coffin_set_pole:Nnx #1 { T }
7819   {
7820     { 0 pt }
7821     {
7822       \dim_eval:n
7823       { \box_ht:N #1 - \box_ht:N \l__coffin_internal_box }
7824     }
7825     { 1000 pt }
7826     { 0 pt }
7827   }
7828   \box_clear:N \l__coffin_internal_box
7829 }
7830 }
7831 \cs_generate_variant:Nn \vcoffin_set:Nnn { c }

```

(End definition for `\vcoffin_set:Nnn` and `\vcoffin_set:cnn`. These functions are documented on page 156.)

`\hcoffin_set:Nw` These are the “begin”/“end” versions of the above: watch the grouping!
`\hcoffin_set:cw`
`\hcoffin_set_end:`

```

7832 \cs_new_protected:Npn \hcoffin_set:Nw #1
7833 {
7834   \__coffin_if_exist:NT #1
7835   {
7836     \hbox_set:Nw #1 \color_group_begin: \color_ensure_current:
7837     \cs_set_protected_nopar:Npn \hcoffin_set_end:
7838     {
7839       \color_group_end:
7840       \hbox_set_end:
7841       \__coffin_reset_structure:N #1

```

```

7842         \_coffin_update_poles:N #1
7843         \_coffin_update_corners:N #1
7844     }
7845 }
7846 }
7847 \cs_new_protected_nopar:Npn \hcoffin_set_end: { }
7848 \cs_generate_variant:Nn \hcoffin_set:Nw { c }

```

(End definition for `\hcoffin_set:Nw` and `\hcoffin_set:cw`. These functions are documented on page 156.)

`\vcoffin_set:Nnw` The same for vertical coffins.

```

\vcoffin_set:cnw 7849 \cs_new_protected:Npn \vcoffin_set:Nnw #1#2
\vcoffin_set_end: 7850 {
7851     \_coffin_if_exist:NT #1
7852     {
7853         \vbox_set:Nw #1
7854         \dim_set:Nn \tex_hsize:D {#2}
7855     }
7856     \dim_set_eq:NN \linewidth \tex_hsize:D
7857     \dim_set_eq:NN \columnwidth \tex_hsize:D
7858 }
7859 \color_group_begin: \color_ensure_current:
7860 \cs_set_protected:Npn \vcoffin_set_end:
7861 {
7862     \color_group_end:
7863     \vbox_set_end:
7864     \_coffin_reset_structure:N #1
7865     \_coffin_update_poles:N #1
7866     \_coffin_update_corners:N #1
7867     \vbox_set_top:Nn \l__coffin_internal_box { \vbox_unpack:N #1 }
7868     \_coffin_set_pole:Nnx #1 { T }
7869     {
7870         { 0 pt }
7871         {
7872             \dim_eval:n
7873             { \box_ht:N #1 - \box_ht:N \l__coffin_internal_box }
7874         }
7875         { 1000 pt }
7876         { 0 pt }
7877     }
7878     \box_clear:N \l__coffin_internal_box
7879 }
7880 }
7881 }
7882 \cs_new_protected_nopar:Npn \vcoffin_set_end: { }
7883 \cs_generate_variant:Nn \vcoffin_set:Nnw { c }

```

(End definition for `\vcoffin_set:Nnw` and `\vcoffin_set:cnw`. These functions are documented on page 156.)

\coffin_set_eq:NN Setting two coffins equal is just a wrapper around other functions.

```

\coffin_set_eq:Nc 7884 \cs_new_protected:Npn \coffin_set_eq:NN #1#2
\coffin_set_eq:cN 7885 {
\coffin_set_eq:cc 7886   \__coffin_if_exist:NT #1
7887   {
7888     \box_set_eq:NN #1 #2
7889     \__coffin_set_eq_structure:NN #1 #2
7890   }
7891 }
7892 \cs_generate_variant:Nn \coffin_set_eq:NN { c , Nc , cc }
```

(End definition for \coffin_set_eq:NN and others. These functions are documented on page 155.)

\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.

```

\l__coffin_aligned_coffin 7893 \coffin_new:N \c_empty_coffin
\l__coffin_aligned_internal_coffin 7894 \hbox_set:Nn \c_empty_coffin { }
7895 \coffin_new:N \l__coffin_aligned_coffin
7896 \coffin_new:N \l__coffin_aligned_internal_coffin
```

(End definition for \c_empty_coffin. This variable is documented on page 158.)

\l_tmpa_coffin The usual scratch space.

```

\l_tmpb_coffin 7897 \coffin_new:N \l_tmpa_coffin
7898 \coffin_new:N \l_tmpb_coffin
```

(End definition for \l_tmpa_coffin and \l_tmpb_coffin. These variables are documented on page 158.)

16.3 Measuring coffins

\coffin_dp:N Coffins are just boxes when it comes to measurement. However, semantically a separate set of functions are required.

```

\coffin_dp:c 7899 \cs_new_eq:NN \coffin_dp:N \box_dp:N
\coffin_ht:N 7900 \cs_new_eq:NN \coffin_dp:c \box_dp:c
\coffin_ht:c 7901 \cs_new_eq:NN \coffin_ht:N \box_ht:N
\coffin_wd:N 7902 \cs_new_eq:NN \coffin_ht:c \box_ht:c
\coffin_wd:c 7903 \cs_new_eq:NN \coffin_wd:N \box_wd:N
7904 \cs_new_eq:NN \coffin_wd:c \box_wd:c
```

(End definition for \coffin_dp:N and others. These functions are documented on page 157.)

16.4 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.

```

7905 \cs_new_protected:Npn \__coffin_get_pole:NnN #1#2#3
7906 {
7907   \prop_get:cnNF
7908     { l__coffin_poles_ \__int_value:w #1 _prop } {#2} #3
7909   {
7910     \__msg_kernel_error:nxxx { kernel } { unknown-coffin-pole }
7911     {#2} { \token_to_str:N #1 }
7912     \tl_set:Nn #3 { { 0 pt } { 0 pt } { 0 pt } { 0 pt } }
7913   }
7914 }
```

(End definition for __coffin_get_pole:NnN. This function is documented on page ??.)

`__coffin_reset_structure:N` Resetting the structure is a simple copy job.

```

7915 \cs_new_protected:Npn \__coffin_reset_structure:N #1
7916 {
7917   \prop_set_eq:cn { l__coffin_corners_ \__int_value:w #1 _prop }
7918   \c__coffin_corners_prop
7919   \prop_set_eq:cn { l__coffin_poles_ \__int_value:w #1 _prop }
7920   \c__coffin_poles_prop
7921 }
```

(End definition for __coffin_reset_structure:N. This function is documented on page ??.)

`_coffin_set_eq_structure:NN` Setting coffin structures equal simply means copying the property list.

```

\_coffin_gset_eq_structure:NN
7922 \cs_new_protected:Npn \__coffin_set_eq_structure:NN #1#2
7923 {
7924   \prop_set_eq:cc { l__coffin_corners_ \__int_value:w #1 _prop }
7925   { l__coffin_corners_ \__int_value:w #2 _prop }
7926   \prop_set_eq:cc { l__coffin_poles_ \__int_value:w #1 _prop }
7927   { l__coffin_poles_ \__int_value:w #2 _prop }
7928 }
7929 \cs_new_protected:Npn \__coffin_gset_eq_structure:NN #1#2
7930 {
7931   \prop_gset_eq:cc { l__coffin_corners_ \__int_value:w #1 _prop }
7932   { l__coffin_corners_ \__int_value:w #2 _prop }
7933   \prop_gset_eq:cc { l__coffin_poles_ \__int_value:w #1 _prop }
7934   { l__coffin_poles_ \__int_value:w #2 _prop }
7935 }
```

(End definition for _coffin_set_eq_structure:NN and __coffin_gset_eq_structure:NN. These functions are documented on page ??.)

`\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.

`\coffin_set_vertical_pole:Nnn`

`\coffin_set_vertical_pole:cnn`

`__coffin_set_pole:Nnn`

`__coffin_set_pole:Nnx`

```

7936 \cs_new_protected:Npn \coffin_set_horizontal_pole:Nnn #1#2#3
7937 {
7938   \__coffin_if_exist:NT #1
7939   {
7940     \__coffin_set_pole:Nnx #1 {#2}
7941     {
7942       { 0 pt } { \dim_eval:n {#3} }
7943       { 1000 pt } { 0 pt }
7944     }
7945   }
7946 }
7947 \cs_new_protected:Npn \coffin_set_vertical_pole:Nnn #1#2#3
7948 {
7949   \__coffin_if_exist:NT #1
7950   {
7951     \__coffin_set_pole:Nnx #1 {#2}
7952     {
7953       { \dim_eval:n {#3} } { 0 pt }
7954       { 0 pt } { 1000 pt }
7955     }
7956   }
7957 }
7958 \cs_new_protected:Npn \__coffin_set_pole:Nnn #1#2#3
7959 { \prop_put:cnn { l__coffin_poles_ } \__int_value:w #1 _prop } {#2} {#3} }
7960 \cs_generate_variant:Nn \coffin_set_horizontal_pole:Nnn { c }
7961 \cs_generate_variant:Nn \coffin_set_vertical_pole:Nnn { c }
7962 \cs_generate_variant:Nn \__coffin_set_pole:Nnn { Nnx }

```

(End definition for `\coffin_set_horizontal_pole:Nnn` and `\coffin_set_vertical_pole:Nnn`. These functions are documented on page 156.)

`__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.

```

7963 \cs_new_protected:Npn \__coffin_update_corners:N #1
7964 {
7965   \prop_put:cnx { l__coffin_corners_ } \__int_value:w #1 _prop } { tl }
7966   { { 0 pt } { \dim_use:N \box_ht:N #1 } }
7967   \prop_put:cnx { l__coffin_corners_ } \__int_value:w #1 _prop } { tr }
7968   { { \dim_use:N \box_wd:N #1 } { \dim_use:N \box_ht:N #1 } }
7969   \prop_put:cnx { l__coffin_corners_ } \__int_value:w #1 _prop } { bl }
7970   { { 0 pt } { \dim_eval:n { - \box_dp:N #1 } } }
7971   \prop_put:cnx { l__coffin_corners_ } \__int_value:w #1 _prop } { br }
7972   { { \dim_use:N \box_wd:N #1 } { \dim_eval:n { - \box_dp:N #1 } } }
7973 }

```

(End definition for `__coffin_update_corners:N`. This function is documented on page ??.)

`__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.

```

7974 \cs_new_protected:Npn \__coffin_update_poles:N #1
7975 {
7976   \prop_put:cnx { l__coffin_poles_ \__int_value:w #1 _prop } { hc }
7977   {
7978     { \dim_eval:n { 0.5 \box_wd:N #1 } }
7979     { 0 pt } { 0 pt } { 1000 pt }
7980   }
7981   \prop_put:cnx { l__coffin_poles_ \__int_value:w #1 _prop } { r }
7982   {
7983     { \dim_use:N \box_wd:N #1 }
7984     { 0 pt } { 0 pt } { 1000 pt }
7985   }
7986   \prop_put:cnx { l__coffin_poles_ \__int_value:w #1 _prop } { vc }
7987   {
7988     { 0 pt }
7989     { \dim_eval:n { ( \box_ht:N #1 - \box_dp:N #1 ) / 2 } }
7990     { 1000 pt }
7991     { 0 pt }
7992   }
7993   \prop_put:cnx { l__coffin_poles_ \__int_value:w #1 _prop } { t }
7994   {
7995     { 0 pt }
7996     { \dim_use:N \box_ht:N #1 }
7997     { 1000 pt }
7998     { 0 pt }
7999   }
8000   \prop_put:cnx { l__coffin_poles_ \__int_value:w #1 _prop } { b }
8001   {
8002     { 0 pt }
8003     { \dim_eval:n { - \box_dp:N #1 } }
8004     { 1000 pt }
8005     { 0 pt }
8006   }
8007 }

```

(End definition for __coffin_update_poles:N. This function is documented on page ??.)

16.5 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.

```

8008 \cs_new_protected:Npn \__coffin_calculate_intersection:Nnn #1#2#3
8009 {
8010   \__coffin_get_pole:NnN #1 {#2} \l__coffin_pole_a_tl
8011   \__coffin_get_pole:NnN #1 {#3} \l__coffin_pole_b_tl
8012   \bool_set_false:N \l__coffin_error_bool

```

```

8013 \exp_last_two_unbraced:Noo
8014 \__coffin_calculate_intersection:nnnnnnnn
8015 \l__coffin_pole_a_tl \l__coffin_pole_b_tl
8016 \bool_if:NT \l__coffin_error_bool
8017 {
8018 \__msg_kernel_error:nn { kernel } { no-pole-intersection }
8019 \dim_zero:N \l__coffin_x_dim
8020 \dim_zero:N \l__coffin_y_dim
8021 }
8022 }

```

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.

```

8023 \cs_new_protected:Npn \__coffin_calculate_intersection:nnnnnnnn
8024 #1#2#3#4#5#6#7#8
8025 {
8026 \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.

```

8027 {
8028 \dim_set:Nn \l__coffin_x_dim {#1}
8029 \dim_compare:nNnTF {#7} = \c_zero_dim
8030 { \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.

```

8031 {
8032 \dim_compare:nNnTF {#8} = \c_zero_dim
8033 { \dim_set:Nn \l__coffin_y_dim {#6} }
8034 {
8035 \__coffin_calculate_intersection_aux:nnnnnN
8036 {#1} {#5} {#6} {#7} {#8} \l__coffin_y_dim
8037 }
8038 }
8039 }

```

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.

```

8040 {
8041 \dim_compare:nNnTF {#4} = \c_zero_dim
8042 {

```

```

8043     \dim_set:Nn \l__coffin_y_dim {#2}
8044     \dim_compare:nNnTF {#8} = { \c_zero_dim }
8045     { \bool_set_true:N \l__coffin_error_bool }
8046     {
8047         \dim_compare:nNnTF {#7} = \c_zero_dim
8048         { \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.

```

8049     {
8050         \__coffin_calculate_intersection_aux:nnnnnN
8051         {#2} {#6} {#5} {#8} {#7} \l__coffin_x_dim
8052     }
8053 }
8054 }

```

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.

```

8055     {
8056         \dim_compare:nNnTF {#7} = \c_zero_dim
8057         {
8058             \dim_set:Nn \l__coffin_x_dim {#5}
8059             \__coffin_calculate_intersection_aux:nnnnnN
8060             {#5} {#1} {#2} {#3} {#4} \l__coffin_y_dim
8061         }
8062         {
8063             \dim_compare:nNnTF {#8} = \c_zero_dim
8064             {
8065                 \dim_set:Nn \l__coffin_y_dim {#6}
8066                 \__coffin_calculate_intersection_aux:nnnnnN
8067                 {#6} {#2} {#1} {#4} {#3} \l__coffin_x_dim
8068             }

```

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.

```

8069     {
8070         \fp_set:Nn \l__coffin_slope_x_fp
8071         { \dim_to_fp:n {#4} / \dim_to_fp:n {#3} }
8072         \fp_set:Nn \l__coffin_slope_y_fp
8073         { \dim_to_fp:n {#8} / \dim_to_fp:n {#7} }
8074         \fp_compare:nNnTF
8075         \l__coffin_slope_x_fp = \l__coffin_slope_y_fp
8076         { \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.

```

8077         {
8078             \dim_set:Nn \l__coffin_x_dim
8079             {
8080                 \fp_to_dim:n
8081                 {
8082                     (
8083                         \dim_to_fp:n {#1} * \l__coffin_slope_x_fp
8084                         - ( \dim_to_fp:n {#5} * \l__coffin_slope_y_fp )
8085                         - \dim_to_fp:n {#2}
8086                         + \dim_to_fp:n {#6}
8087                     )
8088                     /
8089                     ( \l__coffin_slope_x_fp - \l__coffin_slope_y_fp )
8090                 }
8091             }
8092             \__coffin_calculate_intersection_aux:nnnnnN
8093             { \l__coffin_x_dim }
8094             {#5} {#6} {#8} {#7} \l__coffin_y_dim
8095         }
8096     }
8097 }
8098 }
8099 }
8100 }
```

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.

```

8101 \cs_new_protected:Npn \__coffin_calculate_intersection_aux:nnnnnN
8102     #1#2#3#4#5#6
8103     {
8104         \dim_set:Nn #6
8105         {
8106             \fp_to_dim:n
8107             {
8108                 \dim_to_fp:n {#4} *
8109                 ( \dim_to_fp:n {#1} - \dim_to_fp:n {#2} ) /
8110                 \dim_to_fp:n {#5}
8111                 + \dim_to_fp:n {#3}
8112             }
8113         }
8114     }
```

(End definition for __coffin_calculate_intersection:Nnn. This function is documented on page ??.)

16.6 Aligning and typesetting of coffins

`\coffin_join:NnnNnnnn`
`\coffin_join:cnNnnnnn`
`\coffin_join:Nnncnnnn`
`\coffin_join:cnncnnnn`

This command joins two coffins, using a horizontal and vertical pole from each coffin and making an offset between the two. The result is stored as the as a third coffin, which will have all of its handles reset to standard values. First, the more basic alignment function is used to get things started.

```
8115 \cs_new_protected:Npn \coffin_join:NnnNnnnn #1#2#3#4#5#6#7#8
8116 {
8117   \__coffin_align:NnnNnnnnN
8118   #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.

```
8119   \hbox_set:Nn \l__coffin_aligned_coffin
8120   {
8121     \dim_compare:nNnT { \l__coffin_offset_x_dim } < \c_zero_dim
8122     { \tex_kern:D -\l__coffin_offset_x_dim }
8123     \hbox_unpack:N \l__coffin_aligned_coffin
8124     \dim_set:Nn \l__coffin_internal_dim
8125     { \l__coffin_offset_x_dim - \box_wd:N #1 + \box_wd:N #4 }
8126     \dim_compare:nNnT \l__coffin_internal_dim < \c_zero_dim
8127     { \tex_kern:D -\l__coffin_internal_dim }
8128   }
```

The coffin structure is reset, and the corners are cleared: only those from the two parent coffins are needed.

```
8129   \__coffin_reset_structure:N \l__coffin_aligned_coffin
8130   \prop_clear:c
8131   { \l__coffin_corners_ \__int_value:w \l__coffin_aligned_coffin _ prop }
8132   \__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.

```
8133   \dim_compare:nNnTF \l__coffin_offset_x_dim < \c_zero_dim
8134   {
8135     \__coffin_offset_poles:Nnn #1 { -\l__coffin_offset_x_dim } { 0 pt }
8136     \__coffin_offset_poles:Nnn #4 { 0 pt } { \l__coffin_offset_y_dim }
8137     \__coffin_offset_corners:Nnn #1 { -\l__coffin_offset_x_dim } { 0 pt }
8138     \__coffin_offset_corners:Nnn #4 { 0 pt } { \l__coffin_offset_y_dim }
8139   }
8140   {
8141     \__coffin_offset_poles:Nnn #1 { 0 pt } { 0 pt }
8142     \__coffin_offset_poles:Nnn #4
8143     { \l__coffin_offset_x_dim } { \l__coffin_offset_y_dim }
8144     \__coffin_offset_corners:Nnn #1 { 0 pt } { 0 pt }
8145     \__coffin_offset_corners:Nnn #4
8146     { \l__coffin_offset_x_dim } { \l__coffin_offset_y_dim }
```

```

8147     }
8148     \__coffin_update_vertical_poles:NNN #1 #4 \l__coffin_aligned_coffin
8149     \coffin_set_eq:NN #1 \l__coffin_aligned_coffin
8150   }
8151   \cs_generate_variant:Nn \coffin_join:NnnNnnnn { c , Nnnc , cnnc }

```

(End definition for \coffin_join:NnnNnnnn and others. These functions are documented on page 157.)

\coffin_attach:NnnNnnnn A more simple version of the above, as it simply uses the size of the first coffin for the new one. This means that the work here is rather simplified compared to the above code.

\coffin_attach:cnNnnnnn The function used when marking a position is hear also as it is similar but without the structure updates.

\coffin_attach:Nnncnnnn

\coffin_attach_mark:NnnNnnnn

```

8152   \cs_new_protected:Npn \coffin_attach:NnnNnnnn #1#2#3#4#5#6#7#8
8153   {
8154     \__coffin_align:NnnNnnnnN
8155     #1 {#2} {#3} #4 {#5} {#6} {#7} {#8} \l__coffin_aligned_coffin
8156     \box_set_ht:Nn \l__coffin_aligned_coffin { \box_ht:N #1 }
8157     \box_set_dp:Nn \l__coffin_aligned_coffin { \box_dp:N #1 }
8158     \box_set_wd:Nn \l__coffin_aligned_coffin { \box_wd:N #1 }
8159     \__coffin_reset_structure:N \l__coffin_aligned_coffin
8160     \prop_set_eq:cc
8161     { \l__coffin_corners_ \__int_value:w \l__coffin_aligned_coffin _prop }
8162     { \l__coffin_corners_ \__int_value:w #1 _prop }
8163     \__coffin_update_poles:N \l__coffin_aligned_coffin
8164     \__coffin_offset_poles:Nnn #1 { 0 pt } { 0 pt }
8165     \__coffin_offset_poles:Nnn #4
8166     { \l__coffin_offset_x_dim } { \l__coffin_offset_y_dim }
8167     \__coffin_update_vertical_poles:NNN #1 #4 \l__coffin_aligned_coffin
8168     \coffin_set_eq:NN #1 \l__coffin_aligned_coffin
8169   }
8170   \cs_new_protected:Npn \coffin_attach_mark:NnnNnnnn #1#2#3#4#5#6#7#8
8171   {
8172     \__coffin_align:NnnNnnnnN
8173     #1 {#2} {#3} #4 {#5} {#6} {#7} {#8} \l__coffin_aligned_coffin
8174     \box_set_ht:Nn \l__coffin_aligned_coffin { \box_ht:N #1 }
8175     \box_set_dp:Nn \l__coffin_aligned_coffin { \box_dp:N #1 }
8176     \box_set_wd:Nn \l__coffin_aligned_coffin { \box_wd:N #1 }
8177     \box_set_eq:NN #1 \l__coffin_aligned_coffin
8178   }
8179   \cs_generate_variant:Nn \coffin_attach:NnnNnnnn { c , Nnnc , cnnc }

```

(End definition for \coffin_attach:NnnNnnnn and others. These functions are documented on page 157.)

__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.

```

8180 \cs_new_protected:Npn \__coffin_align:NnnNnnnnN #1#2#3#4#5#6#7#8#9
8181 {
8182   \__coffin_calculate_intersection:Nnn #4 {#5} {#6}
8183   \dim_set:Nn \l__coffin_x_prime_dim { \l__coffin_x_dim }
8184   \dim_set:Nn \l__coffin_y_prime_dim { \l__coffin_y_dim }
8185   \__coffin_calculate_intersection:Nnn #1 {#2} {#3}
8186   \dim_set:Nn \l__coffin_offset_x_dim
8187     { \l__coffin_x_dim - \l__coffin_x_prime_dim + #7 }
8188   \dim_set:Nn \l__coffin_offset_y_dim
8189     { \l__coffin_y_dim - \l__coffin_y_prime_dim + #8 }
8190   \hbox_set:Nn \l__coffin_aligned_internal_coffin
8191     {
8192       \box_use:N #1
8193       \tex_kern:D -\box_wd:N #1
8194       \tex_kern:D \l__coffin_offset_x_dim
8195       \box_move_up:nn { \l__coffin_offset_y_dim } { \box_use:N #4 }
8196     }
8197   \coffin_set_eq:NN #9 \l__coffin_aligned_internal_coffin
8198 }

```

(End definition for __coffin_align:NnnNnnnnN. This function is documented on page ??.)

__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.

```

8199 \cs_new_protected:Npn \__coffin_offset_poles:Nnn #1#2#3
8200 {
8201   \prop_map_inline:cn { l__coffin_poles_ \__int_value:w #1 _prop }
8202     { \__coffin_offset_pole:Nnnnnnn #1 {##1} ##2 {#2} {#3} }
8203 }
8204 \cs_new_protected:Npn \__coffin_offset_pole:Nnnnnnn #1#2#3#4#5#6#7#8
8205 {
8206   \dim_set:Nn \l__coffin_x_dim { #3 + #7 }
8207   \dim_set:Nn \l__coffin_y_dim { #4 + #8 }
8208   \tl_if_in:nnTF {#2} { - }
8209     { \tl_set:Nn \l__coffin_internal_tl { {#2} } }
8210     { \tl_set:Nn \l__coffin_internal_tl { { #1 - #2 } } }
8211   \exp_last_unbraced:NNo \__coffin_set_pole:Nnx \l__coffin_aligned_coffin
8212     { \l__coffin_internal_tl }
8213     {
8214       { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim }
8215       {#5} {#6}
8216     }

```

8217 }

(End definition for `_coffin_offset_poles:Nnn`. This function is documented on page ??.)

`_coffin_offset_corners:Nnn` 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.

`_coffin_offset_corner:Nnnnn`

```

8218 \cs_new_protected:Npn \_coffin_offset_corners:Nnn #1#2#3
8219 {
8220   \prop_map_inline:cn { l\_coffin_corners\_int_value:w #1\_prop }
8221   { \_coffin_offset_corner:Nnnnn #1 {##1} ##2 {#2} {#3} }
8222 }
8223 \cs_new_protected:Npn \_coffin_offset_corner:Nnnnn #1#2#3#4#5#6
8224 {
8225   \prop_put:cnx
8226   { l\_coffin_corners\_int_value:w \_coffin_aligned_coffin\_prop }
8227   { #1 - #2 }
8228   {
8229     { \dim_eval:n { #3 + #5 } }
8230     { \dim_eval:n { #4 + #6 } }
8231   }
8232 }
```

(End definition for `_coffin_offset_corners:Nnn`. This function is documented on page ??.)

`_coffin_update_vertical_poles:NNN` 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.

`_coffin_update_T:nnnnnnnnN`

`_coffin_update_B:nnnnnnnnN`

```

8233 \cs_new_protected:Npn \_coffin_update_vertical_poles:NNN #1#2#3
8234 {
8235   \_coffin_get_pole:NnN #3 { #1 -T } \_coffin_pole_a_tl
8236   \_coffin_get_pole:NnN #3 { #2 -T } \_coffin_pole_b_tl
8237   \exp_last_two_unbraced:Noo \_coffin_update_T:nnnnnnnnN
8238   \_coffin_pole_a_tl \_coffin_pole_b_tl #3
8239   \_coffin_get_pole:NnN #3 { #1 -B } \_coffin_pole_a_tl
8240   \_coffin_get_pole:NnN #3 { #2 -B } \_coffin_pole_b_tl
8241   \exp_last_two_unbraced:Noo \_coffin_update_B:nnnnnnnnN
8242   \_coffin_pole_a_tl \_coffin_pole_b_tl #3
8243 }
8244 \cs_new_protected:Npn \_coffin_update_T:nnnnnnnnN #1#2#3#4#5#6#7#8#9
8245 {
8246   \dim_compare:nNnTF {#2} < {#6}
8247   {
8248     \_coffin_set_pole:Nnx #9 { T }
8249     { { 0 pt } {#6} { 1000 pt } { 0 pt } }
8250   }
8251   {
8252     \_coffin_set_pole:Nnx #9 { T }
8253     { { 0 pt } {#2} { 1000 pt } { 0 pt } }
8254   }
8255 }
```

```

8256 \cs_new_protected:Npn \__coffin_update_B:nnnnnnnnN #1#2#3#4#5#6#7#8#9
8257 {
8258   \dim_compare:nNnTF {#2} < {#6}
8259   {
8260     \__coffin_set_pole:Nnx #9 { B }
8261     { { 0 pt } {#2} { 1000 pt } { 0 pt } }
8262   }
8263   {
8264     \__coffin_set_pole:Nnx #9 { B }
8265     { { 0 pt } {#6} { 1000 pt } { 0 pt } }
8266   }
8267 }

```

(End definition for `__coffin_update_vertical_poles:NNN`. This function is documented on page ??.)

`\coffin_typeset:Nnnnn` Typesetting a coffin means aligning it with the current position, which is done using a coffin with no content at all. As well as aligning to the empty coffin, there is also a need to leave vertical mode, if necessary.

`\coffin_typeset:cnnnn`

```

8268 \cs_new_protected:Npn \coffin_typeset:Nnnnn #1#2#3#4#5
8269 {
8270   \hbox_unpack:N \c_empty_box
8271   \__coffin_align:NnnNnnnnN \c_empty_coffin { H } { 1 }
8272   #1 {#2} {#3} {#4} {#5} \l__coffin_aligned_coffin
8273   \box_use:N \l__coffin_aligned_coffin
8274 }
8275 \cs_generate_variant:Nn \coffin_typeset:Nnnnn { c }

```

(End definition for `\coffin_typeset:Nnnnn` and `\coffin_typeset:cnnnn`. These functions are documented on page 157.)

16.7 Coffin diagnostics

`\l__coffin_display_coffin` Used for printing coffins with data structures attached.

```

\l__coffin_display_coord_coffin 8276 \coffin_new:N \l__coffin_display_coffin
\l__coffin_display_pole_coffin 8277 \coffin_new:N \l__coffin_display_coord_coffin
8278 \coffin_new:N \l__coffin_display_pole_coffin

```

(End definition for `\l__coffin_display_coffin`. This variable is documented on page ??.)

`\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.

```

8279 \prop_new:N \l__coffin_display_handles_prop
8280 \prop_put:Nnn \l__coffin_display_handles_prop { tl }
8281 { { b } { r } { -1 } { 1 } }
8282 \prop_put:Nnn \l__coffin_display_handles_prop { thc }
8283 { { b } { hc } { 0 } { 1 } }
8284 \prop_put:Nnn \l__coffin_display_handles_prop { tr }
8285 { { b } { l } { 1 } { 1 } }
8286 \prop_put:Nnn \l__coffin_display_handles_prop { vcl }
8287 { { vc } { r } { -1 } { 0 } }

```

```

8288 \prop_put:Nnn \l__coffin_display_handles_prop { vhc }
8289 { { vc } { hc } { 0 } { 0 } }
8290 \prop_put:Nnn \l__coffin_display_handles_prop { vcr }
8291 { { vc } { l } { 1 } { 0 } }
8292 \prop_put:Nnn \l__coffin_display_handles_prop { bl }
8293 { { t } { r } { -1 } { -1 } }
8294 \prop_put:Nnn \l__coffin_display_handles_prop { bhc }
8295 { { t } { hc } { 0 } { -1 } }
8296 \prop_put:Nnn \l__coffin_display_handles_prop { br }
8297 { { t } { l } { 1 } { -1 } }
8298 \prop_put:Nnn \l__coffin_display_handles_prop { Tl }
8299 { { t } { r } { -1 } { -1 } }
8300 \prop_put:Nnn \l__coffin_display_handles_prop { Thc }
8301 { { t } { hc } { 0 } { -1 } }
8302 \prop_put:Nnn \l__coffin_display_handles_prop { Tr }
8303 { { t } { l } { 1 } { -1 } }
8304 \prop_put:Nnn \l__coffin_display_handles_prop { Hl }
8305 { { vc } { r } { -1 } { 1 } }
8306 \prop_put:Nnn \l__coffin_display_handles_prop { Hhc }
8307 { { vc } { hc } { 0 } { 1 } }
8308 \prop_put:Nnn \l__coffin_display_handles_prop { Hr }
8309 { { vc } { l } { 1 } { 1 } }
8310 \prop_put:Nnn \l__coffin_display_handles_prop { Bl }
8311 { { b } { r } { -1 } { -1 } }
8312 \prop_put:Nnn \l__coffin_display_handles_prop { Bhc }
8313 { { b } { hc } { 0 } { -1 } }
8314 \prop_put:Nnn \l__coffin_display_handles_prop { Br }
8315 { { b } { l } { 1 } { -1 } }

```

(End definition for \l__coffin_display_handles_prop. This variable is documented on page ??.)

`\l__coffin_display_offset_dim` The standard offset for the label from the handle position when displaying handles.

```

8316 \dim_new:N \l__coffin_display_offset_dim
8317 \dim_set:Nn \l__coffin_display_offset_dim { 2 pt }

```

(End definition for \l__coffin_display_offset_dim. This variable is documented on page ??.)

`\l__coffin_display_x_dim` `\l__coffin_display_y_dim` As the intersections of poles have to be calculated to find which ones to print, there is a need to avoid repetition. This is done by saving the intersection into two dedicated values.

```

8318 \dim_new:N \l__coffin_display_x_dim
8319 \dim_new:N \l__coffin_display_y_dim

```

(End definition for \l__coffin_display_x_dim. This variable is documented on page ??.)

`\l__coffin_display_poles_prop` A property list for printing poles: various things need to be deleted from this to get a “nice” output.

```

8320 \prop_new:N \l__coffin_display_poles_prop

```

(End definition for \l__coffin_display_poles_prop. This variable is documented on page ??.)

`\l__coffin_display_font_tl` Stores the settings used to print coffin data: this keeps things flexible.

```

8321 \tl_new:N \l__coffin_display_font_tl
8322 <*initex>
8323 \tl_set:Nn \l__coffin_display_font_tl { } % TODO
8324 </initex>
8325 <*package>
8326 \tl_set:Nn \l__coffin_display_font_tl { \sffamily \tiny }
8327 </package>

```

(End definition for `\l__coffin_display_font_tl`. This variable is documented on page ??.)

`\coffin_mark_handle:Nnnn` Marking a single handle is relatively easy. The standard attachment function is used, meaning that there are two calculations for the location. However, this is likely to be okay given the load expected. Contrast with the more optimised version for showing all handles which comes next.

`\coffin_mark_handle:cnnn`

`_coffin_mark_handle_aux:nnnnNnn`

```

8328 \cs_new_protected:Npn \coffin_mark_handle:Nnnn #1#2#3#4
8329 {
8330   \hcoffin_set:Nn \l__coffin_display_pole_coffin
8331   {
8332     <*initex>
8333     \hbox:n { \tex_vrule:D width 1 pt height 1 pt \scan_stop: } % TODO
8334     </initex>
8335     <*package>
8336     \color {#4}
8337     \rule { 1 pt } { 1 pt }
8338     </package>
8339   }
8340   \coffin_attach_mark:NnnNnnnn #1 {#2} {#3}
8341   \l__coffin_display_pole_coffin { hc } { vc } { 0 pt } { 0 pt }
8342   \hcoffin_set:Nn \l__coffin_display_coord_coffin
8343   {
8344     <*initex>
8345     % TODO
8346     </initex>
8347     <*package>
8348     \color {#4}
8349     </package>
8350     \l__coffin_display_font_tl
8351     ( \tl_to_str:n { #2 , #3 } )
8352   }
8353   \prop_get:NnN \l__coffin_display_handles_prop
8354   { #2 #3 } \l__coffin_internal_tl
8355   \quark_if_no_value:NTF \l__coffin_internal_tl
8356   {
8357     \prop_get:NnN \l__coffin_display_handles_prop
8358     { #3 #2 } \l__coffin_internal_tl
8359     \quark_if_no_value:NTF \l__coffin_internal_tl
8360     {
8361       \coffin_attach_mark:NnnNnnnn #1 {#2} {#3}

```

```

8362         \l__coffin_display_coord_coffin { l } { vc }
8363         { 1 pt } { 0 pt }
8364     }
8365     {
8366         \exp_last_unbraced:No \__coffin_mark_handle_aux:nnnnNnn
8367         \l__coffin_internal_tl #1 {#2} {#3}
8368     }
8369 }
8370 {
8371     \exp_last_unbraced:No \__coffin_mark_handle_aux:nnnnNnn
8372     \l__coffin_internal_tl #1 {#2} {#3}
8373 }
8374 }
8375 \cs_new_protected:Npn \__coffin_mark_handle_aux:nnnnNnn #1#2#3#4#5#6#7
8376 {
8377     \coffin_attach_mark:NnnNnnnn #5 {#6} {#7}
8378     \l__coffin_display_coord_coffin {#1} {#2}
8379     { #3 \l__coffin_display_offset_dim }
8380     { #4 \l__coffin_display_offset_dim }
8381 }
8382 \cs_generate_variant:Nn \coffin_mark_handle:Nnnn { c }

```

(End definition for \coffin_mark_handle:Nnnn and \coffin_mark_handle:cnnn. These functions are documented on page 158.)

\coffin_display_handles:Nn
\coffin_display_handles:cn
 __coffin_display_handles_aux:nnnnnn
 __coffin_display_handles_aux:nnnn
 __coffin_display_attach:Nnnnn

Printing the poles starts by removing any duplicates, for which the H poles is used as the definitive version for the baseline and bottom. Two loops are then used to find the combinations of handles for all of these poles. This is done such that poles are removed during the loops to avoid duplication.

```

8383 \cs_new_protected:Npn \coffin_display_handles:Nn #1#2
8384 {
8385     \hcoffin_set:Nn \l__coffin_display_pole_coffin
8386     {
8387     <*initex>
8388         \hbox:n { \tex_vrule:D width 1 pt height 1 pt \scan_stop: } % TODO
8389     </initex>
8390     <*package>
8391         \color {#2}
8392         \rule { 1 pt } { 1 pt }
8393     </package>
8394     }
8395     \prop_set_eq:Nc \l__coffin_display_poles_prop
8396     { l__coffin_poles_ \__int_value:w #1 _prop }
8397     \__coffin_get_pole:NnN #1 { H } \l__coffin_pole_a_tl
8398     \__coffin_get_pole:NnN #1 { T } \l__coffin_pole_b_tl
8399     \tl_if_eq:NNT \l__coffin_pole_a_tl \l__coffin_pole_b_tl
8400     { \prop_remove:Nn \l__coffin_display_poles_prop { T } }
8401     \__coffin_get_pole:NnN #1 { B } \l__coffin_pole_b_tl
8402     \tl_if_eq:NNT \l__coffin_pole_a_tl \l__coffin_pole_b_tl
8403     { \prop_remove:Nn \l__coffin_display_poles_prop { B } }

```

```

8404 \coffin_set_eq:Nn \l__coffin_display_coffin #1
8405 \prop_map_inline:Nn \l__coffin_display_poles_prop
8406 {
8407   \prop_remove:Nn \l__coffin_display_poles_prop {##1}
8408   \__coffin_display_handles_aux:nnnnnn {##1} ##2 {#2}
8409 }
8410 \box_use:N \l__coffin_display_coffin
8411 }

```

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.

```

8412 \cs_new_protected:Npn \__coffin_display_handles_aux:nnnnnn #1#2#3#4#5#6
8413 {
8414   \prop_map_inline:Nn \l__coffin_display_poles_prop
8415   {
8416     \bool_set_false:N \l__coffin_error_bool
8417     \__coffin_calculate_intersection:nnnnnnnn {#2} {#3} {#4} {#5} ##2
8418     \bool_if:NF \l__coffin_error_bool
8419     {
8420       \dim_set:Nn \l__coffin_display_x_dim { \l__coffin_x_dim }
8421       \dim_set:Nn \l__coffin_display_y_dim { \l__coffin_y_dim }
8422       \__coffin_display_attach:Nnnnn
8423       \l__coffin_display_pole_coffin { hc } { vc }
8424       { 0 pt } { 0 pt }
8425       \hcoffin_set:Nn \l__coffin_display_coord_coffin
8426       {
8427         \*initex
8428         % TODO
8429         \*initex
8430         \*package
8431         \color {#6}
8432         \*package
8433         \l__coffin_display_font_tl
8434         ( \tl_to_str:n { #1 , ##1 } )
8435       }
8436       \prop_get:NnN \l__coffin_display_handles_prop
8437       { #1 ##1 } \l__coffin_internal_tl
8438       \quark_if_no_value:NTF \l__coffin_internal_tl
8439       {
8440         \prop_get:NnN \l__coffin_display_handles_prop
8441         { ##1 #1 } \l__coffin_internal_tl
8442         \quark_if_no_value:NTF \l__coffin_internal_tl
8443         {
8444           \__coffin_display_attach:Nnnnn
8445           \l__coffin_display_coord_coffin { 1 } { vc }
8446           { 1 pt } { 0 pt }
8447         }
8448       }
8449       \exp_last_unbraced:No

```

```

8450         \__coffin_display_handles_aux:nnnn
8451         \l__coffin_internal_tl
8452     }
8453 }
8454 {
8455     \exp_last_unbraced:No \__coffin_display_handles_aux:nnnn
8456     \l__coffin_internal_tl
8457 }
8458 }
8459 }
8460 }
8461 \cs_new_protected:Npn \__coffin_display_handles_aux:nnnn #1#2#3#4
8462 {
8463     \__coffin_display_attach:Nnnnn
8464     \l__coffin_display_coord_coffin {#1} {#2}
8465     { #3 \l__coffin_display_offset_dim }
8466     { #4 \l__coffin_display_offset_dim }
8467 }
8468 \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.

```

8469 \cs_new_protected:Npn \__coffin_display_attach:Nnnnn #1#2#3#4#5
8470 {
8471     \__coffin_calculate_intersection:Nnn #1 {#2} {#3}
8472     \dim_set:Nn \l__coffin_x_prime_dim { \l__coffin_x_dim }
8473     \dim_set:Nn \l__coffin_y_prime_dim { \l__coffin_y_dim }
8474     \dim_set:Nn \l__coffin_offset_x_dim
8475     { \l__coffin_display_x_dim - \l__coffin_x_prime_dim + #4 }
8476     \dim_set:Nn \l__coffin_offset_y_dim
8477     { \l__coffin_display_y_dim - \l__coffin_y_prime_dim + #5 }
8478     \hbox_set:Nn \l__coffin_aligned_coffin
8479     {
8480         \box_use:N \l__coffin_display_coffin
8481         \tex_kern:D -\box_wd:N \l__coffin_display_coffin
8482         \tex_kern:D \l__coffin_offset_x_dim
8483         \box_move_up:nn { \l__coffin_offset_y_dim } { \box_use:N #1 }
8484     }
8485     \box_set_ht:Nn \l__coffin_aligned_coffin
8486     { \box_ht:N \l__coffin_display_coffin }
8487     \box_set_dp:Nn \l__coffin_aligned_coffin
8488     { \box_dp:N \l__coffin_display_coffin }
8489     \box_set_wd:Nn \l__coffin_aligned_coffin
8490     { \box_wd:N \l__coffin_display_coffin }
8491     \box_set_eq:NN \l__coffin_display_coffin \l__coffin_aligned_coffin
8492 }

```

(End definition for `\coffin_display_handles:Nn` and `\coffin_display_handles:cn`. These functions are documented on page 158.)

`\coffin_show_structure:N` For showing the various internal structures attached to a coffin in a way that keeps things relatively readable. If there is no apparent structure then the code complains.

`\coffin_show_structure:c`

```

8493 \cs_new_protected:Npn \coffin_show_structure:N #1
8494 {
8495   \__coffin_if_exist:NT #1
8496   {
8497     \__msg_show_pre:nnxxxx { LaTeX / kernel } { show-coffin }
8498     { \token_to_str:N #1 }
8499     { \dim_eval:n { \coffin_ht:N #1 } }
8500     { \dim_eval:n { \coffin_dp:N #1 } }
8501     { \dim_eval:n { \coffin_wd:N #1 } }
8502     \__msg_show_wrap:n
8503     {
8504       \prop_map_function:cN
8505       { l__coffin_poles_ \__int_value:w #1 _prop }
8506       \__msg_show_item_unbraced:nn
8507     }
8508   }
8509 }
8510 \cs_generate_variant:Nn \coffin_show_structure:N { c }

```

(End definition for `\coffin_show_structure:N` and `\coffin_show_structure:c`. These functions are documented on page 158.)

16.8 Messages

```

8511 \__msg_kernel_new:nnnn { kernel } { no-pole-intersection }
8512 { No~intersection~between~coffin~poles. }
8513 {
8514   \c__msg_coding_error_text_tl
8515   LaTeX~was~asked~to~find~the~intersection~between~two~poles,~
8516   but~they~do~not~have~a~unique~meeting~point:~
8517   the~value~(0~pt,~0~pt)~will~be~used.
8518 }
8519 \__msg_kernel_new:nnnn { kernel } { unknown-coffin }
8520 { Unknown~coffin~'#1'. }
8521 { The~coffin~'#1'~was~never~defined. }
8522 \__msg_kernel_new:nnnn { kernel } { unknown-coffin-pole }
8523 { Pole~'#1'~unknown~for~coffin~'#2'. }
8524 {
8525   \c__msg_coding_error_text_tl
8526   LaTeX~was~asked~to~find~a~typesetting~pole~for~a~coffin,~
8527   but~either~the~coffin~does~not~exist~or~the~pole~name~is~wrong.
8528 }
8529 \__msg_kernel_new:nnn { kernel } { show-coffin }
8530 {
8531   Size~of~coffin~#1 : \\\
8532   > ~ ht~=#2 \\\
8533   > ~ dp~=#3 \\\
8534   > ~ wd~=#4 \\\

```

```

8535     Poles-of-coffin~#1 :
8536   }
8537 </initex | package>

```

17 l3color Implementation

```

8538 (*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.

```

8539 \cs_new_eq:NN \color_group_begin: \group_begin:
8540 \cs_new_protected_nopar:Npn \color_group_end:
8541   {
8542     \tex_par:D
8543     \group_end:
8544   }

```

(End definition for `\color_group_begin:` and `\color_group_end:`. These functions are documented on page 159.)

\color_ensure_current: A driver-independent wrapper for setting the foreground color to the current color “now”.

```

8545 (*initex)
8546 \cs_new_protected_nopar:Npn \color_ensure_current:
8547   { \__driver_color_ensure_current: }
8548 </initex>

```

In package mode, the driver code may not be loaded. To keep down dependencies, if there is no driver code available and no `\set@color` then color is not in use and this function can be a no-op.

```

8549 (*package)
8550 \cs_new_protected_nopar:Npn \color_ensure_current: { }
8551 \AtBeginDocument
8552   {
8553     \cs_if_exist:NTF \__driver_color_ensure_current:
8554       {
8555         \cs_set_protected_nopar:Npn \color_ensure_current:
8556           { \__driver_color_ensure_current: }
8557       }
8558     {
8559       \cs_if_exist:NT \set@color
8560       {
8561         \cs_set_protected_nopar:Npn \color_ensure_current:
8562           { \set@color }
8563       }
8564     }
8565   }
8566 </package>

```

(End definition for `\color_ensure_current:`. This function is documented on page 159.)

```

8567 </initex | package>

```

18 l3msg implementation

```

8568 <*initex | package>
8569 <@@=msg>

\l__msg_internal_tl A general scratch for the module.
8570 \tl_new:N \l__msg_internal_tl

(End definition for \l__msg_internal_tl. This variable is documented on page ??.)

```

18.1 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
8571 \tl_const:Nn \c__msg_text_prefix_tl { msg~text~>~ }
8572 \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. These variables are
documented on page ??.)

```

```

\msg_if_exist_p:nn Test whether the control sequence containing the message text exists or not.
\msg_if_exist:nnTF
8573 \prg_new_conditional:Npnn \msg_if_exist:nn #1#2 { p , T , F , TF }
8574 {
8575   \cs_if_exist:cTF { \c__msg_text_prefix_tl #1 / #2 }
8576   { \prg_return_true: } { \prg_return_false: }
8577 }

(End definition for \msg_if_exist:nnTF. This function is documented on page 161.)

```

```

\__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.

```

```

8578 \cs_new_protected:Npn \__chk_if_free_msg:nn #1#2
8579 {
8580   \msg_if_exist:nnT {#1} {#2}
8581   {
8582     \__msg_kernel_error:nnxx { kernel } { message-already-defined }
8583     {#1} {#2}
8584   }
8585 }
8586 <*package>
8587 \if_bool:N \l@expl@log@functions@bool
8588 \cs_gset_protected:Npn \__chk_if_free_msg:nn #1#2
8589 {
8590   \msg_if_exist:nnT {#1} {#2}
8591   {

```

```

8592         \_msg_kernel_error:nxxx { kernel } { message-already-defined }
8593         {#1} {#2}
8594     }
8595     \_chk_log:x { Defining~message~ #1 / #2 ~\msg_line_context: }
8596 }
8597 \fi:
8598 </package>

```

(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_new:nnn
\msg_gset:nnnn
\msg_gset:nnn
\msg_set:nnnn
\msg_set:nnn
8599 \cs_new_protected:Npn \msg_new:nnnn #1#2
8600 {
8601     \_chk_if_free_msg:nn {#1} {#2}
8602     \msg_gset:nnnn {#1} {#2}
8603 }
8604 \cs_new_protected:Npn \msg_new:nnn #1#2#3
8605 { \msg_new:nnnn {#1} {#2} {#3} { } }
8606 \cs_new_protected:Npn \msg_set:nnnn #1#2#3#4
8607 {
8608     \cs_set:cpn { \c__msg_text_prefix_tl #1 / #2 }
8609     ##1##2##3##4 {#3}
8610     \cs_set:cpn { \c__msg_more_text_prefix_tl #1 / #2 }
8611     ##1##2##3##4 {#4}
8612 }
8613 \cs_new_protected:Npn \msg_set:nnn #1#2#3
8614 { \msg_set:nnnn {#1} {#2} {#3} { } }
8615 \cs_new_protected:Npn \msg_gset:nnnn #1#2#3#4
8616 {
8617     \cs_gset:cpn { \c__msg_text_prefix_tl #1 / #2 }
8618     ##1##2##3##4 {#3}
8619     \cs_gset:cpn { \c__msg_more_text_prefix_tl #1 / #2 }
8620     ##1##2##3##4 {#4}
8621 }
8622 \cs_new_protected:Npn \msg_gset:nnn #1#2#3
8623 { \msg_gset:nnnn {#1} {#2} {#3} { } }

```

(End definition for \msg_new:nnnn and \msg_new:nnn. These functions are documented on page 160.)

18.2 Messages: support functions and text

\c__msg_coding_error_text_tl Simple pieces of text for messages.

```

\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
8624 \tl_const:Nn \c__msg_coding_error_text_tl
8625 {
8626     This-is-a-coding-error.
8627     \\ \\
8628 }
8629 \tl_const:Nn \c__msg_continue_text_tl
8630 { Type~<return>~to~continue }

```

```

8631 \tl_const:Nn \c__msg_critical_text_tl
8632 { Reading~the~current~file~'\g_file_current_name_tl'~will~stop. }
8633 \tl_const:Nn \c__msg_fatal_text_tl
8634 { This~is~a~fatal~error:~LaTeX~will~abort. }
8635 \tl_const:Nn \c__msg_help_text_tl
8636 { For~immediate~help~type~H~<return> }
8637 \tl_const:Nn \c__msg_no_info_text_tl
8638 {
8639 LaTeX~does~not~know~anything~more~about~this~error,~sorry.
8640 \c__msg_return_text_tl
8641 }
8642 \tl_const:Nn \c__msg_on_line_text_tl { on-line }
8643 \tl_const:Nn \c__msg_return_text_tl
8644 {
8645 \\ \\
8646 Try~typing~<return>~to~proceed.
8647 \\
8648 If~that~doesn't~work,~type~X~<return>~to~quit.
8649 }
8650 \tl_const:Nn \c__msg_trouble_text_tl
8651 {
8652 \\ \\
8653 More~errors~will~almost~certainly~follow: \\
8654 the~LaTeX~run~should~be~aborted.
8655 }

```

(End definition for `\c__msg_coding_error_text_tl` and others. These variables are documented on page 170.)

`\msg_line_number:` For writing the line number nicely. `\msg_line_context:` was set up earlier, so this is not new.

```

8656 \cs_new_nopar:Npn \msg_line_number: { \int_use:N \tex_inputlineno:D }
8657 \cs_gset_nopar:Npn \msg_line_context:
8658 {
8659 \c__msg_on_line_text_tl
8660 \c_space_tl
8661 \msg_line_number:
8662 }

```

(End definition for `\msg_line_number:` and `\msg_line_context:`. These functions are documented on page 161.)

18.3 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`.

```

8663 \cs_new_protected:Npn \msg_interrupt:nnn #1#2#3
8664 {

```

```

8665 \tl_if_empty:nTF {#3}
8666 {
8667   \_msg_interrupt_wrap:nn { \ \ \c__msg_no_info_text_tl }
8668   {#1 \\\ \ #2 \\\ \c__msg_continue_text_tl }
8669 }
8670 {
8671   \_msg_interrupt_wrap:nn { \ \ #3 }
8672   {#1 \\\ \ #2 \\\ \c__msg_help_text_tl }
8673 }
8674 }

```

(End definition for `\msg_interrupt:nnn`. This function is documented on page 166.)

```

\_msg_interrupt_wrap:nn
\_msg_interrupt_more_text:n

```

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.

```

8675 \cs_new_protected:Npn \_msg_interrupt_wrap:nn #1#2
8676 {
8677   \iow_wrap:nnnN {#1} { | ~ } { } \_msg_interrupt_more_text:n
8678   \iow_wrap:nnnN {#2} { ! ~ } { } \_msg_interrupt_text:n
8679 }
8680 \cs_new_protected:Npn \_msg_interrupt_more_text:n #1
8681 {
8682   \exp_args:Nx \tex_errhelp:D
8683   {
8684     |,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
8685     #1 \iow_newline:
8686     |.....
8687   }
8688 }

```

(End definition for `_msg_interrupt_wrap:nn`.)

```

\_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.

```

8689 \group_begin:
8690   \char_set_lccode:nn {'\} {'\ }
8691   \char_set_lccode:nn {'\} {'\ }
8692   \char_set_lccode:nn {'&} {'!\}
8693   \char_set_catcode_active:N \&
8694   \tex_lowercase:D
8695   {
8696     \group_end:
8697     \cs_new_protected:Npn \_msg_interrupt_text:n #1
8698     {
8699       \iow_term:x
8700       {
8701         \iow_newline:
8702         !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
8703         \iow_newline:
8704         !
8705       }
8706       \_iow_with:Nnn \tex_newlinechar:D { '\^^J }
8707       {
8708         \_iow_with:Nnn \tex_errorcontextlines:D \c_minus_one
8709         {
8710           \group_begin:
8711           \cs_set_protected_nopar:Npn &
8712           {
8713             \tex_errmessage:D
8714             {
8715               #1
8716               \use_none:n
8717               { ..... }
8718             }
8719           }
8720           \exp_after:wN
8721           \group_end:
8722           &
8723         }
8724       }
8725     }
8726   }

```

(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
`\msg_term:n` work sets things off nicely.

```

8727 \cs_new_protected:Npn \msg_log:n #1
8728 {
8729   \iow_log:n { ..... }

```

```

8730 \iow_wrap:nnnN { . ~ #1} { . ~ } { } \iow_log:n
8731 \iow_log:n { ..... }
8732 }
8733 \cs_new_protected:Npn \msg_term:n #1
8734 {
8735 \iow_term:n { ***** }
8736 \iow_wrap:nnnN { * ~ #1} { * ~ } { } \iow_term:n
8737 \iow_term:n { ***** }
8738 }

```

(End definition for `\msg_log:n`. This function is documented on page 166.)

18.4 Displaying messages

L^AT_EX is handling error messages and so the T_EX ones are disabled. This is already done by the L^AT_EX_{2 ϵ} kernel, so to avoid messing up any deliberate change by a user this is only set in format mode.

```

8739 \*initex
8740 \int_gset_eq:NN \tex_errorcontextlines:D \c_minus_one
8741 \*initex

```

`\msg_fatal_text:n` A function for issuing messages: both the text and order could in principle vary.

```

8742 \cs_new:Npn \msg_fatal_text:n #1 { Fatal~#1~error }
8743 \cs_new:Npn \msg_critical_text:n #1 { Critical~#1~error }
8744 \cs_new:Npn \msg_error_text:n #1 { #1~error }
8745 \cs_new:Npn \msg_warning_text:n #1 { #1~warning }
8746 \cs_new:Npn \msg_info_text:n #1 { #1~info }

```

(End definition for `\msg_fatal_text:n` and others. These functions are documented on page 161.)

`\msg_see_documentation_text:n` Contextual footer information. The L^AT_EX module only comprises L^AT_EX₃ code, so we refer to the L^AT_EX₃ documentation rather than simply “L^AT_EX”.

```

8747 \cs_new:Npn \msg_see_documentation_text:n #1
8748 {
8749 \\\ \ See-the~
8750 \str_if_eq:nnTF {#1} { LaTeX } { LaTeX3 } {#1} ~
8751 documentation~for~further~information.
8752 }

```

(End definition for `\msg_see_documentation_text:n`. This function is documented on page 162.)

`__msg_class_new:nn`

```

8753 \group_begin:
8754 \cs_set_protected:Npn \__msg_class_new:nn #1#2
8755 {
8756 \prop_new:c { l__msg_redirect_ #1 _prop }
8757 \cs_new_protected:cpn { __msg_ #1 _code:nnnnnn }
8758 ##1##2##3##4##5##6 {#2}
8759 \cs_new_protected:cpn { msg_ #1 :nnnnnn } ##1##2##3##4##5##6
8760 {

```

```

8761         \use:x
8762         {
8763             \exp_not:n { \_msg_use:nnnnnn {#1} {##1} {##2} }
8764             { \tl_to_str:n {##3} } { \tl_to_str:n {##4} }
8765             { \tl_to_str:n {##5} } { \tl_to_str:n {##6} }
8766         }
8767     }
8768     \cs_new_protected:cpx { msg_ #1 :nnnnn } ##1##2##3##4##5
8769     { \exp_not:c { msg_ #1 :nnnnnn } {##1} {##2} {##3} {##4} {##5} { } }
8770     \cs_new_protected:cpx { msg_ #1 :nnnn } ##1##2##3##4
8771     { \exp_not:c { msg_ #1 :nnnnnn } {##1} {##2} {##3} {##4} { } { } }
8772     \cs_new_protected:cpx { msg_ #1 :nnn } ##1##2##3
8773     { \exp_not:c { msg_ #1 :nnnnnn } {##1} {##2} {##3} { } { } { } }
8774     \cs_new_protected:cpx { msg_ #1 :nn } ##1##2
8775     { \exp_not:c { msg_ #1 :nnnnnn } {##1} {##2} { } { } { } { } }
8776     \cs_new_protected:cpx { msg_ #1 :nnxxx } ##1##2##3##4##5##6
8777     {
8778         \use:x
8779         {
8780             \exp_not:N \exp_not:n
8781             { \exp_not:c { msg_ #1 :nnnnnn } {##1} {##2} }
8782             {##3} {##4} {##5} {##6}
8783         }
8784     }
8785     \cs_new_protected:cpx { msg_ #1 :nnxxx } ##1##2##3##4##5
8786     { \exp_not:c { msg_ #1 :nnxxx } {##1} {##2} {##3} {##4} {##5} { } }
8787     \cs_new_protected:cpx { msg_ #1 :nnxx } ##1##2##3##4
8788     { \exp_not:c { msg_ #1 :nnxxx } {##1} {##2} {##3} {##4} { } { } }
8789     \cs_new_protected:cpx { msg_ #1 :nnx } ##1##2##3
8790     { \exp_not:c { msg_ #1 :nnxxx } {##1} {##2} {##3} { } { } { } }
8791 }

```

(End definition for _msg_class_new:nn. This function is documented on page ??.)

```

\msg_fatal:nnnnnn For fatal errors, after the error message TEX bails out.
\msg_fatal:nnnnnn
\msg_fatal:nnnn
\msg_fatal:nnn
\msg_fatal:nn
\msg_fatal:nnxxx
\msg_fatal:nnxxx
\msg_fatal:nnxx
\msg_fatal:nnx
8792 \_msg_class_new:nn { fatal }
8793 {
8794     \msg_interrupt:nnn
8795     { \msg_fatal_text:n {#1} : ~ "#2" }
8796     {
8797         \use:c { \c_msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6}
8798         \msg_see_documentation_text:n {#1}
8799     }
8800     { \c_msg_fatal_text_tl }
8801     \tex_end:D
8802 }

```

(End definition for \msg_fatal:nnnnnn and others. These functions are documented on page 162.)

```

\msg_critical:nnnnnn Not quite so bad: just end the current file.
\msg_critical:nnnnnn
\msg_critical:nnnn
\msg_critical:nnn
\msg_critical:nn
\msg_critical:nnxxx
\msg_critical:nnxxx
\msg_critical:nnxx
\msg_critical:nnx

```

```

8803 \_msg\_class\_new:nn { critical }
8804 {
8805   \msg\_interrupt:nnn
8806   { \msg\_critical\_text:n {#1} : ~ "#2" }
8807   {
8808     \use:c { \c\_msg\_text\_prefix\_tl #1 / #2 } {#3} {#4} {#5} {#6}
8809     \msg\_see\_documentation\_text:n {#1}
8810   }
8811   { \c\_msg\_critical\_text\_tl }
8812   \tex\_endinput:D
8813 }

```

(End definition for `\msg_critical:nnnnnn` and others. These functions are documented on page 163.)

`\msg_error:nnnnnn` 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.

```

\msg\_error:nnnnnn
\msg\_error:nnnnn
\msg\_error:nnnn
\msg\_error:nnn
\msg\_error:nn
\msg\_error:nnxxxx
\msg\_error:nnxxx
\msg\_error:nnxx
\msg\_error:nnx
\_msg\_error:cnnnnn
\_msg\_no\_more\_text:nnnn
8814 \_msg\_class\_new:nn { error }
8815 {
8816   \_msg\_error:cnnnnn
8817   { \c\_msg\_more\_text\_prefix\_tl #1 / #2 }
8818   {#3} {#4} {#5} {#6}
8819   {
8820     \msg\_interrupt:nnn
8821     { \msg\_error\_text:n {#1} : ~ "#2" }
8822     {
8823       \use:c { \c\_msg\_text\_prefix\_tl #1 / #2 } {#3} {#4} {#5} {#6}
8824       \msg\_see\_documentation\_text:n {#1}
8825     }
8826   }
8827 }
8828 \cs\_new\_protected:Npn \_msg\_error:cnnnnn #1#2#3#4#5#6
8829 {
8830   \cs\_if\_eq:cNTF {#1} \_msg\_no\_more\_text:nnnn
8831   { #6 { } }
8832   { #6 { \use:c {#1} {#2} {#3} {#4} {#5} } }
8833 }
8834 \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 163.)

`\msg_warning:nnnnnn` Warnings are printed to the terminal.

```

\msg\_warning:nnnnnn
\msg\_warning:nnnnn
\msg\_warning:nnnn
\msg\_warning:nnn
\msg\_warning:nn
\msg\_warning:nnxxxx
\msg\_warning:nnxxx
\msg\_warning:nnxx
\msg\_warning:nnx
8835 \_msg\_class\_new:nn { warning }
8836 {
8837   \msg\_term:n
8838   {
8839     \msg\_warning\_text:n {#1} : ~ "#2" \\ \\
8840     \use:c { \c\_msg\_text\_prefix\_tl #1 / #2 } {#3} {#4} {#5} {#6}
8841   }
8842 }

```

(End definition for `\msg_warning:nnnnnn` and others. These functions are documented on page 163.)

```

\msg_info:nnnnnn Information only goes into the log.
\msg_info:nnnnnn 8843 \__msg_class_new:nn { info }
\msg_info:nnnn 8844 {
\msg_info:nnnn 8845 \msg_log:n
\msg_info:nn 8846 {
\msg_info:nnxxxx 8847 \msg_info_text:n {#1} : ~ "#2" \\ \\
\msg_info:nnxxx 8848 \use:c { \c__msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6}
\msg_info:nnxx 8849 }
\msg_info:nnx 8850 }

```

(End definition for \msg_info:nnnnnn and others. These functions are documented on page 163.)

```

\msg_log:nnnnnn "Log" data is very similar to information, but with no extras added.
\msg_log:nnnnnn 8851 \__msg_class_new:nn { log }
\msg_log:nnnn 8852 {
\msg_log:nnnn 8853 \iow_wrap:nnnN
\msg_log:nn 8854 { \use:c { \c__msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6} }
\msg_log:nnxxxx 8855 { } { } \iow_log:n
\msg_log:nnxxx 8856 }

```

(End definition for \msg_log:nnnnnn and others. These functions are documented on page 164.)

```

\msg_none:nnnnnn The none message type is needed so that input can be gobbled.
\msg_none:nnnnnn 8857 \__msg_class_new:nn { none } { }
\msg_none:nnnn 8858
\msg_none:nnnn (End definition for \msg_none:nnnnnn and others. These functions are documented on page 164.)
\msg_none:nnnn End the group to eliminate \__msg_class_new:nn.
\msg_none:nn 8858 \group_end:
\msg_none:nnxxxx
\msg_none:nnxxx
\__msg_class_chk_exist:nn
\msg_none:nnxx
\msg_none:nnx

```

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.

```

8859 \cs_new:Npn \__msg_class_chk_exist:nT #1
8860 {
8861 \cs_if_free:cTF { __msg_ #1 _code:nnnnnn }
8862 { \__msg_kernel_error:nnx { kernel } { message-class-unknown } {#1} }
8863 }

```

(End definition for __msg_class_chk_exist:nT.)

```

\l__msg_class_tl Support variables needed for the redirection system.
\l__msg_current_class_tl 8864 \tl_new:N \l__msg_class_tl
8865 \tl_new:N \l__msg_current_class_tl

```

(End definition for \l__msg_class_tl and \l__msg_current_class_tl. These variables are documented on page ??.)

```

\l__msg_redirect_prop For redirection of individually-named messages
8866 \prop_new:N \l__msg_redirect_prop

```

(End definition for \l__msg_redirect_prop. This variable is documented on page ??.)

`\l__msg_hierarchy_seq` During redirection, split the message name into a sequence with items `{/module/submodule}`, `{/module}`, and `{}`.

```
8867 \seq_new:N \l__msg_hierarchy_seq
```

(End definition for `\l__msg_hierarchy_seq`. This variable is documented on page ??.)

`\l__msg_class_loop_seq` Classes encountered when following redirections to check for loops.

```
8868 \seq_new:N \l__msg_class_loop_seq
```

(End definition for `\l__msg_class_loop_seq`. This variable is documented on page ??.)

`__msg_use:nnnnnnn` Actually using a message is a multi-step process. First, some safety checks on the message and class requested. The code and arguments are then stored to avoid passing them around. The assignment to `__msg_use_code:` is similar to `\tl_set:Nn`. The message is eventually produced with whatever `\l__msg_class_tl` is when `__msg_use_code:` is called.

```
8869 \cs_new_protected:Npn \__msg_use:nnnnnnn #1#2#3#4#5#6#7
8870 {
8871   \msg_if_exist:nnTF {#2} {#3}
8872   {
8873     \__msg_class_chk_exist:nT {#1}
8874     {
8875       \tl_set:Nn \l__msg_current_class_tl {#1}
8876       \cs_set_protected_nopar:Npx \__msg_use_code:
8877       {
8878         \exp_not:n
8879         {
8880           \use:c { __msg_ \l__msg_class_tl _code:nnnnnnn }
8881           {#2} {#3} {#4} {#5} {#6} {#7}
8882         }
8883       }
8884       \__msg_use_redirect_name:n { #2 / #3 }
8885     }
8886   }
8887   { \__msg_kernel_error:nxxx { kernel } { message-unknown } {#2} {#3} }
8888 }
8889 \cs_new_protected_nopar: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.

```
8890 \cs_new_protected:Npn \__msg_use_redirect_name:n #1
8891 {
8892   \prop_get:NnNTF \l__msg_redirect_prop { / #1 } \l__msg_class_tl
8893   { \__msg_use_code: }
8894   {
8895     \seq_clear:N \l__msg_hierarchy_seq
```

```

8896     \_msg_use_hierarchy:nwwN { }
8897     #1 \q_mark \_msg_use_hierarchy:nwwN
8898     / \q_mark \use_none_delimit_by_q_stop:w
8899     \q_stop
8900     \_msg_use_redirect_module:n { }
8901 }
8902 }
8903 \cs_new_protected:Npn \_msg_use_hierarchy:nwwN #1#2 / #3 \q_mark #4
8904 {
8905     \seq_put_left:Nn \l__msg_hierarchy_seq {#1}
8906     #4 { #1 / #2 } #3 \q_mark #4
8907 }

```

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`.

```

8908 \cs_new_protected:Npn \_msg_use_redirect_module:n #1
8909 {
8910     \seq_map_inline:Nn \l__msg_hierarchy_seq
8911     {
8912         \prop_get:cnNTF { l__msg_redirect_ \l__msg_current_class_tl _prop }
8913         {##1} \l__msg_class_tl
8914         {
8915             \seq_map_break:n
8916             {
8917                 \tl_if_eq:NNTF \l__msg_current_class_tl \l__msg_class_tl
8918                 { \_msg_use_code: }
8919                 {
8920                     \tl_set_eq:NN \l__msg_current_class_tl \l__msg_class_tl
8921                     \_msg_use_redirect_module:n {##1}
8922                 }
8923             }
8924         }
8925         {
8926             \str_if_eq:nnT {##1} {#1}
8927             {
8928                 \tl_set_eq:NN \l__msg_class_tl \l__msg_current_class_tl
8929                 \seq_map_break:n { \_msg_use_code: }
8930             }
8931         }
8932     }
8933 }

```

(End definition for `_msg_use:nnnnnnn`.)

`\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.

```

8934 \cs_new_protected:Npn \msg_redirect_name:nnn #1#2#3
8935 {
8936   \tl_if_empty:nTF {#3}
8937   { \prop_remove:Nn \l__msg_redirect_prop { / #1 / #2 } }
8938   {
8939     \__msg_class_chk_exist:nT {#3}
8940     { \prop_put:Nnn \l__msg_redirect_prop { / #1 / #2 } {#3} }
8941   }
8942 }

```

(End definition for `\msg_redirect_name:nnn`. This function is documented on page 165.)

`\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`.

`\msg_redirect_module:nnn`
`__msg_redirect:nnn`
`__msg_redirect_loop_chk:nnn`
`__msg_redirect_loop_list:n`

```

8943 \cs_new_protected_nopar:Npn \msg_redirect_class:nn
8944 { \__msg_redirect:nnn { } }
8945 \cs_new_protected:Npn \msg_redirect_module:nnn #1
8946 { \__msg_redirect:nnn { / #1 } }
8947 \cs_new_protected:Npn \__msg_redirect:nnn #1#2#3
8948 {
8949   \__msg_class_chk_exist:nT {#2}
8950   {
8951     \tl_if_empty:nTF {#3}
8952     { \prop_remove:cn { l__msg_redirect_ #2 _prop } {#1} }
8953     {
8954       \__msg_class_chk_exist:nT {#3}
8955       {
8956         \prop_put:cnn { l__msg_redirect_ #2 _prop } {#1} {#3}
8957         \tl_set:Nn \l__msg_current_class_tl {#2}
8958         \seq_clear:N \l__msg_class_loop_seq
8959         \__msg_redirect_loop_chk:nnn {#2} {#3} {#1}
8960       }
8961     }
8962   }
8963 }

```

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.

```

8964 \cs_new_protected:Npn \__msg_redirect_loop_chk:nnn #1#2#3
8965 {
8966   \seq_put_right:Nn \l__msg_class_loop_seq {#1}
8967   \prop_get:cnNT { l__msg_redirect_ #1 _prop } {#3} \l__msg_class_tl
8968   {
8969     \str_if_eq:x:nnF { \l__msg_class_tl } {#1}
8970     {
8971       \tl_if_eq:NNTF \l__msg_class_tl \l__msg_current_class_tl
8972       {
8973         \prop_put:cnn { l__msg_redirect_ #2 _prop } {#3} {#2}
8974         \__msg_kernel_warning:nnxxx
8975         { kernel } { message-redirect-loop }
8976         { \seq_item:Nn \l__msg_class_loop_seq { \c_one } }
8977         { \seq_item:Nn \l__msg_class_loop_seq { \c_two } }
8978         {#3}
8979         {
8980           \seq_map_function:NN \l__msg_class_loop_seq
8981             \__msg_redirect_loop_list:n
8982             { \seq_item:Nn \l__msg_class_loop_seq { \c_one } }
8983         }
8984       }
8985       { \__msg_redirect_loop_chk:onn \l__msg_class_tl {#2} {#3} }
8986     }
8987   }
8988 }
8989 \cs_generate_variant:Nn \__msg_redirect_loop_chk:nnn { o }
8990 \cs_new:Npn \__msg_redirect_loop_list:n #1 { {#1} ~ => ~ }

```

(End definition for `\msg_redirect_class:nn` and `\msg_redirect_module:nnn`. These functions are documented on page 165.)

18.5 Kernel-specific functions

`__msg_kernel_new:nnnn`
`__msg_kernel_new:nnn`
`__msg_kernel_set:nnnn`
`__msg_kernel_set:nnn`

The kernel needs some messages of its own. These are created using pre-built functions. Two functions are provided: one more general and one which only has the short text part.

```

8991 \cs_new_protected:Npn \__msg_kernel_new:nnnn #1#2
8992 { \msg_new:nnnn { LaTeX } { #1 / #2 } }
8993 \cs_new_protected:Npn \__msg_kernel_new:nnn #1#2
8994 { \msg_new:nnn { LaTeX } { #1 / #2 } }
8995 \cs_new_protected:Npn \__msg_kernel_set:nnnn #1#2
8996 { \msg_set:nnnn { LaTeX } { #1 / #2 } }
8997 \cs_new_protected:Npn \__msg_kernel_set:nnn #1#2
8998 { \msg_set:nnn { LaTeX } { #1 / #2 } }

```

(End definition for `__msg_kernel_new:nnnn` and `__msg_kernel_new:nnn`. These functions are documented on page 167.)

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.

(End definition for _msg_kernel_class_new:nN.)

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.

516

(End definition for `__msg_kernel_fatal:nnnnnn` and others. These functions are documented on page 167.)

```
9042 \_msg_kernel_class_new:nN { warning } \msg_warning:nnxxxx
9043 \_msg_kernel_class_new:nN { info } \msg_info:nnxxxx
```

End the group to eliminate _msg kernel class new:nN.

Error messages needed to actually implement the message system itself.

```

9047 {
9048   \c__msg_coding_error_text_tl
9049   LaTeX~was~asked~to~define~a~new~message~called~'#2'\
9050   by~the~module~'#1':~this~message~already~exists.
9051   \c__msg_return_text_tl
9052 }

```

```

9056 \c__msg_coding_error_text_tl
9057 LaTeX~was~asked~to~display~a~message~called~'~#2~\'
9058 by~the~module~'~#1~':~this~message~does~not~exist.
9059 \c__msg_return_text_tl
9060 }

```

```

9063 {
9064     LaTeX-has-been-asked-to-redirect-messages-to-a-class~'#1':\\
9065     this-was-never-defined.
9066     \c__msg_return_text_tl
9067 }

```

```

9070 Message~redirection~loop~caused~by~ {#1} ~>~ {#2}
9071 \tl_if_empty:nF {#3} { ~for~module~' \use_none:n #3 ' } .
9072 }

```

517

```

9077 \iow_indent:n { #4 \\\ }
9078 }
    Messages for earlier kernel modules.
9079 \__msg_kernel_new:nnnn { kernel } { bad-number-of-arguments }
9080 { Function~'~#1'~cannot~be~defined~with~#2~arguments. }
9081 {
9082   \c__msg_coding_error_text_tl
9083   LaTeX~has~been~asked~to~define~a~function~'~#1'~with~
9084   #2~arguments.~
9085   TeX~allows~between~0~and~9~arguments~for~a~single~function.
9086 }
9087 \__msg_kernel_new:nnnn { kernel } { command-already-defined }
9088 { Control~sequence~#1~already~defined. }
9089 {
9090   \c__msg_coding_error_text_tl
9091   LaTeX~has~been~asked~to~create~a~new~control~sequence~'~#1'~
9092   but~this~name~has~already~been~used~elsewhere. \\\
9093   The~current~meaning~is:\\
9094   \ \ #2
9095 }
9096 \__msg_kernel_new:nnnn { kernel } { command-not-defined }
9097 { Control~sequence~#1~undefined. }
9098 {
9099   \c__msg_coding_error_text_tl
9100   LaTeX~has~been~asked~to~use~a~control~sequence~'~#1':\\
9101   this~has~not~been~defined~yet.
9102 }
9103 \__msg_kernel_new:nnnn { kernel } { empty-search-pattern }
9104 { Empty~search~pattern. }
9105 {
9106   \c__msg_coding_error_text_tl
9107   LaTeX~has~been~asked~to~replace~an~empty~pattern~by~'~#1':~that~
9108   would~lead~to~an~infinite~loop!
9109 }
9110 \__msg_kernel_new:nnnn { kernel } { out-of-registers }
9111 { No~room~for~a~new~#1. }
9112 {
9113   TeX~only~supports~\int~use:N \c_max_register_int \
9114   of~each~type.~All~the~#1~registers~have~been~used.~
9115   This~run~will~be~aborted~now.
9116 }
9117 \__msg_kernel_new:nnnn { kernel } { missing-colon }
9118 { Function~'~#1'~contains~no~':'. }
9119 {
9120   \c__msg_coding_error_text_tl
9121   Code~level~functions~must~contain~': '~to~separate~the~
9122   argument~specification~from~the~function~name.~This~is~
9123   needed~when~defining~conditionals~or~variants,~or~when~building~a~
9124   parameter~text~from~the~number~of~arguments~of~the~function.

```

```

9125 }
9126 \_msg_kernel_new:nnnn { kernel } { protected-predicate }
9127 { Predicate~'#1'~must-be-expandable. }
9128 {
9129   \c_msg_coding_error_text_tl
9130   LaTeX-has-been-asked-to-define~'#1'~as-a-protected-predicate.~
9131   Only-expandable-tests-can-have-a-predicate-version.
9132 }
9133 \_msg_kernel_new:nnnn { kernel } { conditional-form-unknown }
9134 { Conditional~form~'#1'~for~function~'#2'~unknown. }
9135 {
9136   \c_msg_coding_error_text_tl
9137   LaTeX-has-been-asked-to-define-the-conditional-form~'#1'~of~
9138   the-function~'#2',~but-only~'TF',~'T',~'F',~and~'p'~forms-exist.
9139 }
9140 <*package>
9141 \bool_if:NT \l@expl@check@declarations@bool
9142 {
9143   \_msg_kernel_new:nnnn { check } { non-declared-variable }
9144   { The~variable~#1~has-not-been-declared~\msg_line_context:. }
9145   {
9146     Checking-is-active,~and-you-have-tried-do-so-something-like: \\
9147     \ \ \tl_set:Nn ~ #1 ~ \{ ~ ... ~ \} \\
9148     without~first~having: \\
9149     \ \ \tl_new:N ~ #1 \\
9150     \\
9151     LaTeX~will~create~the~variable~and~continue.
9152   }
9153 }
9154 </package>
9155 \_msg_kernel_new:nnnn { kernel } { scanmark-already-defined }
9156 { Scan-mark~#1~already~defined. }
9157 {
9158   \c_msg_coding_error_text_tl
9159   LaTeX-has-been-asked-to-create-a-new-scan-mark~'#1'~
9160   but~this~name~has~already~been~used~for~a~scan~mark.
9161 }
9162 \_msg_kernel_new:nnnn { kernel } { variable-not-defined }
9163 { Variable~#1~undefined. }
9164 {
9165   \c_msg_coding_error_text_tl
9166   LaTeX-has-been-asked-to-show-a-variable~#1,~but~this~has~not~
9167   been~defined~yet.
9168 }
9169 \_msg_kernel_new:nnnn { kernel } { variant-too-long }
9170 { Variant~form~'#1'~longer~than~base~signature~of~'#2'. }
9171 {
9172   \c_msg_coding_error_text_tl
9173   LaTeX-has-been-asked-to-create-a-variant-of~the~function~'#2'~
9174   with~a~signature~starting~with~'#1',~but~that~is~longer~than~

```

```

9175     the~signature~(part~after~the~colon)~of~'#2'.
9176   }
9177   \__msg_kernel_new:nnnn { kernel } { invalid-variant }
9178   { Variant~form~'#1'~invalid~for~base~form~'#2'. }
9179   {
9180     \c__msg_coding_error_text_tl
9181     LaTeX~has~been~asked~to~create~a~variant~of~the~function~'#2'~
9182     with~a~signature~starting~with~'#1',~but~cannot~change~an~argument~
9183     from~type~'#3'~to~type~'#4'.
9184   }

```

Some errors only appear in expandable settings, hence don't need a “more-text” argument.

```

9185   \__msg_kernel_new:nnn { kernel } { bad-variable }
9186   { Erroneous~variable~#1~used! }
9187   \__msg_kernel_new:nnn { kernel } { misused-sequence }
9188   { A~sequence~was~misused. }
9189   \__msg_kernel_new:nnn { kernel } { misused-prop }
9190   { A~property~list~was~misused. }
9191   \__msg_kernel_new:nnn { kernel } { negative-replication }
9192   { Negative~argument~for~\prg_replicate:nn. }
9193   \__msg_kernel_new:nnn { kernel } { unknown-comparison }
9194   { Relation~'#1'~unknown:~use~=>,~<,>,<=>,>=>,~!=,<=>,>=>. }
9195   \__msg_kernel_new:nnn { kernel } { zero-step }
9196   { Zero~step~size~for~step~function~#1. }

```

Messages used by the “show” functions.

```

9197   \__msg_kernel_new:nnn { kernel } { show-clist }
9198   {
9199     The~comma~list~ \tl_if_empty:nF {#1} { #1 ~ }
9200     \tl_if_empty:nTF {#2}
9201       { is~empty }
9202       { contains~the~items~(without~outer~braces): }
9203   }
9204   \__msg_kernel_new:nnn { kernel } { show-prop }
9205   {
9206     The~property~list~#1~
9207     \tl_if_empty:nTF {#2}
9208       { is~empty }
9209       { contains~the~pairs~(without~outer~braces): }
9210   }
9211   \__msg_kernel_new:nnn { kernel } { show-seq }
9212   {
9213     The~sequence~#1~
9214     \tl_if_empty:nTF {#2}
9215       { is~empty }
9216       { contains~the~items~(without~outer~braces): }
9217   }
9218   \__msg_kernel_new:nnn { kernel } { show-streams }
9219   {

```

```

9220 \tl_if_empty:nTF {#2} { No~ } { The~following~ }
9221 \str_case:nn {#1}
9222 {
9223   { ior } { input ~ }
9224   { iow } { output ~ }
9225 }
9226 streams~are~
9227 \tl_if_empty:nTF {#2} { open } { in~use: }
9228 }

```

18.6 Expandable errors

`__msg_expandable_error:n` 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`.

```

9229 \group_begin:
9230 \char_set_catcode_math_superscript:N \^
9231 \char_set_lccode:nn { '^ } { '\ }
9232 \char_set_lccode:nn { 'L } { 'L }
9233 \char_set_lccode:nn { 'T } { 'T }
9234 \char_set_lccode:nn { 'X } { 'X }
9235 \tex_lowercase:D
9236 {
9237   \cs_new:Npx \__msg_expandable_error:n #1
9238   {
9239     \exp_not:n
9240     {
9241       \exp:w
9242       \exp_after:wN \exp_after:wN
9243       \exp_after:wN \__msg_expandable_error:w
9244       \exp_after:wN \exp_after:wN
9245       \exp_after:wN \exp_end:
9246     }
9247     \exp_not:N \use:n { \exp_not:c { LaTeX3~error: } ^ #1 } ^
9248   }
9249   \cs_new:Npn \__msg_expandable_error:w #1 ^ #2 ^ { #1 }
9250 }
9251 \group_end:

```

(End definition for `__msg_expandable_error:n`.)

`_msg_kernel_expandable_error:nnnnnn`
`_msg_kernel_expandable_error:nnnnn`
`_msg_kernel_expandable_error:nnnn`
`_msg_kernel_expandable_error:nnn`
`_msg_kernel_expandable_error:nn`

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`.

```

9252 \cs_new:Npn \_msg_kernel_expandable_error:nnnnnn #1#2#3#4#5#6
9253 {
9254   \exp_args:Nf \_msg_expandable_error:n
9255   {
9256     \exp_args:Nnc \exp_after:wN \exp_stop_f:
9257     { \c_@@_text_prefix_tl LaTeX / #1 / #2 }
9258     {#3} {#4} {#5} {#6}
9259   }
9260 }
9261 \cs_new:Npn \_msg_kernel_expandable_error:nnnnn #1#2#3#4#5
9262 {
9263   \_msg_kernel_expandable_error:nnnnnn
9264   {#1} {#2} {#3} {#4} {#5} { }
9265 }
9266 \cs_new:Npn \_msg_kernel_expandable_error:nnnn #1#2#3#4
9267 {
9268   \_msg_kernel_expandable_error:nnnnnn
9269   {#1} {#2} {#3} {#4} { } { }
9270 }
9271 \cs_new:Npn \_msg_kernel_expandable_error:nnn #1#2#3
9272 {
9273   \_msg_kernel_expandable_error:nnnnnn
9274   {#1} {#2} {#3} { } { } { }
9275 }
9276 \cs_new:Npn \_msg_kernel_expandable_error:nn #1#2
9277 {
9278   \_msg_kernel_expandable_error:nnnnnn
9279   {#1} {#2} { } { } { } { }
9280 }

```

(End definition for `_msg_kernel_expandable_error:nnnnnn` and others. These functions are documented on page 168.)

18.7 Showing variables

Functions defined in this section are used for diagnostic functions in `l3clist`, `l3file`, `l3prop`, `l3seq`, `xtemplate`

```

\g__msg_log_next_bool
\_msg_log_next:
9281 \bool_new:N \g__msg_log_next_bool
9282 \cs_new_protected_nopar:Npn \_msg_log_next:
9283 { \bool_gset_true:N \g__msg_log_next_bool }

```

(End definition for `\g__msg_log_next_bool`. This variable is documented on page ??.)

```

\_msg_show_pre:nnnnnn Print the text of a message to the terminal or log file without formatting: short cuts
\_msg_show_pre:nnxxxx around \iow_wrap:nnnN. The choice of terminal or log file is done by \_msg_show_-
\_msg_show_pre:nnnnnV pre_aux:n.
\_msg_show_pre_aux:n

```

```

9284 \cs_new_protected:Npn \__msg_show_pre:nnnnnn #1#2#3#4#5#6
9285 {
9286   \exp_args:Nx \iow_wrap:nnnN
9287   {
9288     \exp_not:c { \c__msg_text_prefix_tl #1 / #2 }
9289     { \tl_to_str:n {#3} }
9290     { \tl_to_str:n {#4} }
9291     { \tl_to_str:n {#5} }
9292     { \tl_to_str:n {#6} }
9293   }
9294   { } { } \__msg_show_pre_aux:n
9295 }
9296 \cs_new_protected:Npn \__msg_show_pre:nnxxxx #1#2#3#4#5#6
9297 {
9298   \use:x
9299   { \exp_not:n { \__msg_show_pre:nnnnnn {#1} {#2} } {#3} {#4} {#5} {#6} }
9300 }
9301 \cs_generate_variant:Nn \__msg_show_pre:nnnnnn { nnnnnV }
9302 \cs_new_protected_nopar:Npn \__msg_show_pre_aux:n
9303 { \bool_if:NTF \g__msg_log_next_bool { \iow_log:n } { \iow_term:n } }

```

(End definition for `__msg_show_pre:nnnnnn`, `__msg_show_pre:nnxxxx`, and `__msg_show_pre:nnnnnV`.)

`__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_ii: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 `LaTeX/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`.

```

9304 \cs_new_protected:Npn \__msg_show_variable:NNNnn #1#2#3#4#5
9305 {
9306   #2 #1
9307   {
9308     \tl_if_empty:nF {#4}
9309     {

```

```

9310         \_msg_show_pre:nxxxxx { LaTeX / kernel } { show- #4 }
9311         { \token_to_str:N #1 } { #3 #1 { } { ? } } { } { }
9312     }
9313     \_msg_show_wrap:n {#5}
9314 }
9315 {
9316     \_msg_kernel_error:nxx { kernel } { variable-not-defined }
9317     { \token_to_str:N #1 }
9318     \bool_gset_false:N \g__msg_log_next_bool
9319 }
9320 }

```

(End definition for `_msg_show_variable:NNnn`.)

`_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:nnnN`. This is only important for expressions involving the `\currentgrouplevel` or `\currentgrouptype`. This does not lead to double expansion because the x-expansion of #1 {#2} is a string in all cases where `_msg_show_wrap:Nn` is used.

```

9321 \cs_new_protected:Npn \_msg_show_wrap:Nn #1#2
9322 { \exp_args:Nx \_msg_show_wrap:n { > ~ \tl_to_str:n {#2} = #1 {#2} } }

```

(End definition for `_msg_show_wrap:Nn`.)

`_msg_show_wrap:n` The argument of `_msg_show_wrap:n` is line-wrapped using `\iow_wrap:nnnN`. 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 T_EX, 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 -T_EX's `\showtokens`.

```

9323 \cs_new_protected:Npn \__msg_show_wrap:n #1
9324 { \iow_wrap:nnnN { #1 . } { } { } \__msg_show_wrap_aux:n }
9325 \cs_new_protected:Npn \__msg_show_wrap_aux:n #1
9326 {
9327   \tl_if_single:nTF {#1}
9328   { \tl_clear:N \l__msg_internal_tl }
9329   { \tl_set:Nf \l__msg_internal_tl { \__msg_show_wrap_aux:w #1 \q_stop } }
9330   \bool_if:NTF \g_msg_log_next_bool
9331   {
9332     \iow_log:x { > ~ \l__msg_internal_tl . }
9333     \bool_gset_false:N \g_msg_log_next_bool
9334   }
9335   {
9336     \__iow_with:Nnn \tex_newlinechar:D { 10 }
9337     {
9338       \__iow_with:Nnn \tex_errorcontextlines:D \c_minus_one
9339       {
9340         \etex_showtokens:D \exp_after:wN \exp_after:wN \exp_after:wN
9341         { \exp_after:wN \l__msg_internal_tl }
9342       }
9343     }
9344   }
9345 }
9346 \cs_new:Npn \__msg_show_wrap_aux:w #1 > #2 . \q_stop {#2}

```

(End definition for `__msg_show_wrap:n`.)

`__msg_show_item:n`
`__msg_show_item:nn`
`__msg_show_item_unbraced:nn`

Each item in the variable is formatted using one of the following functions.

```

9347 \cs_new:Npn \__msg_show_item:n #1
9348 {
9349   \\\ > \ \ \{ \tl_to_str:n {#1} \}
9350 }
9351 \cs_new:Npn \__msg_show_item:nn #1#2
9352 {
9353   \\\ > \ \ \{ \tl_to_str:n {#1} \}
9354   \ \ => \ \ \{ \tl_to_str:n {#2} \}
9355 }
9356 \cs_new:Npn \__msg_show_item_unbraced:nn #1#2
9357 {
9358   \\\ > \ \ \{ \tl_to_str:n {#1}
9359   \ \ => \ \ \{ \tl_to_str:n {#2}
9360 }

```

(End definition for `__msg_show_item:n`.)

```

9361 </initex | package>

```

19 l3keys Implementation

9362 $\langle *initex | package \rangle$

19.1 Low-level interface

9363 $\langle @@=keyval \rangle$

For historical reasons this code uses the ‘keyval’ module prefix.

$\backslash g_keyval_level_int$ To allow nesting of $\backslash keyval_parse:Nn$, an integer is needed for the current level.

9364 $\backslash int_new:N \backslash g_keyval_level_int$

(End definition for $\backslash g_keyval_level_int$. This variable is documented on page ??.)

$\backslash l_keyval_key_tl$ The current key name and value.

$\backslash l_keyval_value_tl$ 9365 $\backslash tl_new:N \backslash l_keyval_key_tl$

9366 $\backslash tl_new:N \backslash l_keyval_value_tl$

(End definition for $\backslash l_keyval_key_tl$ and $\backslash l_keyval_value_tl$. These variables are documented on page ??.)

$\backslash l_keyval_sanitise_tl$ Token list variables for dealing with awkward category codes in the input.

$\backslash l_keyval_parse_tl$ 9367 $\backslash tl_new:N \backslash l_keyval_sanitise_tl$

9368 $\backslash tl_new:N \backslash l_keyval_parse_tl$

(End definition for $\backslash l_keyval_sanitise_tl$. This variable is documented on page ??.)

$\backslash _keyval_parse:n$ The parsing function first deals with the category codes for = and , , so that there are no odd events. The input is then handed off to the element by element system.

```

9369 \group_begin:
9370   \char_set_catcode_active:n { '\ = }
9371   \char_set_catcode_active:n { '\ , }
9372   \cs_new_protected:Npx \_keyval_parse:n #1
9373   {
9374     \group_begin:
9375     \tl_set:Nn \exp_not:N \l_keyval_sanitise_tl {#1}
9376     \tl_replace_all:Nnn \exp_not:N \l_keyval_sanitise_tl
9377       { \exp_not:N = } { \token_to_str:N = }
9378     \tl_replace_all:Nnn \exp_not:N \l_keyval_sanitise_tl
9379       { \exp_not:N , } { \token_to_str:N , }
9380     \tl_clear:N \exp_not:N \l_keyval_parse_tl
9381     \exp_not:N \exp_after:wN
9382       \exp_not:N \_keyval_parse_elt:w \exp_not:N \exp_after:wN
9383     \exp_not:N \q_nil \exp_not:N \l_keyval_sanitise_tl
9384     \token_to_str:N , \exp_not:N \q_recursion_tail
9385     \token_to_str:N , \exp_not:N \q_recursion_stop
9386     \exp_not:N \exp_after:wN \group_end:
9387     \exp_not:N \l_keyval_parse_tl
9388   }
9389 \group_end:

```

(End definition for $\backslash _keyval_parse:n$. This function is documented on page ??.)

`_keyval_parse_elt:w` Each item to be parsed will have `\q_nil` added to the front. Hence the blank test here can always be used to find a totally empty argument. To allow rapid matching for an `=` while not stripping any braces, another `\q_nil` needed before the next phase of the parser. Finally, loop around for the next item, adding in the `\q_nil`: this happens whatever the nature of the current argument as the end-of-recursion will clear up in all cases.

```

9390 \cs_new_protected:Npn \_keyval_parse_elt:w #1 ,
9391 {
9392   \tl_if_blank:oF { \use_none:n #1 }
9393   {
9394     \quark_if_recursion_tail_stop:o { \use_none:n #1 }
9395     \_keyval_split_key_value:w #1 \q_nil == \q_stop
9396   }
9397   \_keyval_parse_elt:w \q_nil
9398 }

```

(End definition for `_keyval_parse_elt:w`. This function is documented on page ??.)

`_keyval_split_key_value:w` Split the key and value using a delimited argument. The `\q_nil` values added earlier ensure that no braces will be stripped as part of this process. A blank test can then be used on `#3`: it is only empty if there was no `=` in the original input. In that case, strip a `\q_nil` from the end of the key name then hand on to remove other things and store as `\l_keyval_key_tl` before adding to the output token list. In the case where there is an `=`, first tidy up the key, this time without a trailing `\q_nil`, then do a check to ensure that `#3` is exactly one token (`=`). With that done, the final stage is to hand off to tidy up the value.

```

9399 \cs_new_protected:Npn \_keyval_split_key_value:w #1 = #2 = #3 \q_stop
9400 {
9401   \tl_if_blank:nTF {#3}
9402   {
9403     \_keyval_split_key:w #1 \q_stop
9404     \tl_put_right:Nx \l_keyval_parse_tl
9405     {
9406       \exp_not:c
9407       {
9408         \_keyval_key_no_value_elt_
9409         \int_use:N \g_keyval_level_int
9410         :n
9411       }
9412       { \exp_not:o \l_keyval_key_tl }
9413     }
9414   }
9415   {
9416     \_keyval_split:Nn \l_keyval_key_tl {#1}
9417     \tl_if_blank:oTF { \use_none:n #3 }
9418     { \_keyval_split_value:w \q_nil #2 \q_stop }
9419     { \_msg_kernel_error:nn { kernel } { misplaced-equals-sign } }
9420   }
9421 }
9422 \cs_new_protected:Npn \_keyval_split_key:w #1 \q_nil \q_stop

```

```
9423 { \__keyval_split:Nn \l__keyval_key_tl {#1} }
```

(End definition for __keyval_split_key_value:w. This function is documented on page ??.)

__keyval_split:Nn There are two possible cases here. The first case is that #1 is surrounded by braces, in which case the \use_none:nnn #1 \q_nil \q_nil will yield \q_nil. There, we can remove the leading \q_nil, the braces and any spaces around the outside with \use_ii:nnn. On the other hand, if there are no braces then the second branch removes the leading \q_nil and any surrounding spaces.

__keyval_split:Nw

```
9424 \cs_new_protected:Npn \__keyval_split:Nn #1#2
9425 {
9426   \quark_if_nil:oTF { \use_none:nnn #2 \q_nil \q_nil }
9427   { \tl_set:Nx #1 { \exp_not:o { \use_ii:nnn #2 \q_nil } } }
9428   { \__keyval_split:Nw #1 #2 \q_stop }
9429 }
9430 \cs_new_protected:Npn \__keyval_split:Nw #1 \q_nil #2 \q_stop
9431 { \tl_set:Nx #1 { \tl_trim_spaces:n {#2} } }
```

(End definition for __keyval_split:Nn. This function is documented on page ??.)

__keyval_split_value:w As this stage there is just the value to deal with. The leading and trailing \q_nil tokens are removed in two steps before storing the value with spaces stripped (see __keyval_split:Nn). Doing the storage of key and value in one shot will put exactly the right number of brace groups into the output.

```
9432 \cs_new_protected:Npn \__keyval_split_value:w #1 \q_nil \q_stop
9433 {
9434   \__keyval_split:Nn \l__keyval_value_tl {#1}
9435   \tl_put_right:Nx \l__keyval_parse_tl
9436   {
9437     \exp_not:c
9438     { __keyval_key_value_elt_ \int_use:N \g__keyval_level_int :nn }
9439     { \exp_not:o \l__keyval_key_tl }
9440     { \exp_not:o \l__keyval_value_tl }
9441   }
9442 }
```

(End definition for __keyval_split_value:w. This function is documented on page ??.)

\keyval_parse:NNn The outer parsing routine just sets up the processing functions and hands off.

```
9443 \cs_new_protected:Npn \keyval_parse:NNn #1#2#3
9444 {
9445   \int_gincr:N \g__keyval_level_int
9446   \cs_gset_eq:cN
9447   { __keyval_key_no_value_elt_ \int_use:N \g__keyval_level_int :n } #1
9448   \cs_gset_eq:cN
9449   { __keyval_key_value_elt_ \int_use:N \g__keyval_level_int :nn } #2
9450   \__keyval_parse:n {#3}
9451   \int_gdecr:N \g__keyval_level_int
9452 }
```

(End definition for `\keyval_parse:Nn`. This function is documented on page 183.)

One message for the low level parsing system.

```

9453 \_msg_kernel_new:nnnn { kernel } { misplaced-equals-sign }
9454 { Misplaced-equals-sign-in-key-value-input~\msg_line_number: }
9455 {
9456   LaTeX-is-attempting-to-parse-some-key-value-input-but-found-
9457   two-equals-signs-not-separated-by-a-comma.
9458 }

```

19.2 Constants and variables

```

9459 <@@=keys>

```

`\c__keys_code_root_tl` The prefixes for the code and variables of the keys themselves.

```

\c__keys_info_root_tl 9460 \tl_const:Nn \c__keys_code_root_tl { key-code->~ }
9461 \tl_const:Nn \c__keys_info_root_tl { key-info->~ }

```

(End definition for `\c__keys_code_root_tl` and `\c__keys_info_root_tl`. These variables are documented on page ??.)

`\c__keys_props_root_tl` The prefix for storing properties.

```

9462 \tl_const:Nn \c__keys_props_root_tl { key-prop->~ }

```

(End definition for `\c__keys_props_root_tl`. This variable is documented on page ??.)

`\l_keys_choice_int` Publicly accessible data on which choice is being used when several are generated as a
`\l_keys_choice_tl` set.

```

9463 \int_new:N \l_keys_choice_int
9464 \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 177.)

`\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.

```

9465 \clist_new:N \l__keys_groups_clist

```

(End definition for `\l__keys_groups_clist`. This variable is documented on page ??.)

`\l_keys_key_tl` The name of a key itself: needed when setting keys.

```

9466 \tl_new:N \l_keys_key_tl

```

(End definition for `\l_keys_key_tl`. This variable is documented on page 179.)

`\l__keys_module_tl` The module for an entire set of keys.

```

9467 \tl_new:N \l__keys_module_tl

```

(End definition for `\l__keys_module_tl`. This variable is documented on page ??.)

`\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.

```
9468 \bool_new:N \l__keys_no_value_bool
```

(End definition for \l__keys_no_value_bool. This variable is documented on page ??.)

`\l__keys_only_known_bool` Used to track if only “known” keys are being set.

```
9469 \bool_new:N \l__keys_only_known_bool
```

(End definition for \l__keys_only_known_bool. This variable is documented on page ??.)

`\l_keys_path_tl` The “path” of the current key is stored here: this is available to the programmer and so is public.

```
9470 \tl_new:N \l_keys_path_tl
```

(End definition for \l_keys_path_tl. This variable is documented on page 179.)

`\l__keys_property_tl` The “property” begin set for a key at definition time is stored here.

```
9471 \tl_new:N \l__keys_property_tl
```

(End definition for \l__keys_property_tl. This variable is documented on page ??.)

`\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”).

```
9472 \bool_new:N \l__keys_selective_bool
9473 \bool_new:N \l__keys_filtered_bool
```

(End definition for \l__keys_selective_bool and \l__keys_filtered_bool. These variables are documented on page ??.)

`\l__keys_selective_seq` The list of key groups being filtered in or out during selective setting.

```
9474 \seq_new:N \l__keys_selective_seq
```

(End definition for \l__keys_selective_seq. This variable is documented on page ??.)

`\l__keys_unused_clist` Used when setting only some keys to store those left over.

```
9475 \tl_new:N \l__keys_unused_clist
```

(End definition for \l__keys_unused_clist. This variable is documented on page ??.)

`\l_keys_value_tl` The value given for a key: may be empty if no value was given.

```
9476 \tl_new:N \l_keys_value_tl
```

(End definition for \l_keys_value_tl. This variable is documented on page 179.)

`\l__keys_tmp_bool` Scratch space.

```
9477 \bool_new:N \l__keys_tmp_bool
```

(End definition for \l__keys_tmp_bool. This variable is documented on page ??.)

19.3 The key defining mechanism

`\keys_define:nn` The public function for definitions is just a wrapper for the lower level mechanism, more or less. The outer function is designed to keep a track of the current module, to allow safe nesting. The module is set removing any leading / (which is not needed here).

```

9478 \cs_new_protected:Npn \keys_define:nn
9479 { \__keys_define:onn \l__keys_module_tl }
9480 \cs_new_protected:Npn \__keys_define:nnn #1#2#3
9481 {
9482   \tl_set:Nx \l__keys_module_tl { \tl_to_str:n {#2} }
9483   \keyval_parse:NNn \__keys_define_elt:n \__keys_define_elt:nn {#3}
9484   \tl_set:Nn \l__keys_module_tl {#1}
9485 }
9486 \cs_generate_variant:Nn \__keys_define:nnn { o }

```

(End definition for `\keys_define:nn`. This function is documented on page 172.)

`__keys_define_elt:n` The outer functions here record whether a value was given and then converge on a common internal mechanism. There is first a search for a property in the current key name, then a check to make sure it is known before the code hands off to the next step.

```

9487 \cs_new_protected:Npn \__keys_define_elt:n #1
9488 {
9489   \bool_set_true:N \l__keys_no_value_bool
9490   \__keys_define_elt_aux:nn {#1} { }
9491 }
9492 \cs_new_protected:Npn \__keys_define_elt:nn #1#2
9493 {
9494   \bool_set_false:N \l__keys_no_value_bool
9495   \__keys_define_elt_aux:nn {#1} {#2}
9496 }
9497 \cs_new_protected:Npn \__keys_define_elt_aux:nn #1#2
9498 {
9499   \__keys_property_find:n {#1}
9500   \cs_if_exist:cTF { \c__keys_props_root_tl \l__keys_property_tl }
9501   { \__keys_define_key:n {#2} }
9502   {
9503     \str_if_eq_x:nnF { \l__keys_property_tl } { .abort: }
9504     {
9505       \__msg_kernel_error:nxxx { kernel } { property-unknown }
9506       { \l__keys_property_tl } { \l__keys_path_tl }
9507     }
9508   }
9509 }

```

(End definition for `__keys_define_elt:n`.)

`__keys_property_find:n` Searching for a property means finding the last . in the input, and storing the text before and after it. Everything is turned into strings, so there is no problem using an x-type expansion.

```

9510 \cs_new_protected:Npn \__keys_property_find:n #1

```

```

9511 {
9512     \tl_set:Nx \l_keys_path_tl { \l__keys_module_tl / }
9513     \tl_if_in:nnTF {#1} { . }
9514     { \__keys_property_find:w #1 \q_stop }
9515     {
9516         \__msg_kernel_error:nnx { kernel } { key-no-property } {#1}
9517         \tl_set:Nn \l__keys_property_tl { .abort: }
9518     }
9519 }
9520 \cs_new_protected:Npn \__keys_property_find:w #1 . #2 \q_stop
9521 {
9522     \tl_set:Nx \l_keys_path_tl { \l_keys_path_tl \tl_to_str:n {#1} }
9523     \tl_if_in:nnTF {#2} { . }
9524     {
9525         \tl_set:Nx \l_keys_path_tl { \l_keys_path_tl . }
9526         \__keys_property_find:w #2 \q_stop
9527     }
9528     { \tl_set:Nn \l__keys_property_tl { . #2 } }
9529 }

```

(End definition for __keys_property_find:n.)

__keys_define_key:n
__keys_define_key:w

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.

```

9530 \cs_new_protected:Npn \__keys_define_key:n #1
9531 {
9532     \bool_if:NTF \l__keys_no_value_bool
9533     {
9534         \exp_after:wN \__keys_define_key:w
9535         \l__keys_property_tl \q_stop
9536         { \use:c { \c__keys_props_root_tl \l__keys_property_tl } }
9537         {
9538             \__msg_kernel_error:nnxx { kernel }
9539             { property-requires-value } { \l__keys_property_tl }
9540             { \l_keys_path_tl }
9541         }
9542     }
9543     { \use:c { \c__keys_props_root_tl \l__keys_property_tl } {#1} }
9544 }
9545 \cs_new_protected:Npn \__keys_define_key:w #1 : #2 \q_stop
9546 { \tl_if_empty:nTF {#2} }

```

(End definition for __keys_define_key:n.)

19.4 Turning properties into actions

__keys_ensure_exist:n
__keys_ensure_exist:w

Used to make sure that a key implementation and the related property list will exist whenever this is required. We cannot use for example \prop_clear_new:c here as that

would affect the order in which key properties must be set. As key definitions are never global we use `\cs_set_protected:cpn` not `\cs_new_protected:cpn` here. For the same reason, to avoid issues if the key has been undefined in the current scope but exists at a higher level, we do not use `\prop_new:c` but rather `\prop_set_eq:cN`. The function `__chk_log:x` only writes to the log file if logging all new functions is active: without it keys would not show up (as we are not using `\..._new`).

```

9547 \cs_new_protected:Npn \__keys_ensure_exist:n #1
9548 {
9549   \prop_if_exist:cF { \c__keys_info_root_tl #1 }
9550   {
9551     \prop_set_eq:cN { \c__keys_info_root_tl #1 } \c_empty_prop
9552   }
9553   \cs_if_exist:cF { \c__keys_code_root_tl #1 }
9554   {
9555     \__chk_log:x { Defining~key~#1~ \msg_line_context: }
9556     \cs_set_protected:cpn { \c__keys_code_root_tl #1 } ##1 { }
9557   }
9558 }
9559 \cs_generate_variant:Nn \__keys_ensure_exist:n { V }

```

(End definition for `__keys_ensure_exist:n` and `__keys_ensure_exist:V`.)

`__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.

```

9560 \cs_new_protected:Npn \__keys_bool_set:Nn #1#2
9561 {
9562   \bool_if_exist:NF #1 { \bool_new:N #1 }
9563   \__keys_choice_make:
9564   \__keys_cmd_set:nx { \l_keys_path_tl / true }
9565   { \exp_not:c { bool_ #2 set_true:N } \exp_not:N #1 }
9566   \__keys_cmd_set:nx { \l_keys_path_tl / false }
9567   { \exp_not:c { bool_ #2 set_false:N } \exp_not:N #1 }
9568   \__keys_cmd_set:nn { \l_keys_path_tl / unknown }
9569   {
9570     \__msg_kernel_error:nnx { kernel } { boolean-values-only }
9571     { \l_keys_key_tl }
9572   }
9573   \__keys_default_set:n { true }
9574 }
9575 \cs_generate_variant:Nn \__keys_bool_set:Nn { c }

```

(End definition for `__keys_bool_set:Nn` and `__keys_bool_set:cn`.)

`__keys_bool_set_inverse:Nn` Inverse boolean setting is much the same.

```

9576 \cs_new_protected:Npn \__keys_bool_set_inverse:Nn #1#2
9577 {
9578   \bool_if_exist:NF #1 { \bool_new:N #1 }
9579   \__keys_choice_make:
9580   \__keys_cmd_set:nx { \l_keys_path_tl / true }

```

```

9581     { \exp_not:c { bool_ #2 set_false:N } \exp_not:N #1 }
9582 \__keys_cmd_set:nx { \l_keys_path_tl / false }
9583     { \exp_not:c { bool_ #2 set_true:N } \exp_not:N #1 }
9584 \__keys_cmd_set:nn { \l_keys_path_tl / unknown }
9585     {
9586         \__msg_kernel_error:nxx { kernel } { boolean-values-only }
9587         { \l_keys_key_tl }
9588     }
9589 \__keys_default_set:n { true }
9590 }
9591 \cs_generate_variant:Nn \__keys_bool_set_inverse:Nn { c }

```

(End definition for __keys_bool_set_inverse:Nn and __keys_bool_set_inverse:cn.)

__keys_choice_make: To make a choice from a key, two steps: set the code, and set the unknown key. There is one point to watch here: choice keys cannot be nested! As multichoice and choices are essentially the same bar one function, the code is given together.

```

\__keys_multichoice_make:
\__keys_choice_make:N
\__keys_choice_make_aux:N
\__keys_parent:n
\__keys_parent:o
\__keys_parent:wn
9592 \cs_new_protected_nopar:Npn \__keys_choice_make:
9593     { \__keys_choice_make:N \__keys_choice_find:n }
9594 \cs_new_protected_nopar:Npn \__keys_multichoice_make:
9595     { \__keys_choice_make:N \__keys_multichoice_find:n }
9596 \cs_new_protected_nopar:Npn \__keys_choice_make:N #1
9597     {
9598         \prop_if_exist:cTF
9599         { \c__keys_info_root_tl \__keys_parent:o \l_keys_path_tl }
9600         {
9601             \prop_get:cnNTF
9602             { \c__keys_info_root_tl \__keys_parent:o \l_keys_path_tl }
9603             { choice } \l_keys_value_tl
9604             {
9605                 \__msg_kernel_error:nxxx { kernel } { nested-choice-key }
9606                 { \l_keys_path_tl } { \__keys_parent:o \l_keys_path_tl }
9607             }
9608             { \__keys_choice_make_aux:N #1 }
9609         }
9610         { \__keys_choice_make_aux:N #1 }
9611     }
9612 \cs_new_protected_nopar:Npn \__keys_choice_make_aux:N #1
9613     {
9614         \__keys_cmd_set:nn { \l_keys_path_tl } { #1 {##1} }
9615         \prop_put:cn { \c__keys_info_root_tl \l_keys_path_tl } { choice }
9616         { true }
9617         \__keys_cmd_set:nn { \l_keys_path_tl / unknown }
9618         {
9619             \__msg_kernel_error:nxxx { kernel } { key-choice-unknown }
9620             { \l_keys_path_tl } {##1}
9621         }
9622     }
9623 \cs_new:Npn \__keys_parent:n #1
9624     { \__keys_parent:wn #1 / / \q_stop { } }

```

```

9625 \cs_generate_variant:Nn \__keys_parent:n { o }
9626 \cs_new:Npn \__keys_parent:wn #1 / #2 / #3 \q_stop #4
9627 {
9628   \tl_if_blank:nTF {#2}
9629   { \use_none:n #4 }
9630   {
9631     \__keys_parent:wn #2 / #3 \q_stop { #4 / #1 }
9632   }
9633 }

```

(End definition for __keys_choice_make: and __keys_multichoice_make:.)

__keys_choices_make:nn Auto-generating choices means setting up the root key as a choice, then defining each
 __keys_multichoices_make:nn choice in turn.

```

\__keys_choices_make:Nnn
9634 \cs_new_protected_nopar:Npn \__keys_choices_make:nn
9635 { \__keys_choices_make:Nnn \__keys_choice_make: }
9636 \cs_new_protected_nopar:Npn \__keys_multichoices_make:nn
9637 { \__keys_choices_make:Nnn \__keys_multichoice_make: }
9638 \cs_new_protected:Npn \__keys_choices_make:Nnn #1#2#3
9639 {
9640   #1
9641   \int_zero:N \l_keys_choice_int
9642   \clist_map_inline:nn {#2}
9643   {
9644     \int_incr:N \l_keys_choice_int
9645     \__keys_cmd_set:nx { \l_keys_path_tl / \tl_to_str:n {##1} }
9646     {
9647       \tl_set:Nn \exp_not:N \l_keys_choice_tl {##1}
9648       \int_set:Nn \exp_not:N \l_keys_choice_int
9649       { \int_use:N \l_keys_choice_int }
9650       \exp_not:n {#3}
9651     }
9652   }
9653 }

```

(End definition for __keys_choices_make:nn and __keys_multichoices_make:nn.)

__keys_cmd_set:nn Setting the code for a key first checks that the basic data structures exist, then saves the
 __keys_cmd_set:nx code.

```

\__keys_cmd_set:Vn
\__keys_cmd_set:Vo
9654 \cs_new_protected:Npn \__keys_cmd_set:nn #1#2
9655 {
9656   \__keys_ensure_exist:V \l_keys_path_tl
9657   \cs_set_protected:cpn { \c__keys_code_root_tl #1 } ##1 {#2}
9658 }
9659 \cs_generate_variant:Nn \__keys_cmd_set:nn { nx , Vn , Vo }

```

(End definition for __keys_cmd_set:nn and others.)

__keys_default_set:n Setting a default value is easy.

```

9660 \cs_new_protected:Npn \__keys_default_set:n #1

```

```

9661 {
9662   \__keys_ensure_exist:V \l_keys_path_tl
9663   \tl_if_empty:nTF {#1}
9664   {
9665     \prop_remove:cn { \c__keys_info_root_tl \l_keys_path_tl }
9666     { default }
9667   }
9668   {
9669     \prop_put:cnn { \c__keys_info_root_tl \l_keys_path_tl }
9670     { default } {#1}
9671   }
9672 }

```

(End definition for __keys_default_set:n.)

__keys_groups_set:n Assigning a key to one or more groups uses comma lists. So that the comma list is “well-behaved” later, the storage is done via a stored list here, which does the normalisation.

```

9673 \cs_new_protected:Npn \__keys_groups_set:n #1
9674 {
9675   \__keys_ensure_exist:V \l_keys_path_tl
9676   \clist_set:Nn \l__keys_groups_clist {#1}
9677   \clist_if_empty:NTF \l__keys_groups_clist
9678   {
9679     \prop_remove:cn { \c__keys_info_root_tl \l_keys_path_tl }
9680     { groups }
9681   }
9682   {
9683     \prop_put:cnV { \c__keys_info_root_tl \l_keys_path_tl }
9684     { groups } \l__keys_groups_clist
9685   }
9686 }

```

(End definition for __keys_groups_set:n.)

__keys_initialise:n A set up for initialisation from which the key system requires that the path is split up into a module and a key name. At this stage, \l_keys_path_tl will contain / so a split is easy to do.

```

9687 \cs_new_protected:Npn \__keys_initialise:n #1
9688 {
9689   \__keys_ensure_exist:V \l_keys_path_tl
9690   \exp_after:wN \__keys_initialise:wn \l_keys_path_tl \q_stop {#1}
9691 }
9692 \cs_new_protected:Npn \__keys_initialise:wn #1 / #2 \q_stop #3
9693 { \keys_set:nn {#1} { #2 = {#3} } }

```

(End definition for __keys_initialise:n.)

__keys_meta_make:n To create a meta-key, simply set up to pass data through.

```

9694 \cs_new_protected:Npn \__keys_meta_make:n #1
9695 {

```

```

9696 \__keys_cmd_set:Vo \l_keys_path_tl
9697 {
9698   \exp_after:wN \keys_set:nn
9699   \exp_after:wN { \l__keys_module_tl } {#1}
9700 }
9701 }
9702 \cs_new_protected:Npn \__keys_meta_make:nn #1#2
9703 { \__keys_cmd_set:Vn \l_keys_path_tl { \keys_set:nn {#1} {#2} } }

```

(End definition for __keys_meta_make:n.)

__keys_undefine: Undefining a key has to be done without \cs_undefine:c as that function acts globally.

```

9704 \cs_new_protected_nopar:Npn \__keys_undefine:
9705 {
9706   \cs_set_eq:cN { \c__keys_code_root_tl \l_keys_path_tl } \tex_undefined:D
9707   \cs_set_eq:cN { \c__keys_info_root_tl \l_keys_path_tl } \tex_undefined:D
9708 }

```

(End definition for __keys_undefine:.)

__keys_value_requirement:nn Values can be required or forbidden by having the appropriate marker set. First, both the required and forbidden ones are clear, just in case!

```

9709 \cs_new_protected:Npn \__keys_value_requirement:nn #1#2
9710 {
9711   \__keys_ensure_exist:V \l_keys_path_tl
9712   \prop_remove:cn { \c__keys_info_root_tl \l_keys_path_tl }
9713     { required }
9714   \prop_remove:cn { \c__keys_info_root_tl \l_keys_path_tl }
9715     { forbidden }
9716   \str_if_eq:nnTF {#2} { true }
9717   {
9718     \prop_put:cnn { \c__keys_info_root_tl \l_keys_path_tl }
9719       {#1} { true }
9720   }
9721   {
9722     \str_if_eq:nnF {#2} { false }
9723     {
9724       \__msg_kernel_error:nxx { kernel } { property-boolean-values-only }
9725       { .value_ #1 :n }
9726     }
9727   }
9728 }

```

(End definition for __keys_value_requirement:nn.)

__keys_variable_set:NnnN Setting a variable takes the type and scope separately so that it is easy to make a new variable if needed.

```

9729 \cs_new_protected:Npn \__keys_variable_set:NnnN #1#2#3#4
9730 {
9731   \use:c { #2_if_exist:Nf } #1 { \use:c { #2_new:N } #1 }

```

```

9732     \_keys_cmd_set:nx { \l_keys_path_tl }
9733     {
9734         \exp_not:c { #2 _ #3 set:N #4 }
9735         \exp_not:N #1
9736         \exp_not:n { {##1} }
9737     }
9738 }
9739 \cs_generate_variant:Nn \_keys_variable_set:NnnN { c }

```

(End definition for `_keys_variable_set:NnnN` and `_keys_variable_set:cnnN`.)

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.
.bool_set:c 9740 \cs_new_protected:cpn { \c__keys_props_root_tl .bool_set:N } #1
          { \_keys_bool_set:Nn #1 { } }
.bool_gset:N 9741 { \_keys_bool_set:Nn #1 { } }
.bool_gset:c 9742 \cs_new_protected:cpn { \c__keys_props_root_tl .bool_set:c } #1
          { \_keys_bool_set:cn {#1} { } }
          9743 { \_keys_bool_set:cn {#1} { } }
          9744 \cs_new_protected:cpn { \c__keys_props_root_tl .bool_gset:N } #1
          { \_keys_bool_set:Nn #1 { g } }
          9745 { \_keys_bool_set:Nn #1 { g } }
          9746 \cs_new_protected:cpn { \c__keys_props_root_tl .bool_gset:c } #1
          { \_keys_bool_set:cn {#1} { g } }
          9747 { \_keys_bool_set:cn {#1} { g } }

```

(End definition for `.bool_set:N` and `.bool_set:c`. These functions are documented on page 173.)

```

.bool_set_inverse:N One function for this.
.bool_set_inverse:c 9748 \cs_new_protected:cpn { \c__keys_props_root_tl .bool_set_inverse:N } #1
          { \_keys_bool_set_inverse:Nn #1 { } }
.bool_gset_inverse:N 9749 { \_keys_bool_set_inverse:Nn #1 { } }
.bool_gset_inverse:c 9750 \cs_new_protected:cpn { \c__keys_props_root_tl .bool_set_inverse:c } #1
          { \_keys_bool_set_inverse:cn {#1} { } }
          9751 { \_keys_bool_set_inverse:cn {#1} { } }
          9752 \cs_new_protected:cpn { \c__keys_props_root_tl .bool_gset_inverse:N } #1
          { \_keys_bool_set_inverse:Nn #1 { g } }
          9753 { \_keys_bool_set_inverse:Nn #1 { g } }
          9754 \cs_new_protected:cpn { \c__keys_props_root_tl .bool_gset_inverse:c } #1
          { \_keys_bool_set_inverse:cn {#1} { g } }
          9755 { \_keys_bool_set_inverse:cn {#1} { g } }

```

(End definition for `.bool_set_inverse:N` and `.bool_set_inverse:c`. These functions are documented on page 173.)

.choice: Making a choice is handled internally, as it is also needed by `.generate_choices:n`.

```

9756 \cs_new_protected_nopar:cpn { \c__keys_props_root_tl .choice: }
9757 { \_keys_choice_make: }

```

(End definition for `.choice:.` This function is documented on page 173.)

.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.

```
.choices:Vn 9758 \cs_new_protected:cpn { \c__keys_props_root_tl .choices:nn } #1
.choices:on 9759 { \__keys_choices_make:nn #1 }
.choices:xn 9760 \cs_new_protected:cpn { \c__keys_props_root_tl .choices:Vn } #1
9761 { \exp_args:NV \__keys_choices_make:nn #1 }
9762 \cs_new_protected:cpn { \c__keys_props_root_tl .choices:on } #1
9763 { \exp_args:No \__keys_choices_make:nn #1 }
9764 \cs_new_protected:cpn { \c__keys_props_root_tl .choices:xn } #1
9765 { \exp_args:Nx \__keys_choices_make:nn #1 }
```

(End definition for .choices:nn and others. These functions are documented on page 173.)

.code:n Creating code is simply a case of passing through to the underlying set function.

```
9766 \cs_new_protected:cpn { \c__keys_props_root_tl .code:n } #1
9767 { \__keys_cmd_set:nn { \l_keys_path_tl } {#1} }
```

(End definition for .code:n. This function is documented on page 174.)

.clist_set:N

```
.clist_set:c 9768 \cs_new_protected:cpn { \c__keys_props_root_tl .clist_set:N } #1
.clist_gset:N 9769 { \__keys_variable_set:NnnN #1 { clist } { } n }
.clist_gset:c 9770 \cs_new_protected:cpn { \c__keys_props_root_tl .clist_set:c } #1
9771 { \__keys_variable_set:cnnN {#1} { clist } { } n }
9772 \cs_new_protected:cpn { \c__keys_props_root_tl .clist_gset:N } #1
9773 { \__keys_variable_set:NnnN #1 { clist } { g } n }
9774 \cs_new_protected:cpn { \c__keys_props_root_tl .clist_gset:c } #1
9775 { \__keys_variable_set:cnnN {#1} { clist } { g } n }
```

(End definition for .clist_set:N and .clist_set:c. These functions are documented on page 174.)

.default:n Expansion is left to the internal functions.

```
.default:V 9776 \cs_new_protected:cpn { \c__keys_props_root_tl .default:n } #1
.default:o 9777 { \__keys_default_set:n {#1} }
.default:x 9778 \cs_new_protected:cpn { \c__keys_props_root_tl .default:V } #1
9779 { \exp_args:NV \__keys_default_set:n #1 }
9780 \cs_new_protected:cpn { \c__keys_props_root_tl .default:o } #1
9781 { \exp_args:No \__keys_default_set:n {#1} }
9782 \cs_new_protected:cpn { \c__keys_props_root_tl .default:x } #1
9783 { \exp_args:Nx \__keys_default_set:n {#1} }
```

(End definition for .default:n and others. These functions are documented on page 174.)

.dim_set:N Setting a variable is very easy: just pass the data along.

```
.dim_set:c 9784 \cs_new_protected:cpn { \c__keys_props_root_tl .dim_set:N } #1
.dim_gset:N 9785 { \__keys_variable_set:NnnN #1 { dim } { } n }
.dim_gset:c 9786 \cs_new_protected:cpn { \c__keys_props_root_tl .dim_set:c } #1
9787 { \__keys_variable_set:cnnN {#1} { dim } { } n }
9788 \cs_new_protected:cpn { \c__keys_props_root_tl .dim_gset:N } #1
9789 { \__keys_variable_set:NnnN #1 { dim } { g } n }
9790 \cs_new_protected:cpn { \c__keys_props_root_tl .dim_gset:c } #1
9791 { \__keys_variable_set:cnnN {#1} { dim } { g } n }
```

(End definition for `.dim_set:N` and `.dim_set:c`. These functions are documented on page 174.)

```
.fp_set:N Setting a variable is very easy: just pass the data along.
.fp_set:c 9792 \cs_new_protected:cpn { \c__keys_props_root_tl .fp_set:N } #1
.fp_gset:N 9793 { \__keys_variable_set:NnnN #1 { fp } { } n }
.fp_gset:c 9794 \cs_new_protected:cpn { \c__keys_props_root_tl .fp_set:c } #1
          9795 { \__keys_variable_set:cnnN {#1} { fp } { } n }
          9796 \cs_new_protected:cpn { \c__keys_props_root_tl .fp_gset:N } #1
          9797 { \__keys_variable_set:NnnN #1 { fp } { g } n }
          9798 \cs_new_protected:cpn { \c__keys_props_root_tl .fp_gset:c } #1
          9799 { \__keys_variable_set:cnnN {#1} { fp } { g } n }
```

(End definition for `.fp_set:N` and `.fp_set:c`. These functions are documented on page 174.)

```
.groups:n A single property to create groups of keys.
          9800 \cs_new_protected:cpn { \c__keys_props_root_tl .groups:n } #1
          9801 { \__keys_groups_set:n {#1} }
```

(End definition for `.groups:n`. This function is documented on page 174.)

```
.initial:n The standard hand-off approach.
.initial:V 9802 \cs_new_protected:cpn { \c__keys_props_root_tl .initial:n } #1
.initial:o 9803 { \__keys_initialise:n {#1} }
.initial:x 9804 \cs_new_protected:cpn { \c__keys_props_root_tl .initial:V } #1
          9805 { \exp_args:NV \__keys_initialise:n #1 }
          9806 \cs_new_protected:cpn { \c__keys_props_root_tl .initial:o } #1
          9807 { \exp_args:No \__keys_initialise:n {#1} }
          9808 \cs_new_protected:cpn { \c__keys_props_root_tl .initial:x } #1
          9809 { \exp_args:Nx \__keys_initialise:n {#1} }
```

(End definition for `.initial:n` and others. These functions are documented on page 175.)

```
.int_set:N Setting a variable is very easy: just pass the data along.
.int_set:c 9810 \cs_new_protected:cpn { \c__keys_props_root_tl .int_set:N } #1
.int_gset:N 9811 { \__keys_variable_set:NnnN #1 { int } { } n }
.int_gset:c 9812 \cs_new_protected:cpn { \c__keys_props_root_tl .int_set:c } #1
          9813 { \__keys_variable_set:cnnN {#1} { int } { } n }
          9814 \cs_new_protected:cpn { \c__keys_props_root_tl .int_gset:N } #1
          9815 { \__keys_variable_set:NnnN #1 { int } { g } n }
          9816 \cs_new_protected:cpn { \c__keys_props_root_tl .int_gset:c } #1
          9817 { \__keys_variable_set:cnnN {#1} { int } { g } n }
```

(End definition for `.int_set:N` and `.int_set:c`. These functions are documented on page 175.)

```
.meta:n Making a meta is handled internally.
          9818 \cs_new_protected:cpn { \c__keys_props_root_tl .meta:n } #1
          9819 { \__keys_meta_make:n {#1} }
```

(End definition for `.meta:n`. This function is documented on page 175.)

.meta:nn Meta with path: potentially lots of variants, but for the moment no so many defined.

```
9820 \cs_new_protected:cpn { \c__keys_props_root_tl .meta:nn } #1
9821 { \__keys_meta_make:nn #1 }
```

(End definition for .meta:nn. This function is documented on page 175.)

.multichoice: The same idea as .choice: and .choices:nn, but where more than one choice is allowed.

```
.multichoices:nn 9822 \cs_new_protected_nopar:cpn { \c__keys_props_root_tl .multichoice: }
.multichoices:Vn 9823 { \__keys_multichoice_make: }
.multichoices:on 9824 \cs_new_protected:cpn { \c__keys_props_root_tl .multichoices:nn } #1
.multichoices:xn 9825 { \__keys_multichoices_make:nn #1 }
9826 \cs_new_protected:cpn { \c__keys_props_root_tl .multichoices:Vn } #1
9827 { \exp_args:NV \__keys_multichoices_make:nn #1 }
9828 \cs_new_protected:cpn { \c__keys_props_root_tl .multichoices:on } #1
9829 { \exp_args:No \__keys_multichoices_make:nn #1 }
9830 \cs_new_protected:cpn { \c__keys_props_root_tl .multichoices:xn } #1
9831 { \exp_args:Nx \__keys_multichoices_make:nn #1 }
```

(End definition for .multichoice:. This function is documented on page 175.)

.skip_set:N Setting a variable is very easy: just pass the data along.

```
.skip_set:c 9832 \cs_new_protected:cpn { \c__keys_props_root_tl .skip_set:N } #1
.skip_gset:N 9833 { \__keys_variable_set:NnnN #1 { skip } { } n }
.skip_gset:c 9834 \cs_new_protected:cpn { \c__keys_props_root_tl .skip_set:c } #1
9835 { \__keys_variable_set:cnnN {#1} { skip } { } n }
9836 \cs_new_protected:cpn { \c__keys_props_root_tl .skip_gset:N } #1
9837 { \__keys_variable_set:NnnN #1 { skip } { g } n }
9838 \cs_new_protected:cpn { \c__keys_props_root_tl .skip_gset:c } #1
9839 { \__keys_variable_set:cnnN {#1} { skip } { g } n }
```

(End definition for .skip_set:N and .skip_set:c. These functions are documented on page 175.)

.tl_set:N Setting a variable is very easy: just pass the data along.

```
.tl_set:c 9840 \cs_new_protected:cpn { \c__keys_props_root_tl .tl_set:N } #1
.tl_gset:N 9841 { \__keys_variable_set:NnnN #1 { tl } { } n }
.tl_gset:c 9842 \cs_new_protected:cpn { \c__keys_props_root_tl .tl_set:c } #1
.tl_set_x:N 9843 { \__keys_variable_set:cnnN {#1} { tl } { } n }
.tl_set_x:c 9844 \cs_new_protected:cpn { \c__keys_props_root_tl .tl_set_x:N } #1
.tl_gset_x:N 9845 { \__keys_variable_set:NnnN #1 { tl } { } x }
.tl_gset_x:c 9846 \cs_new_protected:cpn { \c__keys_props_root_tl .tl_set_x:c } #1
9847 { \__keys_variable_set:cnnN {#1} { tl } { } x }
9848 \cs_new_protected:cpn { \c__keys_props_root_tl .tl_gset:N } #1
9849 { \__keys_variable_set:NnnN #1 { tl } { g } n }
9850 \cs_new_protected:cpn { \c__keys_props_root_tl .tl_gset:c } #1
9851 { \__keys_variable_set:cnnN {#1} { tl } { g } n }
9852 \cs_new_protected:cpn { \c__keys_props_root_tl .tl_gset_x:N } #1
9853 { \__keys_variable_set:NnnN #1 { tl } { g } x }
9854 \cs_new_protected:cpn { \c__keys_props_root_tl .tl_gset_x:c } #1
9855 { \__keys_variable_set:cnnN {#1} { tl } { g } x }
```

(End definition for .tl_set:N and .tl_set:c. These functions are documented on page 175.)

.undefine: Another simple wrapper.

```
9856 \cs_new_protected_nopar:cpn { \c__keys_props_root_tl .undefine: }
9857 { \__keys_undefine: }
```

(End definition for .undefine:. This function is documented on page 176.)

.value_forbidden:n These are very similar, so both call the same function.

```
.value_required:n 9858 \cs_new_protected:cpn { \c__keys_props_root_tl .value_forbidden:n } #1
9859 { \__keys_value_requirement:nn { forbidden } {#1} }
9860 \cs_new_protected:cpn { \c__keys_props_root_tl .value_required:n } #1
9861 { \__keys_value_requirement:nn { required } {#1} }
```

(End definition for .value_forbidden:n. This function is documented on page 176.)

19.6 Setting keys

\keys_set:nn A simple wrapper again.

```
\keys_set:nV 9862 \cs_new_protected_nopar:Npn \keys_set:nn
\keys_set:nv 9863 { \__keys_set:onn { \l__keys_module_tl } }
\keys_set:no 9864 \cs_new_protected:Npn \__keys_set:nnn #1#2#3
\__keys_set:nnn 9865 {
\__keys_set:onn 9866 \tl_set:Nx \l__keys_module_tl { \tl_to_str:n {#2} }
9867 \keyval_parse:NNn \__keys_set_elt:n \__keys_set_elt:nn {#3}
9868 \tl_set:Nn \l__keys_module_tl {#1}
9869 }
9870 \cs_generate_variant:Nn \keys_set:nn { nV , nv , no }
9871 \cs_generate_variant:Nn \__keys_set:nnn { o }
```

(End definition for \keys_set:nn and others. These functions are documented on page 179.)

\keys_set_known:nnN Setting known keys simply means setting the appropriate flag, then running the standard code. To allow for nested setting, any existing value of \l__keys_unused_clist is saved on the stack and reset afterwards. Note that for speed/simplicity reasons we use a `tl` operation to set the `clist` here!

```
\keys_set_known:nVN 9872 \cs_new_protected_nopar:Npn \keys_set_known:nnN
\keys_set_known:nvN 9873 { \__keys_set_known:onnN \l__keys_unused_clist }
\keys_set_known:noN 9874 \cs_generate_variant:Nn \keys_set_known:nnN { nV , nv , no }
\__keys_set_known:nnnN 9875 \cs_new_protected:Npn \__keys_set_known:nnnN #1#2#3#4
\__keys_set_known:onnN 9876 {
\keys_set_known:nV 9877 \clist_clear:N \l__keys_unused_clist
\keys_set_known:nv 9878 \keys_set_known:nn {#2} {#3}
\keys_set_known:no 9879 \tl_set:Nx #4 { \exp_not:o { \l__keys_unused_clist } }
9880 \tl_set:Nn \l__keys_unused_clist {#1}
9881 }
9882 \cs_generate_variant:Nn \__keys_set_known:nnnN { o }
9883 \cs_new_protected:Npn \keys_set_known:nn #1#2
9884 {
9885 \bool_set_true:N \l__keys_only_known_bool
9886 \keys_set:nn {#1} {#2}
```

```

9887     \bool_set_false:N \l__keys_only_known_bool
9888   }
9889   \cs_generate_variant:Nn \keys_set_known:nn { nV , nv , no }

```

(End definition for \keys_set_known:nnN and others. These functions are documented on page 180.)

```

\keys_set_filter:nnnN The idea of setting keys in a selective manner again uses flags wrapped around the basic
\keys_set_filter:nnVN code. The comments on \keys_set_known:nnN also apply here.
\keys_set_filter:nnvN \keys_set_filter:nnnN
\__keys_set_filter:nnnnN 9890 \cs_new_protected_nopar:Npn \keys_set_filter:nnnN
\__keys_set_filter:nnnnN 9891 { \__keys_set_filter:nnnnN \l__keys_unused_clist }
\keys_set_filter:nnn 9892 \cs_generate_variant:Nn \keys_set_filter:nnnN { nnV , nnv , nno }
\keys_set_filter:nnn 9893 \cs_new_protected:Npn \__keys_set_filter:nnnnN #1#2#3#4#5
\keys_set_filter:nnV 9894 {
\keys_set_filter:nnv \keys_set_filter:nnn 9895 \clist_clear:N \l__keys_unused_clist
\keys_set_filter:nnv \keys_set_filter:nnn {#2} {#3} {#4} 9896
\keys_set_groups:nnn 9897 \tl_set:Nx #5 { \exp_not:o { \l__keys_unused_clist } }
\keys_set_groups:nnV 9898 \tl_set:Nn \l__keys_unused_clist {#1}
\keys_set_groups:nnv \keys_set_groups:nnn 9899 }
9900 \cs_generate_variant:Nn \__keys_set_filter:nnnnN { o }
9901 \cs_new_protected:Npn \keys_set_filter:nnn #1#2#3
9902 {
9903   \bool_set_true:N \l__keys_selective_bool
9904   \bool_set_true:N \l__keys_filtered_bool
9905   \seq_set_from_clist:Nn \l__keys_selective_seq {#2}
9906   \keys_set:nn {#1} {#3}
9907   \bool_set_false:N \l__keys_selective_bool
9908 }
9909 \cs_generate_variant:Nn \keys_set_filter:nnn { nnV , nnv , nno }
9910 \cs_new_protected:Npn \keys_set_groups:nnn #1#2#3
9911 {
9912   \bool_set_true:N \l__keys_selective_bool
9913   \bool_set_false:N \l__keys_filtered_bool
9914   \seq_set_from_clist:Nn \l__keys_selective_seq {#2}
9915   \keys_set:nn {#1} {#3}
9916   \bool_set_false:N \l__keys_selective_bool
9917 }
9918 \cs_generate_variant:Nn \keys_set_groups:nnn { nnV , nnv , nno }

```

(End definition for \keys_set_filter:nnnN, \keys_set_filter:nnVN, and \keys_set_filter:nnvN \keys_set_filter:nnnN. These functions are documented on page 181.)

```

\__keys_set_elt:n A shared system once again. First, set the current path and add a default if needed.
\__keys_set_elt:nn There are then checks to see if the a value is required or forbidden. If everything passes,
\__keys_set_elt_aux:nnn move on to execute the code.
\__keys_set_elt_aux:onn 9919 \cs_new_protected:Npn \__keys_set_elt:n #1
\__keys_find_key_module:w 9920 {
\__keys_set_elt_aux: 9921 \bool_set_true:N \l__keys_no_value_bool
\__keys_set_elt_selective: 9922 \__keys_set_elt_aux:onn \l__keys_module_tl {#1} { }
9923 }
9924 \cs_new_protected:Npn \__keys_set_elt:nn #1#2

```

```

9925 {
9926     \bool_set_false:N \l__keys_no_value_bool
9927     \__keys_set_elt_aux:onnn \l__keys_module_tl {#1} {#2}
9928 }

```

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.

```

9929 \cs_new_protected:Npn \__keys_set_elt_aux:nnn #1#2#3
9930 {
9931     \tl_set:Nx \l_keys_path_tl { \l__keys_module_tl / \tl_to_str:n {#2} }
9932     \tl_clear:N \l__keys_module_tl
9933     \exp_after:wN \__keys_find_key_module:w \l_keys_path_tl / \q_stop
9934     \__keys_value_or_default:n {#3}
9935     \bool_if:NTF \l__keys_selective_bool
9936     { \__keys_set_elt_selective: }
9937     { \__keys_set_elt_aux: }
9938     \tl_set:Nn \l__keys_module_tl {#1}
9939 }
9940 \cs_generate_variant:Nn \__keys_set_elt_aux:nnn { o }
9941 \cs_new_protected:Npn \__keys_find_key_module:w #1 / #2 \q_stop
9942 {
9943     \tl_if_blank:nTF {#2}
9944     { \tl_set:Nn \l_keys_key_tl {#1} }
9945     {
9946         \tl_put_right:Nx \l__keys_module_tl
9947         {
9948             \tl_if_empty:NF \l__keys_module_tl { / }
9949             #1
9950         }
9951         \__keys_find_key_module:w #2 \q_stop
9952     }
9953 }
9954 \cs_new_protected_nopar:Npn \__keys_set_elt_aux:
9955 {
9956     \bool_if:nTF
9957     {
9958         \__keys_if_value_p:n { required } &&
9959         \l__keys_no_value_bool
9960     }
9961     {
9962         \__msg_kernel_error:nmx { kernel } { value-required }
9963         { \l_keys_path_tl }
9964     }
9965     {
9966         \bool_if:nTF
9967         {
9968             \__keys_if_value_p:n { forbidden } &&
9969             ! \l__keys_no_value_bool

```

```

9970     }
9971     {
9972         \_msg_kernel_error:nxxx { kernel } { value-forbidden }
9973         { \l_keys_path_tl } { \l_keys_value_tl }
9974     }
9975     { \_keys_execute: }
9976 }
9977 }

```

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.

```

9978 \cs_new_protected_nopar:Npn \_keys_set_elt_selective:
9979 {
9980     \prop_if_exist:cTF { \c__keys_info_root_tl \l_keys_path_tl }
9981     {
9982         \prop_get:cnNTF { \c__keys_info_root_tl \l_keys_path_tl }
9983         { groups } \l__keys_groups_clist
9984         { \_keys_check_groups: }
9985         {
9986             \bool_if:NTF \l__keys_filtered_bool
9987             { \_keys_set_elt_aux: }
9988             { \_keys_store_unused: }
9989         }
9990     }
9991     {
9992         \bool_if:NTF \l__keys_filtered_bool
9993         { \_keys_set_elt_aux: }
9994         { \_keys_store_unused: }
9995     }
9996 }

```

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.

```

9997 \cs_new_protected_nopar:Npn \_keys_check_groups:
9998 {
9999     \bool_set_false:N \l__keys_tmp_bool
10000     \seq_map_inline:Nn \l__keys_selective_seq
10001     {
10002         \clist_map_inline:Nn \l__keys_groups_clist
10003         {
10004             \str_if_eq:nnT {##1} {####1}
10005             {
10006                 \bool_set_true:N \l__keys_tmp_bool
10007                 \clist_map_break:n { \seq_map_break: }
10008             }
10009         }
10010     }
10011     \bool_if:NTF \l__keys_tmp_bool

```

```

10012     {
10013         \bool_if:NTF \l__keys_filtered_bool
10014         { \__keys_store_unused: }
10015         { \__keys_set_elt_aux: }
10016     }
10017     {
10018         \bool_if:NTF \l__keys_filtered_bool
10019         { \__keys_set_elt_aux: }
10020         { \__keys_store_unused: }
10021     }
10022 }

```

(End definition for __keys_set_elt:n and __keys_set_elt:nn.)

__keys_value_or_default:n If a value is given, return it as #1, otherwise send a default if available.

```

10023 \cs_new_protected:Npn \__keys_value_or_default:n #1
10024 {
10025     \bool_if:NTF \l__keys_no_value_bool
10026     {
10027         \prop_get:cnNF { \c__keys_info_root_tl \l_keys_path_tl }
10028         { default } \l_keys_value_tl
10029         { \tl_clear:N \l_keys_value_tl }
10030     }
10031     { \tl_set:Nn \l_keys_value_tl {#1} }
10032 }

```

(End definition for __keys_value_or_default:n.)

__keys_if_value_p:n To test if a value is required or forbidden. A simple check for the existence of the appropriate marker.

```

10033 \prg_new_conditional:Npnn \__keys_if_value:n #1 { p }
10034 {
10035     \prop_if_exist:cTF { \c__keys_info_root_tl \l_keys_path_tl }
10036     {
10037         \prop_if_in:cnTF { \c__keys_info_root_tl \l_keys_path_tl } {#1}
10038         { \prg_return_true: }
10039         { \prg_return_false: }
10040     }
10041     { \prg_return_false: }
10042 }

```

(End definition for __keys_if_value_p: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.

```

10043 \cs_new_protected_nopar:Npn \__keys_execute:
10044 { \__keys_execute:nn { \l_keys_path_tl } { \__keys_execute_unknown: } }
10045 \cs_new_protected_nopar:Npn \__keys_execute_unknown:

```

```

10046 {
10047     \bool_if:NTF \l__keys_only_known_bool
10048     { \__keys_store_unused: }
10049     {
10050         \__keys_execute:nn { \l__keys_module_tl / unknown }
10051         {
10052             \__msg_kernel_error:nxxx { kernel } { key-unknown }
10053             { \l_keys_path_tl } { \l__keys_module_tl }
10054         }
10055     }
10056 }
10057 \cs_new:Npn \__keys_execute:nn #1#2
10058 {
10059     \cs_if_exist:cTF { \c__keys_code_root_tl #1 }
10060     {
10061         \exp_args:Nc \exp_args:No { \c__keys_code_root_tl #1 }
10062         \l_keys_value_tl
10063     }
10064     {#2}
10065 }
10066 \cs_new_protected_nopar:Npn \__keys_store_unused:
10067 {
10068     \clist_put_right:Nx \l__keys_unused_clist
10069     {
10070         \exp_not:o \l_keys_key_tl
10071         \bool_if:NF \l__keys_no_value_bool
10072         { = { \exp_not:o \l_keys_value_tl } }
10073     }
10074 }

```

(End definition for __keys_execute:.)

__keys_choice_find:n Executing a choice has two parts. First, try the choice given, then if that fails call the
 __keys_multichoice_find:n 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.

```

10075 \cs_new:Npn \__keys_choice_find:n #1
10076 {
10077     \__keys_execute:nn { \l_keys_path_tl / \tl_to_str:n {#1} }
10078     { \__keys_execute:nn { \l_keys_path_tl / unknown } { } }
10079 }
10080 \cs_new:Npn \__keys_multichoice_find:n #1
10081 { \clist_map_function:nN {#1} \__keys_choice_find:n }

```

(End definition for __keys_choice_find:n.)

19.7 Utilities

\keys_if_exist_p:nn A utility for others to see if a key exists.

\keys_if_exist:nnTF 10082 \prg_new_conditional:Npnn \keys_if_exist:nn #1#2 { p , T , F , TF }
 10083 {

```

10084     \cs_if_exist:cTF { \c__keys_code_root_tl \tl_to_str:n { #1 / #2 } }
10085     { \prg_return_true: }
10086     { \prg_return_false: }
10087 }

```

(End definition for \keys_if_exist:nnTF. This function is documented on page 181.)

\keys_if_choice_exist_p:nnn Just an alternative view on \keys_if_exist:nn(TF).

```

\keys_if_choice_exist:nnnTF 10088 \prg_new_conditional:Npnn \keys_if_choice_exist:nnn #1#2#3
10089 { p , T , F , TF }
10090 {
10091     \cs_if_exist:cTF { \c__keys_code_root_tl \tl_to_str:n { #1 / #2 / #3 } }
10092     { \prg_return_true: }
10093     { \prg_return_false: }
10094 }

```

(End definition for \keys_if_choice_exist:nnnTF. This function is documented on page 182.)

\keys_show:nn 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.

__keys_show:NN

```

10095 \cs_new_protected:Npn \keys_show:nn #1#2
10096 {
10097     \keys_if_exist:nnTF {#1} {#2}
10098     {
10099         \__msg_show_pre:nnxxxx { LaTeX / kernel } { show-key }
10100         { \tl_to_str:n { #1 / #2 } } { t } { } { }
10101         \exp_args:Ncc \__keys_show:NN
10102         { \c__keys_code_root_tl \tl_to_str:n { #1 / #2 } }
10103         { \c__keys_info_root_tl \tl_to_str:n { #1 / #2 } }
10104     }
10105     {
10106         \__msg_show_pre:nnxxxx { LaTeX / kernel } { show-key }
10107         { \tl_to_str:n { #1 / #2 } } { f } { } { }
10108         \__msg_show_wrap:n { }
10109     }
10110 }
10111 \cs_new_protected:Npn \__keys_show:NN #1#2
10112 {
10113     \use:x
10114     {
10115         \__msg_show_wrap:n
10116         {
10117             \exp_not:N \__msg_show_item_unbraced:nn { code }
10118             { \token_get_replacement_spec:N #1 }
10119             \exp_not:n
10120             { \prop_map_function:NN #2 \__msg_show_item_unbraced:nn }
10121         }
10122     }
10123 }

```

(End definition for \keys_show:nn. This function is documented on page 182.)

19.8 Messages

For when there is a need to complain.

```

10124 \_msg_kernel_new:nnnn { kernel } { boolean-values-only }
10125 { Key~'#1'~accepts~boolean-values-only. }
10126 { The~key~'#1'~only~accepts~the~values~'true'~and~'false'. }
10127 \_msg_kernel_new:nnnn { kernel } { choice-unknown }
10128 { Choice~'#2'~unknown~for~key~'#1'. }
10129 {
10130   The~key~'#1'~takes~a~limited~number~of~values.\\
10131   The~input~given,~'#2',~is~not~on~the~list~accepted.
10132 }
10133 \_msg_kernel_new:nnnn { kernel } { key-choice-unknown }
10134 { Key~'#1'~accepts~only~a~fixed~set~of~choices. }
10135 {
10136   The~key~'#1'~only~accepts~predefined~values,~
10137   and~'#2'~is~not~one~of~these.
10138 }
10139 \_msg_kernel_new:nnnn { kernel } { key-no-property }
10140 { No~property~given~in~definition~of~key~'#1'. }
10141 {
10142   \c_msg_coding_error_text_tl
10143   Inside~\keys_define:nn  each~key~name~
10144   needs~a~property:  \\ \\
10145   \iow_indent:n { #1 .<property> } \\ \\
10146   LaTeX~did~not~find~a~'.'~to~indicate~the~start~of~a~property.
10147 }
10148 \_msg_kernel_new:nnnn { kernel } { key-unknown }
10149 { The~key~'#1'~is~unknown~and~is~being~ignored. }
10150 {
10151   The~module~'#2'~does~not~have~a~key~called~'#1'.\\
10152   Check~that~you~have~spelled~the~key~name~correctly.
10153 }
10154 \_msg_kernel_new:nnnn { kernel } { nested-choice-key }
10155 { Attempt~to~define~'#1'~as~a~nested~choice~key. }
10156 {
10157   The~key~'#1'~cannot~be~defined~as~a~choice~as~the~parent~key~'#2'~is~
10158   itself~a~choice.
10159 }
10160 \_msg_kernel_new:nnnn { kernel } { property-boolean-values-only }
10161 { The~property~'#1'~accepts~boolean-values-only. }
10162 {
10163   \c_msg_coding_error_text_tl
10164   The~property~'#1'~only~accepts~the~values~'true'~and~'false'.
10165 }
10166 \_msg_kernel_new:nnnn { kernel } { property-requires-value }
10167 { The~property~'#1'~requires~a~value. }

```

```

10168 {
10169   \c__msg_coding_error_text_tl
10170   LaTeX~was~asked~to~set~property~'#1'~for~key~'#2'.\\
10171   No~value~was~given~for~the~property,~and~one~is~required.
10172 }
10173 \__msg_kernel_new:nnnn { kernel } { property-unknown }
10174 { The~key~property~'#1'~is~unknown. }
10175 {
10176   \c__msg_coding_error_text_tl
10177   LaTeX~has~been~asked~to~set~the~property~'#1'~for~key~'#2':~
10178   this~property~is~not~defined.
10179 }
10180 \__msg_kernel_new:nnnn { kernel } { value-forbidden }
10181 { The~key~'#1'~does~not~take~a~value. }
10182 {
10183   The~key~'#1'~should~be~given~without~a~value.\\
10184   The~value~'#2'~was~present:~the~key~will~be~ignored.
10185 }
10186 \__msg_kernel_new:nnnn { kernel } { value-required }
10187 { The~key~'#1'~requires~a~value. }
10188 {
10189   The~key~'#1'~must~have~a~value.\\
10190   No~value~was~present:~the~key~will~be~ignored.
10191 }
10192 \__msg_kernel_new:nnn { kernel } { show-key }
10193 {
10194   The~key~'#1'~
10195   \str_if_eq:nnTF {#2} { t }
10196   { has~the~properties: }
10197   { is~undefined. }
10198 }

```

19.9 Deprecated functions

`.value_forbidden:` Deprecated 2015-07-14.

```

.value_required: 10199 \cs_new_protected_nopar:cpn { \c__keys_props_root_tl .value_forbidden: }
                  10200 { \__keys_value_requirement:nn { forbidden } { true } }
                  10201 \cs_new_protected_nopar:cpn { \c__keys_props_root_tl .value_required: }
                  10202 { \__keys_value_requirement:nn { required } { true } }

```

(End definition for `.value_forbidden:`. This function is documented on page ??.)

```
10203 </initex | package>
```

20 l3file implementation

The following test files are used for this code: `m3file001`.

```

10204 <*initex | package>
10205 <@@=file>

```

20.1 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`.

```

10206 \tl_new:N \g_file_current_name_tl
10207 <*initex>
10208 \tex_everyjob:D \exp_after:wN
10209 {
10210     \tex_the:D \tex_everyjob:D
10211     \tl_gset:Nx \g_file_current_name_tl { \tex_jobname:D }
10212 }
10213 </initex>
10214 <*package>
10215 \cs_if_exist:NT \@currname
10216 { \tl_gset_eq:NN \g_file_current_name_tl \@currname }
10217 </package>

```

(End definition for `\g_file_current_name_tl`. This variable is documented on page 184.)

`\g__file_stack_seq` The input list of files is stored as a sequence stack.

```

10218 \seq_new:N \g__file_stack_seq

```

(End definition for `\g__file_stack_seq`. This variable is documented on page ??.)

`\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 T_EX run. In package mode we will eventually copy the contents of `\@filelist`.

```

10219 \seq_new:N \g__file_record_seq
10220 <*initex>
10221 \tex_everyjob:D \exp_after:wN
10222 {
10223     \tex_the:D \tex_everyjob:D
10224     \seq_gput_right:NV \g__file_record_seq \g_file_current_name_tl
10225 }
10226 </initex>

```

(End definition for `\g__file_record_seq`. This variable is documented on page ??.)

`\l__file_internal_tl` Used as a short-term scratch variable. It may be possible to reuse `\l__file_internal_name_tl` there.

```

10227 \tl_new:N \l__file_internal_tl

```

(End definition for `\l__file_internal_tl`. This variable is documented on page ??.)

`\l__file_internal_name_tl` Used to return the fully-qualified name of a file.

```

10228 \tl_new:N \l__file_internal_name_tl

```

(End definition for `\l__file_internal_name_tl`. This variable is documented on page 190.)

`\l__file_search_path_seq` The current search path.

```

10229 \seq_new:N \l__file_search_path_seq

(End definition for \l__file_search_path_seq. This variable is documented on page ??.)

```

`\l_file_saved_search_path_seq` The current search path has to be saved for package use.

```

10230 <*package>
10231 \seq_new:N \l_file_saved_search_path_seq
10232 </package>

(End definition for \l_file_saved_search_path_seq. This variable is documented on page ??.)

```

`\l__file_internal_seq` Scratch space for comma list conversion in package mode.

```

10233 <*package>
10234 \seq_new:N \l__file_internal_seq
10235 </package>

(End definition for \l__file_internal_seq. This variable is documented on page ??.)

```

`__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.

`__file_name_sanitize_aux:n`

```

10236 \cs_new_protected:Npn \__file_name_sanitize:nn #1#2
10237 {
10238   \group_begin:
10239     \seq_map_inline:Nn \l_char_active_seq
10240       { \cs_set_nopar:Npx ##1 { \token_to_str:N ##1 } }
10241     \tl_set:Nx \l__file_internal_name_tl {#1}
10242     \tl_set:Nx \l__file_internal_name_tl
10243       { \tl_to_str:N \l__file_internal_name_tl }
10244     \int_compare:nNnTF
10245       {
10246         \int_mod:nn
10247           {
10248             0 \tl_map_function:NN \l__file_internal_name_tl
10249               \__file_name_sanitize_aux:n
10250           }
10251         \c_two
10252       }
10253     = \c_zero
10254     {
10255       \tl_remove_all:Nn \l__file_internal_name_tl { " }
10256       \tl_if_in:NnT \l__file_internal_name_tl { ~ }
10257       {
10258         \tl_set:Nx \l__file_internal_name_tl
10259           { " \exp_not:V \l__file_internal_name_tl " }
10260       }
10261     }
10262   {

```

```

10263         \_msg_kernel_error:nxx
10264         { kernel } { unbalanced-quote-in-filename }
10265         { \l__file_internal_name_tl }
10266     }
10267     \use:x
10268     {
10269         \group_end:
10270         \exp_not:n {#2} { \l__file_internal_name_tl }
10271     }
10272 }
10273 \cs_new:Npn \__file_name_sanitiz_aux:n #1
10274 {
10275     \token_if_eq_charcode:NNT #1 "
10276     { + \c_one }
10277 }

```

(End definition for __file_name_sanitiz: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 T_EX 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.

```

10278 \cs_new_protected:Npn \file_add_path:nN #1
10279 { \__file_name_sanitiz_aux:n {#1} { \__file_add_path:nN } }
10280 \cs_new_protected:Npn \__file_add_path:nN #1#2
10281 {
10282     \_ior_open:Nn \g__file_internal_ior {#1}
10283     \ior_if_eof:NNTF \g__file_internal_ior
10284     { \__file_add_path_search:nN {#1} #2 }
10285     { \tl_set:Nn #2 {#1} }
10286     \ior_close:N \g__file_internal_ior
10287 }
10288 \cs_new_protected:Npn \__file_add_path_search:nN #1#2
10289 {
10290     \tl_set:Nn #2 { \q_no_value }
10291 }
10292 \cs_if_exist:NT \input@path
10293 {
10294     \seq_set_eq:NN \l__file_saved_search_path_seq
10295     \l__file_search_path_seq
10296     \seq_set_split:NnV \l__file_internal_seq { , } \input@path
10297     \seq_concat:NNN \l__file_search_path_seq
10298     \l__file_search_path_seq \l__file_internal_seq
10299 }
10300 \</package>
10301 \seq_map_inline:Nn \l__file_search_path_seq
10302 {
10303     \_ior_open:Nn \g__file_internal_ior { ##1 #1 }
10304     \ior_if_eof:NNTF \g__file_internal_ior

```

```

10305         {
10306             \tl_set:Nx #2 { ##1 #1 }
10307             \seq_map_break:
10308         }
10309     }
10310 <*package>
10311     \cs_if_exist:NT \input@path
10312     {
10313         \seq_set_eq:NN \l__file_search_path_seq
10314         \l__file_saved_search_path_seq
10315     }
10316 </package>
10317 }

```

(End definition for `\file_add_path:nN`. This function is documented on page 184.)

`\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`.

```

10318 \prg_new_protected_conditional:Npnn \file_if_exist:n #1 { T , F , TF }
10319 {
10320     \file_add_path:nN {#1} \l__file_internal_name_tl
10321     \quark_if_no_value:NTF \l__file_internal_name_tl
10322     { \prg_return_false: }
10323     { \prg_return_true: }
10324 }

```

(End definition for `\file_if_exist:nTF`. This function is documented on page 184.)

`\file_input:n` Loading a file is done in a safe way, checking first that the file exists and loading only if it does. Push the file name on the `\g__file_stack_seq`, and add it to the file list, either `\g__file_record_seq`, or `\@filelist` in package mode.

```

\__file_if_exist:nT
\__file_input:n \__file_input:V
\__file_input_aux:n
\__file_input_aux:o
10325 \cs_new_protected:Npn \file_input:n #1
10326 {
10327     \__file_if_exist:nT {#1}
10328     { \__file_input:V \l__file_internal_name_tl }
10329 }

```

This code is spun out as a separate function to encapsulate the error message into a easy-to-reuse form.

```

10330 \cs_new_protected:Npn \__file_if_exist:nT #1#2
10331 {
10332     \file_if_exist:nTF {#1}
10333     {#2}
10334     {
10335         \__file_name_sanitiz:nn {#1}
10336         { \__msg_kernel_error:nxx { kernel } { file-not-found } }
10337     }
10338 }
10339 \cs_new_protected:Npn \__file_input:n #1

```

```

10340 {
10341   \tl_if_in:nnTF {#1} { . }
10342   { \__file_input_aux:n {#1} }
10343   { \__file_input_aux:o { \tl_to_str:n { #1 . tex } } }
10344 }
10345 \cs_generate_variant:Nn \__file_input:n { V }
10346 \cs_new_protected:Npn \__file_input_aux:n #1
10347 {
10348   \*initex
10349   \seq_gput_right:Nn \g__file_record_seq {#1}
10350   \*initex
10351   \*package
10352   \clist_if_exist:NTF \@filelist
10353   { \@addtofilelist {#1} }
10354   { \seq_gput_right:Nn \g__file_record_seq {#1} }
10355   \*package
10356   \seq_gpush:Nn \g__file_stack_seq \g_file_current_name_tl
10357   \tl_gset:Nn \g_file_current_name_tl {#1}
10358   \tex_input:D #1 \c_space_tl
10359   \seq_gpop:NN \g__file_stack_seq \l__file_internal_tl
10360   \tl_gset_eq:NN \g_file_current_name_tl \l__file_internal_tl
10361 }
10362 \cs_generate_variant:Nn \__file_input_aux:n { o }

```

(End definition for `\file_input:n`. This function is documented on page 184.)

`\file_path_include:n` Wrapper functions to manage the search path.

`\file_path_remove:n`

`__file_path_include:n`

```

10363 \cs_new_protected:Npn \file_path_include:n #1
10364 { \__file_name_sanitiz:nn {#1} { \__file_path_include:n } }
10365 \cs_new_protected:Npn \__file_path_include:n #1
10366 {
10367   \seq_if_in:NnF \l__file_search_path_seq {#1}
10368   { \seq_put_right:Nn \l__file_search_path_seq {#1} }
10369 }
10370 \cs_new_protected:Npn \file_path_remove:n #1
10371 {
10372   \__file_name_sanitiz:nn {#1}
10373   { \seq_remove_all:Nn \l__file_search_path_seq }
10374 }

```

(End definition for `\file_path_include:n`. This function is documented on page 185.)

`\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.

```

10375 \cs_new_protected_nopar:Npn \file_list:
10376 {
10377   \seq_set_eq:NN \l__file_internal_seq \g__file_record_seq
10378   \*package

```

```

10379 \clist_if_exist:NT \@filelist
10380 {
10381     \clist_map_inline:Nn \@filelist
10382     {
10383         \seq_put_right:No \l__file_internal_seq
10384         { \tl_to_str:n {##1} }
10385     }
10386 }
10387 </package>
10388 \seq_remove_duplicates:N \l__file_internal_seq
10389 \iow_log:n { *~File~List~* }
10390 \seq_map_inline:Nn \l__file_internal_seq { \iow_log:n {##1} }
10391 \iow_log:n { ***** }
10392 }

```

(End definition for `\file_list`:. This function is documented on page 185.)

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`.

```

10393 <*package>
10394 \AtBeginDocument
10395 {
10396     \clist_map_inline:Nn \@filelist
10397     { \seq_gput_right:No \g__file_record_seq { \tl_to_str:n {##1} } }
10398 }
10399 </package>

```

20.2 Input operations

```
10400 <@@=ior>
```

20.2.1 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.

```
10401 \cs_new_eq:NN \c_term_ior \c_sixteen
```

(End definition for `\c_term_ior`. This variable is documented on page 190.)

`\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!

```

10402 \seq_new:N \g__ior_streams_seq
10403 <*initex>
10404 \seq_gset_split:Nnn \g__ior_streams_seq { , }
10405 { 0 , 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 , 10 , 11 , 12 , 13 , 14 , 15 }
10406 </initex>

```

(End definition for `\g__ior_streams_seq`. This variable is documented on page ??.)

`\l__ior_stream_tl` Used to recover the raw stream number from the stack.

```
10407 \tl_new:N \l__ior_stream_tl
```

(End definition for `\l__ior_stream_tl`. This variable is documented on page ??.)

`\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.

```
10408 \prop_new:N \g__ior_streams_prop
10409 <*package>
10410 \int_step_inline:nnnn
10411 { \c_zero }
10412 { \c_one }
10413 {
10414   \cs_if_exist:NTF \normalend
10415   { \tex_count:D 38 \scan_stop: }
10416   {
10417     \tex_count:D 16 \scan_stop:
10418     \cs_if_exist:NT \loccount { - \c_one }
10419   }
10420 }
10421 {
10422   \prop_gput:Nnn \g__ior_streams_prop {#1} { Reserved-by-format }
10423 }
10424 </package>
```

(End definition for `\g__ior_streams_prop`. This variable is documented on page ??.)

20.2.2 Stream management

`\ior_new:N` Reserving a new stream is done by defining the name as equal to using the terminal.

```
\ior_new:c 10425 \cs_new_protected:Npn \ior_new:N #1 { \cs_new_eq:NN #1 \c_term_ior }
10426 \cs_generate_variant:Nn \ior_new:N { c }
```

(End definition for `\ior_new:N` and `\ior_new:c`. These functions are documented on page 185.)

`\ior_open: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.

`\ior_open:cn`
`__ior_open_aux:Nn`

```
10427 \cs_new_protected:Npn \ior_open:Nn #1#2
10428 { \__file_name_sanitize:nn {#2} { \__ior_open_aux:Nn #1 } }
10429 \cs_generate_variant:Nn \ior_open:Nn { c }
10430 \cs_new_protected:Npn \__ior_open_aux:Nn #1#2
10431 {
10432   \file_add_path:nN {#2} \l__file_internal_name_tl
```

```

10433 \quark_if_no_value:NTF \l__file_internal_name_tl
10434 { \__msg_kernel_error:nxx { kernel } { file-not-found } {#2} }
10435 { \__ior_open:No #1 \l__file_internal_name_tl }
10436 }

```

(End definition for `\ior_open:Nn` and `\ior_open:cn`. These functions are documented on page 185.)

`\ior_open:NnTF` Much the same idea for opening a read with a conditional, except the auxiliary function
`\ior_open:cnTF` does not issue an error if the file is not found.

```

\__ior_open_aux:NnTF
10437 \prg_new_protected_conditional:Npnn \ior_open:Nn #1#2 { T , F , TF }
10438 { \__file_name_sanitize:nn {#2} { \__ior_open_aux:NnTF #1 } }
10439 \cs_generate_variant:Nn \ior_open:NnT { c }
10440 \cs_generate_variant:Nn \ior_open:NnF { c }
10441 \cs_generate_variant:Nn \ior_open:NnTF { c }
10442 \cs_new_protected:Npn \__ior_open_aux:NnTF #1#2
10443 {
10444   \file_add_path:nN {#2} \l__file_internal_name_tl
10445   \quark_if_no_value:NTF \l__file_internal_name_tl
10446   { \prg_return_false: }
10447   {
10448     \__ior_open:No #1 \l__file_internal_name_tl
10449     \prg_return_true:
10450   }
10451 }

```

(End definition for `\ior_open:NnTF` and `\ior_open:cnTF`. These functions are documented on page 185.)

`__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`.

```

10452 \*package>
10453 \exp_args:Nnf \cs_new_protected_nopar:Npn \__ior_new:N
10454 { \exp_args:Nnc \exp_after:wN \exp_stop_f: { newread } }
10455 \</package>

```

(End definition for `__ior_new:N`.)

`__ior_open:Nn` The stream allocation itself uses the fact that there is a list of all of those available, so
`__ior_open:No` allocation is simply a question of using the number at the top of the list. In package
`__ior_open_stream:Nn` 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 $\text{\LaTeX} 2_{\epsilon}$ for a new stream and use that number (after a bit of conversion).

```

10456 \cs_new_protected:Npn \__ior_open:Nn #1#2
10457 {
10458   \ior_close:N #1
10459   \seq_gpop:NNTF \g__ior_streams_seq \l__ior_stream_tl

```

```

10460 { \_ior\_open\_stream:Nn #1 {#2} }
10461 <*initex>
10462 { \_msg\_kernel\_fatal:nn { kernel } { input-streams-exhausted } }
10463 </initex>
10464 <*package>
10465 {
10466   \_ior\_new:N #1
10467   \tl\_set:Nx \l\_ior\_stream\_tl { \int\_eval:n {#1} }
10468   \_ior\_open\_stream:Nn #1 {#2}
10469 }
10470 </package>
10471 }
10472 \cs\_generate\_variant:Nn \_ior\_open:Nn { No }
10473 \cs\_new\_protected:Npn \_ior\_open\_stream:Nn #1#2
10474 {
10475   \tex\_global:D \tex\_chardef:D #1 = \l\_ior\_stream\_tl \scan\_stop:
10476   \prop\_gput:Nvn \g\_ior\_streams\_prop #1 {#2}
10477   \tex\_openin:D #1 #2 \scan\_stop:
10478 }

```

(End definition for _ior_open:Nn and _ior_open:No.)

\ior_close:N Closing a stream means getting rid of it at the T_EX level and removing from the various
\ior_close:c 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.

```

10479 \cs\_new\_protected:Npn \ior\_close:N #1
10480 {
10481   \int\_compare:nT { \c\_minus\_one < #1 < \c\_sixteen }
10482   {
10483     \tex\_closein:D #1
10484     \prop\_gremove:Nv \g\_ior\_streams\_prop #1
10485     \seq\_if\_in:NVF \g\_ior\_streams\_seq #1
10486     { \seq\_gpush:Nv \g\_ior\_streams\_seq #1 }
10487     \cs\_gset\_eq:NN #1 \c\_term\_ior
10488   }
10489 }
10490 \cs\_generate\_variant:Nn \ior\_close:N { c }

```

(End definition for \ior_close:N and \ior_close:c. These functions are documented on page 186.)

\ior_list_streams: Show the property lists, but with some “pretty printing”. See the l3msg module. The
_ior_list_streams:Nn 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.

```

10491 \cs\_new\_protected\_nopar:Npn \ior\_list\_streams:
10492 { \_ior\_list\_streams:Nn \g\_ior\_streams\_prop { ior } }
10493 \cs\_new\_protected:Npn \_ior\_list\_streams:Nn #1#2
10494 {

```

```

10495     \_msg_show_pre:nnxxxx { LaTeX / kernel } { show-streams }
10496     {#2} { \prop_if_empty:NF #1 { ? } } { } { }
10497     \_msg_show_wrap:n
10498     { \prop_map_function:NN #1 \_msg_show_item_unbraced:nn }
10499   }

```

(End definition for `\ior_list_streams:`. This function is documented on page 186.)

20.2.3 Reading input

`\if_eof:w` The primitive conditional

```

10500 \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
10501 \prg_new_conditional:Nnn \ior_if_eof:N { p , T , F , TF }
10502 {
10503   \cs_if_exist:NTF #1
10504   {
10505     \if_int_compare:w #1 = \c_sixteen
10506     \prg_return_true:
10507   } else:
10508     \if_eof:w #1
10509     \prg_return_true:
10510   } else:
10511     \prg_return_false:
10512   \fi:
10513 \fi:
10514 }
10515 { \prg_return_true: }
10516 }

```

(End definition for `\ior_if_eof:NTF`. This function is documented on page 187.)

`\ior_get:NN` And here we read from files.

```

10517 \cs_new_protected:Npn \ior_get:NN #1#2
10518 { \tex_read:D #1 to #2 }

```

(End definition for `\ior_get:NN`. This function is documented on page 186.)

`\ior_get_str:NN` Reading as strings is a more complicated wrapper, as we wish to remove the endline character.

```

10519 \cs_new_protected:Npn \ior_get_str:NN #1#2
10520 {
10521   \use:x
10522   {
10523     \int_set_eq:NN \tex_endlinechar:D \c_minus_one
10524     \exp_not:n { \etex_readline:D #1 to #2 }
10525     \int_set:Nn \tex_endlinechar:D { \int_use:N \tex_endlinechar:D }
10526   }
10527 }

```

(End definition for `\ior_get_str:NN`. This function is documented on page 187.)

`\g__file_internal_ior` Needed by the higher-level code, but cannot be created until here.

```
10528 \ior_new:N \g__file_internal_ior
```

(End definition for `\g__file_internal_ior`. This variable is documented on page 190.)

20.3 Output operations

```
10529 <@@=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.

20.3.1 Variables and constants

`\c_log_iow` Here we allocate two output streams for writing to the transcript file only (`\c_log_iow`)
`\c_term_iow` and to both the terminal and transcript file (`\c_term_iow`).

```
10530 \cs_new_eq:NN \c_log_iow \c_minus_one
```

```
10531 \cs_new_eq:NN \c_term_iow \c_sixteen
```

(End definition for `\c_log_iow` and `\c_term_iow`. These variables are documented on page 190.)

`\g__iow_streams_seq` A list of the currently-available input streams to be used as a stack. Things are done differently in format and package mode, so the starting point varies!

```
10532 \seq_new:N \g__iow_streams_seq
```

```
10533 <*initex>
```

```
10534 \seq_gset_eq:NN \g__iow_streams_seq \g__ior_streams_seq
```

```
10535 </initex>
```

(End definition for `\g__iow_streams_seq`. This variable is documented on page ??.)

`\l__iow_stream_tl` Used to recover the raw stream number from the stack.

```
10536 \tl_new:N \l__iow_stream_tl
```

(End definition for `\l__iow_stream_tl`. This variable is documented on page ??.)

`\g__iow_streams_prop` As for reads with the appropriate adjustment of the register numbers to check on.

```
10537 \prop_new:N \g__iow_streams_prop
```

```
10538 <*package>
```

```
10539 \int_step_inline:nnnn
```

```
10540 { \c_zero }
```

```
10541 { \c_one }
```

```
10542 {
```

```
10543   \cs_if_exist:NTF \normalend
```

```
10544   { \tex_count:D 39 \scan_stop: }
```

```
10545   {
```

```
10546     \tex_count:D 17 \scan_stop:
```

```
10547     \cs_if_exist:NT \loccount { - \c_one }
```

```
10548   }
```

```
10549 }
```

```
10550 {
```

```

10551 \prop_gput:Nnn \g__iow_streams_prop {#1} { Reserved-by-format }
10552 }
10553 \</package>

```

(End definition for \g__iow_streams_prop. This variable is documented on page ??.)

20.4 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.

```

10554 \cs_new_protected:Npn \iow_new:N #1 { \cs_new_eq:NN #1 \c_term_iow }
10555 \cs_generate_variant:Nn \iow_new:N { c }

```

(End definition for \iow_new:N and \iow_new:c. These functions are documented on page 185.)

__iow_new:N As for read streams, copy \newwrite in package mode, making sure that it is not \outer.

```

10556 \*package>
10557 \exp_args:Nnf \cs_new_protected_nopar:Npn \__iow_new:N
10558 { \exp_args:Nnc \exp_after:wN \exp_stop_f: { newwrite } }
10559 \</package>

```

(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
\iow_open:cn conditional version.

```

10560 \cs_new_protected:Npn \iow_open:Nn #1#2
10561 { \__file_name_sanitize:nn {#2} { \__iow_open:Nn #1 } }
10562 \cs_generate_variant:Nn \iow_open:Nn { c }
10563 \cs_new_protected:Npn \__iow_open:Nn #1#2
10564 {
10565   \iow_close:N #1
10566   \seq_gpop:NNTF \g__iow_streams_seq \l__iow_stream_tl
10567   { \__iow_open_stream:Nn #1 {#2} }
10568 \*initex>
10569 { \__msg_kernel_fatal:nn { kernel } { output-streams-exhausted } }
10570 \</initex>
10571 \*package>
10572 {
10573   \__iow_new:N #1
10574   \tl_set:Nx \l__iow_stream_tl { \int_eval:n {#1} }
10575   \__iow_open_stream:Nn #1 {#2}
10576 }
10577 \</package>
10578 }
10579 \cs_generate_variant:Nn \__iow_open:Nn { No }
10580 \cs_new_protected:Npn \__iow_open_stream:Nn #1#2
10581 {
10582   \tex_global:D \tex_chardef:D #1 = \l__iow_stream_tl \scan_stop:
10583   \prop_gput:Nvn \g__iow_streams_prop #1 {#2}
10584   \tex_immediate:D \tex_openout:D #1 #2 \scan_stop:
10585 }

```

(End definition for `\iow_open:Nn` and `\iow_open:cn`. These functions are documented on page 186.)

`\iow_close:N` Closing a stream is not quite the reverse of opening one. First, the close operation is easier than the open one, and second as the stream is actually a number we can use it directly to show that the slot has been freed up.

`\iow_close:c`

```

10586 \cs_new_protected:Npn \iow_close:N #1
10587 {
10588   \int_compare:nT { \c_minus_one < #1 < \c_sixteen }
10589   {
10590     \tex_immediate:D \tex_closeout:D #1
10591     \prop_gremove:NV \g__iow_streams_prop #1
10592     \seq_if_in:NVF \g__iow_streams_seq #1
10593     { \seq_gpush:NV \g__iow_streams_seq #1 }
10594     \cs_gset_eq:NN #1 \c_term_ior
10595   }
10596 }
10597 \cs_generate_variant:Nn \iow_close:N { c }

```

(End definition for `\iow_close:N` and `\iow_close:c`. These functions are documented on page 186.)

`\iow_list_streams:` Done as for input, but with a copy of the auxiliary so the name is correct.
`__iow_list_streams:Nn`

```

10598 \cs_new_protected_nopar:Npn \iow_list_streams:
10599 { \__iow_list_streams:Nn \g__iow_streams_prop { iow } }
10600 \cs_new_eq:NN \__iow_list_streams:Nn \__ior_list_streams:Nn

```

(End definition for `\iow_list_streams:`. This function is documented on page 186.)

20.4.1 Deferred writing

`\iow_shipout_x:Nn` First the easy part, this is the primitive, which expects its argument to be braced.

`\iow_shipout_x:Nx`

`\iow_shipout_x:cn`

`\iow_shipout_x:cx`

```

10601 \cs_new_protected:Npn \iow_shipout_x:Nn #1#2
10602 { \tex_write:D #1 {#2} }
10603 \cs_generate_variant:Nn \iow_shipout_x:Nn { c, Nx, cx }

```

(End definition for `\iow_shipout_x:Nn` and others. These functions are documented on page 188.)

`\iow_shipout:Nn` With ε -TEX available deferred writing without expansion is easy.

`\iow_shipout:Nx`

`\iow_shipout:cn`

`\iow_shipout:cx`

```

10604 \cs_new_protected:Npn \iow_shipout:Nn #1#2
10605 { \tex_write:D #1 { \exp_not:n {#2} } }
10606 \cs_generate_variant:Nn \iow_shipout:Nn { c, Nx, cx }

```

(End definition for `\iow_shipout:Nn` and others. These functions are documented on page 188.)

20.4.2 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.

```

10607 \cs_new_protected:Npn \__iow_with:Nnn #1#2
10608 {
10609   \int_compare:nNnTF {#1} = {#2}
10610   { \use:n }
10611   { \exp_args:No \__iow_with_aux:nNnn { \int_use:N #1 } #1 {#2} }
10612 }
10613 \cs_new_protected:Npn \__iow_with_aux:nNnn #1#2#3#4
10614 {
10615   \int_set:Nn #2 {#3}
10616   #4
10617   \int_set:Nn #2 {#1}
10618 }

```

(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
`\iow_now:Nx` this stream isn't open, the output goes to the terminal instead. If the first argument is
`\iow_now:cn` no output stream at all, we get an internal error. We don't use the expansion done by
`\iow_now:cx` `\write` to get the Nx variant, because it differs in subtle ways from x-expansion, namely, macro parameter characters would not need to be doubled. We set the `\newlinechar` to 10 using `__iow_with:Nnn` to support formats such as plain T_EX: otherwise, `\iow_newline:` would not work. We do not do this for `\iow_shipout:Nn` or `\iow_shipout_x:Nn`, as T_EX looks at the value of the `\newlinechar` at shipout time in those cases.

```

10619 \cs_new_protected:Npn \iow_now:Nn #1#2
10620 {
10621   \__iow_with:Nnn \tex_newlinechar:D { '\^^J }
10622   { \tex_immediate:D \tex_write:D #1 { \exp_not:n {#2} } }
10623 }
10624 \cs_generate_variant:Nn \iow_now:Nn { c, Nx, cx }

```

(End definition for `\iow_now:Nn` and others. These functions are documented on page 187.)

`\iow_log:n` Writing to the log and the terminal directly are relatively easy.
`\iow_log:x` 10625 `\cs_set_protected_nopar:Npn \iow_log:x { \iow_now:Nx \c_log_iow }`
`\iow_term:n` 10626 `\cs_new_protected_nopar:Npn \iow_log:n { \iow_now:Nn \c_log_iow }`
`\iow_term:x` 10627 `\cs_set_protected_nopar:Npn \iow_term:x { \iow_now:Nx \c_term_iow }`
10628 `\cs_new_protected_nopar:Npn \iow_term:n { \iow_now:Nn \c_term_iow }`

(End definition for `\iow_log:n` and `\iow_log:x`. These functions are documented on page 187.)

20.4.3 Special characters for writing

\iow_newline: Global variable holding the character that forces a new line when something is written to an output stream.

```
10629 \cs_new_nopar:Npn \iow_newline: { ^^J }
```

(End definition for \iow_newline:. This function is documented on page 188.)

\iow_char:N Function to write any escaped char to an output stream.

```
10630 \cs_new_eq:NN \iow_char:N \cs_to_str:N
```

(End definition for \iow_char:N. This function is documented on page 188.)

20.4.4 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.

\l_iow_line_count_int This is the “raw” number of characters in a line which can be written to the terminal. The standard value is the line length typically used by T_EXLive and MikT_EX.

```
10631 \int_new:N \l_iow_line_count_int
10632 \int_set:Nn \l_iow_line_count_int { 78 }
```

(End definition for \l_iow_line_count_int. This variable is documented on page 189.)

\l__iow_target_count_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.

```
10633 \int_new:N \l__iow_target_count_int
```

(End definition for \l__iow_target_count_int.)

\l_iow_current_line_int These store the number of characters in the line and word currently being constructed,
\l_iow_current_word_int and the current indentation, respectively.

```
\l_iow_current_indentation_int
10634 \int_new:N \l_iow_current_line_int
10635 \int_new:N \l_iow_current_word_int
10636 \int_new:N \l_iow_current_indentation_int
```

(End definition for \l_iow_current_line_int, \l_iow_current_word_int, and \l_iow_current_indentation_int.)

\l__iow_current_line_tl These hold the current line of text and current word, and a number of spaces for inden-
\l__iow_current_word_tl tation, respectively.

```
\l_iow_current_indentation_tl
10637 \tl_new:N \l__iow_current_line_tl
10638 \tl_new:N \l__iow_current_word_tl
10639 \tl_new:N \l__iow_current_indentation_tl
```

(End definition for \l__iow_current_line_tl, \l__iow_current_word_tl, and \l__iow_current_indentation_tl.)

`\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.

```

10640 \tl_new:N \l__iow_wrap_tl

```

(End definition for `\l__iow_wrap_tl`.)

`\l__iow_newline_tl` The token list inserted to produce the new line, with the *⟨run-on text⟩*.

```

10641 \tl_new:N \l__iow_newline_tl

```

(End definition for `\l__iow_newline_tl`.)

`\l__iow_line_start_bool` Boolean to avoid adding a space at the beginning of forced newlines, and to know when to add the indentation.

```

10642 \bool_new:N \l__iow_line_start_bool

```

(End definition for `\l__iow_line_start_bool`.)

`\c_catcode_other_space_tl` Lowercase a character with category code 12 to produce an “other” space. We can do everything within the group, because `\tl_const:Nn` defines its argument globally.

```

10643 \group_begin:
10644   \char_set_catcode_other:N \*
10645   \char_set_lccode:nn {'\*} {'\ }
10646   \tex_lowercase:D { \tl_const:Nn \c_catcode_other_space_tl { * } }
10647 \group_end:

```

(End definition for `\c_catcode_other_space_tl`. This function is documented on page 190.)

`\c__iow_wrap_marker_tl` Every special action of the wrapping code is preceded by the same recognizable string, `\c__iow_wrap_marker_tl`. Upon seeing that “word”, the wrapping code reads one space-delimited argument to know what operation to perform. The setting of `\escapechar` here is not very important, but makes `\c__iow_wrap_marker_tl` look nicer.

```

\c__iow_wrap_end_marker_tl
\c__iow_wrap_newline_marker_tl
\c__iow_wrap_indent_marker_tl
\c__iow_wrap_unindent_marker_tl
10648 \group_begin:
10649   \int_set_eq:NN \tex_escapechar:D \c_minus_one
10650   \tl_const:Nx \c__iow_wrap_marker_tl
10651     { \tl_to_str:n { ^^I ^^O ^^W ^^_ ^^W ^^R ^^A ^^P } }
10652 \group_end:
10653 \tl_map_inline:nn
10654 { { end } { newline } { indent } { unindent } }
10655 {
10656   \tl_const:cx { c__iow_wrap_ #1 _marker_tl }
10657   {
10658     \c_catcode_other_space_tl
10659     \c__iow_wrap_marker_tl
10660     \c_catcode_other_space_tl
10661     #1
10662     \c_catcode_other_space_tl
10663   }
10664 }

```

(End definition for `\c__iow_wrap_marker_tl`.)

`\iow_indent:n` We give a (protected) error definition to `\iow_indent:n` when outside messages. Within wrapped message, it places the instruction for increasing the indentation before its argument, and the instruction for unindenting afterwards. Note that there will be no forced line-break, so the indentation only changes when the next line is started.

`__iow_indent:n`

`__iow_indent_error:n`

```

10665 \cs_new:Npx \__iow_indent:n #1
10666 {
10667   \c__iow_wrap_indent_marker_tl
10668   #1
10669   \c__iow_wrap_unindent_marker_tl
10670 }
10671 \cs_new:Npn \__iow_indent_error:n #1
10672 {
10673   \_msg_kernel_expandable_error:nn { kernel } { indent-outside-wrapping-code }
10674   #1
10675 }
10676 \cs_new_protected_nopar:Npn \iow_indent:n { \__iow_indent_error:n }

```

(End definition for `\iow_indent:n`. This function is documented on page 189.)

`\iow_wrap:nnnN` The main wrapping function works as follows. First give `\`, `_` and other formatting commands the correct definition for messages, before fully-expanding the input. In package mode, the expansion uses L^AT_EX 2_ε's `\protect` mechanism. Afterwards, set the newline marker (two assignments to fully expand, then convert to a string) and its length, and initialize some registers. There is then a loop over each word in the input, which will do the actual wrapping. After the loop, the resulting text is passed on to the function which has been given as a post-processor. The argument `#4` is available for additional set up steps for the output. The definition of `\` and `_` use an “other” space rather than a normal space, because the latter might be absorbed by T_EX to end a number or other f-type expansions. The `\tl_to_str:N` step converts the “other” space back to a normal space.

`__iow_wrap_set:Nx`

```

10677 \cs_new_protected:Npn \iow_wrap:nnnN #1#2#3#4
10678 {
10679   \group_begin:
10680     \int_set_eq:NN \tex_escapechar:D \c_minus_one
10681     \cs_set_nopar:Npx \{ { \token_to_str:N \{ }
10682     \cs_set_nopar:Npx \# { \token_to_str:N \# }
10683     \cs_set_nopar:Npx \} { \token_to_str:N \} }
10684     \cs_set_nopar:Npx \% { \token_to_str:N \% }
10685     \cs_set_nopar:Npx \~ { \token_to_str:N \~ }
10686     \int_set:Nn \tex_escapechar:D { 92 }
10687     \cs_set_eq:NN \ \ \c__iow_wrap_newline_marker_tl
10688     \cs_set_eq:NN \_ \_ \c_catcode_other_space_tl
10689     \cs_set_eq:NN \iow_indent:n \__iow_indent:n
10690     #3
10691     <*initex>
10692     \tl_set:Nx \l__iow_wrap_tl {#1}
10693     </initex>
10694     <*package>
10695     \__iow_wrap_set:Nx \l__iow_wrap_tl {#1}

```

```
10696 </package>
```

To warn users that `\iow_indent:n` only works in the first argument of `\iow_wrap:nnnN` reset `\iow_indent:n` to its error definition. Then store a newline character and the run-on text as a string in `\l__iow_newline_tl`, and set some variables. The first line's target count is equal to the length of the whole line. The value `\l__iow_target_count_int` is altered later on by `__iow_wrap_set_target:`.

```
10697     \cs_set_eq:NN \iow_indent:n \__iow_indent_error:n
10698     \tl_set:Nx \l__iow_newline_tl { \iow_newline: #2 }
10699     \tl_set:Nx \l__iow_newline_tl { \tl_to_str:N \l__iow_newline_tl }
10700     \int_set_eq:NN \l__iow_target_count_int \l_iow_line_count_int
10701     \tl_clear:N \l__iow_current_indentation_tl
10702     \int_zero:N \l__iow_current_line_int
10703     \tl_set:Nn \l__iow_current_line_tl { \use_none:n }
10704     \bool_set_true:N \l__iow_line_start_bool
```

After some setup above (in particular the odd setting of the current line to `\use_none:n`), a loop goes through space-delimited words in the message, recognizing special markers. To make sure that the first line behaves identically to others, start with a newline marker: the `\use_none:n` above avoids actually getting a new line in the output.

```
10705     \use:x
10706     {
10707         \exp_not:n { \tl_clear:N \l__iow_wrap_tl }
10708         \__iow_wrap_loop:w
10709         \tl_to_str:N \c__iow_wrap_newline_marker_tl
10710         \tl_to_str:N \l__iow_wrap_tl
10711         \tl_to_str:N \c__iow_wrap_end_marker_tl
10712         \c_space_tl \c_space_tl
10713         \exp_not:N \q_stop
10714     }
10715     \exp_args:NNo \group_end:
10716     #4 \l__iow_wrap_tl
10717 }
```

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.

```
10718 <*package>
10719 \cs_new_eq:NN \__iow_wrap_set:Nx \protected@edef
10720 </package>
```

(End definition for `\iow_wrap:nnnN`. This function is documented on page 189.)

`__iow_wrap_set_target:` This is called at the beginning of every line (both those forced by `\\` and those due to line-breaking). The initial call does nothing except redefine `__iow_wrap_set_target:` itself (within the group in which `\iow_wrap:nnnN` works). The next call (at the beginning of the second line) disables any later call and sets the `\l__iow_target_count_int` to the correct value, namely the `\l_iow_line_count_int` shortened by the length of the run-on text (the shift by 1 is due to the presence of `\iow_newline:` in `\l__iow_newline_tl`). This is a bit of a hack to measure the string length of the run on text without the `l3str`

module (which is still experimental). This should be replaced once the string module is finalised with something a little cleaner.

```

10721 \cs_new_protected_nopar:Npn \__iow_wrap_set_target:
10722 {
10723   \cs_set_protected_nopar:Npn \__iow_wrap_set_target:
10724   {
10725     \cs_set_protected_nopar:Npn \__iow_wrap_set_target: { }
10726     \tl_replace_all:Nnn \l__iow_newline_tl { ~ } { \c_space_tl }
10727     \int_set:Nn \l__iow_target_count_int
10728       { \l__iow_line_count_int - \tl_count:N \l__iow_newline_tl + \c_one }
10729   }
10730 }

```

(End definition for __iow_wrap_set_target:.)

__iow_wrap_loop:w The loop grabs one word in the input, and checks whether it is the special marker, or a normal word.

```

10731 \cs_new_protected:Npn \__iow_wrap_loop:w #1~ %
10732 {
10733   \tl_set:Nn \l__iow_current_word_tl {#1}
10734   \tl_if_eq:NNTF \l__iow_current_word_tl \c__iow_wrap_marker_tl
10735     { \__iow_wrap_special:w }
10736     { \__iow_wrap_word: }
10737 }

```

(End definition for __iow_wrap_loop:w.)

__iow_wrap_word: For a normal word, update the line count, then test if the current word would fit in the current line, and call the appropriate function. If the word fits in the current line, add it to the line, preceded by a space unless it is the first word of the line. Otherwise, the current line is added to the result, with the run-on text. The current word (and its character count) are then put in the new line.

```

10738 \cs_new_protected_nopar:Npn \__iow_wrap_word:
10739 {
10740   \int_set:Nn \l__iow_current_word_int
10741     { \exp_args:No \str_count_ignore_spaces:n \l__iow_current_word_tl }
10742   \int_add:Nn \l__iow_current_line_int { \l__iow_current_word_int }
10743   \int_compare:nNnTF \l__iow_current_line_int < \l__iow_target_count_int
10744     { \__iow_wrap_word_fits: }
10745     { \__iow_wrap_word_newline: }
10746   \__iow_wrap_loop:w
10747 }
10748 \cs_new_protected_nopar:Npn \__iow_wrap_word_fits:
10749 {
10750   \bool_if:NNTF \l__iow_line_start_bool
10751     {
10752       \bool_set_false:N \l__iow_line_start_bool
10753       \tl_put_right:Nx \l__iow_current_line_tl
10754         { \l__iow_current_indentation_tl \l__iow_current_word_tl }

```

```

10755         \int_add:Nn \l__iow_current_line_int
10756         { \l__iow_current_indentation_int }
10757     }
10758     {
10759         \tl_put_right:Nx \l__iow_current_line_tl
10760         { ~ \l__iow_current_word_tl }
10761         \int_incr:N \l__iow_current_line_int
10762     }
10763 }
10764 \cs_new_protected_nopar:Npn \__iow_wrap_word_newline:
10765 {
10766     \__iow_wrap_set_target:
10767     \tl_put_right:Nx \l__iow_wrap_tl
10768     { \l__iow_current_line_tl \l__iow_newline_tl }
10769     \int_set:Nn \l__iow_current_line_int
10770     {
10771         \l__iow_current_word_int
10772         + \l__iow_current_indentation_int
10773     }
10774     \tl_set:Nx \l__iow_current_line_tl
10775     { \l__iow_current_indentation_tl \l__iow_current_word_tl }
10776 }

```

(End definition for __iow_wrap_word:.)

```

\__iow_wrap_special:w
\__iow_wrap_newline:w
\__iow_wrap_indent:w
\__iow_wrap_unindent:w
\__iow_wrap_end:w

```

When the “special” marker is encountered, read what operation to perform, as a space-delimited argument, perform it, and remember to loop. In fact, to avoid spurious spaces when two special actions follow each other, we look ahead for another copy of the marker. Forced newlines are almost identical to those caused by overflow, except that here the word is empty. To indent more, add four spaces to the start of the indentation token list. To reduce indentation, rebuild the indentation token list using `\prg_replicate:nn`. At the end, we simply save the last line (without the run-on text), and prevent the loop.

```

10777 \cs_new_protected:Npn \__iow_wrap_special:w #1 ~ #2 ~ #3 ~ %
10778 {
10779     \use:c { __iow_wrap_#1: }
10780     \str_if_eq_x:nnTF { #2~#3 } { ~ \c__iow_wrap_marker_tl }
10781     { \__iow_wrap_special:w }
10782     { \__iow_wrap_loop:w #2 ~ #3 ~ }
10783 }
10784 \cs_new_protected_nopar:Npn \__iow_wrap_newline:
10785 {
10786     \__iow_wrap_set_target:
10787     \tl_put_right:Nx \l__iow_wrap_tl
10788     { \l__iow_current_line_tl \l__iow_newline_tl }
10789     \int_zero:N \l__iow_current_line_int
10790     \tl_clear:N \l__iow_current_line_tl
10791     \bool_set_true:N \l__iow_line_start_bool
10792 }
10793 \cs_new_protected_nopar:Npx \__iow_wrap_indent:

```

```

10794 {
10795   \int_add:Nn \l__iow_current_indentation_int \c_four
10796   \tl_put_right:Nx \exp_not:N \l__iow_current_indentation_tl
10797   { \c_space_tl \c_space_tl \c_space_tl \c_space_tl }
10798 }
10799 \cs_new_protected_nopar:Npn \__iow_wrap_unindent:
10800 {
10801   \int_sub:Nn \l__iow_current_indentation_int \c_four
10802   \tl_set:Nx \l__iow_current_indentation_tl
10803   { \prg_replicate:nn \l__iow_current_indentation_int { ~ } }
10804 }
10805 \cs_new_protected_nopar:Npn \__iow_wrap_end:
10806 {
10807   \tl_put_right:Nx \l__iow_wrap_tl
10808   { \l__iow_current_line_tl }
10809   \use_none_delimit_by_q_stop:w
10810 }

```

(End definition for __iow_wrap_special:w.)

20.5 Messages

```

10811 \__msg_kernel_new:nnnn { kernel } { file-not-found }
10812 { File~'#1'~not-found. }
10813 {
10814   The~requested~file~could~not~be~found~in~the~current~directory,~
10815   in~the~TeX~search~path~or~in~the~LaTeX~search~path.
10816 }
10817 \__msg_kernel_new:nnnn { kernel } { input-streams-exhausted }
10818 { Input~streams~exhausted }
10819 {
10820   TeX~can~only~open~up~to~16~input~streams~at~one~time.\\
10821   All~16~are~currently~in~use,~and~something~wanted~to~open~
10822   another~one.
10823 }
10824 \__msg_kernel_new:nnnn { kernel } { output-streams-exhausted }
10825 { Output~streams~exhausted }
10826 {
10827   TeX~can~only~open~up~to~16~output~streams~at~one~time.\\
10828   All~16~are~currently~in~use,~and~something~wanted~to~open~
10829   another~one.
10830 }
10831 \__msg_kernel_new:nnnn { kernel } { unbalanced-quote-in-filename }
10832 { Unbalanced~quotes~in~file~name~'#1'. }
10833 {
10834   File~names~must~contain~balanced~numbers~of~quotes~(").
10835 }
10836 \__msg_kernel_new:nnn { kernel } { indent-outside-wrapping-code }
10837 { Only~\iow_wrap:nnnN~(arg~1)~allows~\iow_indent:n }
10838 </initex | package)

```

21 l3fp implementation

Nothing to see here: everything is in the subfiles!

22 l3fp-aux implementation

```
10839 <*initex | package>
```

```
10840 <@@=fp>
```

22.1 Internal representation

Internally, a floating point number $\langle X \rangle$ is a token list containing

```
\s__fp \__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`.

Since floating point numbers are always accessed by the various operations using f-expansion, we can safely let them be protected: x-expansion will then leave them untouched. However, when used directly without an accessor function, floating points should produce an error. `\s__fp` will do nothing, and `__fp_chk:w` produces an error.

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. The various possibilities will be distinguished by their $\langle case \rangle$, which is a single digit:⁸

- 0 zeros: `+0` and `-0`,
- 1 “normal” numbers (positive and negative),
- 2 infinities: `+inf` and `-inf`,
- 3 quiet and signalling `nan`.

The $\langle sign \rangle$ is 0 (positive) or 2 (negative), except in the case of `nan`, which have $\langle sign \rangle = 1$. This ensures that changing the $\langle sign \rangle$ digit to $2 - \langle sign \rangle$ is exactly equivalent to changing the sign of the number.

Special floating point numbers have the form

```
\s__fp \__fp_chk:w <case> <sign> \s__fp_... ;
```

where `\s__fp_...` is a scan mark carrying information about how the number was formed (useful for debugging).

Normal floating point numbers ($\langle case \rangle = 1$) have the form

⁸Bruno: I need to implement subnormal numbers. Also, quiet and signalling `nan` must be better distinguished.

Table 1: Internal representation of floating point numbers.

Representation	Meaning
0 0 \s_fp_... ;	Positive zero.
0 2 \s_fp_... ;	Negative zero.
1 0 {<exponent>} {<X ₁ >} {<X ₂ >} {<X ₃ >} {<X ₄ >} ;	Positive floating point.
1 2 {<exponent>} {<X ₁ >} {<X ₂ >} {<X ₃ >} {<X ₄ >} ;	Negative floating point.
2 0 \s_fp_... ;	Positive infinity.
2 2 \s_fp_... ;	Negative infinity.
3 1 \s_fp_... ;	Quiet nan.
3 1 \s_fp_... ;	Signalling nan.

\s_fp _fp_chk:w 1 <sign> {<exponent>} {<X₁>} {<X₂>} {<X₃>} {<X₄>} ;

Here, the <exponent> is an integer, at most \c_fp_max_exponent_int = 10000 in absolute value. The body consists in four blocks of exactly 4 digits, $0000 \leq \langle X_i \rangle \leq 9999$, such that

$$\langle X \rangle = (-1)^{\langle sign \rangle} 10^{-\langle exponent \rangle} \sum_{i=1}^4 \langle X_i \rangle 10^{-4i}$$

and such that the <exponent> is minimal. This implies $1000 \leq \langle X_1 \rangle \leq 9999$.

22.2 Internal storage of floating points numbers

A floating point number <X> is stored as

\s_fp _fp_chk:w <case> <sign> <body> ;

Here, <case> is 0 for ± 0 , 1 for normal numbers, 2 for $\pm \infty$, and 3 for nan, and <sign> is 0 for positive numbers, 1 for nans, and 2 for negative numbers. The <body> of normal numbers is {<exponent>} {<X₁>} {<X₂>} {<X₃>} {<X₄>}, with

$$\langle X \rangle = (-1)^{\langle sign \rangle} 10^{-\langle exponent \rangle} \sum_i \langle X_i \rangle 10^{-4i}.$$

Calculations are done in base 10000, *i.e.* one myriad. The <exponent> lies between $\pm \text{\c_fp_max_exponent_int} = \pm 10000$ inclusive.

Additionally, positive and negative floating point numbers may only be stored with $1000 \leq \langle X_1 \rangle < 10000$. This requirement is necessary in order to preserve accuracy and speed.

22.3 Using arguments and semicolons

_fp_use_none_stop_f:n This function removes an argument (typically a digit) and replaces it by \exp_stop_f:, a marker which stops f-type expansion.

10841 \cs_new:Npn _fp_use_none_stop_f:n #1 { \exp_stop_f: }

(End definition for `__fp_use_none_stop_f:n`.)

`__fp_use_s:n` Those functions place a semicolon after one or two arguments (typically digits).

`__fp_use_s:nn` 10842 `\cs_new:Npn __fp_use_s:n #1 { #1; }`
 10843 `\cs_new:Npn __fp_use_s:nn #1#2 { #1#2; }`

(End definition for `__fp_use_s:n` and `__fp_use_s:nn`.)

`__fp_use_none_until_s:w` Those functions select specific arguments among a set of arguments delimited by a semicolon.
`__fp_use_i_until_s:nw`

`__fp_use_ii_until_s:nnw` 10844 `\cs_new:Npn __fp_use_none_until_s:w #1; { }`
 10845 `\cs_new:Npn __fp_use_i_until_s:nw #1#2; {#1}`
 10846 `\cs_new:Npn __fp_use_ii_until_s:nnw #1#2#3; {#2}`

(End definition for `__fp_use_none_until_s:w`, `__fp_use_i_until_s:nw`, and `__fp_use_ii_until_s:nnw`.)

`__fp_reverse_args:Nww` Many internal functions take arguments delimited by semicolons, and it is occasionally useful to swap two such arguments.

10847 `\cs_new:Npn __fp_reverse_args:Nww #1 #2; #3; { #1 #3; #2; }`

(End definition for `__fp_reverse_args:Nww`.)

`__fp_rrot:www` Rotate three arguments delimited by semicolons. This is the inverse (or the square) of the Forth primitive ROT.

10848 `\cs_new:Npn __fp_rrot:www #1; #2; #3; { #2; #3; #1; }`

(End definition for `__fp_rrot:www`.)

`__fp_use_i:ww` Many internal functions take arguments delimited by semicolons, and it is occasionally useful to remove one or two such arguments.
`__fp_use_i:www`

10849 `\cs_new:Npn __fp_use_i:ww #1; #2; { #1; }`
 10850 `\cs_new:Npn __fp_use_i:www #1; #2; #3; { #1; }`

(End definition for `__fp_use_i:ww` and `__fp_use_i:www`.)

22.4 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
`__fp_chk:w` 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.

10851 `__scan_new:N \s__fp`
 10852 `\cs_new_protected:Npn __fp_chk:w #1 ;`
 10853 `{`
 10854 `__msg_kernel_error:nnx { kernel } { misused-fp }`
 10855 `{ \fp_to_tl:n { \s__fp __fp_chk:w #1 ; } }`
 10856 `}`

(End definition for `\s__fp` and `__fp_chk:w`.)

`\s__fp_mark` Aliases of `\tex_relax:D`, used to terminate expressions.

`\s__fp_stop` 10857 `__scan_new:N \s__fp_mark`
 10858 `__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` 10859 `__scan_new:N \s__fp_invalid`
`\s__fp_overflow` 10860 `__scan_new:N \s__fp_underflow`
`\s__fp_division` 10861 `__scan_new:N \s__fp_overflow`
`\s__fp_exact` 10862 `__scan_new:N \s__fp_division`
 10863 `__scan_new:N \s__fp_exact`

(End definition for `\s__fp_invalid` and others.)

`\c_zero_fp` The special floating points. All of them have the form

`\c_minus_zero_fp` `\s__fp __fp_chk:w <case> <sign> \s__fp...` ;
`\c_inf_fp`

`\c_minus_inf_fp` where the dots in `\s__fp...` are one of `invalid`, `underflow`, `overflow`, `division`,
`\c_nan_fp` `exact`, describing how the floating point was created. We define the floating points here
 as “exact”.

10864 `\tl_const:Nn \c_zero_fp { \s__fp __fp_chk:w 0 0 \s__fp_exact ; }`
 10865 `\tl_const:Nn \c_minus_zero_fp { \s__fp __fp_chk:w 0 2 \s__fp_exact ; }`
 10866 `\tl_const:Nn \c_inf_fp { \s__fp __fp_chk:w 2 0 \s__fp_exact ; }`
 10867 `\tl_const:Nn \c_minus_inf_fp { \s__fp __fp_chk:w 2 2 \s__fp_exact ; }`
 10868 `\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 199.)

`\c__fp_max_exponent_int` Normal floating point numbers have an exponent at most `max_exponent` in absolute value. Larger numbers are rounded to $\pm\infty$. Smaller numbers are subnormal (not implemented yet), and digits beyond $10^{-\text{max_exponent}}$ are rounded away, hence the true minimum exponent is $-\text{max_exponent} - 16$; beyond this, numbers are rounded to zero. Why this choice of limits? When computing $(a \cdot 10^n)^{(b \cdot 10^p)}$, we need to evaluate $\log(a \cdot 10^n) = \log(a) + n \log(10)$ as a fixed point number, which we manipulate as blocks of 4 digits. Multiplying such a fixed point number by $n < 10000$ is much cheaper than larger n , because we can multiply n with each block safely.

10869 `\int_const:Nn \c__fp_max_exponent_int { 10000 }`

(End definition for `\c__fp_max_exponent_int`.)

`__fp_zero_fp:N` In case of overflow or underflow, we have to output a zero or infinity with a given sign.

`__fp_inf_fp:N` 10870 `\cs_new:Npn __fp_zero_fp:N #1`
 10871 `{ \s__fp __fp_chk:w 0 #1 \s__fp_underflow ; }`
 10872 `\cs_new:Npn __fp_inf_fp:N #1`
 10873 `{ \s__fp __fp_chk:w 2 #1 \s__fp_overflow ; }`

(End definition for `__fp_zero_fp:N` and `__fp_inf_fp:N`.)

`__fp_max_fp:N` `__fp_min_fp:N` In some cases, we need to output the smallest or biggest positive or negative finite numbers.

```

10874 \cs_new:Npn \__fp_min_fp:N #1
10875   {
10876     \s__fp \__fp_chk:w 1 #1
10877     { \int_eval:n { - \c__fp_max_exponent_int } }
10878     {1000} {0000} {0000} {0000} ;
10879   }
10880 \cs_new:Npn \__fp_max_fp:N #1
10881   {
10882     \s__fp \__fp_chk:w 1 #1
10883     { \int_use:N \c__fp_max_exponent_int }
10884     {9999} {9999} {9999} {9999} ;
10885   }

```

(End definition for `__fp_max_fp:N` and `__fp_min_fp:N`.)

`__fp_exponent:w` For normal numbers, the function expands to the exponent, otherwise to 0.

```

10886 \cs_new:Npn \__fp_exponent:w \s__fp \__fp_chk:w #1
10887   {
10888     \if_meaning:w 1 #1
10889     \exp_after:wN \__fp_use_ii_until_s:nnw
10890     \else:
10891     \exp_after:wN \__fp_use_i_until_s:nw
10892     \exp_after:wN 0
10893     \fi:
10894   }

```

(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.

```

10895 \cs_new:Npn \__fp_neg_sign:N #1
10896   { \__int_eval:w \c_two - #1 \__int_eval_end: }

```

(End definition for `__fp_neg_sign:N`.)

22.5 Overflow, underflow, and exact zero

`__fp_sanitize:Nw` `__fp_sanitize:wN` `__fp_sanitize_zero:w` 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`.

```

10897 \cs_new:Npn \__fp_sanitize:Nw #1 #2;
10898   {
10899     \if_case:w

```

```

10900         \if_int_compare:w #2 > \c__fp_max_exponent_int \c_one \else:
10901         \if_int_compare:w #2 < - \c__fp_max_exponent_int \c_two \else:
10902         \if_meaning:w 1 #1 \c_three \else: \c_zero \fi: \fi: \fi:
10903     \or: \exp_after:wN \__fp_overflow:w
10904     \or: \exp_after:wN \__fp_underflow:w
10905     \or: \exp_after:wN \__fp_sanitization_zero:w
10906     \fi:
10907     \s__fp \__fp_chk:w 1 #1 {#2}
10908 }
10909 \cs_new:Npn \__fp_sanitization:wN #1; #2 { \__fp_sanitization:Nw #2 #1; }
10910 \cs_new:Npn \__fp_sanitization_zero:w \s__fp \__fp_chk:w #1 #2 #3;
10911 { \c_zero_fp }

```

(End definition for `__fp_sanitization:Nw` and `__fp_sanitization:wN`.)

22.6 Expanding after a floating point number

`__fp_exp_after_o:w` Places *tokens* (empty in the case of `__fp_exp_after_o:w`) between the *floating point* and the *more tokens*, 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.

```

10912 \cs_new:Npn \__fp_exp_after_o:w \s__fp \__fp_chk:w #1
10913 {
10914     \if_meaning:w 1 #1
10915         \exp_after:wN \__fp_exp_after_normal:nNNw
10916     \else:
10917         \exp_after:wN \__fp_exp_after_special:nNNw
10918     \fi:
10919     { }
10920     #1
10921 }
10922 \cs_new:Npn \__fp_exp_after_o:nw #1 \s__fp \__fp_chk:w #2
10923 {
10924     \if_meaning:w 1 #2
10925         \exp_after:wN \__fp_exp_after_normal:nNNw
10926     \else:
10927         \exp_after:wN \__fp_exp_after_special:nNNw
10928     \fi:
10929     { #1 }
10930     #2
10931 }
10932 \cs_new:Npn \__fp_exp_after_f:nw #1 \s__fp \__fp_chk:w #2
10933 {
10934     \if_meaning:w 1 #2
10935         \exp_after:wN \__fp_exp_after_normal:nNNw
10936     \else:
10937         \exp_after:wN \__fp_exp_after_special:nNNw
10938     \fi:

```

```

10939     { \exp:w \exp_end_continue_f:w #1 }
10940     #2
10941   }

```

(End definition for _fp_exp_after_o:w.)

_fp_exp_after_special:nNNw Special floating point numbers are easy to jump over since they contain few tokens.

```

10942 \cs_new:Npn \_fp_exp_after_special:nNNw #1#2#3#4;
10943 {
10944   \exp_after:wN \s__fp
10945   \exp_after:wN \_fp_chk:w
10946   \exp_after:wN #2
10947   \exp_after:wN #3
10948   \exp_after:wN #4
10949   \exp_after:wN ;
10950   #1
10951 }

```

(End definition for _fp_exp_after_special:nNNw.)

_fp_exp_after_normal:nNNw 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.

```

10952 \cs_new:Npn \_fp_exp_after_normal:nNNw #1 1 #2 #3 #4#5#6#7;
10953 {
10954   \exp_after:wN \_fp_exp_after_normal:Nwwwww
10955   \exp_after:wN #2
10956   \__int_value:w #3 \exp_after:wN ;
10957   \__int_value:w 1 #4 \exp_after:wN ;
10958   \__int_value:w 1 #5 \exp_after:wN ;
10959   \__int_value:w 1 #6 \exp_after:wN ;
10960   \__int_value:w 1 #7 \exp_after:wN ; #1
10961 }
10962 \cs_new:Npn \_fp_exp_after_normal:Nwwwww
10963   #1 #2; 1 #3 ; 1 #4 ; 1 #5 ; 1 #6 ;
10964   { \s__fp \_fp_chk:w 1 #1 {#2} {#3} {#4} {#5} {#6} ; }

```

(End definition for _fp_exp_after_normal:nNNw.)

_fp_exp_after_array_f:w

_fp_exp_after_stop_f:nw

```

10965 \cs_new:Npn \_fp_exp_after_array_f:w #1
10966 {
10967   \cs:w \_fp_exp_after \_fp_type_from_scan:N #1 _f:nw \cs_end:
10968   { \_fp_exp_after_array_f:w }
10969   #1
10970 }
10971 \cs_new_eq:NN \_fp_exp_after_stop_f:nw \use_none:nn

```

(End definition for _fp_exp_after_array_f:w.)

22.7 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_use:N \__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_use:N \__int_eval:w - 5 0000
\exp_after:wN \pack:NNNNw
\int_use:N \__int_eval:w 4 9995 0000
+ 12345 * 6677
\exp_after:wN \pack:NNNNw
\int_use:N \__int_eval:w 5 0000 0000
+ 12345 * 8899 ;
```

The `\exp_after:wN` triggers `\int_use:N __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_use:N __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.

```
\__fp_pack:NNNNw
\c__fp_trailing_shift_int
\c__fp_middle_shift_int
\c__fp_leading_shift_int
```

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.

```

10972 \int_const:Nn \c__fp_leading_shift_int { - 5 0000 }
10973 \int_const:Nn \c__fp_middle_shift_int { 5 0000 * 9999 }
10974 \int_const:Nn \c__fp_trailing_shift_int { 5 0000 * 10000 }
10975 \cs_new:Npn \__fp_pack:NNNNNw #1 #2#3#4#5 #6; { + #1#2#3#4#5 ; {#6} }

```

(End definition for __fp_pack:NNNNNw.)

__fp_pack_big:NNNNNw 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 .

```

10976 \int_const:Nn \c__fp_big_leading_shift_int { - 15 2374 }
10977 \int_const:Nn \c__fp_big_middle_shift_int { 15 2374 * 9999 }
10978 \int_const:Nn \c__fp_big_trailing_shift_int { 15 2374 * 10000 }
10979 \cs_new:Npn \__fp_pack_big:NNNNNw #1#2 #3#4#5#6 #7;
10980 { + #1#2#3#4#5#6 ; {#7} }

```

(End definition for __fp_pack_big:NNNNNw.)

__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.

```

10981 \int_const:Nn \c__fp_Bigg_leading_shift_int { - 20 0000 }
10982 \int_const:Nn \c__fp_Bigg_middle_shift_int { 20 0000 * 9999 }
10983 \int_const:Nn \c__fp_Bigg_trailing_shift_int { 20 0000 * 10000 }
10984 \cs_new:Npn \__fp_pack_Bigg:NNNNNw #1#2 #3#4#5#6 #7;
10985 { + #1#2#3#4#5#6 ; {#7} }

```

(End definition for __fp_pack_Bigg:NNNNNw.)

_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.

```

10986 \cs_new:Npn \_fp_pack_twice_four:wNNNNNNNN #1; #2#3#4#5 #6#7#8#9
10987 { #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.

```

10988 \cs_new:Npn \__fp_pack_eight:wNNNNNNNN #1; #2#3#4#5 #6#7#8#9
10989 { #1 {#2#3#4#5#6#7#8#9} ; }

```

(End definition for __fp_pack_eight:wNNNNNNNN.)

22.8 Decimate (dividing by a power of 10)

`_fp_decimate:nNnnnn` 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 numbers, 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 $\langle X'_1 \rangle \langle X'_2 \rangle$ 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 fails for negative $\langle shift \rangle$.

```

10990 \cs_new:Npn \_fp_decimate:nNnnnn #1
10991 {
10992   \cs:w
10993     \_fp_decimate_
10994     \if_int_compare:w \_int_eval:w #1 > \c_sixteen
10995       tiny
10996     \else:
10997       \_int_to_roman:w \_int_eval:w #1
10998     \fi:
10999     :Nnnnn
11000   \cs_end:
11001 }
```

Each of the auxiliaries see the function $\langle f_1 \rangle$, followed by 4 blocks of 4 digits.

(End definition for `_fp_decimate:nNnnnn`.)

`_fp_decimate_:Nnnnn`
`_fp_decimate_tiny:Nnnnn`

If the $\langle shift \rangle$ is zero, or too big, life is very easy.

```

11002 \cs_new:Npn \_fp_decimate_:Nnnnn #1 #2#3#4#5
11003 { #1 0 {#2#3} {#4#5} ; }
11004 \cs_new:Npn \_fp_decimate_tiny:Nnnnn #1 #2#3#4#5
11005 { #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. This triggers the f-expansion of `_fp_decimate_pack:nnnnnnnnnnw`,⁹ responsible for building two blocks of 8 digits, and removing the

⁹No, the argument spec is not a mistake: the function calls an auxiliary to do half of the job.

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.

```

11006 \cs_new:Npn \__fp_tmp:w #1 #2 #3
11007 {
11008   \cs_new:cpn { __fp_decimate_ #1 :Nnnnn } ##1 ##2##3##4##5
11009   {
11010     \exp_after:wN ##1
11011     \__int_value:w
11012     \exp_after:wN \__fp_round_digit:Nw #2 ;
11013     \__fp_decimate_pack:nnnnnnnnnw #3 ;
11014   }
11015 }
11016 \__fp_tmp:w {i} {\use_none:nnn #50}{ 0{#2}#3{#4}#5 }
11017 \__fp_tmp:w {ii} {\use_none:nn #5 }{ 00{#2}#3{#4}#5 }
11018 \__fp_tmp:w {iii} {\use_none:n #5 }{ 000{#2}#3{#4}#5 }
11019 \__fp_tmp:w {iv} { #5 }{ {0000}#2{#3}#4 #5 }
11020 \__fp_tmp:w {v} {\use_none:nnn #4#5 }{ 0{0000}#2{#3}#4 #5 }
11021 \__fp_tmp:w {vi} {\use_none:nn #4#5 }{ 00{0000}#2{#3}#4 #5 }
11022 \__fp_tmp:w {vii} {\use_none:n #4#5 }{ 000{0000}#2{#3}#4 #5 }
11023 \__fp_tmp:w {viii}{ #4#5 }{ {0000}0000{#2}#3 #4 #5 }
11024 \__fp_tmp:w {ix} {\use_none:nnn #3#4+#5}{ 0{0000}0000{#2}#3 #4 #5 }
11025 \__fp_tmp:w {x} {\use_none:nn #3#4+#5}{ 00{0000}0000{#2}#3 #4 #5 }
11026 \__fp_tmp:w {xi} {\use_none:n #3#4+#5}{ 000{0000}0000{#2}#3 #4 #5 }
11027 \__fp_tmp:w {xii} { #3#4+#5}{ {0000}0000{0000}#2 #3 #4 #5 }
11028 \__fp_tmp:w {xiii}{\use_none:nnn#2#3+#4#5}{ 0{0000}0000{0000}#2 #3 #4 #5 }
11029 \__fp_tmp:w {xiv} {\use_none:nn #2#3+#4#5}{ 00{0000}0000{0000}#2 #3 #4 #5 }
11030 \__fp_tmp:w {xv} {\use_none:n #2#3+#4#5}{ 000{0000}0000{0000}#2 #3 #4 #5 }
11031 \__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_round_digit:Nw __fp_round_digit:Nw will receive the “extra digits” as its argument, and its expansion is triggered by __int_value:w. If the first digit is neither 0 nor 5, then it is the *rounding* digit. Otherwise, if the remaining digits are not all zero, we need to add 1 to that leading digit to get the rounding digit. Some caution is required, though, because there may be more than 10 “extra digits”, and this may overflow T_EX’s integers. Instead of feeding the digits directly to __fp_round_digit:Nw, they come split into several blocks, separated by +. Hence the first __int_eval:w here.

The computation of the *rounding* digit leaves an unfinished __int_value:w, which expands the following functions. This allows us to repack nicely the digits we keep. Those digits come as an alternation of unbraced and braced blocks of 4 digits, such that the first 5 groups of token consist in 4 single digits, and one brace group (in some order), and the next 5 have the same structure. This is followed by some digits and a semicolon.

```

11032 \cs_new:Npn \__fp_decimate_pack:nnnnnnnnnw #1#2#3#4#5
11033 { \__fp_decimate_pack:nnnnnnw { #1#2#3#4#5 } }
11034 \cs_new:Npn \__fp_decimate_pack:nnnnnnw #1 #2#3#4#5#6
11035 { {#1} {#2#3#4#5#6} }

```

(End definition for __fp_round_digit:Nw and __fp_decimate_pack:nnnnnnnnnw.)

22.9 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.

```
11036 \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.

```
11037 \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.

```
11038 \cs_new:Npn \__fp_case_return_o:Nw #1#2 \fi: #3 \s__fp #4 ;
11039 { \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.

```
11040 \cs_new:Npn \__fp_case_return_same_o:w #1 \fi: #2 \s__fp
11041 { \fi: \__fp_exp_after_o:w \s__fp }
```

(End definition for `__fp_case_return_same_o:w`.)

`__fp_case_return_o:Nww` Same as `__fp_case_return_o:Nw` but with two trailing floating points.

```
11042 \cs_new:Npn \__fp_case_return_o:Nww #1#2 \fi: #3 \s__fp #4 ; #5 ;
11043 { \fi: \exp_after:wN #1 }
```

(End definition for _fp_case_return_o:Nww.)

_fp_case_return_i_o:ww Similar to _fp_case_return_same_o:w, but this returns the first or second of two
_fp_case_return_ii_o:ww trailing floating point numbers, expanding once after the result.

```
11044 \cs_new:Npn \_fp_case_return_i_o:ww #1 \fi: #2 \s_fp #3 ; \s_fp #4 ;
11045 { \fi: \_fp_exp_after_o:w \s_fp #3 ; }
11046 \cs_new:Npn \_fp_case_return_ii_o:ww #1 \fi: #2 \s_fp #3 ;
11047 { \fi: \_fp_exp_after_o:w }
```

(End definition for _fp_case_return_i_o:ww and _fp_case_return_ii_o:ww.)

22.10 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
_fp_small_int_true:wTF in the range $[-10^8, 10^8]$ and fed as a braced argument to the *⟨true code⟩*. Otherwise, the
_fp_small_int_normal:NnwTF *⟨false code⟩* is performed. First filter special cases: neither **nan** nor infinities are integers.
_fp_small_int_test:NnnwNTF Normal numbers with a non-positive exponent are never integers. When the exponent is
greater than 8, the number is too large for the range. Otherwise, decimate, and test the
digits after the decimal separator. The \use_iii:nnn remove a trailing ; and the true
branch, leaving only the false branch. The _int_value:w appearing in the case where
the normal floating point is an integer takes care of expanding all the conditionals until
the trailing ;. That integer is fed to _fp_small_int_true:wTF which places it as a
braced argument of the true branch. The \use_i:nn in _fp_small_int_test:NnnwNTF
removes the top-level \else: coming from _fp_small_int_normal:NnwTF, hence will
call the \use_iii:nnn which follows, taking the false branch.

```
11048 \cs_new:Npn \_fp_small_int:wTF \s_fp \_fp_chk:w #1#2
11049 {
11050   \if_case:w #1 \exp_stop_f:
11051     \_fp_case_return:nw { \_fp_small_int_true:wTF 0 ; }
11052   \or: \exp_after:wN \_fp_small_int_normal:NnwTF
11053   \or:
11054     \_fp_case_return:nw
11055     {
11056       \exp_after:wN \_fp_small_int_true:wTF \_int_value:w
11057       \if_meaning:w 2 #2 - \fi: 1 0000 0000 ;
11058     }
11059   \else: \_fp_case_return:nw \use_ii:nn
11060   \fi:
11061   #2
11062 }
11063 \cs_new:Npn \_fp_small_int_true:wTF #1; #2#3 { #2 {#1} }
11064 \cs_new:Npn \_fp_small_int_normal:NnwTF #1#2#3;
11065 {
11066   \if_int_compare:w #2 > \c_zero
11067     \_fp_decimate:nNnnnn { \c_sixteen - #2 }
11068     \_fp_small_int_test:NnnwNnw
11069     #3 #1 {#2}
11070   \else:
```

```

11071     \exp_after:wN \use_iii:nnn
11072     \fi:
11073     ;
11074 }
11075 \cs_new:Npn \__fp_small_int_test:NnnwNnw #1#2#3#4; #5#6
11076 {
11077     \if_meaning:w 0 #1
11078     \exp_after:wN \__fp_small_int_true:wTF
11079     \__int_value:w \if_meaning:w 2 #5 - \fi:
11080     \if_int_compare:w #6 > \c_eight
11081     1 0000 0000
11082     \else:
11083     #3
11084     \fi:
11085 \else:
11086     \use_i:nn
11087     \fi:
11088 }

```

(End definition for __fp_small_int:wTF.)

22.11 Length of a floating point array

`__fp_array_count:n` 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.

```

11089 \cs_new:Npn \__fp_array_count:n #1
11090 {
11091     \int_use:N \__int_eval:w \c_zero
11092     \__fp_array_count_loop:Nw #1 { ? \__prg_break: } ;
11093     \__prg_break_point:
11094     \__int_eval_end:
11095 }
11096 \cs_new:Npn \__fp_array_count_loop:Nw #1#2;
11097 { \use_none:n #1 + \c_one \__fp_array_count_loop:Nw }

```

(End definition for __fp_array_count:n.)

22.12 x-like expansion expandably

`__fp_expand:n` 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.

```

11098 \cs_new:Npn \__fp_expand:n #1
11099 {

```

```

11100 \_fp_expand_loop:nwnN { }
11101   #1 \prg_do_nothing:
11102   \s__fp_mark { } \_fp_expand_loop:nwnN
11103   \s__fp_mark { } \_fp_use_i_until_s:nw ;
11104 }
11105 \cs_new:Npn \_fp_expand_loop:nwnN #1#2 \s__fp_mark #3 #4
11106 {
11107   \exp_after:wN #4 \exp:w \exp_end_continue_f:w
11108   #2
11109   \s__fp_mark { #3 #1 } #4
11110 }

```

(End definition for _fp_expand:n.)

22.13 Messages

Using a floating point directly is an error.

```

11111 \_msg_kernel_new:nnnn { kernel } { misused-fp }
11112 { A~floating~point~with~value~'#1'~was~misused. }
11113 {
11114   To~obtain~the~value~of~a~floating~point~variable,~use~
11115   '\token_to_str:N \fp_to_decimal:N',~
11116   '\token_to_str:N \fp_to_scientific:N',~or~other~
11117   conversion~functions.
11118 }
11119 </initex | package>

```

23 l3fp-traps Implementation

```

11120 <*initex | package>
11121 <@@=fp>

```

Exceptions should be accessed by an n-type argument, among

- `invalid_operation`
- `division_by_zero`
- `overflow`
- `underflow`
- `inexact` (actually never used).

23.1 Flags

`\fp_flag_off:n` Function to turn a flag off. Simply undefine it.

```
11122 \cs_new_protected:Npn \fp_flag_off:n #1
11123 { \cs_set_eq:cN { l__fp_ #1 _flag_token } \tex_undefined:D }
```

(End definition for `\fp_flag_off:n`. This function is documented on page 200.)

`\fp_flag_on:n` Function to turn a flag on expandably: use TeX’s automatic assignment to `\scan_stop:.`

```
11124 \cs_new:Npn \fp_flag_on:n #1
11125 { \exp_args:Nc \use_none:n { l__fp_ #1 _flag_token } }
```

(End definition for `\fp_flag_on:n`. This function is documented on page 200.)

`\fp_if_flag_on_p:n` Returns true if the flag is on, false otherwise.

```
\fp_if_flag_on:nTF 11126 \prg_new_conditional:Npnn \fp_if_flag_on:n #1 { p , T , F , TF }
11127 {
11128   \if_cs_exist:w l__fp_ #1 _flag_token \cs_end:
11129   \prg_return_true:
11130   \else:
11131   \prg_return_false:
11132   \fi:
11133 }
```

(End definition for `\fp_if_flag_on:nTF`. This function is documented on page 200.)

`\l_fp_invalid_operation_flag_token` `\l_fp_division_by_zero_flag_token` `\l__fp_overflow_flag_token` `\l__fp_underflow_flag_token` The IEEE standard defines five exceptions. We currently don’t support the “inexact” exception.

```
11134 \cs_new_eq:NN \l__fp_invalid_operation_flag_token \tex_undefined:D
11135 \cs_new_eq:NN \l__fp_division_by_zero_flag_token \tex_undefined:D
11136 \cs_new_eq:NN \l__fp_overflow_flag_token \tex_undefined:D
11137 \cs_new_eq:NN \l__fp_underflow_flag_token \tex_undefined:D
```

(End definition for `\l__fp_invalid_operation_flag_token` and others.)

23.2 Traps

Exceptions can be trapped to obtain custom behaviour. When an invalid operation or a division by zero is trapped, the trap receives as arguments the result as an N -type floating point number, the function name (multiple letters for prefix operations, or a single symbol for infix operations), and the operand(s). When an overflow or underflow is trapped, the trap receives the resulting overly large or small floating point number if it is not too big, otherwise it receives $+\infty$. Currently, the inexact exception is entirely ignored.

The behaviour when an exception occurs is controlled by the definitions of the functions

- `__fp_invalid_operation:nnw`,
- `__fp_invalid_operation_o:Nww`,

- `__fp_invalid_operation_tl_o:ff`,
- `__fp_division_by_zero_o:Nnw`,
- `__fp_division_by_zero_o:NNww`,
- `__fp_overflow:w`,
- `__fp_underflow:w`.

Rather than changing them directly, we provide a user interface as `\fp_trap:nn` $\{\langle exception \rangle\}$ $\{\langle way of trapping \rangle\}$, where the $\langle way of trapping \rangle$ 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`

```

11138 \cs_new_protected:Npn \fp_trap:nn #1#2
11139 {
11140   \cs_if_exist_use:cF { __fp_trap_#1_set_#2: }
11141   {
11142     \clist_if_in:nnTF
11143     { invalid_operation , division_by_zero , overflow , underflow }
11144     {#1}
11145     {
11146       \__msg_kernel_error:nnxx { kernel }
11147       { unknown-fpu-trap-type } {#1} {#2}
11148     }
11149     {
11150       \__msg_kernel_error:nnx
11151       { kernel } { unknown-fpu-exception } {#1}
11152     }
11153   }
11154 }
```

(End definition for `\fp_trap:nn`. This function is documented on page 201.)

`__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.

```

11155 \cs_new_protected_nopar:Npn \__fp_trap_invalid_operation_set_error:
11156 { \__fp_trap_invalid_operation_set:N \prg_do_nothing: }
11157 \cs_new_protected_nopar:Npn \__fp_trap_invalid_operation_set_flag:
11158 { \__fp_trap_invalid_operation_set:N \use_none:nnnnn }
11159 \cs_new_protected_nopar:Npn \__fp_trap_invalid_operation_set_none:
11160 { \__fp_trap_invalid_operation_set:N \use_none:nnnnnnn }
11161 \cs_new_protected:Npn \__fp_trap_invalid_operation_set:N #1
11162 {
11163   \exp_args:Nno \use:n
11164   { \cs_set:Npn \__fp_invalid_operation:nnw ##1##2##3; }
```

```

11165     {
11166         #1
11167         \__fp_error:nfn { invalid } {##2} { \fp_to_tl:n { ##3; } } { }
11168         \fp_flag_on:n { invalid_operation }
11169         ##1
11170     }
11171     \exp_args:Nno \use:n
11172     { \cs_set:Npn \__fp_invalid_operation_o:Nww ##1##2; ##3; }
11173     {
11174         #1
11175         \__fp_error:nfn { invalid-ii }
11176         { \fp_to_tl:n { ##2; } } { \fp_to_tl:n { ##3; } } {##1}
11177         \fp_flag_on:n { invalid_operation }
11178         \exp_after:wN \c_nan_fp
11179     }
11180     \exp_args:Nno \use:n
11181     { \cs_set:Npn \__fp_invalid_operation_tl_o:ff ##1##2 }
11182     {
11183         #1
11184         \__fp_error:nfn { invalid } {##1} {##2} { }
11185         \fp_flag_on:n { invalid_operation }
11186         \exp_after:wN \c_nan_fp
11187     }
11188 }

```

(End definition for __fp_trap_invalid_operation_set_error: and others.)

<pre> __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 </pre>	<p>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.</p>
---	---

```

11189 \cs_new_protected_nopar:Npn \__fp_trap_division_by_zero_set_error:
11190 { \__fp_trap_division_by_zero_set:N \prg_do_nothing: }
11191 \cs_new_protected_nopar:Npn \__fp_trap_division_by_zero_set_flag:
11192 { \__fp_trap_division_by_zero_set:N \use_none:nnnnn }
11193 \cs_new_protected_nopar:Npn \__fp_trap_division_by_zero_set_none:
11194 { \__fp_trap_division_by_zero_set:N \use_none:nnnnnnn }
11195 \cs_new_protected:Npn \__fp_trap_division_by_zero_set:N #1
11196 {
11197     \exp_args:Nno \use:n
11198     { \cs_set:Npn \__fp_division_by_zero_o:Nnw ##1##2##3; }
11199     {
11200         #1
11201         \__fp_error:nfn { zero-div } {##2} { \fp_to_tl:n { ##3; } } { }
11202         \fp_flag_on:n { division_by_zero }
11203         \exp_after:wN ##1
11204     }
11205     \exp_args:Nno \use:n
11206     { \cs_set:Npn \__fp_division_by_zero_o:NNww ##1##2##3; ##4; }
11207     {

```

```

11208         #1
11209         \__fp_error:nffn { zero-div-ii }
11210         { \fp_to_tl:n { ##3; } } { \fp_to_tl:n { ##4; } } {##2}
11211         \fp_flag_on:n { division_by_zero }
11212         \exp_after:wN ##1
11213     }
11214 }

```

(End definition for __fp_trap_division_by_zero_set_error: and others.)

__fp_trap_overflow_set_error: Just as for invalid operations and division by zero, the three different behaviours are 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 T_EX, 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.

```

11215 \cs_new_protected_nopar:Npn \__fp_trap_overflow_set_error:
11216 { \__fp_trap_overflow_set:N \prg_do_nothing: }
11217 \cs_new_protected_nopar:Npn \__fp_trap_overflow_set_flag:
11218 { \__fp_trap_overflow_set:N \use_none:nnnnn }
11219 \cs_new_protected_nopar:Npn \__fp_trap_overflow_set_none:
11220 { \__fp_trap_overflow_set:N \use_none:nnnnnnn }
11221 \cs_new_protected:Npn \__fp_trap_overflow_set:N #1
11222 { \__fp_trap_overflow_set:NnNn #1 { overflow } \__fp_inf_fp:N { inf } }
11223 \cs_new_protected_nopar:Npn \__fp_trap_underflow_set_error:
11224 { \__fp_trap_underflow_set:N \prg_do_nothing: }
11225 \cs_new_protected_nopar:Npn \__fp_trap_underflow_set_flag:
11226 { \__fp_trap_underflow_set:N \use_none:nnnnn }
11227 \cs_new_protected_nopar:Npn \__fp_trap_underflow_set_none:
11228 { \__fp_trap_underflow_set:N \use_none:nnnnnnn }
11229 \cs_new_protected:Npn \__fp_trap_underflow_set:N #1
11230 { \__fp_trap_overflow_set:NnNn #1 { underflow } \__fp_zero_fp:N { 0 } }
11231 \cs_new_protected:Npn \__fp_trap_overflow_set:NnNn #1#2#3#4
11232 {
11233     \exp_args:Nno \use:n
11234     { \cs_set:cpn { __fp_ #2 :w } \s__fp \__fp_chk:w ##1##2##3; }
11235     {
11236         #1
11237         \__fp_error:nffn
11238         { flow \if_meaning:w 1 ##1 -to \fi: }
11239         { \fp_to_tl:n { \s__fp \__fp_chk:w ##1##2##3; } }
11240         { \token_if_eq_meaning:NNF 0 ##2 { - } #4 }
11241         {#2}
11242         \fp_flag_on:n {#2}
11243         #3 ##2
11244     }
11245 }

```

(End definition for `_fp_trap_overflow_set_error:` and others.)

`_fp_invalid_operation:nnw` Initialize the two 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_invalid_operation_o:Nnw
\__fp_invalid_operation_tl:off
\__fp_division_by_zero_o:Nnw
\__fp_division_by_zero_o:NNww
\__fp_overflow:w
\__fp_underflow:w
11246 \cs_new:Npn \__fp_invalid_operation:nnw #1#2#3; { }
11247 \cs_new:Npn \__fp_invalid_operation_o:Nnw #1#2; #3; { }
11248 \cs_new:Npn \__fp_invalid_operation_tl:off #1 #2 { }
11249 \cs_new:Npn \__fp_division_by_zero_o:Nnw #1#2#3; { }
11250 \cs_new:Npn \__fp_division_by_zero_o:NNww #1#2#3; #4; { }
11251 \cs_new:Npn \__fp_overflow:w { }
11252 \cs_new:Npn \__fp_underflow:w { }
11253 \fp_trap:nn { invalid_operation } { error }
11254 \fp_trap:nn { division_by_zero } { flag }
11255 \fp_trap:nn { overflow } { flag }
11256 \fp_trap:nn { underflow } { flag }

```

(End definition for `_fp_invalid_operation:nnw` and others.)

`_fp_invalid_operation_o:nw` Convenient short-hands for returning `\c_nan_fp` for a unary or binary operation, and `_fp_invalid_operation_o:fw` expanding after.

```

11257 \cs_new_nopar:Npn \__fp_invalid_operation_o:nw
11258 { \__fp_invalid_operation:nnw { \exp_after:wN \c_nan_fp } }
11259 \cs_generate_variant:Nn \__fp_invalid_operation_o:nw { f }

```

(End definition for `_fp_invalid_operation_o:nw` and `_fp_invalid_operation_o:fw`.)

23.3 Errors

```

\__fp_error:nnnn
\__fp_error:nnfn
\__fp_error:nffn
11260 \cs_new:Npn \__fp_error:nnnn #1
11261 { \_msg_kernel_expandable_error:nnnnn { kernel } { fp - #1 } }
11262 \cs_generate_variant:Nn \__fp_error:nnnn { nnf, nff }

```

(End definition for `_fp_error:nnnn`, `_fp_error:nnfn`, and `_fp_error:nffn`.)

23.4 Messages

Some messages.

```

11263 \_msg_kernel_new:nnnn { kernel } { unknown-fpu-exception }
11264 {
11265   The~FPU~exception~'#1'~is~not~known:~
11266   that~trap~will~never~be~triggered.
11267 }
11268 {
11269   The~only~exceptions~to~which~traps~can~be~attached~are \
11270   \iow_indent:n
11271   {
11272     * ~ invalid_operation \

```

```

11273         * ~ division_by_zero \\
11274         * ~ overflow \\
11275         * ~ underflow
11276     }
11277 }
11278 \_msg_kernel_new:nnnn { kernel } { unknown-fpu-trap-type }
11279 { The~FPU~trap~type~'#2'~is~not~known. }
11280 {
11281     The~trap~type~must~be~one~of \\
11282     \iow_indent:n
11283     {
11284         * ~ error \\
11285         * ~ flag \\
11286         * ~ none
11287     }
11288 }
11289 \_msg_kernel_new:nnn { kernel } { fp-flow }
11290 { An ~ #3 ~ occurred. }
11291 \_msg_kernel_new:nnn { kernel } { fp-flow-to }
11292 { #1 ~ #3 ed ~ to ~ #2 . }
11293 \_msg_kernel_new:nnn { kernel } { fp-zero-div }
11294 { Division~by~zero~in~ #1 (#2) }
11295 \_msg_kernel_new:nnn { kernel } { fp-zero-div-ii }
11296 { Division~by~zero~in~ (#1) #3 (#2) }
11297 \_msg_kernel_new:nnn { kernel } { fp-invalid }
11298 { Invalid~operation~ #1 (#2) }
11299 \_msg_kernel_new:nnn { kernel } { fp-invalid-ii }
11300 { Invalid~operation~ (#1) #3 (#2) }
11301 </initex | package>

```

24 l3fp-round implementation

```

11302 <*initex | package>
11303 <@@=fp>

```

24.1 Rounding tools

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 `\c_zero` or `\c_one` depending on whether the final result should be rounded up or down.

- `__fp_round:NNN <sign> <digit1> <digit2>` can expand to `\c_zero` or `\c_one`.
- `__fp_round_s:NNNw <sign> <digit1> <digit2> <more digits>`; can expand to `\c_zero` ; or `\c_one` ;.
- `__fp_round_neg:NNN <sign> <digit1> <digit2>` can expand to `\c_zero` or `\c_one`.

See implementation comments for details on the syntax.

```
\__fp_round:NNN
\__fp_round_to_nearest:NNN
  \_fp_round_to_nearest_ninf:NNN
  \_fp_round_to_nearest_zero:NNN
  \_fp_round_to_nearest_pinf:NNN
\__fp_round_to_ninf:NNN
\__fp_round_to_zero:NNN
\__fp_round_to_pinf:NNN
```

If rounding the number $\langle final\ sign \rangle \langle digit_1 \rangle . \langle digit_2 \rangle$ to an integer rounds it towards zero (truncates it), this function expands to `\c_zero`, and otherwise to `\c_one`. 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 $\langle final\ sign \rangle$ 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 $\langle final\ sign \rangle$ is 0 for positive, and 2 for negative.

By default, the functions below return `\c_zero`, but this is superseded by `__fp_round_return_one:`, which instead returns `\c_one`, expanding everything and removing `\c_zero` 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 $\langle digit_2 \rangle$ is larger than 5, then round up. If it is less than 5, round down. If it is exactly 5, then round such that $\langle digit_1 \rangle$ plus the result is even. In other words, round up if $\langle digit_1 \rangle$ 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 $+\infty$.

```
11304 \cs_new:Npn \__fp_round_return_one:
11305 { \exp_after:wN \c_one \exp:w }
11306 \cs_new:Npn \__fp_round_to_ninf:NNN #1 #2 #3
11307 {
11308   \if_meaning:w 2 #1
11309     \if_int_compare:w #3 > \c_zero
11310       \__fp_round_return_one:
11311     \fi:
11312   \fi:
```

```

11313     \c_zero
11314 }
11315 \cs_new:Npn \__fp_round_to_zero:NNN #1 #2 #3 { \c_zero }
11316 \cs_new:Npn \__fp_round_to_pinf:NNN #1 #2 #3
11317 {
11318     \if_meaning:w 0 #1
11319     \if_int_compare:w #3 > \c_zero
11320     \__fp_round_return_one:
11321     \fi:
11322     \fi:
11323     \c_zero
11324 }
11325 \cs_new:Npn \__fp_round_to_nearest:NNN #1 #2 #3
11326 {
11327     \if_int_compare:w #3 > \c_five
11328     \__fp_round_return_one:
11329     \else:
11330     \if_meaning:w 5 #3
11331     \if_int_odd:w #2 \exp_stop_f:
11332     \__fp_round_return_one:
11333     \fi:
11334     \fi:
11335     \fi:
11336     \c_zero
11337 }
11338 \cs_new:Npn \__fp_round_to_nearest_ninf:NNN #1 #2 #3
11339 {
11340     \if_int_compare:w #3 > \c_five
11341     \__fp_round_return_one:
11342     \else:
11343     \if_meaning:w 5 #3
11344     \if_meaning:w 2 #1
11345     \__fp_round_return_one:
11346     \fi:
11347     \fi:
11348     \fi:
11349     \c_zero
11350 }
11351 \cs_new:Npn \__fp_round_to_nearest_zero:NNN #1 #2 #3
11352 {
11353     \if_int_compare:w #3 > \c_five
11354     \__fp_round_return_one:
11355     \fi:
11356     \c_zero
11357 }
11358 \cs_new:Npn \__fp_round_to_nearest_pinf:NNN #1 #2 #3
11359 {
11360     \if_int_compare:w #3 > \c_five
11361     \__fp_round_return_one:
11362     \else:

```

```

11363     \if_meaning:w 5 #3
11364     \if_meaning:w 0 #1
11365         \__fp_round_return_one:
11366     \fi:
11367     \fi:
11368     \fi:
11369     \c_zero
11370 }
11371 \cs_new_eq:NN \__fp_round:NNN \__fp_round_to_nearest:NNN

```

(End definition for __fp_round:NNN.)

__fp_round_s:NNNw Similar to __fp_round:NNN, but with an extra semicolon, this function expands to \c_zero ; if rounding $\langle final\ sign \rangle \langle digit \rangle . \langle more\ digits \rangle$ to an integer truncates, and to \c_one ; 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.

```

11372 \cs_new:Npn \__fp_round_s:NNNw #1 #2 #3 #4;
11373 {
11374     \exp_after:wN \__fp_round:NNN
11375     \exp_after:wN #1
11376     \exp_after:wN #2
11377     \int_use:N \__int_eval:w
11378     \if_int_odd:w 0 \if_meaning:w 0 #3 1 \fi:
11379         \if_meaning:w 5 #3 1 \fi:
11380         \exp_stop_f:
11381     \if_int_compare:w \__int_eval:w #4 > \c_zero
11382         1 +
11383     \fi:
11384     \fi:
11385     #3
11386 ;
11387 }

```

(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.

```

11388 \cs_new:Npn \__fp_round_digit:Nw #1 #2;
11389 {
11390     \if_int_odd:w \if_meaning:w 0 #1 \c_one \else:
11391         \if_meaning:w 5 #1 \c_one \else:
11392             \c_zero \fi: \fi:
11393     \if_int_compare:w \__int_eval:w #2 > \c_zero
11394         \__int_eval:w \c_one +
11395     \fi:
11396     \fi:
11397     #1

```

```
11398 }
```

(End definition for `_fp_round_digit:Nw`.)

```
\_fp\_round\_neg:NNN
\_fp\_round\_to\_nearest\_neg:NNN
\_fp\_round\_to\_nearest\_ninf\_neg:NNN
\_fp\_round\_to\_nearest\_zero\_neg:NNN
\_fp\_round\_to\_nearest\_pinf\_neg:NNN
\_fp\_round\_to\_ninf\_neg:NNN
\_fp\_round\_to\_zero\_neg:NNN
\_fp\_round\_to\_pinf\_neg:NNN
```

This expands to `\c_zero` or `\c_one` after doing the following test. Starting from a number of the form $\langle final\ sign \rangle 0.\langle 15\ digits \rangle \langle digit_1 \rangle$ with exactly 15 (non-all-zero) digits before $\langle digit_1 \rangle$, subtract from it $\langle final\ sign \rangle 0.0\dots 0\langle digit_2 \rangle$, where there are 16 zeros. If in the current rounding mode the result should be rounded down, then this function returns `\c_one`. Otherwise, *i.e.*, if the result is rounded back to the first operand, then this function returns `\c_zero`.

It turns out that this negative “round to nearest” is identical to the positive one. And this is the default mode.

```
11399 \cs_new_eq:NN \_fp\_round\_to\_ninf\_neg:NNN \_fp\_round\_to\_pinf:NNN
11400 \cs_new:Npn \_fp\_round\_to\_zero\_neg:NNN #1 #2 #3
11401 {
11402   \if_int_compare:w #3 > \c\_zero
11403     \_fp\_round\_return\_one:
11404   \fi:
11405   \c\_zero
11406 }
11407 \cs_new_eq:NN \_fp\_round\_to\_pinf\_neg:NNN \_fp\_round\_to\_ninf:NNN
11408 \cs_new_eq:NN \_fp\_round\_to\_nearest\_neg:NNN \_fp\_round\_to\_nearest:NNN
11409 \cs_new_eq:NN \_fp\_round\_to\_nearest\_ninf\_neg:NNN \_fp\_round\_to\_nearest\_pinf:NNN
11410 \cs_new:Npn \_fp\_round\_to\_nearest\_zero\_neg:NNN #1 #2 #3
11411 {
11412   \if_int_compare:w #3 > \c\_four
11413     \_fp\_round\_return\_one:
11414   \fi:
11415   \c\_zero
11416 }
11417 \cs_new_eq:NN \_fp\_round\_to\_nearest\_pinf\_neg:NNN \_fp\_round\_to\_nearest\_ninf:NNN
11418 \cs_new_eq:NN \_fp\_round\_neg:NNN \_fp\_round\_to\_nearest\_neg:NNN
```

(End definition for `_fp_round_neg:NNN`.)

24.2 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.

```
11419 \cs_new:Npn \_fp\_round\_o:Nw #1#2 @
11420 {
11421   \if_case:w
11422     \_int_eval:w \_fp\_array\_count:n {#2} - \c\_one \_int\_eval\_end:
11423     \_fp\_round:Nwn #1 #2 {0} \exp:w
11424   \or: \_fp\_round:Nww #1 #2 \exp:w
11425   \else: \_fp\_round:Nwww #1 #2 @ \exp:w
11426   \fi:
11427   \exp\_end\_continue\_f:w
11428 }
```

(End definition for _fp_round_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.

```

11429 \cs_new:Npn \_fp\_round:Nwww #1#2 ; #3 ; \s\_fp \_fp\_chk:w #4#5#6 ; #7 @
11430 {
11431   \cs_if_eq:NNTF #1 \_fp\_round\_to\_nearest:NNN
11432   {
11433     \tl_if_empty:nTF {#7}
11434     {
11435       \exp_args:Nc \_fp\_round:Nww
11436       {
11437         \_fp\_round\_to\_nearest
11438         \if_meaning:w 0 #4 _zero \else:
11439         \if_case:w #5 \exp\_stop\_f: _pinf \or: \else: _ninf \fi: \fi:
11440         :NNN
11441       }
11442       #2 ; #3 ;
11443     }
11444     {
11445       \_fp\_error:nnnn { num-args } { round () } { 1 } { 3 }
11446       \exp\_after:wN \c\_nan\_fp
11447     }
11448   }
11449   {
11450     \_fp\_error:nffn { num-args }
11451     { \_fp\_round\_name\_from\_cs:N #1 () } { 1 } { 2 }
11452     \exp\_after:wN \c\_nan\_fp
11453   }
11454 }

```

(End definition for _fp_round:Nwww.)

_fp_round_name_from_cs:N

```

11455 \cs_new:Npn \_fp\_round\_name\_from\_cs:N #1
11456 {
11457   \cs_if_eq:NNTF #1 \_fp\_round\_to\_zero:NNN { trunc }
11458   {
11459     \cs_if_eq:NNTF #1 \_fp\_round\_to\_ninf:NNN { floor }
11460     {
11461       \cs_if_eq:NNTF #1 \_fp\_round\_to\_pinf:NNN { ceil }
11462       { round }
11463     }
11464   }
11465 }

```

(End definition for _fp_round_name_from_cs:N.)

```

    \__fp_round:Nww
    \__fp_round:Nwn
11466 \cs_new:Npn \__fp_round:Nww #1#2 ; #3 ;
11467 {
11468     \__fp_small_int:wTF #3; { \__fp_round:Nwn #1#2; }
11469     {
11470         \__fp_invalid_operation_tl_o:ff
11471         { \__fp_round_name_from_cs:N #1 }
11472         { \__fp_array_to_clist:n { #2; #3; } }
11473     }
11474 }
11475 \cs_new:Npn \__fp_round:Nwn #1 \s__fp \__fp_chk:w #2#3#4; #5
11476 {
11477     \if_meaning:w 1 #2
11478     \exp_after:wN \__fp_round_normal:NwNNnw
11479     \exp_after:wN #1
11480     \__int_value:w #5
11481     \else:
11482     \exp_after:wN \__fp_exp_after_o:w
11483     \fi:
11484     \s__fp \__fp_chk:w #2#3#4;
11485 }
11486 \cs_new:Npn \__fp_round_normal:NwNNnw #1#2 \s__fp \__fp_chk:w 1#3#4#5;
11487 {
11488     \__fp_decimate:nNnnnn { \c_sixteen - #4 - #2 }
11489     \__fp_round_normal:NnnwNNnn #5 #1 #3 {#4} {#2}
11490 }
11491 \cs_new:Npn \__fp_round_normal:NnnwNNnn #1#2#3#4; #5#6
11492 {
11493     \exp_after:wN \__fp_round_normal:NNwNnn
11494     \int_use:N \__int_eval:w
11495     \if_int_compare:w #2 > \c_zero
11496     1 \__int_value:w #2
11497     \exp_after:wN \__fp_round_pack:Nw
11498     \int_use:N \__int_eval:w 1#3 +
11499     \else:
11500     \if_int_compare:w #3 > \c_zero
11501     1 \__int_value:w #3 +
11502     \fi:
11503     \fi:
11504     \exp_after:wN #5
11505     \exp_after:wN #6
11506     \use_none:nnnnnnn #3
11507     #1
11508     \__int_eval_end:
11509     0000 0000 0000 0000 ; #6
11510 }
11511 \cs_new:Npn \__fp_round_pack:Nw #1
11512 { \if_meaning:w 2 #1 + \c_one \fi: \__int_eval_end: }
11513 \cs_new:Npn \__fp_round_normal:NNwNnn #1 #2

```

```

11514 {
11515   \if_meaning:w 0 #2
11516     \exp_after:wN \__fp_round_special:NwwNnn
11517     \exp_after:wN #1
11518   \fi:
11519   \__fp_pack_twice_four:wNNNNNNNN
11520   \__fp_pack_twice_four:wNNNNNNNN
11521   \__fp_round_normal_end:wwNnn
11522   ; #2
11523 }
11524 \cs_new:Npn \__fp_round_normal_end:wwNnn #1;#2;#3#4#5
11525 {
11526   \exp_after:wN \__fp_exp_after_o:w \exp:w \exp_end_continue_f:w
11527   \__fp_sanitize:Nw #3 #4 ; #1 ;
11528 }
11529 \cs_new:Npn \__fp_round_special:NwwNnn #1#2;#3;#4#5#6
11530 {
11531   \if_meaning:w 0 #1
11532     \__fp_case_return:nw
11533     { \exp_after:wN \__fp_zero_fp:N \exp_after:wN #4 }
11534   \else:
11535     \exp_after:wN \__fp_round_special_aux:Nw
11536     \exp_after:wN #4
11537     \int_use:N \__int_eval:w \c_one
11538     \if_meaning:w 1 #1 -#6 \else: +#5 \fi:
11539   \fi:
11540   ;
11541 }
11542 \cs_new:Npn \__fp_round_special_aux:Nw #1#2;
11543 {
11544   \exp_after:wN \__fp_exp_after_o:w \exp:w \exp_end_continue_f:w
11545   \__fp_sanitize:Nw #1#2; {1000}{0000}{0000}{0000};
11546 }

```

(End definition for __fp_round:Nww and __fp_round:Nwn.)

```

11547 </initex | package>

```

25 l3fp-parse implementation

```

11548 <*initex | package>
11549 <@@=fp>

```

25.1 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.

T_EXhackers note: Registers (integers, toks, etc.) are automatically unpacked, without requiring a function such as `\int_use:N`. Invalid tokens remaining after `f`-expansion 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.
- 14 Binary `**` and `^` (right to left).
- 12 Unary `+`, `-`, `!` (right to left).
- 10 Binary `*`, `/`, and juxtaposition (implicit `*`).
- 9 Binary `+` and `-`.
- 7 Comparisons.
- 5 Logical `and`, denoted by `&&`.
- 4 Logical `or`, denoted by `||`.
- 3 Ternary operator `?:`, piece `?`.
- 2 Ternary operator `?:`, piece `:`.
- 1 Commas, and parentheses accepting commas.
- 0 Parentheses expecting exactly one argument.
- 1 Start and end of the expression.

25.1.1 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 `\c_zero`, 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 \c_zero 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 `\c_zero`, hence expands to nothing. Now, `__int_value:w` sees the `;`, which cannot be part of a number. The expansion stops, and we are left with

```
\add:ww 12345 ; 333444 ;
```

which can safely perform the addition by grabbing two arguments delimited by `;`.

If we were to continue parsing the expression, then the following number should also be cleaned up before the next use of a binary operation such as `\add:ww`. Just like `__int_value:w 12345 \exp_after:wN ;` expanded what follows once, we need `\add:ww` to do the calculation, and in the process to expand the following once. This is also true in our real application: all the functions of the form `__fp_..._o:ww` expand what follows once. This comes at the cost of leaving tokens in the input stack, and we 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.

25.1.2 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_three`, `\c_nine`, ...) 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 \wedge .
- Compare the precedences of $-$ and \wedge . Since the latter is higher, we need to compute the exponentiation. For this, find the second operand with a nested call to `\operand:Nw \wedge`.
- Clean up 3 and find $*$.
- Compare the precedences of \wedge and $*$. Since the former is higher, `\operand:Nw \wedge` 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 $\langle precedence \rangle$ the argument of `__fp_parse_operand:Nw`, that is, the precedence of the binary operator whose operand we are trying to find. The basic action is to read numbers from the input stream. This is done by `__fp_parse_one:Nw`. A first approximation of this function is that it reads one $\langle number \rangle$, performing no computation, and finds the following binary $\langle operator \rangle$. Then it expands to

```
 $\langle number \rangle$ 
\__fp_parse_infix_ $\langle operator \rangle$ :N  $\langle precedence \rangle$ 
```

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_` $\langle operator \rangle$: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 $\langle precedence \rangle$ (of the earlier operator) to the `infix` auxiliary for the following $\langle operator \rangle$, to know whether to perform the computation of the $\langle operator \rangle$. If it should not be performed, the `infix` auxiliary expands to

```
@ \use_none:n \__fp_parse_infix_
```

and otherwise it calls `__fp_parse_operand:Nw` with the precedence of the $\langle operator \rangle$ to find its second operand $\langle number_2 \rangle$ and the next $\langle operator_2 \rangle$, and expands to

```
@ \__fp_parse_apply_binary:NwNwN
 $\langle operator \rangle$   $\langle number_2 \rangle$ 
@ \__fp_parse_infix_
```

The `infix` function is responsible for comparing precedences, but cannot directly call the computation functions, because the first operand $\langle number \rangle$ 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` $\langle precedence \rangle$ with some of the expansion control removed is

```
\exp_after:wN \__fp_parse_continue:NwN
\exp_after:wN  $\langle precedence \rangle$ 
\exp:w \exp_end_continue_f:w
\__fp_parse_one:Nw  $\langle precedence \rangle$ 
```

This expands `__fp_parse_one:Nw` $\langle precedence \rangle$ completely, which finds a number, wraps the next $\langle operator \rangle$ into an infix function, feeds this function the $\langle precedence \rangle$, and expands it, yielding either

```
\__fp_parse_continue:NwN  $\langle precedence \rangle$ 
 $\langle number \rangle$  @
\use_none:n \__fp_parse_infix_ $\langle operator \rangle$ :N
```

or

```
\__fp_parse_continue:NwN  $\langle precedence \rangle$ 
 $\langle number \rangle$  @
\__fp_parse_apply_binary:NwNwN
 $\langle operator \rangle$   $\langle number_2 \rangle$ 
@ \__fp_parse_infix_ $\langle operator_2 \rangle$ :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  $\langle precedence \rangle$   $\langle number \rangle$  @
\__fp_parse_infix_ $\langle operator \rangle$ :N
```

then $\langle number \rangle$ @ `__fp_parse_infix_ $\langle operator \rangle$:N`. In the second case, `#3` is `__fp_parse_apply_binary:NwNwN`, whose role is to compute $\langle number \rangle$ $\langle operator \rangle$ $\langle number_2 \rangle$ and to prepare for the next comparison of precedences: first we get

```
\__fp_parse_apply_binary:NwNwN
 $\langle precedence \rangle$   $\langle number \rangle$  @
 $\langle operator \rangle$   $\langle number_2 \rangle$ 
@ \__fp_parse_infix_ $\langle operator_2 \rangle$ :N
```

then

```
\exp_after:wN \__fp_parse_continue:NwN
\exp_after:wN  $\langle precedence \rangle$ 
\exp:w \exp_end_continue_f:w
\__fp_ $\langle operator \rangle$ _o:ww  $\langle number \rangle$   $\langle number_2 \rangle$ 
\exp:w \exp_end_continue_f:w
\__fp_parse_infix_ $\langle operator_2 \rangle$ :N  $\langle precedence \rangle$ 
```

where `__fp_ $\langle operator \rangle$ _o:ww` computes $\langle number \rangle$ $\langle operator \rangle$ $\langle number_2 \rangle$ and expands after the result, thus triggers the comparison of the precedence of the $\langle operator_2 \rangle$ and the $\langle precedence \rangle$, continuing the loop.

We have introduced the most important functions here, and the next few paragraphs will describe various subtleties.

25.1.3 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 $\langle number \rangle$ and the next infix $\langle operator \rangle$. If what follows `_fp_parse_one:Nw` $\langle precedence \rangle$ 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 $\langle precedence \rangle$ of the previous operator, but $0>-2+3$ is then parsed as $0>-(2+3)$: the addition is performed because it binds more tightly than the comparison which precedes $-$. The correct approach is for a unary $-$ to perform operations whose precedence is greater than both that of the previous operation, and that of the unary $-$ itself. The unary $-$ is given a precedence higher than multiplication and division. This does not lead to any surprising result, since $-(x/y) = (-x)/y$ and similarly for multiplication, and it reduces the number of nested calls to `_fp_parse_operand:Nw`.

Functions are implemented as prefix operators with very high precedence, so that their argument is the first number that can possibly be built.

Note that contrarily to the `infix` functions discussed earlier, the `prefix` functions do perform tests on the previous $\langle precedence \rangle$ to decide whether to find an argument or not, since we know that we need a number, and must never stop there.

25.1.4 Numbers and reading tokens one by one

So far, we have glossed over one important point: what is a “number”? A number is typically given in the form $\langle significand \rangle \mathbf{e} \langle exponent \rangle$, where the $\langle significand \rangle$ is any non-empty string composed of decimal digits and at most one decimal separator (a period), the exponent “ $\mathbf{e} \langle exponent \rangle$ ” 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 $\langle significand \rangle$ can also be an integer, dimension, skip, or muskip variable, in which case dimensions are converted from points (or mu units) to floating points, and the $\langle exponent \rangle$

can also be an integer variable. Numbers can also be given as floating point variables, or as named constants such as `nan`, `inf` or `pi`. We may add more types in the future.

When `__fp_parse_one:Nw` is looking for a “number”, here is what happens.

- If the next token is a control sequence with the meaning of `\scan_stop:`, it can be: `\s__fp`, in which case our job is done, as what follows is an internal floating point number, or `\s__fp_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_three`, 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_three 2` is disallowed.

In the above, we need to test whether a character token `#1` is a digit:

```
\if_int_compare:w \c_nine < 1 \token_to_str:N #1 \exp_stop_f:
  is a digit
\else:
```

```

    not a digit
\fi:

```

To exclude 0, replace `\c_nine` by `\c_ten`. 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 = \c_three
  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`.

25.2 Main auxiliary functions

`__fp_parse_operand:Nw` Reads the “...”, performing every computation with a precedence higher than $\langle precedence \rangle$, then expands to where the $\langle operation \rangle$ is the first operation with a lower precedence, possibly `end`, and the “...” start just after the $\langle operation \rangle$.

(End definition for `__fp_parse_operand:Nw`.)

`__fp_parse_infix_+:N` If `+` has a precedence higher than the $\langle precedence \rangle$, cleans up a second $\langle operand \rangle$ and finds the $\langle operation_2 \rangle$ 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 $\langle precedence \rangle$. If the following $\langle operation \rangle$ has a precedence higher than $\langle precedence \rangle$, expands to and otherwise expands to

(End definition for `__fp_parse_one:Nw`.)

25.3 Helpers

`__fp_parse_expand:w` This function must always come within a `\exp:w` expansion. The $\langle tokens \rangle$ should be the part of the expression that we have not yet read. This requires in particular closing all conditionals properly before expanding.

```
11550 \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.

```
11551 \cs_new:Npn \__fp_parse_return_semicolon:w
11552     #1 \fi: \__fp_parse_expand:w { \fi: ; #1 }
```

(End definition for `__fp_parse_return_semicolon:w`.)

`__fp_type_from_scan:N` Grabs the pieces of the stringified $\langle token \rangle$ which lies after the first `s__fp`. If the $\langle token \rangle$ does not contain that string, the result is `_?`.

```
\__fp_type_from_scan:w
11553 \cs_new:Npx \__fp_type_from_scan:N #1
11554 {
11555     \exp_not:N \exp_after:wN \exp_not:N \__fp_type_from_scan:w
11556     \exp_not:N \token_to_str:N #1 \exp_not:N \q_mark
11557     \tl_to_str:n { s__fp _? } \exp_not:N \q_mark \exp_not:N \q_stop
11558 }
11559 \use:x
11560 {
11561     \cs_new:Npn \exp_not:N \__fp_type_from_scan:w
11562         ##1 \tl_to_str:n { s__fp } ##2 \exp_not:N \q_mark ##3 \exp_not:N \q_stop
11563         {##2}
11564 }
```

(End definition for `__fp_type_from_scan:N` and `__fp_type_from_scan:w`.)

`__fp_parse_digits_vii: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

`__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`

$\langle digits \rangle$; $\langle filling\ 0 \rangle$; $\langle length \rangle$

where $\langle filling\ 0 \rangle$ is a string of zeros such that $\langle digits \rangle \langle filling\ 0 \rangle$ has the length given by the index of the function, and $\langle length \rangle$ is the number of zeros in the $\langle filling\ 0 \rangle$ 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.

```

11565 \cs_set_protected:Npn \__fp_tmp:w #1 #2 #3
11566 {
11567   \cs_new:cpn { __fp_parse_digits_ #1 :N } ##1
11568   {
11569     \if_int_compare:w \c_nine < 1 \token_to_str:N ##1 \exp_stop_f:
11570     \token_to_str:N ##1 \exp_after:wN #2 \exp:w
11571   \else:
11572     \__fp_parse_return_semicolon:w #3 ##1
11573   \fi:
11574   \__fp_parse_expand:w
11575 }
11576 }
11577 \__fp_tmp:w {vii} \__fp_parse_digits_vii:N { 0000000 ; 7 }
11578 \__fp_tmp:w {vi} \__fp_parse_digits_vi:N { 000000 ; 6 }
11579 \__fp_tmp:w {v} \__fp_parse_digits_iv:N { 00000 ; 5 }
11580 \__fp_tmp:w {iv} \__fp_parse_digits_iii:N { 0000 ; 4 }
11581 \__fp_tmp:w {iii} \__fp_parse_digits_ii:N { 000 ; 3 }
11582 \__fp_tmp:w {ii} \__fp_parse_digits_i:N { 00 ; 2 }
11583 \__fp_tmp:w {i} \__fp_parse_digits_:N { 0 ; 1 }
11584 \cs_new_nopar:Npn \__fp_parse_digits_:N { ; ; 0 }

```

(End definition for `__fp_parse_digits_vii:N` and others.)

25.4 Parsing one number

`__fp_parse_one:Nw` This function finds one number, and packs the symbol which follows in an `\..._infix...` csname. `#1` is the previous $\langle precedence \rangle$, 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 $\text{\LaTeX} 2_{\epsilon}$ command `\protect`. Using a well placed `\reverse_if:N`, this case is sent to `__fp_parse_one_fp:NN` which deals with it robustly.

```

11585 \cs_new:Npn \__fp_parse_one:Nw #1 #2
11586 {
11587   \if_catcode:w \scan_stop: \exp_not:N #2
11588   \exp_after:wN \if_meaning:w \exp_not:N #2 #2 \else:
11589   \exp_after:wN \reverse_if:N
11590   \fi:
11591   \if_meaning:w \scan_stop: #2
11592   \exp_after:wN \exp_after:wN
11593   \exp_after:wN \__fp_parse_one_fp:NN
11594   \else:

```

```

11595         \exp_after:wN \exp_after:wN
11596         \exp_after:wN \__fp_parse_one_register:NN
11597     \fi:
11598 \else:
11599     \if_int_compare:w \c_nine < 1 \token_to_str:N #2 \exp_stop_f:
11600         \exp_after:wN \exp_after:wN
11601         \exp_after:wN \__fp_parse_one_digit:NN
11602     \else:
11603         \exp_after:wN \exp_after:wN
11604         \exp_after:wN \__fp_parse_one_other:NN
11605     \fi:
11606 \fi:
11607 #1 #2
11608 }

```

(End definition for __fp_parse_one:Nw.)

```

\__fp_parse_one_fp:NN
\__fp_exp_after_mark_f:nw
\__fp_exp_after_?_f: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”.

```

11609 \cs_new:Npn \__fp_parse_one_fp:NN #1#2
11610 {
11611     \cs:w __fp_exp_after \__fp_type_from_scan:N #2 _f:nw \cs_end:
11612     {
11613         \exp_after:wN \__fp_parse_infix:NN
11614         \exp_after:wN #1 \exp:w \__fp_parse_expand:w
11615     }
11616     #2
11617 }
11618 \cs_new:Npn \__fp_exp_after_mark_f:nw #1
11619 {
11620     \_msg_kernel_expandable_error:nn { kernel } { fp-early-end }
11621     \exp_after:wN \c_nan_fp \exp:w \exp_end_continue_f:w #1
11622 }

```

```

11623 \cs_new:cpn { __fp_exp_after_?_f:nw } #1#2
11624 {
11625   \__msg_kernel_expandable_error:nnn { kernel } { bad-variable } {#2}
11626   \exp_after:wN \c_nan_fp \exp:w \exp_end_continue_f:w #1
11627 }
11628 <*package>
11629 \group_begin:
11630   \char_set_catcode_letter:N \@
11631   \cs_if_exist:NT \@unexpandable@protect
11632   {
11633     \cs_gset:cpn { __fp_exp_after_?_f:nw } #1#2
11634     {
11635       \exp_after:wN \c_nan_fp \exp:w \exp_end_continue_f:w #1
11636       \str_if_eq:nnTF {#2} { \protect }
11637       {
11638         \cs_if_eq:NNTF #2 \@unexpandable@protect { \use_i:nn } { \use:n }
11639         { \__msg_kernel_expandable_error:nnn { kernel } { fp-robust-cmd } }
11640       }
11641       { \__msg_kernel_expandable_error:nnn { kernel } { bad-variable } {#2} }
11642     }
11643   }
11644 \group_end:
11645 </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

```

This is called whenever #2 is a control sequence other than \scan_stop: in meaning. We assume that it is a register, but carefully unpacking it with \tex_the:D within braces. First, we find the exponent following #2. Then we unpack #2 with \tex_the:D, and the auxii auxiliary distinguishes integer registers from dimensions/skips from muskips, according to the presence of a period and/or of pt. For integers, simply convert $\langle value \rangle e \langle exponent \rangle$ to a floating point number with \fp_parse:n (this is somewhat wasteful). For other registers, the decimal rounding provided by T_EX does not accurately represent the binary value that it manipulates, so we extract this binary value as a number of scaled points with __int_value:w __dim_eval:w $\langle decimal value \rangle$ pt, and use an auxiliary of \dim_to_fp:n, which performs the multiplication by 2^{-16} , correctly rounded.

```

11646 \cs_new:Npn \__fp_parse_one_register:NN #1#2
11647 {
11648   \exp_after:wN \__fp_parse_infix_after_operand:NwN
11649   \exp_after:wN #1
11650   \exp:w \exp_end_continue_f:w
11651   \exp_after:wN \__fp_parse_one_register_aux:Nw
11652   \exp_after:wN #2
11653   \__int_value:w
11654   \exp_after:wN \__fp_parse_exponent:N
11655   \exp:w \__fp_parse_expand:w
11656 }

```

```

11657 \cs_new:Npx \__fp_parse_one_register_aux:Nw #1
11658 {
11659   \exp_not:n
11660   {
11661     \exp_after:wN \use:nn
11662     \exp_after:wN \__fp_parse_one_register_auxii:wwwNw
11663   }
11664   \exp_not:N \exp_after:wN { \exp_not:N \tex_the:D #1 }
11665   ; \exp_not:N \__fp_parse_one_register_dim:ww
11666   \tl_to_str:n { pt } ; \exp_not:N \__fp_parse_one_register_mu:www
11667   . \tl_to_str:n { pt } ; \exp_not:N \__fp_parse_one_register_int:www
11668   \exp_not:N \q_stop
11669 }
11670 \use:x
11671 {
11672   \cs_new:Npn \exp_not:N \__fp_parse_one_register_auxii:wwwNw
11673     ##1 . ##2 \tl_to_str:n { pt } ##3 ; ##4##5 \exp_not:N \q_stop
11674     { ##4 ##1.##2; }
11675   \cs_new:Npn \exp_not:N \__fp_parse_one_register_mu:www
11676     ##1 \tl_to_str:n { mu } ; ##2 ;
11677     { \exp_not:N \__fp_parse_one_register_dim:ww ##1 ; }
11678 }
11679 \cs_new:Npn \__fp_parse_one_register_int:www #1; #2.; #3;
11680 { \__fp_parse:n { #1 e #3 } }
11681 \cs_new:Npn \__fp_parse_one_register_dim:ww #1; #2;
11682 {
11683   \exp_after:wN \__fp_from_dim_test:ww
11684   \__int_value:w #2 \exp_after:wN ,
11685   \__int_value:w \__dim_eval:w #1 pt ;
11686 }

```

(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.

```

11687 \cs_new:Npn \__fp_parse_one_digit:NN #1
11688 {
11689   \exp_after:wN \__fp_parse_infix_after_operand:NwN
11690   \exp_after:wN #1
11691   \exp:w \exp_end_continue_f:w
11692   \exp_after:wN \__fp_sanitize:wN
11693   \int_use:N \__int_eval:w \c_zero \__fp_parse_trim_zeros:N
11694 }

```

(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 a 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`.

```

11695 \cs_new:Npn \__fp_parse_one_other:NN #1 #2
11696 {
11697   \if_int_compare:w
11698     \__int_eval:w
11699     ( '#2 \if_int_compare:w '#2 > 'Z - \c_thirty_two \fi: ) / 26
11700     = \c_three
11701     \exp_after:wN \__fp_parse_word:Nw
11702     \exp_after:wN #1
11703     \exp_after:wN #2
11704     \exp:w \exp_after:wN \__fp_parse_letters:N
11705     \exp:w
11706   \else:
11707     \exp_after:wN \__fp_parse_prefix:NNN
11708     \exp_after:wN #1
11709     \exp_after:wN #2
11710     \cs:w
11711     __fp_parse_prefix_ \token_to_str:N #2 :Nw
11712     \exp_after:wN
11713     \cs_end:
11714     \exp:w
11715   \fi:
11716   \__fp_parse_expand:w
11717 }

```

(End definition for `__fp_parse_one_other:NN`.)

`__fp_parse_word:Nw` Finding letters is a simple recursion. Once `__fp_parse_letters:N` has done its job, `__fp_parse_letters:N` we try to build a control sequence from the word #2. If it is a known word, then the corresponding action is taken, and otherwise, we complain about an unknown word, yield `\c_nan_fp`, and look for the following infix operator. Note that the unknown word could be a mistyped function as well as a mistyped constant, so there is no way to tell whether to look for arguments; we do not.

```

11718 \cs_new:Npn \__fp_parse_word:Nw #1#2;
11719 {
11720   \cs_if_exist_use:cF { __fp_parse_word_#2:N }
11721   {
11722     \__msg_kernel_expandable_error:nnn
11723     { kernel } { unknown-fp-word } {#2}
11724     \exp_after:wN \c_nan_fp \exp:w \exp_end_continue_f:w
11725     \__fp_parse_infix:NN
11726   }
11727   #1
11728 }
11729 \cs_new:Npn \__fp_parse_letters:N #1
11730 {

```

```

11731 \exp_end_continue_f:w
11732 \if_int_compare:w
11733   \if_catcode:w \scan_stop: \exp_not:N #1
11734   \c_zero
11735   \else:
11736     \__int_eval:w
11737     ( '#1 \if_int_compare:w '#1 > 'Z - \c_thirty_two \fi: )
11738     / 26
11739   \fi:
11740   = \c_three
11741   \exp_after:wN #1
11742   \exp:w \exp_after:wN \__fp_parse_letters:N
11743   \exp:w
11744   \else:
11745     \__fp_parse_return_semicolon:w #1
11746   \fi:
11747   \__fp_parse_expand:w
11748 }

```

(End definition for __fp_parse_word:Nw.)

__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.

```

11749 \cs_new:Npn \__fp_parse_prefix:NNN #1#2#3
11750 {
11751   \if_meaning:w \scan_stop: #3
11752   \exp_after:wN \__fp_parse_prefix_unknown:NNN
11753   \exp_after:wN #2
11754   \fi:
11755   #3 #1
11756 }
11757 \cs_new:Npn \__fp_parse_prefix_unknown:NNN #1#2#3
11758 {
11759   \cs_if_exist:cTF { __fp_parse_infix_ \token_to_str:N #1 :N }
11760   {
11761     \__msg_kernel_expandable_error:nnn
11762     { kernel } { fp-missing-number } {#1}
11763     \exp_after:wN \c_nan_fp \exp:w \exp_end_continue_f:w
11764     \__fp_parse_infix:NN #3 #1
11765   }
11766   {
11767     \__msg_kernel_expandable_error:nnn
11768     { kernel } { fp-unknown-symbol } {#1}
11769     \__fp_parse_one:Nw #3

```

```

11770     }
11771 }

```

(End definition for `__fp_parse_prefix:NNN` and `__fp_parse_prefix_unknown:NNN`.)

25.4.1 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_trim_zeros:N`; otherwise, our number is exactly zero, and we call `__fp_parse_zero:` to take care of that case.

```

11772 \cs_new:Npn __fp_parse_trim_zeros:N #1
11773 {
11774   \if:w 0 \exp_not:N #1
11775     \exp_after:wN __fp_parse_trim_zeros:N
11776     \exp:w
11777   \else:
11778     \if:w . \exp_not:N #1
11779       \exp_after:wN __fp_parse_trim_zeros:N
11780       \exp:w
11781     \else:
11782       __fp_parse_trim_end:w #1
11783     \fi:
11784   \fi:
11785   __fp_parse_expand:w
11786 }
11787 \cs_new:Npn __fp_parse_trim_end:w #1 \fi: \fi: __fp_parse_expand:w
11788 {
11789   \fi:
11790   \fi:
11791   \if_int_compare:w \c_nine < 1 \token_to_str:N #1 \exp_stop_f:
11792     \exp_after:wN __fp_parse_large:N
11793   \else:
11794     \exp_after:wN __fp_parse_zero:
11795   \fi:
11796   #1
11797 }

```

(End definition for `__fp_parse_trim_zeros:N` and `__fp_parse_trim_end:w`.)

```

__fp_parse_trim_zeros:N
__fp_parse_trim_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”.

```

11798 \cs_new:Npn \__fp_parse_strim_zeros:N #1
11799 {
11800   \if:w 0 \exp_not:N #1
11801     - \c_one
11802     \exp_after:wN \__fp_parse_strim_zeros:N \exp:w
11803   \else:
11804     \__fp_parse_strim_end:w #1
11805   \fi:
11806   \__fp_parse_expand:w
11807 }
11808 \cs_new:Npn \__fp_parse_strim_end:w #1 \fi: \__fp_parse_expand:w
11809 {
11810   \fi:
11811   \if_int_compare:w \c_nine < 1 \token_to_str:N #1 \exp_stop_f:
11812     \exp_after:wN \__fp_parse_small:N
11813   \else:
11814     \exp_after:wN \__fp_parse_zero:
11815   \fi:
11816   #1
11817 }

```

(End definition for `__fp_parse_strim_zeros:N` and `__fp_parse_strim_end:w`.)

`__fp_parse_zero:` After reading a significand of 0, we need to remove any exponent, then put a sign of 1 for `__fp_sanitize:wN`, small hack to denote an exact zero (rather than an underflow).

```

11818 \cs_new:Npn \__fp_parse_zero:
11819 {
11820   \exp_after:wN ; \exp_after:wN 1
11821   \__int_value:w \__fp_parse_exponent:N
11822 }

```

(End definition for `__fp_parse_zero:`.)

25.4.2 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.

```

11823 \cs_new:Npn \__fp_parse_small:N #1
11824 {

```

```

11825 \exp_after:wN \_fp_parse_pack_leading:NNNNNww
11826 \int_use:N \_int_eval:w 1 \token_to_str:N #1
11827 \exp_after:wN \_fp_parse_small_leading:wwNN
11828 \_int_value:w 1
11829 \exp_after:wN \_fp_parse_digits_vii:N
11830 \exp:w \_fp_parse_expand:w
11831 }

```

(End definition for _fp_parse_small:N.)

_fp_parse_small_leading:wwNN We leave $\langle \text{digits} \rangle$ $\langle \text{zeros} \rangle$ 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 $\backslash\text{c_zero}$ (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.

```

11832 \cs_new:Npn \_fp_parse_small_leading:wwNN 1 #1 ; #2; #3 #4
11833 {
11834   #1 #2
11835   \exp_after:wN \_fp_parse_pack_trailing:NNNNNNww
11836   \exp_after:wN \c_zero
11837   \int_use:N \_int_eval:w 1
11838   \if_int_compare:w \c_nine < 1 \token_to_str:N #4 \exp_stop_f:
11839     \token_to_str:N #4
11840     \exp_after:wN \_fp_parse_small_trailing:wwNN
11841     \_int_value:w 1
11842     \exp_after:wN \_fp_parse_digits_vi:N
11843     \exp:w
11844   \else:
11845     0000 0000 \_fp_parse_exponent:Nw #4
11846   \fi:
11847   \_fp_parse_expand:w
11848 }

```

(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 $\langle \text{next token} \rangle$ 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 $+\backslash\text{c_zero}$ or to $+\backslash\text{c_one}$. Otherwise, there is no 16-th digit, so we put a 0, and look for an exponent.

```

11849 \cs_new:Npn \_fp_parse_small_trailing:wwNN 1 #1 ; #2; #3 #4
11850 {
11851   #1 #2
11852   \if_int_compare:w \c_nine < 1 \token_to_str:N #4 \exp_stop_f:
11853     \token_to_str:N #4
11854     \exp_after:wN \_fp_parse_small_round:NN
11855     \exp_after:wN #4
11856   \exp:w

```

```

11857     \else:
11858         0 \__fp_parse_exponent:Nw #4
11859     \fi:
11860     \__fp_parse_expand:w
11861 }

```

(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 (+ `\c_one` 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.

```

11862 \cs_new:Npn \__fp_parse_pack_trailing:NNNNNNww #1 #2 #3#4#5#6 #7; #8 ;
11863 {
11864     \if_meaning:w 2 #2 + \c_one \fi:
11865     ; #8 + #1 ; {#3#4#5#6} {#7};
11866 }
11867 \cs_new:Npn \__fp_parse_pack_leading:NNNNNNww #1 #2#3#4#5 #6; #7;
11868 {
11869     + #7
11870     \if_meaning:w 2 #1 \__fp_parse_pack_carry:w \fi:
11871     ; 0 {#2#3#4#5} {#6}
11872 }
11873 \cs_new:Npn \__fp_parse_pack_carry:w \fi: ; 0 #1
11874 { \fi: + \c_one ; 0 {1000} }

```

(End definition for __fp_parse_pack_trailing:NNNNNNww, __fp_parse_pack_leading:NNNNNNww, and __fp_parse_pack_carry:w.)

25.4.3 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.

```

11875 \cs_new:Npn \__fp_parse_large:N #1
11876 {
11877     \exp_after:wN \__fp_parse_large_leading:wwNN
11878     \__int_value:w 1 \token_to_str:N #1

```

```

11879      \exp_after:wN \__fp_parse_digits_vii:N
11880      \exp:w \__fp_parse_expand:w
11881    }

```

(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 *<number of zeros>* (number of digits missing). Then prepare to pack the 8 first digits. If the *<next token>* is a digit, read up to 6 more digits (digits 10 to 15). If it is a period, try to grab the end of our 8 first digits, branching to the `small` functions since the number of digit does not affect the exponent anymore. Finally, if this is the end of the significand, insert the *<zeros>* to complete the 8 first digits, insert 8 more, and look for an exponent.

```

11882 \cs_new:Npn \__fp_parse_large_leading:wwNN 1 #1 ; #2; #3 #4
11883 {
11884   + \c_eight - #3
11885   \exp_after:wN \__fp_parse_pack_leading:NNNNNww
11886   \int_use:N \__int_eval:w 1 #1
11887   \if_int_compare:w \c_nine < 1 \token_to_str:N #4 \exp_stop_f:
11888     \exp_after:wN \__fp_parse_large_trailing:wwNN
11889     \__int_value:w 1 \token_to_str:N #4
11890     \exp_after:wN \__fp_parse_digits_vi:N
11891     \exp:w
11892   \else:
11893     \if:w . \exp_not:N #4
11894       \exp_after:wN \__fp_parse_small_leading:wwNN
11895       \__int_value:w 1
11896       \cs:w
11897         __fp_parse_digits_
11898         \__int_to_roman:w #3
11899         :N \exp_after:wN
11900       \cs_end:
11901       \exp:w
11902     \else:
11903       #2
11904       \exp_after:wN \__fp_parse_pack_trailing:NNNNNNww
11905       \exp_after:wN \c_zero
11906       \__int_value:w 1 0000 0000
11907       \__fp_parse_exponent:Nw #4
11908     \fi:
11909   \fi:
11910   \__fp_parse_expand:w
11911 }

```

(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.

```

11912 \cs_new:Npn \__fp_parse_large_trailing:wwNN 1 #1 ; #2; #3 #4
11913 {
11914   \if_int_compare:w \c_nine < 1 \token_to_str:N #4 \exp_stop_f:
11915     \exp_after:wN \__fp_parse_pack_trailing:NNNNNNww
11916     \exp_after:wN \c_eight
11917     \int_use:N \__int_eval:w 1 #1 \token_to_str:N #4
11918     \exp_after:wN \__fp_parse_large_round:NN
11919     \exp_after:wN #4
11920     \exp:w
11921   \else:
11922     \exp_after:wN \__fp_parse_pack_trailing:NNNNNNww
11923     \int_use:N \__int_eval:w \c_seven - #3 \exp_stop_f:
11924     \int_use:N \__int_eval:w 1 #1
11925     \if:w . \exp_not:N #4
11926       \exp_after:wN \__fp_parse_small_trailing:wwNN
11927       \__int_value:w 1
11928       \cs:w
11929         __fp_parse_digits_
11930         \__int_to_roman:w #3
11931         :N \exp_after:wN
11932         \cs_end:
11933         \exp:w
11934     \else:
11935       #2 0 \__fp_parse_exponent:Nw #4
11936     \fi:
11937   \fi:
11938   \__fp_parse_expand:w
11939 }

```

(End definition for `__fp_parse_large_trailing:wwNN`.)

25.4.4 Number: beyond 16 digits, rounding

`__fp_parse_round_loop:N` This loop is called when rounding a number (whether the mantissa is small or large).
`__fp_parse_round_up:N` It should appear in an integer expression. This function reads digits one by one, until reaching a non-digit, and adds 1 to the integer expression for each digit. If all digits found are 0, the function ends the expression by `;\c_zero`, otherwise by `;\c_one`. 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.

```

11940 \cs_new:Npn \__fp_parse_round_loop:N #1
11941 {
11942   \if_int_compare:w \c_nine < 1 \token_to_str:N #1 \exp_stop_f:
11943     + \c_one
11944   \if:w 0 \token_to_str:N #1
11945     \exp_after:wN \__fp_parse_round_loop:N

```

```

11946         \exp:w
11947     \else:
11948         \exp_after:wN \__fp_parse_round_up:N
11949         \exp:w
11950     \fi:
11951 \else:
11952     \__fp_parse_return_semicolon:w \c_zero #1
11953 \fi:
11954 \__fp_parse_expand:w
11955 }
11956 \cs_new:Npn \__fp_parse_round_up:N #1
11957 {
11958     \if_int_compare:w \c_nine < 1 \token_to_str:N #1 \exp_stop_f:
11959         + \c_one
11960         \exp_after:wN \__fp_parse_round_up:N
11961         \exp:w
11962     \else:
11963         \__fp_parse_return_semicolon:w \c_one #1
11964     \fi:
11965     \__fp_parse_expand:w
11966 }

```

(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 \c_zero or \c_one is added to the surrounding integer expression.

```

11967 \cs_new:Npn \__fp_parse_round_after:wN #1; #2
11968 {
11969     + #2 \exp_after:wN ;
11970     \int_use:N \__int_eval:w #1 + \__fp_parse_exponent:N
11971 }

```

(End definition for __fp_parse_round_after:wN.)

__fp_parse_small_round:NN Here, #1 is the digit that we are currently rounding (we only care whether it is even or odd). If #2 is not a digit, then fetch an exponent and expand to ;*<exponent>* only. Otherwise, we will expand to +\c_zero or +\c_one, 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 +\c_zero or +\c_one 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.

```

11972 \cs_new:Npn \__fp_parse_small_round:NN #1#2
11973 {
11974     \if_int_compare:w \c_nine < 1 \token_to_str:N #2 \exp_stop_f:
11975     +

```

```

11976     \exp_after:wN \__fp_round_s:NNNw
11977     \exp_after:wN 0
11978     \exp_after:wN #1
11979     \exp_after:wN #2
11980     \int_use:N \__int_eval:w
11981     \exp_after:wN \__fp_parse_round_after:wN
11982     \int_use:N \__int_eval:w \c_zero * \__int_eval:w \c_zero
11983     \exp_after:wN \__fp_parse_round_loop:N
11984     \exp:w
11985   \else:
11986     \__fp_parse_exponent:Nw #2
11987   \fi:
11988   \__fp_parse_expand:w
11989 }

```

(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.

```

11990 \cs_new:Npn \__fp_parse_large_round:NN #1#2
11991 {
11992   \if_int_compare:w \c_nine < 1 \token_to_str:N #2 \exp_stop_f:
11993   +
11994   \exp_after:wN \__fp_round_s:NNNw
11995   \exp_after:wN 0
11996   \exp_after:wN #1
11997   \exp_after:wN #2
11998   \int_use:N \__int_eval:w
11999   \exp_after:wN \__fp_parse_large_round_aux:wNN
12000   \int_use:N \__int_eval:w \c_one
12001   \exp_after:wN \__fp_parse_round_loop:N
12002   \else: %^^A could be dot, or e, or other
12003   \exp_after:wN \__fp_parse_large_round_test:NN
12004   \exp_after:wN #1
12005   \exp_after:wN #2
12006   \fi:
12007 }
12008 \cs_new:Npn \__fp_parse_large_round_test:NN #1#2
12009 {
12010   \if:w . \exp_not:N #2
12011   \exp_after:wN \__fp_parse_small_round:NN
12012   \exp_after:wN #1

```

```

12013     \exp:w
12014   \else:
12015     \__fp_parse_exponent:Nw #2
12016   \fi:
12017   \__fp_parse_expand:w
12018 }
12019 \cs_new:Npn \__fp_parse_large_round_aux:wNN #1 ; #2 #3
12020 {
12021   + #2
12022   \exp_after:wN \__fp_parse_round_after:wN
12023   \int_use:N \__int_eval:w #1
12024   \if:w . \exp_not:N #3
12025     + \c_zero * \__int_eval:w \c_zero
12026     \exp_after:wN \__fp_parse_round_loop:N
12027     \exp:w \exp_after:wN \__fp_parse_expand:w
12028   \else:
12029     \exp_after:wN ;
12030     \exp_after:wN \c_zero
12031     \exp_after:wN #3
12032   \fi:
12033 }

```

(End definition for __fp_parse_large_round:NN, __fp_parse_large_round_test:NN, and __fp_parse_large_round_aux:wNN.)

25.4.5 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.

```

12034 \cs_new:Npn \__fp_parse_exponent:Nw #1 #2 \__fp_parse_expand:w
12035 {
12036   \exp_after:wN ;
12037   \__int_value:w #2 \__fp_parse_exponent:N #1
12038 }

```

(End definition for __fp_parse_exponent:Nw.)

`__fp_parse_exponent: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.

`__fp_parse_exponent_aux:N`

```

12039 \cs_new:Npn \__fp_parse_exponent:N #1
12040 {
12041   \if:w e \exp_not:N #1
12042     \exp_after:wN \__fp_parse_exponent_aux:N
12043     \exp:w
12044   \else:
12045     0 \__fp_parse_return_semicolon:w #1
12046   \fi:
12047   \__fp_parse_expand:w
12048 }
12049 \cs_new:Npn \__fp_parse_exponent_aux:N #1
12050 {
12051   \if_int_compare:w \if_catcode:w \scan_stop: \exp_not:N #1
12052     \c_zero \else: ' #1 \fi: > '9 \exp_stop_f:
12053     0 \exp_after:wN ; \exp_after:wN e
12054   \else:
12055     \exp_after:wN \__fp_parse_exponent_sign:N
12056   \fi:
12057   #1
12058 }

```

(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).

```

12059 \cs_new:Npn \__fp_parse_exponent_sign:N #1
12060 {
12061   \if:w + \if:w - \exp_not:N #1 + \fi: \token_to_str:N #1
12062     \exp_after:wN \__fp_parse_exponent_sign:N
12063     \exp:w \exp_after:wN \__fp_parse_expand:w
12064   \else:
12065     \exp_after:wN \__fp_parse_exponent_body:N

```

```

12066     \exp_after:wN #1
12067     \fi:
12068 }

```

(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).

```

12069 \cs_new:Npn \__fp_parse_exponent_body:N #1
12070 {
12071   \if_int_compare:w \c_nine < 1 \token_to_str:N #1 \exp_stop_f:
12072   \token_to_str:N #1
12073   \exp_after:wN \__fp_parse_exponent_digits:N
12074   \exp:w
12075   \else:
12076     \__fp_parse_exponent_keep:NTF #1
12077     { \__fp_parse_return_semicolon:w #1 }
12078     {
12079       \exp_after:wN ;
12080       \exp:w
12081     }
12082   \fi:
12083   \__fp_parse_expand:w
12084 }

```

(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.

```

12085 \cs_new:Npn \_fp_parse_exponent_digits:N #1
12086 {
12087   \if_int_compare:w \c_nine < 1 \token_to_str:N #1 \exp_stop_f:
12088   \token_to_str:N #1
12089   \exp_after:wN \_fp_parse_exponent_digits:N
12090   \exp:w
12091   \else:
12092     \__fp_parse_return_semicolon:w #1
12093   \fi:
12094   \_fp_parse_expand:w
12095 }

```

(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.

```

12096 \prg_new_conditional:Npnn \__fp_parse_exponent_keep:N #1 { TF }
12097 {
12098   \if_catcode:w \scan_stop: \exp_not:N #1
12099   \if_meaning:w \scan_stop: #1
12100   \if_int_compare:w
12101     \__str_if_eq_x:nn { \s__fp } { \exp_not:N #1 } = \c_zero
12102     0
12103     \__msg_kernel_expandable_error:nnn
12104     { kernel } { fp-after-e } { floating~point~ }
12105     \prg_return_true:
12106   \else:
12107     0
12108     \__msg_kernel_expandable_error:nnn
12109     { kernel } { bad-variable } { #1 }
12110     \prg_return_false:
12111   \fi:
12112 \else:
12113   \if_int_compare:w
12114     \__str_if_eq_x:nn { \__int_value:w #1 } { \tex_the:D #1 }
12115     = \c_zero
12116     \__int_value:w #1
12117   \else:
12118     0
12119     \__msg_kernel_expandable_error:nnn
12120     { kernel } { fp-after-e } { dimension~#1 }
12121   \fi:
12122   \prg_return_false:
12123 \fi:
12124 \else:
12125   0
12126   \__msg_kernel_expandable_error:nnn
12127   { kernel } { fp-missing } { exponent }
12128   \prg_return_true:
12129 \fi:
12130 }

```

(End definition for __fp_parse_exponent_keep:N_{TF}.)

25.5 Constants, functions and prefix operators

25.5.1 Prefix operators

__fp_parse_prefix+:N_w A unary + does nothing: we should continue looking for a number.

```

12131 \cs_new_eq:cN { \__fp_parse_prefix+:Nw } \__fp_parse_one:Nw

```

(End definition for __fp_parse_prefix+:N_w.)

`_fp_parse_apply_unary:NNWwN` 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.

```

12132 \cs_new:Npn \_fp_parse_apply_unary:NNWwN #1#2#3#4#5
12133 {
12134     #3 #2 #4 @
12135     \exp:w \exp_end_continue_f:w #5 #1
12136 }

```

(End definition for `_fp_parse_apply_unary:NNWwN`.)

`_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_twelve` of the unary operator, then call the appropriate `_fp_⟨operation⟩_o:w` function, where the `⟨operation⟩` is `set_sign` or `not`.

`_fp_parse_prefix !:Nw`

```

12137 \cs_set_protected:Npn \_fp_tmp:w #1#2#3#4
12138 {
12139     \cs_new:cpn { \_fp_parse_prefix_ #1 :Nw } ##1
12140     {
12141         \exp_after:wN \_fp_parse_apply_unary:NNWwN
12142         \exp_after:wN ##1
12143         \exp_after:wN #4
12144         \exp_after:wN #3
12145         \exp:w
12146         \if_int_compare:w #2 < ##1
12147             \_fp_parse_operand:Nw ##1
12148         \else:
12149             \_fp_parse_operand:Nw #2
12150         \fi:
12151         \_fp_parse_expand:w
12152     }
12153 }
12154 \_fp_tmp:w - \c_twelve \_fp_set_sign_o:w 2
12155 \_fp_tmp:w ! \c_twelve \_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.

```

12156 \cs_new:cpn { \_fp_parse_prefix .:Nw } #1
12157 {
12158     \exp_after:wN \_fp_parse_infix_after_operand:NwN
12159     \exp_after:wN #1
12160     \exp:w \exp_end_continue_f:w
12161     \exp_after:wN \_fp_sanitize:wN

```

```

12162         \int_use:N \__int_eval:w \c_zero \__fp_parse_strim_zeros:N
12163     }

```

(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) or 13 (unary boolean “not”). 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.

```

12164 \group_begin:
12165   \char_set_catcode_letter:N (
12166   \char_set_catcode_letter:N )
12167   \cs_new:Npn \__fp_parse_prefix_(:Nw #1
12168   {
12169     \exp_after:wN \__fp_parse_lparen_after:NwN
12170     \exp_after:wN #1
12171     \exp:w
12172     \if_int_compare:w #1 = \c_sixteen
12173       \__fp_parse_operand:Nw \c_one
12174     \else:
12175       \__fp_parse_operand:Nw \c_zero
12176     \fi:
12177     \__fp_parse_expand:w
12178   }
12179   \cs_new:Npn \__fp_parse_lparen_after:NwN #1#2 @ #3
12180   {
12181     \token_if_eq_meaning:NNTF #3 \__fp_parse_infix_):N
12182     {
12183       \__fp_exp_after_array_f:w #2 \s__fp_stop
12184       \exp_after:wN \__fp_parse_infix:NN
12185       \exp_after:wN #1
12186       \exp:w \__fp_parse_expand:w
12187     }
12188     {
12189       \__msg_kernel_expandable_error:nnn
12190       { kernel } { fp-missing } { { } }
12191       #2 @ \use_none:n #3
12192     }
12193   }
12194 \group_end:

```

(End definition for __fp_parse_prefix_(:Nw and __fp_parse_lparen_after:NwN.)

25.5.2 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`.

```

12195 \cs_set_protected:Npn \__fp_tmp:w #1 #2
12196 {
12197   \cs_new_nopar:cpn { __fp_parse_word_#1:N }
12198   { \exp_after:wN #2 \exp:w \exp_end_continue_f:w \__fp_parse_infix:NN }
12199 }
12200 \__fp_tmp:w { inf } \c_inf_fp
12201 \__fp_tmp:w { nan } \c_nan_fp
12202 \__fp_tmp:w { pi } \c_pi_fp
12203 \__fp_tmp:w { deg } \c_one_degree_fp
12204 \__fp_tmp:w { true } \c_one_fp
12205 \__fp_tmp:w { false } \c_zero_fp

```

(End definition for __fp_parse_word_inf:N and others.)

__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.

```

\__fp_parse_word_in:N
\__fp_parse_word_pc:N
\__fp_parse_word_cm:N
\__fp_parse_word_mm:N
\__fp_parse_word_dd:N
\__fp_parse_word_cc:N
\__fp_parse_word_nd:N
\__fp_parse_word_nc:N
\__fp_parse_word_bp:N
\__fp_parse_word_sp:N
12206 \cs_set_protected:Npn \__fp_tmp:w #1 #2
12207 {
12208   \cs_new_nopar:cpn { __fp_parse_word_#1:N }
12209   {
12210     \__fp_exp_after_f:nw { \__fp_parse_infix:NN }
12211     \s__fp \__fp_chk:w 10 #2 ;
12212   }
12213 }
12214 \__fp_tmp:w {pt} { {1} {1000} {0000} {0000} {0000} }
12215 \__fp_tmp:w {in} { {2} {7227} {0000} {0000} {0000} }
12216 \__fp_tmp:w {pc} { {2} {1200} {0000} {0000} {0000} }
12217 \__fp_tmp:w {cm} { {2} {2845} {2755} {9055} {1181} }
12218 \__fp_tmp:w {mm} { {1} {2845} {2755} {9055} {1181} }
12219 \__fp_tmp:w {dd} { {1} {1070} {0085} {6496} {0630} }
12220 \__fp_tmp:w {cc} { {2} {1284} {0102} {7795} {2756} }
12221 \__fp_tmp:w {nd} { {1} {1066} {9783} {4645} {6693} }
12222 \__fp_tmp:w {nc} { {2} {1280} {3740} {1574} {8031} }
12223 \__fp_tmp:w {bp} { {1} {1003} {7500} {0000} {0000} }
12224 \__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.

```

\__fp_parse_word_ex:N
12225 \tl_map_inline:nn { {em} {ex} }
12226 {
12227   \cs_new_nopar:cpn { __fp_parse_word_#1:N }
12228   {
12229     \exp_after:wN \__fp_from_dim_test:ww
12230     \exp_after:wN 0 \exp_after:wN ,
12231     \__int_value:w \__dim_eval:w 1 #1 \exp_after:wN ;
12232     \exp:w \exp_end_continue_f:w \__fp_parse_infix:NN
12233   }
12234 }

```

(End definition for __fp_parse_word_em:N and __fp_parse_word_ex:N.)

25.5.3 Functions

```

\__fp_parse_unary_function:nNN
\__fp_parse_function:NNN
12235 \cs_new:Npn \__fp_parse_unary_function:nNN #1#2#3
12236 {
12237   \exp_after:wN \__fp_parse_apply_unary:NNNwN
12238   \exp_after:wN #3
12239   \exp_after:wN #2
12240   \cs:w \__fp_#1_o:w \exp_after:wN \cs_end:
12241   \exp:w
12242   \__fp_parse_operand:Nw \c_fifteen \__fp_parse_expand:w
12243 }
12244 \cs_new:Npn \__fp_parse_function:NNN #1#2#3
12245 {
12246   \exp_after:wN \__fp_parse_apply_unary:NNNwN
12247   \exp_after:wN #3
12248   \exp_after:wN #2
12249   \exp_after:wN #1
12250   \exp:w
12251   \__fp_parse_operand:Nw \c_sixteen \__fp_parse_expand:w
12252 }

```

(End definition for __fp_parse_unary_function:nNN and __fp_parse_function:NNN.)

```

\__fp_parse_word_acot:N
\__fp_parse_word_acotd:N
\__fp_parse_word_atan:N
\__fp_parse_word_atand:N
\__fp_parse_word_max:N
\__fp_parse_word_min:N
12253 \cs_new_nopar:Npn \__fp_parse_word_acot:N
12254 { \__fp_parse_function:NNN \__fp_acot_o:Nw \use_i:nn }
12255 \cs_new_nopar:Npn \__fp_parse_word_acotd:N
12256 { \__fp_parse_function:NNN \__fp_acot_o:Nw \use_ii:nn }
12257 \cs_new_nopar:Npn \__fp_parse_word_atan:N
12258 { \__fp_parse_function:NNN \__fp_atan_o:Nw \use_i:nn }
12259 \cs_new_nopar:Npn \__fp_parse_word_atand:N
12260 { \__fp_parse_function:NNN \__fp_atan_o:Nw \use_ii:nn }
12261 \cs_new_nopar:Npn \__fp_parse_word_max:N
12262 { \__fp_parse_function:NNN \__fp_minmax_o:Nw 2 }
12263 \cs_new_nopar:Npn \__fp_parse_word_min:N
12264 { \__fp_parse_function:NNN \__fp_minmax_o:Nw 0 }

```

(End definition for __fp_parse_word_acot:N and others.)

```

\__fp_parse_word_abs:N
\__fp_parse_word_exp:N
\__fp_parse_word_ln:N
\__fp_parse_word_sqrt:N
Unary functions.
12265 \cs_new:Npn \__fp_parse_word_abs:N
12266 { \__fp_parse_unary_function:nNN { set_sign } 0 }
12267 \cs_new_nopar:Npn \__fp_parse_word_exp:N
12268 { \__fp_parse_unary_function:nNN {exp} ? }
12269 \cs_new_nopar:Npn \__fp_parse_word_ln:N
12270 { \__fp_parse_unary_function:nNN {ln} ? }
12271 \cs_new_nopar:Npn \__fp_parse_word_sqrt:N
12272 { \__fp_parse_unary_function:nNN {sqrt} ? }

```

(End definition for `__fp_parse_word_abs:N` and others.)

```

\__fp_parse_word_acos:N
\__fp_parse_word_acosd:N
\__fp_parse_word_acsc:N
\__fp_parse_word_acscd:N
\__fp_parse_word_asec:N
\__fp_parse_word_asecd:N
\__fp_parse_word_asin:N
\__fp_parse_word_asind:N
\__fp_parse_word_cos:N
\__fp_parse_word_cosd:N
\__fp_parse_word_cot:N
\__fp_parse_word_cotd:N
\__fp_parse_word_csc:N
\__fp_parse_word_cscd:N
\__fp_parse_word_trunc:N
\__fp_parse_word_sec:N
\__fp_parse_word_floor:N
\__fp_parse_word_ceil:N
\__fp_parse_word_sin:N
\__fp_parse_word_sind:N
\__fp_parse_word_tan:N
\__fp_parse_word_tand:N

```

Unary functions.

```

12273 \tl_map_inline:nn
12274 {
12275   {acos} {acsc} {asec} {asin}
12276   {cos} {cot} {csc} {sec} {sin} {tan}
12277 }
12278 {
12279   \cs_new_nopar:cpn { __fp_parse_word_#1:N }
12280     { \__fp_parse_unary_function:nnn {#1} \use_i:nn }
12281   \cs_new_nopar:cpn { __fp_parse_word_#1d:N }
12282     { \__fp_parse_unary_function:nnn {#1} \use_ii:nn }
12283 }

```

(End definition for `__fp_parse_word_acos:N` and others.)

```

\__fp_parse_word_cscd:N
\__fp_parse_word_trunc:N
\__fp_parse_word_sec:N
\__fp_parse_word_floor:N
\__fp_parse_word_ceil:N
\__fp_parse_word_sin:N
\__fp_parse_word_sind:N
\__fp_parse_word_tan:N
\__fp_parse_word_tand:N

```

```

12284 \cs_new_nopar:Npn \__fp_parse_word_trunc:N
12285   { \__fp_parse_function:NNN \__fp_round_o:Nw \__fp_round_to_zero:NNN }
12286 \cs_new_nopar:Npn \__fp_parse_word_floor:N
12287   { \__fp_parse_function:NNN \__fp_round_o:Nw \__fp_round_to_ninf:NNN }
12288 \cs_new_nopar:Npn \__fp_parse_word_ceil:N
12289   { \__fp_parse_function:NNN \__fp_round_o:Nw \__fp_round_to_pinf:NNN }

```

(End definition for `__fp_parse_word_trunc:N`, `__fp_parse_word_floor:N`, and `__fp_parse_word_ceil:N`.)

```

\__fp_parse_word_round:N
\__fp_parse_round:Nw

```

```

12290 \cs_new:Npn \__fp_parse_word_round:N #1#2
12291 {
12292   \if_meaning:w + #2
12293     \__fp_parse_round:Nw \__fp_round_to_pinf:NNN
12294   \else:
12295     \if_meaning:w 0 #2
12296       \__fp_parse_round:Nw \__fp_round_to_zero:NNN
12297     \else:
12298       \if_meaning:w - #2
12299         \__fp_parse_round:Nw \__fp_round_to_ninf:NNN
12300       \fi:
12301     \fi:
12302   \fi:
12303   \__fp_parse_function:NNN
12304   \__fp_round_o:Nw \__fp_round_to_nearest:NNN #1
12305   #2
12306 }
12307 \cs_new:Npn \__fp_parse_round:Nw
12308   #1 #2 \__fp_round_to_nearest:NNN #3#4 { #2 #1 #3 }

```

(End definition for `__fp_parse_word_round:N` and `__fp_parse_round:Nw`.)

25.6 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_minus_one` 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:.`

```

12309 \cs_new:Npn \__fp_parse:n #1
12310 {
12311   \exp:w
12312   \exp_after:wN \__fp_parse_after:ww
12313   \exp:w
12314   \__fp_parse_operand:Nw \c_minus_one
12315   \__fp_parse_expand:w #1
12316   \s__fp_mark \__fp_parse_infix_end:N
12317   \s__fp_stop
12318 }
12319 \cs_new:Npn \__fp_parse_after:ww
12320   #1@ \__fp_parse_infix_end:N \s__fp_stop
12321 { \exp_end: #1 }

```

(End definition for `__fp_parse:n`.)

`__fp_parse_operand:Nw` The `__fp_parse_operand` 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`. This function should be used with much care.

```

12322 \cs_new:Npn \__fp_parse_operand:Nw #1
12323 {
12324   \exp_end_continue_f:w
12325   \exp_after:wN \__fp_parse_continue:NwN
12326   \exp_after:wN #1
12327   \exp:w \exp_end_continue_f:w
12328   \exp_after:wN \__fp_parse_one:Nw
12329   \exp_after:wN #1
12330   \exp:w
12331 }
12332 \cs_new:Npn \__fp_parse_continue:NwN #1 #2 @ #3 { #3 #1 #2 @ }

```

(End definition for `__fp_parse_operand:Nw`.)

`__fp_parse_apply_binary:NwNwN` Receives $\langle precedence \rangle \langle operand_1 \rangle @ \langle operation \rangle \langle operand_2 \rangle @ \langle infix command \rangle$. Builds the appropriate call to the $\langle operation \rangle$ #3.

```

12333 \cs_new:Npn \__fp_parse_apply_binary:NwNwN #1 #2@ #3 #4@ #5
12334 {
12335   \exp_after:wN \__fp_parse_continue:NwN
12336   \exp_after:wN #1
12337   \exp:w \exp_end_continue_f:w \cs:w __fp_#3_o:ww \cs_end: #2 #4
12338   \exp:w \exp_end_continue_f:w #5 #1
12339 }

```

(End definition for _fp_parse_apply_binary:NwNwN.)

25.7 Infix operators

_fp_parse_infix_after_operand:NwN

```

12340 \cs_new:Npn \_fp_parse_infix_after_operand:NwN #1 #2;
12341 {
12342   \_fp_exp_after_f:nw { \_fp_parse_infix:NN #1 }
12343   #2;
12344 }
12345 \group_begin:
12346   \char_set_catcode_letter:N \*
12347   \cs_new:Npn \_fp_parse_infix:NN #1 #2
12348   {
12349     \if_catcode:w \scan_stop: \exp_not:N #2
12350     \if_int_compare:w
12351       \_str_if_eq_x:nn { \s__fp_mark } { \exp_not:N #2 }
12352       = \c_zero
12353       \exp_after:wN \exp_after:wN
12354       \exp_after:wN \_fp_parse_infix_mark:NNN
12355     \else:
12356       \exp_after:wN \exp_after:wN
12357       \exp_after:wN \_fp_parse_infix_juxtapose:N
12358     \fi:
12359   \else:
12360     \if_int_compare:w
12361       \_int_eval:w
12362       ( '#2 \if_int_compare:w '#2 > 'Z - \c_thirty_two \fi: )
12363       / 26
12364       = \c_three
12365       \exp_after:wN \exp_after:wN
12366       \exp_after:wN \_fp_parse_infix_juxtapose:N
12367     \else:
12368       \exp_after:wN \_fp_parse_infix_check:NNN
12369       \cs:w
12370       \_fp_parse_infix_ \token_to_str:N #2 :N
12371       \exp_after:wN \exp_after:wN \exp_after:wN
12372     \cs_end:
12373     \fi:
12374   \fi:
12375   #1
12376   #2
12377 }
12378 \cs_new:Npn \_fp_parse_infix_check:NNN #1#2#3
12379 {
12380   \if_meaning:w \scan_stop: #1
12381     \_msg_kernel_expandable_error:nnn
12382     { kernel } { fp-missing } { * }
12383   \exp_after:wN \_fp_parse_infix_*:N

```

```

12384         \exp_after:wN #2
12385         \exp_after:wN #3
12386     \else:
12387         \exp_after:wN #1
12388         \exp_after:wN #2
12389         \exp:w \exp_after:wN \__fp_parse_expand:w
12390     \fi:
12391 }
12392 \group_end:

```

(End definition for __fp_parse_infix_after_operand:NwN.)

25.7.1 Closing parentheses and commas

__fp_parse_infix_mark:NNN 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.

```

12393 \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.

```

12394 \cs_new:Npn \__fp_parse_infix_end:N #1
12395 { @ \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.

```

12396 \group_begin:
12397   \char_set_catcode_letter:N \)
12398   \cs_new:Npn \__fp_parse_infix_):N #1
12399   {
12400     \if_int_compare:w #1 < \c_zero
12401       \__msg_kernel_expandable_error:nnn { kernel } { fp-extra } { ) }
12402       \exp_after:wN \__fp_parse_infix:NN
12403       \exp_after:wN #1
12404       \exp:w \exp_after:wN \__fp_parse_expand:w
12405     \else:
12406       \exp_after:wN @
12407       \exp_after:wN \use_none:n
12408       \exp_after:wN \__fp_parse_infix_):N
12409     \fi:
12410   }
12411 \group_end:

```

(End definition for __fp_parse_infix_):N.)

```

\__fp_parse_infix_
:N
12412 \group_begin:
12413 \char_set_catcode_letter:N \,
12414 \cs_new:Npn \__fp_parse_infix_,:N #1
12415 {
12416   \if_int_compare:w #1 > \c_one
12417     \exp_after:wN @
12418     \exp_after:wN \use_none:n
12419     \exp_after:wN \__fp_parse_infix_,:N
12420   \else:
12421     \if_int_compare:w #1 = \c_one
12422       \exp_after:wN \__fp_parse_infix_comma:w
12423       \exp:w
12424     \else:
12425       \exp_after:wN \__fp_parse_infix_comma_gobble:w
12426       \exp:w
12427     \fi:
12428     \__fp_parse_operand:Nw \c_one
12429     \exp_after:wN \__fp_parse_expand:w
12430   \fi:
12431 }
12432 \cs_new:Npn \__fp_parse_infix_comma:w #1 @
12433 { #1 @ \use_none:n }
12434 \cs_new:Npn \__fp_parse_infix_comma_gobble:w #1 @
12435 {
12436   \__msg_kernel_expandable_error:nn { kernel } { fp-extra-comma }
12437   @ \use_none:n
12438 }
12439 \group_end:

(End definition for \__fp_parse_infix_ and :N.)

```

25.7.2 Usual infix operators

__fp_parse_infix_+:N As described in the “work plan”, each infix operator has an associated \..._infix...
 __fp_parse_infix_-:N function, a computing function, and precedence, given as arguments to __fp_tmp:w.
 __fp_parse_infix_/:N Using the general mechanism for arithmetic operations. The power operation must be
 __fp_parse_infix_mul:N associative in the opposite order from all others. For this, we use two distinct precedences.
 __fp_parse_infix_and:N The odd requirement to set \+ here is to cover the case where expl3 is loaded by
 __fp_parse_infix_or:N plain TeX: \+ is an \outer macro there, and so the following code would otherwise give
 __fp_parse_infix^:N an error in that case.

```

12440 \group_begin:
12441 <*package>
12442 \cs_set_nopar:Npn \+ { }
12443 </package>
12444 \char_set_catcode_other:N \&
12445 \char_set_catcode_letter:N \^
12446 \char_set_catcode_letter:N \/
12447 \char_set_catcode_letter:N \-

```

```

12448 \char_set_catcode_letter:N \+
12449 \cs_set_protected:Npn \__fp_tmp:w #1#2#3#4
12450 {
12451   \cs_new:Npn #1 ##1
12452   {
12453     \if_int_compare:w ##1 < #3
12454       \exp_after:wN @
12455       \exp_after:wN \__fp_parse_apply_binary:NwNwN
12456       \exp_after:wN #2
12457       \exp:w
12458       \__fp_parse_operand:Nw #4
12459       \exp_after:wN \__fp_parse_expand:w
12460     \else:
12461       \exp_after:wN @
12462       \exp_after:wN \use_none:n
12463       \exp_after:wN #1
12464     \fi:
12465   }
12466 }
12467 \__fp_tmp:w \__fp_parse_infix_~:N ~ \c_fifteen \c_fourteen
12468 \__fp_tmp:w \__fp_parse_infix_/:N / \c_ten \c_ten
12469 \__fp_tmp:w \__fp_parse_infix_mul:N * \c_ten \c_ten
12470 \__fp_tmp:w \__fp_parse_infix_-:N - \c_nine \c_nine
12471 \__fp_tmp:w \__fp_parse_infix_+:N + \c_nine \c_nine
12472 \__fp_tmp:w \__fp_parse_infix_and:N & \c_five \c_five
12473 \__fp_tmp:w \__fp_parse_infix_or:N | \c_four \c_four
12474 \group_end:

```

(End definition for __fp_parse_infix_+:N and others.)

25.7.3 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`.

```

12475 \cs_new:cpn { __fp_parse_infix_(:N } #1
12476 { \__fp_parse_infix_juxtapose:N #1 ( }

```

(End definition for __fp_parse_infix_(:N.)

`_fp_parse_infix_juxtapose:N` Juxtaposition follows the same scheme as other binary operations, but calls `__fp_parse_apply_juxtapose:NwNwN` rather than directly calling `__fp_parse_apply_binary:NwNwN`. This lets us catch errors such as `...(1,2,3)pt` where one operand of the juxtaposition is not a single number: both #3 and #5 of the apply auxiliary must be empty.

```

12477 \cs_new:Npn \__fp_parse_infix_juxtapose:N #1
12478 {
12479   \if_int_compare:w #1 < \c_ten
12480     \exp_after:wN @

```

```

12481     \exp_after:wN \_fp_parse_apply_juxtapose:NwwN
12482     \exp:w
12483     \_fp_parse_operand:Nw \c_ten
12484     \exp_after:wN \_fp_parse_expand:w
12485     \else:
12486     \exp_after:wN @
12487     \exp_after:wN \use_none:n
12488     \exp_after:wN \_fp_parse_infix_juxtapose:N
12489     \fi:
12490   }
12491   \cs_new:Npn \_fp_parse_apply_juxtapose:NwwN #1 #2;#3@ #4;#5@
12492   {
12493     \if_catcode:w ^ \tl_to_str:n { #3 #5 } ^
12494     \else:
12495       \_fp_error:nffn { invalid-ii }
12496       { \_fp_array_to_clist:n { #2; #3 } }
12497       { \_fp_array_to_clist:n { #4; #5 } }
12498       { }
12499     \fi:
12500     \_fp_parse_apply_binary:NwNwN #1 #2;@ * #4;@
12501   }

```

(End definition for `_fp_parse_infix_juxtapose:N` and `_fp_parse_apply_juxtapose:NwwN`.)

25.7.4 Multi-character cases

`_fp_parse_infix_*:N`

```

12502 \group_begin:
12503   \char_set_catcode_letter:N ^
12504   \cs_new:cpn { \_fp_parse_infix_*:N } #1#2
12505   {
12506     \if:w * \exp_not:N #2
12507       \exp_after:wN \_fp_parse_infix_^:N
12508       \exp_after:wN #1
12509     \else:
12510       \exp_after:wN \_fp_parse_infix_mul:N
12511       \exp_after:wN #1
12512       \exp_after:wN #2
12513     \fi:
12514   }
12515 \group_end:

```

(End definition for `_fp_parse_infix_*:N`.)

`_fp_parse_infix_|:Nw`

`_fp_parse_infix_&:Nw`

```

12516 \group_begin:
12517   \char_set_catcode_letter:N \|
12518   \char_set_catcode_letter:N \&
12519   \cs_new:Npn \_fp_parse_infix_|:N #1#2
12520   {

```

```

12521 \if:w | \exp_not:N #2
12522 \exp_after:wN \__fp_parse_infix_|:N
12523 \exp_after:wN #1
12524 \exp:w \exp_after:wN \__fp_parse_expand:w
12525 \else:
12526 \exp_after:wN \__fp_parse_infix_or:N
12527 \exp_after:wN #1
12528 \exp_after:wN #2
12529 \fi:
12530 }
12531 \cs_new:Npn \__fp_parse_infix_&:N #1#2
12532 {
12533 \if:w & \exp_not:N #2
12534 \exp_after:wN \__fp_parse_infix_&:N
12535 \exp_after:wN #1
12536 \exp:w \exp_after:wN \__fp_parse_expand:w
12537 \else:
12538 \exp_after:wN \__fp_parse_infix_and:N
12539 \exp_after:wN #1
12540 \exp_after:wN #2
12541 \fi:
12542 }
12543 \group_end:

```

(End definition for __fp_parse_infix_/:Nw.)

25.7.5 Ternary operator

```

\__fp_parse_infix_?:N
\__fp_parse_infix_:N
12544 \group_begin:
12545 \char_set_catcode_letter:N \?
12546 \cs_new:Npn \__fp_parse_infix_?:N #1
12547 {
12548 \if_int_compare:w #1 < \c_three
12549 \exp_after:wN @
12550 \exp_after:wN \__fp_ternary:NwwN
12551 \exp:w
12552 \__fp_parse_operand:Nw \c_three
12553 \exp_after:wN \__fp_parse_expand:w
12554 \else:
12555 \exp_after:wN @
12556 \exp_after:wN \use_none:n
12557 \exp_after:wN \__fp_parse_infix_?:N
12558 \fi:
12559 }
12560 \cs_new:Npn \__fp_parse_infix_:N #1
12561 {
12562 \if_int_compare:w #1 < \c_three
12563 \__msg_kernel_expandable_error:nnnn
12564 { kernel } { fp-missing } { ? } { ~for~?: }

```

```

12565         \exp_after:wN @
12566         \exp_after:wN \__fp_ternary_auxii:NwwN
12567         \exp:w
12568         \__fp_parse_operand:Nw \c_two
12569         \exp_after:wN \__fp_parse_expand:w
12570     \else:
12571         \exp_after:wN @
12572         \exp_after:wN \use_none:n
12573         \exp_after:wN \__fp_parse_infix_::N
12574     \fi:
12575 }
12576 \group_end:

```

(End definition for __fp_parse_infix_?:N and __fp_parse_infix_::N.)

25.7.6 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

```

```

12577 \cs_new:cpn { __fp_parse_infix_<:N } #1
12578 {
12579     \__fp_parse_compare:NNNNNNN #1 \c_one
12580     \c_zero \c_zero \c_zero \c_zero <
12581 }
12582 \cs_new:cpn { __fp_parse_infix_=:N } #1
12583 {
12584     \__fp_parse_compare:NNNNNNN #1 \c_one
12585     \c_zero \c_zero \c_zero \c_zero =
12586 }
12587 \cs_new:cpn { __fp_parse_infix_>:N } #1
12588 {
12589     \__fp_parse_compare:NNNNNNN #1 \c_one
12590     \c_zero \c_zero \c_zero \c_zero >
12591 }
12592 \cs_new:cpn { __fp_parse_infix_!:N } #1
12593 {
12594     \exp_after:wN \__fp_parse_compare:NNNNNNN
12595     \exp_after:wN #1
12596     \exp_after:wN \c_zero
12597     \exp_after:wN \c_one
12598     \exp_after:wN \c_one
12599     \exp_after:wN \c_one
12600     \exp_after:wN \c_one
12601 }
12602 \cs_new:Npn \__fp_parse_excl_error:
12603 {
12604     \_msg_kernel_expandable_error:nnnn
12605     { kernel } { fp-missing } { = } { ~after~!. }
12606 }
12607 \cs_new:Npn \__fp_parse_compare:NNNNNNN #1
12608 {

```

```

12609 \if_int_compare:w #1 < \c_seven
12610 \exp_after:wN \__fp_parse_compare_auxi:NNNNNNN
12611 \exp_after:wN \__fp_parse_excl_error:
12612 \else:
12613 \exp_after:wN @
12614 \exp_after:wN \use_none:n
12615 \exp_after:wN \__fp_parse_compare:NNNNNNN
12616 \fi:
12617 }
12618 \cs_new:Npn \__fp_parse_compare_auxi:NNNNNNN #1#2#3#4#5#6#7
12619 {
12620 \if_case:w
12621 \if_catcode:w \scan_stop: \exp_not:N #7
12622 \c_minus_one
12623 \else:
12624 \__int_eval:w '#7 - '< \__int_eval_end:
12625 \fi:
12626 \__fp_parse_compare_auxii:NNNNN #2#2#4#5#6
12627 \or: \__fp_parse_compare_auxii:NNNNN #2#3#2#5#6
12628 \or: \__fp_parse_compare_auxii:NNNNN #2#3#4#2#6
12629 \or: \__fp_parse_compare_auxii:NNNNN #2#3#4#5#2
12630 \else: #1 \__fp_parse_compare_end:NNNNw #3#4#5#6#7
12631 \fi:
12632 }
12633 \cs_new:Npn \__fp_parse_compare_auxii:NNNNN #1#2#3#4#5
12634 {
12635 \exp_after:wN \__fp_parse_compare_auxi:NNNNNNN
12636 \exp_after:wN \prg_do_nothing:
12637 \exp_after:wN #1
12638 \exp_after:wN #2
12639 \exp_after:wN #3
12640 \exp_after:wN #4
12641 \exp_after:wN #5
12642 \exp:w \exp_after:wN \__fp_parse_expand:w
12643 }
12644 \cs_new:Npn \__fp_parse_compare_end:NNNNw #1#2#3#4#5 \fi:
12645 {
12646 \fi:
12647 \exp_after:wN @
12648 \exp_after:wN \__fp_parse_apply_compare:NwNNNNNNwN
12649 \exp_after:wN \c_one_fp
12650 \exp_after:wN #1
12651 \exp_after:wN #2
12652 \exp_after:wN #3
12653 \exp_after:wN #4
12654 \exp:w
12655 \__fp_parse_operand:Nw \c_seven \__fp_parse_expand:w #5
12656 }
12657 \cs_new:Npn \__fp_parse_apply_compare:NwNNNNNNwN
12658 #1 #2@ #3 #4#5#6#7 #8@ #9

```

```

12659 {
12660     \if_int_odd:w
12661         \if_meaning:w \c_zero_fp #3
12662         \c_zero
12663     \else:
12664         \if_case:w \__fp_compare_back:ww #8 #2 \exp_stop_f:
12665             #5 \or: #6 \or: #7 \else: #4
12666         \fi:
12667     \fi:
12668     \exp_after:wN \__fp_parse_apply_compare_aux:NNwN
12669     \exp_after:wN \c_one_fp
12670 \else:
12671     \exp_after:wN \__fp_parse_apply_compare_aux:NNwN
12672     \exp_after:wN \c_zero_fp
12673 \fi:
12674 #1 #8 #9
12675 }
12676 \cs_new:Npn \__fp_parse_apply_compare_aux:NNwN #1 #2 #3; #4
12677 {
12678     \if_meaning:w \__fp_parse_compare:NNNNNN #4
12679     \exp_after:wN \__fp_parse_continue_compare:NNwNN
12680     \exp_after:wN #1
12681     \exp_after:wN #2
12682     \exp:w \exp_end_continue_f:w
12683     \__fp_exp_after_o:w #3;
12684     \exp:w \exp_end_continue_f:w
12685 \else:
12686     \exp_after:wN \__fp_parse_continue:NwN
12687     \exp_after:wN #2
12688     \exp:w \exp_end_continue_f:w
12689     \exp_after:wN #1
12690     \exp:w \exp_end_continue_f:w
12691 \fi:
12692 #4 #2
12693 }
12694 \cs_new:Npn \__fp_parse_continue_compare:NNwNN #1#2 #3@ #4#5
12695 { #4 #2 #3@ #1 }

```

(End definition for __fp_parse_infix_<:N and others.)

25.8 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.

```

12696 \cs_new:Npn \fp_function:Nw #1
12697 {
12698     \exp_after:wN \__fp_function_apply:nw
12699     \exp_after:wN #1
12700     \exp:w
12701     \__fp_parse_operand:Nw \c_sixteen \__fp_parse_expand:w

```

```
12702 }
```

(End definition for \fp_function:Nw. This function is documented on page ??.)

```
\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).

```
12703 \cs_new_protected:Npn \fp_new_function:Npn #1#2#
12704 {
12705   \__fp_new_function:Ncfnn #1
12706   { \__fp_user_ \cs_to_str:N #1 }
12707   { \int_eval:n { \tl_count:n {#2} / \c_two } }
12708   {#2}
12709 }
12710 \cs_new_protected:Npn \__fp_new_function:NNnnn #1#2#3#4#5
12711 {
12712   \cs_new_nopar:Npn #1
12713   {
12714     \exp_after:wN \__fp_function_apply:nw \exp_after:wN
12715     {
12716       \exp_after:wN \__fp_function_args:Nwn
12717       \exp_after:wN #2
12718       \__int_value:w #3 \exp_after:wN ; \exp_after:wN
12719     }
12720     \exp:w
12721     \__fp_parse_operand:Nw \c_sixteen \__fp_parse_expand:w
12722   }
12723   \cs_new:Npn #2 #4 {#5}
12724 }
12725 \cs_generate_variant:Nn \__fp_new_function:NNnnn { Ncf }
12726 \cs_new:Npn \__fp_function_args:Nwn #1#2; #3
12727 {
12728   \int_compare:nNnTF { \tl_count:n {#3} } = {#2}
12729   { #1 #3 }
12730   {
12731     \__msg_kernel_expandable_error:nnnnn
12732     { kernel } { fp-num-args } { #1() } {#2} {#2}
12733     \c_nan_fp
12734   }
12735 }
```

(End definition for \fp_new_function:Npn. This function is documented on page ??.)

```
\__fp_function_apply:nw
\__fp_function_store:wwNwnn
\__fp_function_store_end:wnnn
```

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`.

```

12736 \cs_new:Npn \__fp_function_apply:nw #1#2 @
12737 {
12738   \__fp_parse:n
12739   {
12740     \__fp_function_store:wwNwnn #2
12741     \s__fp_mark \__fp_function_store:wwNwnn ;
12742     \s__fp_mark \__fp_function_store_end:wnnn
12743     \s__fp_stop { } { } {#1}
12744   }
12745   \s__fp_mark
12746 }
12747 \cs_new:Npn \__fp_function_store:wwNwnn
12748   #1; #2 \s__fp_mark #3#4 \s__fp_stop #5#6
12749   { #3 #2 \s__fp_mark #3#4 \s__fp_stop { #5 #6 } { { #1; } } }
12750 \cs_new:Npn \__fp_function_store_end:wnnn
12751   #1 \s__fp_stop #2#3#4
12752   { #4 {#2} }

```

(End definition for `__fp_function_apply:nw`, `__fp_function_store:wwNwnn`, and `__fp_function_store_end:wnnn`.)

25.9 Messages

```

12753 \__msg_kernel_new:nnn { kernel } { unknown-fp-word }
12754 { Unknown~fp-word~#1. }
12755 \__msg_kernel_new:nnn { kernel } { fp-missing }
12756 { Missing~#1~inserted #2. }
12757 \__msg_kernel_new:nnn { kernel } { fp-extra }
12758 { Extra~#1~ignored. }
12759 \__msg_kernel_new:nnn { kernel } { fp-early-end }
12760 { Premature~end~in~fp-expression. }
12761 \__msg_kernel_new:nnn { kernel } { fp-after-e }
12762 { Cannot~use~#1 after~'e'. }
12763 \__msg_kernel_new:nnn { kernel } { fp-missing-number }
12764 { Missing~number~before~'#1'. }
12765 \__msg_kernel_new:nnn { kernel } { fp-unknown-symbol }
12766 { Unknown~symbol~#1~ignored. }
12767 \__msg_kernel_new:nnn { kernel } { fp-extra-comma }
12768 { Unexpected~comma:~extra~arguments~ignored. }
12769 \__msg_kernel_new:nnn { kernel } { fp-num-args }
12770 { #1~expects~between~#2~and~#3~arguments. }
12771 {*package}
12772 \cs_if_exist:cT { @unexpandable@protect }

```

```

12773 {
12774   \_msg_kernel_new:nnn { kernel } { fp-robust-cmd }
12775   { Robust~command~#1 invalid-in~fp~expression! }
12776 }
12777 </package>
12778 </initex | package>

```

26 l3fp-logic Implementation

```

12779 <*initex | package>
12780 <@@=fp>

```

26.1 Syntax of internal functions

- `_fp_compare_npos:nwnw {<exp0>} <body1> ; {<exp0>} <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.

26.2 Existence test

`\fp_if_exist_p:N` Copies of the cs functions defined in l3basics.

```

\fp_if_exist_p:c 12781 \prg_new_eq_conditional:NNn \fp_if_exist:N \cs_if_exist:N { TF , T , F , p }
\fp_if_exist:NTF 12782 \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` and `\fp_if_exist:cTF`. These functions are documented on page 196.)

26.3 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:nTF
\_fp_compare_return:w 12783 \prg_new_conditional:Npnn \fp_compare:n #1 { p , T , F , TF }
12784 {
12785   \exp_after:wN \_fp_compare_return:w
12786   \exp:w \exp_end_continue_f:w \_fp_parse:n {#1}
12787 }
12788 \cs_new:Npn \_fp_compare_return:w \s__fp \_fp_chk:w #1#2;
12789 {
12790   \if_meaning:w 0 #1
12791   \prg_return_false:
12792   \else:

```

```

12793     \prg_return_true:
12794     \fi:
12795 }

```

(End definition for `\fp_compare:nTF`. This function is documented on page 197.)

`\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 `\fp_compare:nNnTF` ‘#2-‘=, which is -1 for <, 0 for =, 1 for > and 2 for ?.

```

12796 \prg_new_conditional:Npnn \fp_compare:nNn #1#2#3 { p , T , F , TF }
12797 {
12798   \if_int_compare:w
12799     \exp_after:wN \__fp_compare_aux:wn
12800     \exp:w \exp_end_continue_f:w \__fp_parse:n {#1} {#3}
12801     = \__int_eval:w '#2 - '= \__int_eval_end:
12802     \prg_return_true:
12803   \else:
12804     \prg_return_false:
12805   \fi:
12806 }
12807 \cs_new:Npn \__fp_compare_aux:wn #1; #2
12808 {
12809   \exp_after:wN \__fp_compare_back:ww
12810   \exp:w \exp_end_continue_f:w \__fp_parse:n {#2} #1;
12811 }

```

(End definition for `\fp_compare:nNnTF`. This function is documented on page 196.)

`__fp_compare_back:ww`
`__fp_compare_nan:w`

`__fp_compare_back:ww <y> ; <x> ;`
 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.

```

12812 \cs_new:Npn \__fp_compare_back:ww
12813   \s__fp \__fp_chk:w #1 #2 #3;
12814   \s__fp \__fp_chk:w #4 #5 #6;
12815 {
12816   \__int_value:w
12817   \if_meaning:w 3 #1 \exp_after:wN \__fp_compare_nan:w \fi:
12818   \if_meaning:w 3 #4 \exp_after:wN \__fp_compare_nan:w \fi:
12819   \if_meaning:w 2 #5 - \fi:
12820   \if_meaning:w #2 #5
12821     \if_meaning:w #1 #4
12822     \if_meaning:w 1 #1
12823     \__fp_compare_npos:nwnw #6; #3;
12824   \else:

```

```

12825         0
12826         \fi:
12827     \else:
12828         \if_int_compare:w #4 < #1 - \fi: 1
12829     \fi:
12830 \else:
12831     \if_int_compare:w #1#4 = \c_zero
12832     0
12833     \else:
12834     1
12835     \fi:
12836 \fi:
12837 \exp_stop_f:
12838 }
12839 \cs_new:Npn \__fp_compare_nan:w #1 \exp_stop_f: { \c_two }

```

(End definition for __fp_compare_back:ww and __fp_compare_nan:w.)

__fp_compare_npos:nwnw __fp_compare_npos:nwnw { $\langle expo_1 \rangle$ } $\langle body_1 \rangle$; { $\langle expo_2 \rangle$ } $\langle body_2 \rangle$;
__fp_compare_significand:nnnnnnnn

Within an __int_value:w ... \exp_stop_f: construction, this expands to 0 if the two numbers are equal, -1 if the first is smaller, and 1 if the first is bigger. First compare the exponents: the larger one denotes the larger number. If they are equal, we must compare significands. If both the first 8 digits and the next 8 digits coincide, the numbers are equal. If only the first 8 digits coincide, the next 8 decide. Otherwise, the first 8 digits are compared.

```

12840 \cs_new:Npn \__fp_compare_npos:nwnw #1#2; #3#4;
12841 {
12842     \if_int_compare:w #1 = #3 \exp_stop_f:
12843     \__fp_compare_significand:nnnnnnnn #2 #4
12844     \else:
12845         \if_int_compare:w #1 < #3 - \fi: 1
12846     \fi:
12847 }
12848 \cs_new:Npn \__fp_compare_significand:nnnnnnnn #1#2#3#4#5#6#7#8
12849 {
12850     \if_int_compare:w #1#2 = #5#6 \exp_stop_f:
12851     \if_int_compare:w #3#4 = #7#8 \exp_stop_f:
12852     0
12853     \else:
12854         \if_int_compare:w #3#4 < #7#8 - \fi: 1
12855     \fi:
12856     \else:
12857         \if_int_compare:w #1#2 < #5#6 - \fi: 1
12858     \fi:
12859 }

```

(End definition for __fp_compare_npos:nwnw.)

26.4 Floating point expression loops

`\fp_do_until:nn` These are quite easy given the above functions. The `do_until` and `do_while` versions execute the body, then test. The `until_do` and `while_do` do it the other way round.

```

\fp_do_while:nn
\fp_until_do:nn
\fp_while_do:nn
12860 \cs_new:Npn \fp_do_until:nn #1#2
12861 {
12862     #2
12863     \fp_compare:nF {#1}
12864     { \fp_do_until:nn {#1} {#2} }
12865 }
12866 \cs_new:Npn \fp_do_while:nn #1#2
12867 {
12868     #2
12869     \fp_compare:nT {#1}
12870     { \fp_do_while:nn {#1} {#2} }
12871 }
12872 \cs_new:Npn \fp_until_do:nn #1#2
12873 {
12874     \fp_compare:nF {#1}
12875     {
12876         #2
12877         \fp_until_do:nn {#1} {#2}
12878     }
12879 }
12880 \cs_new:Npn \fp_while_do:nn #1#2
12881 {
12882     \fp_compare:nT {#1}
12883     {
12884         #2
12885         \fp_while_do:nn {#1} {#2}
12886     }
12887 }
```

(End definition for `\fp_do_until:nn` and others. These functions are documented on page 198.)

`\fp_do_until:nNn` As above but not using the `nNn` syntax.

```

\fp_do_while:nNn
\fp_until_do:nNn
\fp_while_do:nNn
12888 \cs_new:Npn \fp_do_until:nNn #1#2#3#4
12889 {
12890     #4
12891     \fp_compare:nNnF {#1} #2 {#3}
12892     { \fp_do_until:nNn {#1} #2 {#3} {#4} }
12893 }
12894 \cs_new:Npn \fp_do_while:nNn #1#2#3#4
12895 {
12896     #4
12897     \fp_compare:nNnT {#1} #2 {#3}
12898     { \fp_do_while:nNn {#1} #2 {#3} {#4} }
12899 }
12900 \cs_new:Npn \fp_until_do:nNn #1#2#3#4
12901 {
```

```

12902     \fp_compare:nNnF {#1} #2 {#3}
12903     {
12904         #4
12905         \fp_until_do:nNnn {#1} #2 {#3} {#4}
12906     }
12907 }
12908 \cs_new:Npn \fp_while_do:nNnn #1#2#3#4
12909 {
12910     \fp_compare:nNnT {#1} #2 {#3}
12911     {
12912         #4
12913         \fp_while_do:nNnn {#1} #2 {#3} {#4}
12914     }
12915 }

```

(End definition for `\fp_do_until:nNnn` and others. These functions are documented on page 197.)

26.5 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`.

```

12916 \cs_new:Npn \__fp_minmax_o:Nw #1#2 @
12917 {
12918     \if_meaning:w 0 #1
12919     \exp_after:wN \__fp_minmax_loop:Nww \exp_after:wN \c_one
12920     \else:
12921     \exp_after:wN \__fp_minmax_loop:Nww \exp_after:wN \c_minus_one
12922     \fi:
12923     #2
12924     \s__fp \__fp_chk:w 2 #1 \s__fp_exact ;
12925     \s__fp \__fp_chk:w { 3 \__fp_minmax_break_o:w } ;
12926 }

```

(End definition for `__fp_minmax_o:Nw`.)

`__fp_minmax_loop:Nww` The first argument is `-1` or `1` 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.

```

12927 \cs_new:Npn \__fp_minmax_loop:Nww
12928     #1 \s__fp \__fp_chk:w #2#3; \s__fp \__fp_chk:w #4#5;

```

```

12929 {
12930   \if_meaning:w 3 #4
12931   \if_meaning:w 3 #2
12932   \__fp_minmax_auxi:ww
12933   \else:
12934   \__fp_minmax_auxii:ww
12935   \fi:
12936   \else:
12937   \if_int_compare:w
12938   \__fp_compare_back:ww
12939   \s__fp \__fp_chk:w #4#5;
12940   \s__fp \__fp_chk:w #2#3;
12941   = #1
12942   \__fp_minmax_auxii:ww
12943   \else:
12944   \__fp_minmax_auxi:ww
12945   \fi:
12946   \fi:
12947   \__fp_minmax_loop:Nww #1
12948   \s__fp \__fp_chk:w #2#3;
12949   \s__fp \__fp_chk:w #4#5;
12950 }

```

(End definition for __fp_minmax_loop:Nww.)

```

\__fp_minmax_auxi:ww Keep the first/second number, and remove the other.
\__fp_minmax_auxii:ww
12951 \cs_new:Npn \__fp_minmax_auxi:ww #1 \fi: \fi: #2 \s__fp #3 ; \s__fp #4;
12952 { \fi: \fi: #2 \s__fp #3 ; }
12953 \cs_new:Npn \__fp_minmax_auxii:ww #1 \fi: \fi: #2 \s__fp #3 ;
12954 { \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.

```

12955 \cs_new:Npn \__fp_minmax_break_o:w #1 \fi: \fi: #2 \s__fp #3; #4;
12956 { \fi: \__fp_exp_after_o:w \s__fp #3; }

```

(End definition for __fp_minmax_break_o:w.)

26.6 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.

```

12957 \cs_new:cpn { __fp_not_o:w } #1 \s__fp \__fp_chk:w #2#3; @
12958 {
12959   \if_meaning:w 0 #2
12960   \exp_after:wN \exp_after:wN \exp_after:wN \c_one_fp
12961   \else:

```

```

12962         \exp_after:wN \exp_after:wN \exp_after:wN \c_zero_fp
12963     \fi:
12964 }

```

(End definition for `_fp_not_o:w`.)

`_fp_&o:ww` For **and**, if the first number is zero, return it (with the same sign). Otherwise, return
`_fp_|o:ww` the second one. For **or**, the logic is reversed: if the first number is non-zero, return
`_fp_and_return:wNw` 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.

```

12965 \group_begin:
12966     \char_set_catcode_letter:N &
12967     \char_set_catcode_letter:N |
12968     \cs_new:Npn \_fp_&o:ww #1 \s_fp \_fp_chk:w #2#3;
12969     {
12970         \if_meaning:w 0 #2 #1
12971             \_fp_and_return:wNw \s_fp \_fp_chk:w #2#3;
12972         \fi:
12973         \_fp_exp_after_o:w
12974     }
12975     \cs_new_nopar:Npn \_fp_|o:ww { \_fp_&o:ww \else: }
12976 \group_end:
12977 \cs_new:Npn \_fp_and_return:wNw #1; \fi: #2#3; { \fi: #2 #1; }

```

(End definition for `_fp_&o:ww`.)

26.7 Ternary operator

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.

```

12978 \cs_new:Npn \_fp_ternary:NwwN #1 #2@ #3@ #4
12979 {
12980     \if_meaning:w \_fp_parse_infix_::N #4
12981         \_fp_ternary_loop:Nw
12982         #2
12983         \s_fp \_fp_chk:w { \_fp_ternary_loop_break:w } ;
12984         \_fp_ternary_break_point:n { \exp_after:wN \_fp_ternary_auxi:NwwN }
12985         \exp_after:wN #1
12986         \exp:w \exp_end_continue_f:w
12987         \_fp_exp_after_array_f:w #3 \s_fp_stop
12988         \exp_after:wN @
12989         \exp:w
12990         \_fp_parse_operand:Nw \c_two
12991         \_fp_parse_expand:w
12992     \else:

```

```

12993     \_msg_kernel_expandable_error:nnnn
12994     { kernel } { fp-missing } { : } { ~for~?: }
12995     \exp_after:wN \_fp_parse_continue:NwN
12996     \exp_after:wN #1
12997     \exp:w \exp_end_continue_f:w
12998     \_fp_exp_after_array_f:w #3 \s\_fp_stop
12999     \exp_after:wN #4
13000     \exp_after:wN #1
13001     \fi:
13002   }
13003   \cs_new:Npn \_fp_ternary_loop_break:w
13004     #1 \fi: #2 \_fp_ternary_break_point:n #3
13005   {
13006     \c_zero = \c_zero \fi:
13007     \exp_after:wN \_fp_ternary_auxii:NwwN
13008   }
13009   \cs_new:Npn \_fp_ternary_loop:Nw \s\_fp \_fp_chk:w #1#2;
13010   {
13011     \if_int_compare:w #1 > \c_zero
13012       \exp_after:wN \_fp_ternary_map_break:
13013     \fi:
13014     \_fp_ternary_loop:Nw
13015   }
13016   \cs_new:Npn \_fp_ternary_map_break: #1 \_fp_ternary_break_point:n #2 {#2}
13017   \cs_new:Npn \_fp_ternary_auxi:NwwN #1#2@#3@#4
13018   {
13019     \exp_after:wN \_fp_parse_continue:NwN
13020     \exp_after:wN #1
13021     \exp:w \exp_end_continue_f:w
13022     \_fp_exp_after_array_f:w #2 \s\_fp_stop
13023     #4 #1
13024   }
13025   \cs_new:Npn \_fp_ternary_auxii:NwwN #1#2@#3@#4
13026   {
13027     \exp_after:wN \_fp_parse_continue:NwN
13028     \exp_after:wN #1
13029     \exp:w \exp_end_continue_f:w
13030     \_fp_exp_after_array_f:w #3 \s\_fp_stop
13031     #4 #1
13032   }

```

(End definition for _fp_ternary:NwwN, _fp_ternary_auxi:NwwN, and _fp_ternary_auxii:NwwN.)

```

13033 \</initex | package>

```

27 l3fp-basics Implementation

```

13034 \*initex | package>
13035 \@@=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>.

27.1 Common to several operations

```

\__fp_basics_pack_low:NNNNNw
  \_fp_basics_pack_high:NNNNNw
  \_fp_basics_pack_high_carry:w

```

Addition and multiplication of significands are done in two steps: first compute a (more or less) exact result, then round and pack digits in the final (braced) form. These functions take care of the packing, with special attention given to the case where rounding has caused a carry. Since rounding can only shift the final digit by 1, a carry always produces an exact power of 10. Thus, `__fp_basics_pack_high_carry:w` is always followed by four times `{0000}`.

```

13036 \cs_new:Npn \__fp_basics_pack_low:NNNNNw #1 #2#3#4#5 #6;
13037 { + #1 - \c_one ; {#2#3#4#5} {#6} ; }
13038 \cs_new:Npn \__fp_basics_pack_high:NNNNNw #1 #2#3#4#5 #6;
13039 {
13040   \if_meaning:w 2 #1
13041     \__fp_basics_pack_high_carry:w
13042   \fi:
13043   ; {#2#3#4#5} {#6}
13044 }
13045 \cs_new:Npn \__fp_basics_pack_high_carry:w \fi: ; #1
13046 { \fi: + \c_one ; {1000} }

```

(End definition for `__fp_basics_pack_low:NNNNNw`, `__fp_basics_pack_high:NNNNNw`, and `__fp-basics_pack_high_carry:w`.)

```

\_fp_basics_pack_weird_low:NNNNw
\__fp_basics_pack_weird_high:NNNNNNNNw

```

I don’t fully understand those functions, used for additions and divisions. Hence the name.

```

13047 \cs_new:Npn \__fp_basics_pack_weird_low:NNNNw #1 #2#3#4 #5;
13048 {
13049   \if_meaning:w 2 #1
13050     + \c_one
13051   \fi:
13052   \__int_eval_end:
13053   #2#3#4; {#5} ;
13054 }
13055 \cs_new:Npn \__fp_basics_pack_weird_high:NNNNNNNNw
13056 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`.)

27.2 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.

27.2.1 Sign, exponent, and special numbers

`__fp_-_o:ww` A previous version of this function grabbed its two operands, changed the sign of the second, and called `__fp+_o:ww`. However, for efficiency reasons, the operands were swapped in the process, which means that error messages ended up wrong. Now, 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 check that it was followed by `\s__fp` and not arbitrary junk.

```

13057 \cs_new_nopar:cpx { __fp_-_o:ww } \s__fp
13058 {
13059   \exp_not:c { __fp+_o:ww }
13060   \exp_not:n { \s__fp \__fp_neg_sign:N }
13061 }
```

(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 (equivalent to changing the $\langle sign_2 \rangle$ of the second operand). 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.

```

13062 \cs_new:cpn { __fp+_o:ww }
13063   \s__fp #1 \__fp_chk:w #2 #3 ; \s__fp \__fp_chk:w #4 #5
13064   {
13065     \if_case:w
13066       \if_meaning:w #2 #4
13067         #2 \exp_stop_f:
13068     \else:
13069       \if_int_compare:w #2 > #4 \exp_stop_f:
13070       \c_three
13071     \else:
13072       \c_minus_one
13073     \fi:
13074   \fi:
13075     \exp_after:wN \__fp_add_zeros_o:Nww \__int_value:w
13076 \or:   \exp_after:wN \__fp_add_normal_o:Nww \__int_value:w
13077 \or:   \exp_after:wN \__fp_add_inf_o:Nww \__int_value:w
13078 \or:   \__fp_case_return_i_o:ww
13079 \else: \exp_after:wN \__fp_add_return_ii_o:Nww \__int_value:w
13080 \fi:
13081 #1 #5
13082 \s__fp \__fp_chk:w #2 #3 ;
13083 \s__fp \__fp_chk:w #4 #5
13084 }

```

(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.

```

13085 \cs_new:Npn \__fp_add_return_ii_o:Nww #1 #2 ; \s__fp \__fp_chk:w #3 #4
13086   { \__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`.

```

13087 \cs_new:Npn \__fp_add_zeros_o:Nww #1 \s__fp \__fp_chk:w 0 #2
13088   {
13089     \if_int_compare:w #2 #1 = 20 \exp_stop_f:
13090     \exp_after:wN \__fp_add_return_ii_o:Nww
13091   \else:
13092     \__fp_case_return_i_o:ww
13093   \fi:
13094   #1
13095   \s__fp \__fp_chk:w 0 #2
13096 }

```

(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.

```

13097 \cs_new:Npn \_fp\_add\_inf\_o:Nww
13098   #1 \s\_fp \_fp\_chk:w 2 #2 #3; \s\_fp \_fp\_chk:w 2 #4
13099   {
13100     \if_meaning:w #1 #2
13101       \_fp\_case\_return\_i\_o:ww
13102     \else:
13103       \_fp\_case\_use:nw
13104       {
13105         \if_meaning:w #1 #4
13106           \exp\_after:wN \_fp\_invalid\_operation\_o:Nww
13107           \exp\_after:wN +
13108         \else:
13109           \exp\_after:wN \_fp\_invalid\_operation\_o:Nww
13110           \exp\_after:wN -
13111         \fi:
13112       }
13113     \fi:
13114     \s\_fp \_fp\_chk:w 2 #2 #3;
13115     \s\_fp \_fp\_chk:w 2 #4
13116   }

```

(End definition for `_fp_add_inf_o:Nww`.)

`_fp_add_normal_o:Nww` `_fp_add_normal_o:Nww` $\langle sign_2 \rangle$ `\s_fp _fp_chk:w 1` $\langle sign_1 \rangle$ $\langle exp_1 \rangle$
 $\langle body_1 \rangle$; `\s_fp _fp_chk:w 1` $\langle initial\ sign_2 \rangle$ $\langle exp_2 \rangle$ $\langle body_2 \rangle$;

We now have two normal numbers to add, and we have to check signs and exponents more carefully before performing the addition.

```

13117 \cs_new:Npn \_fp\_add\_normal\_o:Nww #1 \s\_fp \_fp\_chk:w 1 #2
13118   {
13119     \if_meaning:w #1#2
13120       \exp\_after:wN \_fp\_add\_npos\_o:NnwNnw
13121     \else:
13122       \exp\_after:wN \_fp\_sub\_npos\_o:NnwNnw
13123     \fi:
13124     #2
13125   }

```

(End definition for `_fp_add_normal_o:Nww`.)

27.2.2 Absolute addition

In this subsection, we perform the addition of two positive normal numbers.

```

\__fp_add_npos_o:NnwNnw \__fp_add_npos_o:NnwNnw  $\langle sign_1 \rangle$   $\langle exp_1 \rangle$   $\langle body_1 \rangle$  ; \s__fp \__fp_chk:w 1
 $\langle initial\ sign_2 \rangle$   $\langle exp_2 \rangle$   $\langle body_2 \rangle$  ;

```

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.

```

13126 \cs_new:Npn \__fp_add_npos_o:NnwNnw #1#2#3 ; \s__fp \__fp_chk:w 1 #4 #5
13127 {
13128   \exp_after:wN \__fp_sanitize:Nw
13129   \exp_after:wN #1
13130   \int_use:N \__int_eval:w
13131   \if_int_compare:w #2 > #5 \exp_stop_f:
13132     #2
13133     \exp_after:wN \__fp_add_big_i_o:wNww \__int_value:w -
13134   \else:
13135     #5
13136     \exp_after:wN \__fp_add_big_ii_o:wNww \__int_value:w
13137   \fi:
13138   \__int_eval:w #5 - #2 ; #1 #3;
13139 }

```

(End definition for `__fp_add_npos_o:NnwNnw`.)

```

\__fp_add_big_i_o:wNww \__fp_add_big_i_o:wNww  $\langle shift \rangle$  ;  $\langle final\ sign \rangle$   $\langle body_1 \rangle$  ;  $\langle body_2 \rangle$  ;
\__fp_add_big_ii_o:wNww Shift the significand of the small number, then add with \__fp_add_significand_
o:NnnwnnnnN.

```

```

13140 \cs_new:Npn \__fp_add_big_i_o:wNww #1; #2 #3; #4;
13141 {
13142   \__fp_decimate:nNnnnn {#1}
13143   \__fp_add_significand_o:NnnwnnnnN
13144   #4
13145   #3
13146   #2
13147 }
13148 \cs_new:Npn \__fp_add_big_ii_o:wNww #1; #2 #3; #4;
13149 {
13150   \__fp_decimate:nNnnnn {#1}
13151   \__fp_add_significand_o:NnnwnnnnN
13152   #3
13153   #4
13154   #2
13155 }

```

(End definition for `__fp_add_big_i_o:wNww`.)

```

\__fp_add_significand_o:NnnwnnnnN
\__fp_add_significand_pack:NNNNNNN
\__fp_add_significand_test_o:N

```

$\backslash_fp_add_significand_o:NnnwnnnnN$ $\langle rounding\ digit \rangle \{ \langle Y'_1 \rangle \} \{ \langle Y'_2 \rangle \}$
 $\langle extra-digits \rangle ; \{ \langle X_1 \rangle \} \{ \langle X_2 \rangle \} \{ \langle X_3 \rangle \} \{ \langle X_4 \rangle \} \langle final\ sign \rangle$

To round properly, we must know at which digit the rounding should occur. This requires to know whether the addition produces an overall carry or not. Thus, we do the computation now and check for a carry, then go back and do the rounding. The rounding may cause a carry in very rare cases such as $0.99 \dots 95 \rightarrow 1.00 \dots 0$, but this situation always give an exact power of 10, for which it is easy to correct the result at the end.

```

13156 \cs_new:Npn \__fp_add_significand_o:NnnwnnnnN #1 #2#3 #4; #5#6#7#8
13157 {
13158   \exp_after:wN \__fp_add_significand_test_o:N
13159   \int_use:N \__int_eval:w 1#5#6 + #2
13160   \exp_after:wN \__fp_add_significand_pack:NNNNNNN
13161   \int_use:N \__int_eval:w 1#7#8 + #3 ; #1
13162 }
13163 \cs_new:Npn \__fp_add_significand_pack:NNNNNNN #1 #2#3#4#5#6#7
13164 {
13165   \if_meaning:w 2 #1
13166     + \c_one
13167   \fi:
13168   ; #2 #3 #4 #5 #6 #7 ;
13169 }
13170 \cs_new:Npn \__fp_add_significand_test_o:N #1
13171 {
13172   \if_meaning:w 2 #1
13173     \exp_after:wN \__fp_add_significand_carry_o:wwwNN
13174   \else:
13175     \exp_after:wN \__fp_add_significand_no_carry_o:wwwNN
13176   \fi:
13177 }

```

(End definition for $\backslash_fp_add_significand_o:NnnwnnnnN$.)

```

\__fp_add_significand_no_carry_o:wwwNN

```

$\backslash_fp_add_significand_no_carry_o:wwwNN$ $\langle 8d \rangle ; \langle 6d \rangle ; \langle 2d \rangle ; \langle rounding\ digit \rangle \langle sign \rangle$

If there's no carry, grab all the digits again and round. The packing function $\backslash_fp_basics_pack_high:NNNNNw$ takes care of the case where rounding brings a carry.

```

13178 \cs_new:Npn \__fp_add_significand_no_carry_o:wwwNN
13179   #1; #2; #3#4 ; #5#6
13180 {
13181   \exp_after:wN \__fp_basics_pack_high:NNNNNw
13182   \int_use:N \__int_eval:w 1 #1
13183   \exp_after:wN \__fp_basics_pack_low:NNNNNw
13184   \int_use:N \__int_eval:w 1 #2 #3#4
13185     + \__fp_round:NNN #6 #4 #5
13186   \exp_after:wN ;
13187 }

```

(End definition for $\backslash_fp_add_significand_no_carry_o:wwwNN$.)

`_fp_add_significand_carry_o:wwwNN` $\langle 8d \rangle$; $\langle 6d \rangle$; $\langle 2d \rangle$; $\langle \text{rounding digit} \rangle$ $\langle \text{sign} \rangle$

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.

```

13188 \cs_new:Npn \_fp\_add\_significand\_carry\_o:wwwNN
13189   #1; #2; #3#4; #5#6
13190   {
13191     + \c_one
13192     \exp\_after:wN \_fp\_basics\_pack\_weird\_high:NNNNNNNNw
13193     \int\_use:N \_int\_eval:w 1 1 #1
13194     \exp\_after:wN \_fp\_basics\_pack\_weird\_low:NNNNw
13195     \int\_use:N \_int\_eval:w 1 #2#3 +
13196     \exp\_after:wN \_fp\_round:NNN
13197     \exp\_after:wN #6
13198     \exp\_after:wN #3
13199     \_int\_value:w \_fp\_round\_digit:Nw #4 #5 ;
13200     \exp\_after:wN ;
13201   }

```

(End definition for `_fp_add_significand_carry_o:wwwNN`.)

27.2.3 Absolute subtraction

`_fp_sub_npos_o:NnwNnw` $\langle \text{sign}_1 \rangle$ $\langle \text{exp}_1 \rangle$ $\langle \text{body}_1 \rangle$; `\s_fp` `_fp_chk:w 1`
`_fp_sub_eq_o:Nnwnw` $\langle \text{initial sign}_2 \rangle$ $\langle \text{exp}_2 \rangle$ $\langle \text{body}_2 \rangle$;
`_fp_sub_npos_ii_o:Nnwnw`

Rounding properly in some modes requires to know what the sign of the result will be. Thus, we start by comparing the exponents and significands. If the numbers coincide, return zero. If the second number is larger, swap the numbers and call `_fp_sub_npos_i_o:Nnwnw` with the opposite of $\langle \text{sign}_1 \rangle$.

```

13202 \cs_new:Npn \_fp\_sub\_npos\_o:NnwNnw #1#2#3; \s\_fp \_fp\_chk:w 1 #4#5#6;
13203   {
13204     \if\_case:w \_fp\_compare\_npos:nwnw {#2} #3; {#5} #6; \exp\_stop\_f:
13205     \exp\_after:wN \_fp\_sub\_eq\_o:Nnwnw
13206     \or:
13207     \exp\_after:wN \_fp\_sub\_npos\_i\_o:Nnwnw
13208     \else:
13209     \exp\_after:wN \_fp\_sub\_npos\_ii\_o:Nnwnw
13210     \fi:
13211     #1 {#2} #3; {#5} #6;
13212   }
13213 \cs_new:Npn \_fp\_sub\_eq\_o:Nnwnw #1#2; #3; { \exp\_after:wN \c\_zero\_fp }
13214 \cs_new:Npn \_fp\_sub\_npos\_ii\_o:Nnwnw #1 #2; #3;
13215   {
13216     \exp\_after:wN \_fp\_sub\_npos\_i\_o:Nnwnw
13217     \int\_use:N \_int\_eval:w \c\_two - #1 \_int\_eval\_end:
13218     #3; #2;
13219   }

```

(End definition for `_fp_sub_npos_o:NnwNnw`.)

`__fp_sub_npos_i_o:Nnwnw` After the computation is done, `__fp_sanitiz: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.

```

13220 \cs_new:Npn \__fp_sub_npos_i_o:Nnwnw #1 #2#3; #4#5;
13221 {
13222   \exp_after:wN \__fp_sanitiz:Nw
13223   \exp_after:wN #1
13224   \int_use:N \__int_eval:w
13225   #2
13226   \if_int_compare:w #2 = #4 \exp_stop_f:
13227     \exp_after:wN \__fp_sub_back_near_o:nnnnnnnnN
13228   \else:
13229     \exp_after:wN \__fp_decimate:nNnnnn \exp_after:wN
13230     { \int_use:N \__int_eval:w #2 - #4 - \c_one \exp_after:wN }
13231     \exp_after:wN \__fp_sub_back_far_o:NnnwnnnnnN
13232   \fi:
13233   #5
13234   #3
13235   #1
13236 }

```

(End definition for `__fp_sub_npos_i_o:Nnwnw`.)

`__fp_sub_back_near_o:nnnnnnnnN` `__fp_sub_back_near_o:nnnnnnnnN { $\langle Y_1 \rangle$ } { $\langle Y_2 \rangle$ } { $\langle Y_3 \rangle$ } { $\langle Y_4 \rangle$ } { $\langle X_1 \rangle$ }`
`__fp_sub_back_near_pack:NNNNNNw` `{ $\langle X_2 \rangle$ } { $\langle X_3 \rangle$ } { $\langle X_4 \rangle$ } $\langle final\ sign \rangle$`
`__fp_sub_back_near_after:wNNNNw`

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.

```

13237 \cs_new:Npn \__fp_sub_back_near_o:nnnnnnnnN #1#2#3#4 #5#6#7#8 #9
13238 {
13239   \exp_after:wN \__fp_sub_back_near_after:wNNNNw
13240   \int_use:N \__int_eval:w 10#5#6 - #1#2 - \c_eleven
13241   \exp_after:wN \__fp_sub_back_near_pack:NNNNNNw
13242   \int_use:N \__int_eval:w 11#7#8 - #3#4 \exp_after:wN ;
13243 }
13244 \cs_new:Npn \__fp_sub_back_near_pack:NNNNNNw #1#2#3#4#5#6#7 ;
13245 { + #1#2 ; {#3#4#5#6} {#7} ; }
13246 \cs_new:Npn \__fp_sub_back_near_after:wNNNNw 10 #1#2#3#4 #5 ;
13247 {
13248   \if_meaning:w 0 #1
13249     \exp_after:wN \__fp_sub_back_shift:wnnnn
13250   \fi:
13251   ; {#1#2#3#4} {#5}
13252 }

```

(End definition for _fp_sub_back_near_o:nnnnnnnnN.)

_fp_sub_back_shift:wnnnn
 _fp_sub_back_shift_ii:ww
 _fp_sub_back_shift_iii:NNNNNNNNw
 _fp_sub_back_shift_iv:nnnnw

_fp_sub_back_shift:wnnnn ; { $\langle Z_1 \rangle$ } { $\langle Z_2 \rangle$ } { $\langle Z_3 \rangle$ } { $\langle Z_4 \rangle$ } ;
 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 T_EX'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.

```

13253 \cs_new:Npn \_fp_sub_back_shift:wnnnn ; #1#2
13254 {
13255   \exp_after:wN \_fp_sub_back_shift_ii:ww
13256   \_int_value:w #1 #2 0 ;
13257 }
13258 \cs_new:Npn \_fp_sub_back_shift_ii:ww #1 0 ; #2#3 ;
13259 {
13260   \if_meaning:w @ #1 @
13261   - \c_seven
13262   - \exp_after:wN \use_i:nnn
13263   \exp_after:wN \_fp_sub_back_shift_iii:NNNNNNNNw
13264   \_int_value:w #2#3 0 ~ 123456789;
13265   \else:
13266   - \_fp_sub_back_shift_iii:NNNNNNNNw #1 123456789;
13267   \fi:
13268   \exp_after:wN \_fp_pack_twice_four:wNNNNNNNN
13269   \exp_after:wN \_fp_pack_twice_four:wNNNNNNNN
13270   \exp_after:wN \_fp_sub_back_shift_iv:nnnnw
13271   \exp_after:wN ;
13272   \_int_value:w
13273   #1 ~ #2#3 0 ~ 0000 0000 0000 000 ;
13274 }
13275 \cs_new:Npn \_fp_sub_back_shift_iii:NNNNNNNNw #1#2#3#4#5#6#7#8#9; {#8}
13276 \cs_new:Npn \_fp_sub_back_shift_iv:nnnnw #1 ; #2 ; { ; #1 ; }

```

(End definition for _fp_sub_back_shift:wnnnn.)

_fp_sub_back_far_o:NnnwnnnnN

_fp_sub_back_far_o:NnnwnnnnN $\langle \text{rounding} \rangle$ { $\langle Y'_1 \rangle$ } { $\langle Y'_2 \rangle$ }
 $\langle \text{extra-digits} \rangle$; { $\langle X_1 \rangle$ } { $\langle X_2 \rangle$ } { $\langle X_3 \rangle$ } { $\langle X_4 \rangle$ } $\langle \text{final sign} \rangle$

If the difference is greater than $10^{\langle expo_x \rangle}$, call the **very_far** auxiliary. If the result is less than $10^{\langle expo_x \rangle}$, call the **not_far** auxiliary. If it is too close a call to know yet, namely if $1\langle Y'_1 \rangle \langle Y'_2 \rangle = \langle X_1 \rangle \langle X_2 \rangle \langle X_3 \rangle \langle X_4 \rangle 0$, then call the **quite_far** auxiliary. We use the odd combination of space and semi-colon delimiters to allow the **not_far** auxiliary to grab each piece individually, the **very_far** auxiliary to use _fp_pack_eight:wNNNNNNNN, and the **quite_far** to ignore the significands easily (using the ; delimiter).

```

13277 \cs_new:Npn \_fp_sub_back_far_o:NnnwnnnnN #1 #2#3 #4; #5#6#7#8
13278 {

```

```

13279 \if_case:w
13280 \if_int_compare:w 1 #2 = #5#6 \use_i:nnnn #7 \exp_stop_f:
13281 \if_int_compare:w #3 = \use_none:n #7#8 0 \exp_stop_f:
13282 \c_zero
13283 \else:
13284 \if_int_compare:w #3 > \use_none:n #7#8 0 - \fi: \c_one
13285 \fi:
13286 \else:
13287 \if_int_compare:w 1 #2 > #5#6 \use_i:nnnn #7 - \fi: \c_one
13288 \fi:
13289 \exp_after:wN \__fp_sub_back_quite_far_o:wwNN
13290 \or: \exp_after:wN \__fp_sub_back_very_far_o:wwwNN
13291 \else: \exp_after:wN \__fp_sub_back_not_far_o:wwwNN
13292 \fi:
13293 #2 ~ #3 ; #5 #6 ~ #7 #8 ; #1
13294 }

```

(End definition for __fp_sub_back_far_o:NnnwnnnnN.)

__fp_sub_back_quite_far_o:wwNN
__fp_sub_back_quite_far_ii:NN

The easiest case is when $x - y$ is extremely close to a power of 10, namely the first digit of x is 1, and all others vanish when subtracting y . Then the *rounding* #3 and the *final sign* #4 control whether we get 1 or 0.9999999999999999. In the usual round-to-nearest mode, we will get 1 whenever the *rounding* digit is less than or equal to 5 (remember that the *rounding* digit is only equal to 5 if there was no further non-zero digit).

```

13295 \cs_new:Npn \__fp_sub_back_quite_far_o:wwNN #1; #2; #3#4
13296 {
13297 \exp_after:wN \__fp_sub_back_quite_far_ii:NN
13298 \exp_after:wN #3
13299 \exp_after:wN #4
13300 }
13301 \cs_new:Npn \__fp_sub_back_quite_far_ii:NN #1#2
13302 {
13303 \if_case:w \__fp_round_neg:NNN #2 0 #1
13304 \exp_after:wN \use_i:nn
13305 \else:
13306 \exp_after:wN \use_ii:nn
13307 \fi:
13308 { ; {1000} {0000} {0000} {0000} ; }
13309 { - \c_one ; {9999} {9999} {9999} {9999} ; }
13310 }

```

(End definition for __fp_sub_back_quite_far_o:wwNN.)

__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 $- \c_one$). 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.

```

13311 \cs_new:Npn \__fp_sub_back_not_far_o:wwwNN #1 ~ #2; #3 ~ #4; #5#6
13312 {
13313   - \c_one
13314   \exp_after:wN \__fp_sub_back_near_after:wNNNNw
13315   \int_use:N \__int_eval:w 1#30 - #1 - \c_eleven
13316   \exp_after:wN \__fp_sub_back_near_pack:NNNNNNw
13317   \int_use:N \__int_eval:w 11 0000 0000 + #40 - #2
13318   - \exp_after:wN \__fp_round_neg:NNN
13319   \exp_after:wN #6
13320   \use_none:nnnnnn #2 #5
13321   \exp_after:wN ;
13322 }

```

(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.

```

13323 \cs_new:Npn \__fp_sub_back_very_far_o:wwwNN #1#2#3#4#5#6#7
13324 {
13325   \__fp_pack_eight:wNNNNNNNN
13326   \__fp_sub_back_very_far_ii_o:nnNwwNN
13327   { 0 #1#2#3 #4#5#6#7 }
13328   ;
13329 }
13330 \cs_new:Npn \__fp_sub_back_very_far_ii_o:nnNwwNN #1#2 ; #3 ; #4 ~ #5; #6#7
13331 {
13332   \exp_after:wN \__fp_basics_pack_high:NNNNNw
13333   \int_use:N \__int_eval:w 1#4 - #1 - \c_one
13334   \exp_after:wN \__fp_basics_pack_low:NNNNNw
13335   \int_use:N \__int_eval:w 2#5 - #2
13336   - \exp_after:wN \__fp_round_neg:NNN
13337   \exp_after:wN #7
13338   \__int_value:w
13339   \if_int_odd:w \__int_eval:w #5 - #2 \__int_eval_end:
13340     1 \else: 2 \fi:
13341   \__int_value:w \__fp_round_digit:Nw #3 #6 ;
13342   \exp_after:wN ;
13343 }

```

(End definition for `__fp_sub_back_very_far_o:wwwNN`.)

27.3 Multiplication

27.3.1 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`.

```

13344 \cs_new_nopar:cpn { __fp*_o:ww }
13345 {
13346   \__fp_mul_cases_o:NnNnw
13347   *
13348   { - \c_two + }
13349   \__fp_mul_npos_o:Nww
13350   { }
13351 }
```

(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.

```

13352 \cs_new:Npn \__fp_mul_cases_o:NnNnw
13353   #1#2#3#4 \s__fp \__fp_chk:w #5#6#7; \s__fp \__fp_chk:w #8#9
13354 {
13355   \if_case:w \__int_eval:w
13356     \if_int_compare:w #5 #8 = \c_eleven
13357     \c_one
13358   \else:
13359     \if_meaning:w 3 #8
13360     \c_three
13361   \else:
13362     \if_meaning:w 3 #5
13363     \c_two
13364   \else:
13365     \if_int_compare:w #5 #8 = \c_ten
13366     \c_nine #2 - \c_two
13367   \else:
13368     (#5 #2 #8) / \c_two * \c_two + \c_seven
13369   \fi:
```

```

13370         \fi:
13371     \fi:
13372     \fi:
13373     \if_meaning:w #6 #9 - \c_one \fi:
13374     \__int_eval_end:
13375     \__fp_case_use:nw { #3 0 }
13376 \or: \__fp_case_use:nw { #3 2 }
13377 \or: \__fp_case_return_i_o:ww
13378 \or: \__fp_case_return_ii_o:ww
13379 \or: \__fp_case_return_o:Nww \c_zero_fp
13380 \or: \__fp_case_return_o:Nww \c_minus_zero_fp
13381 \or: \__fp_case_use:nw { \__fp_invalid_operation_o:Nww #1 }
13382 \or: \__fp_case_use:nw { \__fp_invalid_operation_o:Nww #1 }
13383 \or: \__fp_case_return_o:Nww \c_inf_fp
13384 \or: \__fp_case_return_o:Nww \c_minus_inf_fp
13385 #4
13386 \fi:
13387 \s__fp \__fp_chk:w #5 #6 #7;
13388 \s__fp \__fp_chk:w #8 #9
13389 }

```

(End definition for __fp_mul_cases_o:nNnnww.)

27.3.2 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.

```

13390 \cs_new:Npn \__fp_mul_npos_o:Nww
13391 #1 \s__fp \__fp_chk:w #2 #3 #4 #5 ; \s__fp \__fp_chk:w #6 #7 #8 #9 ;
13392 {
13393     \exp_after:wN \__fp_sanitize:Nw
13394     \exp_after:wN #1
13395     \int_use:N \__int_eval:w
13396     #4 + #8
13397     \__fp_mul_significand_o:nnnnNnnnn #5 #1 #9
13398 }

```

(End definition for __fp_mul_npos_o:Nww.)

```

\__fp_mul_significand_o:nnnnNnnnn \__fp_mul_significand_o:nnnnNnnnn {<X1>} {<X2>} {<X3>} {<X4>} <sign>
\__fp_mul_significand_drop:NNNNNw {<Y1>} {<Y2>} {<Y3>} {<Y4>}
\__fp_mul_significand_keep:NNNNNw

```

Note the three semicolons at the end of the definition. One is for the last __fp_mul_significand_drop:NNNNNw; one is for __fp_round_digit:Nw later on; and one,

preceded by `\exp_after:wN`, which is correctly expanded (within an `__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`.

```

13399 \cs_new:Npn \__fp_mul_significand_o:nnnnNnnnn #1#2#3#4 #5 #6#7#8#9
13400 {
13401   \exp_after:wN \__fp_mul_significand_test_f:NNN
13402   \exp_after:wN #5
13403   \int_use:N \__int_eval:w 99990000 + #1*#6 +
13404   \exp_after:wN \__fp_mul_significand_keep:NNNNNw
13405   \int_use:N \__int_eval:w 99990000 + #1*#7 + #2*#6 +
13406   \exp_after:wN \__fp_mul_significand_keep:NNNNNw
13407   \int_use:N \__int_eval:w 99990000 + #1*#8 + #2*#7 + #3*#6 +
13408   \exp_after:wN \__fp_mul_significand_drop:NNNNNw
13409   \int_use:N \__int_eval:w 99990000 + #1*#9 + #2*#8 + #3*#7 + #4*#6 +
13410   \exp_after:wN \__fp_mul_significand_drop:NNNNNw
13411   \int_use:N \__int_eval:w 99990000 + #2*#9 + #3*#8 + #4*#7 +
13412   \exp_after:wN \__fp_mul_significand_drop:NNNNNw
13413   \int_use:N \__int_eval:w 99990000 + #3*#9 + #4*#8 +
13414   \exp_after:wN \__fp_mul_significand_drop:NNNNNw
13415   \int_use:N \__int_eval:w 100000000 + #4*#9 ;
13416   ; \exp_after:wN ;
13417 }
13418 \cs_new:Npn \__fp_mul_significand_drop:NNNNNw #1#2#3#4#5 #6;
13419 { #1#2#3#4#5 ; + #6 }
13420 \cs_new:Npn \__fp_mul_significand_keep:NNNNNw #1#2#3#4#5 #6;
13421 { #1#2#3#4#5 ; #6 ; }

```

(End definition for `__fp_mul_significand_o:nnnnNnnnn`.)

```

\__fp_mul_significand_test_f:NNN \__fp_mul_significand_test_f:NNN <sign> 1 <digits 1-8> ; <digits 9-12> ;
<digits 13-16> ; + <digits 17-20> + <digits 21-24> + <digits 25-28> + <digits
29-32> ; \exp_after:wN ;

```

If the *<digit 1>* is non-zero, then for rounding we only care about the digits 16 and 17, and whether further digits are zero or not (check for exact ties). On the other hand, if *<digit 1>* is zero, we care about digits 17 and 18, and whether further digits are zero.

```

13422 \cs_new:Npn \__fp_mul_significand_test_f:NNN #1 #2 #3
13423 {
13424   \if_meaning:w 0 #3
13425   \exp_after:wN \__fp_mul_significand_small_f:NNwwwN
13426   \else:
13427   \exp_after:wN \__fp_mul_significand_large_f:NwwNNNN
13428   \fi:
13429   #1 #3
13430 }

```

(End definition for `_fp_mul_significand_test_f:NNN`.)

`_fp_mul_significand_large_f:NwwNNN`

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`.

```

13431 \cs_new:Npn \_fp_mul_significand_large_f:NwwNNN #1 #2; #3; #4#5#6#7; +
13432 {
13433   \exp_after:wN \_fp_basics_pack_high:NNNNNw
13434   \int_use:N \_int_eval:w 1#2
13435   \exp_after:wN \_fp_basics_pack_low:NNNNNw
13436   \int_use:N \_int_eval:w 1#3#4#5#6#7
13437   + \exp_after:wN \_fp_round:NNN
13438   \exp_after:wN #1
13439   \exp_after:wN #7
13440   \_int_value:w \_fp_round_digit:Nw
13441 }

```

(End definition for `_fp_mul_significand_large_f:NwwNNN`.)

`_fp_mul_significand_small_f:NNwwN`

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.

```

13442 \cs_new:Npn \_fp_mul_significand_small_f:NNwwN #1 #2#3; #4#5; #6; + #7
13443 {
13444   - \c_one
13445   \exp_after:wN \_fp_basics_pack_high:NNNNNw
13446   \int_use:N \_int_eval:w 1#3#4
13447   \exp_after:wN \_fp_basics_pack_low:NNNNNw
13448   \int_use:N \_int_eval:w 1#5#6#7
13449   + \exp_after:wN \_fp_round:NNN
13450   \exp_after:wN #1
13451   \exp_after:wN #7
13452   \_int_value:w \_fp_round_digit:Nw
13453 }

```

(End definition for `_fp_mul_significand_small_f:NNwwN`.)

27.4 Division

27.4.1 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 `- \c_two +` 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:NnNnw` are provided as the fourth argument here.

```

13454 \cs_new_nopar:cpn { __fp/_o:ww }
13455 {
13456   \__fp_mul_cases_o:NnNnw
13457   /
13458   { - }
13459   \__fp_div_npos_o:Nww
13460   {
13461     \or:
13462       \__fp_case_use:nw
13463       { \__fp_division_by_zero_o:NNnw \c_inf_fp / }
13464     \or:
13465       \__fp_case_use:nw
13466       { \__fp_division_by_zero_o:NNnw \c_minus_inf_fp / }
13467   }
13468 }

```

(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.

```

13469 \cs_new:Npn \__fp_div_npos_o:Nww
13470   #1 \s__fp \__fp_chk:w 1 #2 #3 #4 ; \s__fp \__fp_chk:w 1 #5 #6 #7#8#9;
13471 {
13472   \exp_after:wN \__fp_sanitize:Nw
13473   \exp_after:wN #1
13474   \int_use:N \__int_eval:w
13475   #3 - #6
13476   \exp_after:wN \__fp_div_significand_i_o:wnnw
13477   \int_use:N \__int_eval:w #7 \use_i:nnnn #8 + \c_one ;
13478   #4
13479   {\#7}{\#8}\#9 ;
13480   #1
13481 }

```

(End definition for `__fp_div_npos_o:Nww`.)

27.4.2 Work plan

In this subsection, we explain how to avoid overflowing $\text{T}_{\text{E}}\text{X}$'s integers when performing the division of two positive normal numbers.

We are given two numbers, $A = 0.A_1A_2A_3A_4$ and $Z = 0.Z_1Z_2Z_3Z_4$, in blocks of 4 digits, and we know that the first digits of A_1 and of Z_1 are non-zero. To compute A/Z , we proceed as follows.

- Find an integer $Q_A \simeq 10^4 A/Z$.
- Replace A by $B = 10^4 A - Q_A Z$.
- Find an integer $Q_B \simeq 10^4 B/Z$.
- Replace B by $C = 10^4 B - Q_B Z$.
- Find an integer $Q_C \simeq 10^4 C/Z$.
- Replace C by $D = 10^4 C - Q_C Z$.
- Find an integer $Q_D \simeq 10^4 D/Z$.
- Consider $E = 10^4 D - Q_D Z$, and ensure correct rounding.

The result is then $Q = 10^{-4}Q_A + 10^{-8}Q_B + 10^{-12}Q_C + 10^{-16}Q_D + \text{rounding}$. Since the Q_i are integers, B , C , D , and E are all exact multiples of 10^{-16} , in other words, computing with 16 digits after the decimal separator yields exact results. The problem 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

$$\backslash\text{int_eval:n} \left\{ \frac{A_1 A_2}{Z_1 + 1} - 1 \right\} \leq 10^4 \frac{A}{Z}$$

Subtracting 1 at the end takes care of the fact that $\varepsilon\text{-T}_{\text{E}}\text{X}$'s $\backslash_ \text{int_eval:w}$ 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 $\text{T}_{\text{E}}\text{X}$ can only handle integers less than roughly $2 \cdot 10^9$.

A better formula is to take

$$Q_A = \backslash\text{int_eval:n} \left\{ \frac{10 \cdot A_1 A_2}{\lfloor 10^{-3} \cdot Z_1 Z_2 \rfloor + 1} - 1 \right\}.$$

This is always less than $10^9 A / (10^5 Z)$, as we wanted. In words, we take the 5 first digits of Z into account, and the 8 first digits of A , using 0 as a 9-th digit rather than the true

digit for efficiency reasons. We shall prove that using this formula to define all the Q_i avoids any overflow. For convenience, let us denote

$$y = \lfloor 10^{-3} \cdot Z_1 Z_2 \rfloor + 1,$$

so that, taking into account the fact that $\varepsilon\text{-TeX}$ rounds ties away from zero,

$$\begin{aligned} Q_A &= \left\lfloor \frac{A_1 A_2 0}{y} - \frac{1}{2} \right\rfloor \\ &> \frac{A_1 A_2 0}{y} - \frac{3}{2}. \end{aligned}$$

Note that $10^4 < y \leq 10^5$, and $999 \leq Q_A \leq 99989$. Also note that this formula does not cause an overflow as long as $A < (2^{31} - 1)/10^9 \simeq 2.147 \dots$, since the numerator involves an integer slightly smaller than $10^9 A$.

Let us bound B :

$$\begin{aligned} 10^5 B &= A_1 A_2 0 + 10 \cdot 0.A_3 A_4 - 10 \cdot Z_1.Z_2 Z_3 Z_4 \cdot Q_A \\ &< A_1 A_2 0 \cdot \left(1 - 10 \cdot \frac{Z_1.Z_2 Z_3 Z_4}{y} \right) + \frac{3}{2} \cdot 10 \cdot Z_1.Z_2 Z_3 Z_4 + 10 \\ &\leq \frac{A_1 A_2 0 \cdot (y - 10 \cdot Z_1.Z_2 Z_3 Z_4)}{y} + \frac{3}{2} y + 10 \\ &\leq \frac{A_1 A_2 0 \cdot 1}{y} + \frac{3}{2} y + 10 \leq \frac{10^9 A}{y} + 1.6 \cdot y. \end{aligned}$$

At the last step, we hide 10 into the second term for later convenience. The same reasoning yields

$$\begin{aligned} 10^5 B &< 10^9 A / y + 1.6y, \\ 10^5 C &< 10^9 B / y + 1.6y, \\ 10^5 D &< 10^9 C / y + 1.6y, \\ 10^5 E &< 10^9 D / y + 1.6y. \end{aligned}$$

The goal is now to prove that none of B , C , D , and E can go beyond $(2^{31} - 1)/10^9 = 2.147 \dots$.

Combining the various inequalities together with $A < 1$, we get

$$\begin{aligned} 10^5 B &< 10^9 / y + 1.6y, \\ 10^5 C &< 10^{13} / y^2 + 1.6(y + 10^4), \\ 10^5 D &< 10^{17} / y^3 + 1.6(y + 10^4 + 10^8 / y), \\ 10^5 E &< 10^{21} / y^4 + 1.6(y + 10^4 + 10^8 / y + 10^{12} / y^2). \end{aligned}$$

All of those bounds are convex functions of y (since every power of y involved is convex, and the coefficients are positive), and thus maximal at one of the end-points of the allowed range $10^4 < y \leq 10^5$. Thus,

$$\begin{aligned} 10^5 B &< \max(1.16 \cdot 10^5, 1.7 \cdot 10^5), \\ 10^5 C &< \max(1.32 \cdot 10^5, 1.77 \cdot 10^5), \\ 10^5 D &< \max(1.48 \cdot 10^5, 1.777 \cdot 10^5), \\ 10^5 E &< \max(1.64 \cdot 10^5, 1.7777 \cdot 10^5). \end{aligned}$$

All of those bounds are less than $2.147 \cdot 10^5$, and we are thus within $\text{T}_{\text{E}}\text{X}$'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 $\varepsilon\text{-T}_{\text{E}}\text{X}$ round

$$P = \backslash\text{int_eval:n} \left\{ \frac{2 \cdot E_1 E_2}{Z_1 Z_2} \right\},$$

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 $\varepsilon\text{-T}_{\text{E}}\text{X}$'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.

27.4.3 Implementing the significand division

`_fp_div_significand_i_o:wnnw` $\langle y \rangle$; $\{\langle A_1 \rangle\} \{\langle A_2 \rangle\} \{\langle A_3 \rangle\} \{\langle A_4 \rangle\}$
 $\{\langle Z_1 \rangle\} \{\langle Z_2 \rangle\} \{\langle Z_3 \rangle\} \{\langle Z_4 \rangle\}$; $\langle sign \rangle$

Compute $10^6 + Q_A$ (a 7 digit number thanks to the shift), unbrace $\langle A_1 \rangle$ and $\langle A_2 \rangle$, and prepare the $\langle continuation \rangle$ arguments for 4 consecutive calls to `_fp_div_significand_calc:wnnnnnnnn`. Each of these calls 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.

```

13482 \cs_new:Npn \_fp_div_significand_i_o:wnnw #1 ; #2#3 #4 ;
13483 {
13484   \exp_after:wN \_fp_div_significand_test_o:w
13485   \int_use:N \_int_eval:w
13486   \exp_after:wN \_fp_div_significand_calc:wnnnnnnnn
13487   \int_use:N \_int_eval:w 999999 + #2 #3 0 / #1 ;
13488   #2 #3 ;
13489   #4
13490   { \exp_after:wN \_fp_div_significand_ii:wnn \_int_value:w #1 }
13491   { \exp_after:wN \_fp_div_significand_ii:wnn \_int_value:w #1 }
13492   { \exp_after:wN \_fp_div_significand_ii:wnn \_int_value:w #1 }
13493   { \exp_after:wN \_fp_div_significand_iii:wnnnnnn \_int_value:w #1 }
13494 }

```

(End definition for `_fp_div_significand_i_o:wnnw`.)

`_fp_div_significand_calc:wnnnnnnnn` $\langle 10^6 + Q_A \rangle$; $\langle A_1 \rangle \langle A_2 \rangle$; $\{\langle A_3 \rangle\}$
`_fp_div_significand_calc_i:wnnnnnnnn` $\{\langle A_4 \rangle\} \{\langle Z_1 \rangle\} \{\langle Z_2 \rangle\} \{\langle Z_3 \rangle\} \{\langle Z_4 \rangle\} \{\langle continuation \rangle\}$
`_fp_div_significand_calc_ii:wnnnnnnnn` expands to

$\langle 10^6 + Q_A \rangle \langle continuation \rangle$; $\langle B_1 \rangle \langle B_2 \rangle$; $\{\langle B_3 \rangle\} \{\langle B_4 \rangle\} \{\langle Z_1 \rangle\} \{\langle Z_2 \rangle\} \{\langle Z_3 \rangle\}$
 $\{\langle Z_4 \rangle\}$

where $B = 10^4 A - Q_A \cdot Z$. This function is also used to compute C , D , E (with the input shifted accordingly), and is used in `l3fp-expo`.

We know that $0 < Q_A < 1.8 \cdot 10^5$, so the product of Q_A with each Z_i is within $\text{T}_{\text{E}}\text{X}$'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 $\langle continuation \rangle$, this is not possible because the macro has 9 parameters.

The result we want is then (the overall power of 10 is arbitrary):

$$\begin{aligned}
& 10^{-4}(\#2 - \#1 \cdot \#5 - 10 \cdot \langle i \rangle \cdot \#5\#6) + 10^{-8}(\#3 - \#1 \cdot \#6 - 10 \cdot \langle i \rangle \cdot \#7) \\
& + 10^{-12}(\#4 - \#1 \cdot \#7 - 10 \cdot \langle i \rangle \cdot \#8) + 10^{-16}(-\#1 \cdot \#8),
\end{aligned}$$

where $\langle i \rangle$ stands for the 10^5 digit of Q_A , which is 0 or 1, and #1, #2, etc. are the parameters of either auxiliary. The factors of 10 come from the fact that $Q_A = 10 \cdot$

$10^4 \cdot \langle i \rangle + \#1$. As usual, to combine all the terms, we need to choose some shifts which must ensure that the number of digits of the second, third, and fourth terms are each fixed. Here, the positive contributions are at most 10^8 and the negative contributions can go up to 10^9 . Indeed, for the auxiliary with $\langle i \rangle = 1$, $\#1$ is at most 80000, leading to contributions of at worst $-8 \cdot 10^8 4$, 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 T_EX's limits once more.

```

13495 \cs_new:Npn \__fp_div_significand_calc:wwnnnnnnn #1
13496 {
13497   \if_meaning:w 1 #1
13498     \exp_after:wN \__fp_div_significand_calc_i:wwnnnnnnn
13499   \else:
13500     \exp_after:wN \__fp_div_significand_calc_ii:wwnnnnnnn
13501   \fi:
13502 }
13503 \cs_new:Npn \__fp_div_significand_calc_i:wwnnnnnnn #1; #2;#3#4 #5#6#7#8 #9
13504 {
13505   1 1 #1
13506   #9 \exp_after:wN ;
13507   \int_use:N \__int_eval:w \c__fp_Bigg_leading_shift_int
13508     + #2 - #1 * #5 - #5#60
13509   \exp_after:wN \__fp_pack_Bigg:NNNNNNw
13510   \int_use:N \__int_eval:w \c__fp_Bigg_middle_shift_int
13511     + #3 - #1 * #6 - #70
13512   \exp_after:wN \__fp_pack_Bigg:NNNNNNw
13513   \int_use:N \__int_eval:w \c__fp_Bigg_middle_shift_int
13514     + #4 - #1 * #7 - #80
13515   \exp_after:wN \__fp_pack_Bigg:NNNNNNw
13516   \int_use:N \__int_eval:w \c__fp_Bigg_trailing_shift_int
13517     - #1 * #8 ;
13518   {#5}{#6}{#7}{#8}
13519 }
13520 \cs_new:Npn \__fp_div_significand_calc_ii:wwnnnnnnn #1; #2;#3#4 #5#6#7#8 #9
13521 {
13522   1 0 #1
13523   #9 \exp_after:wN ;
13524   \int_use:N \__int_eval:w \c__fp_Bigg_leading_shift_int
13525     + #2 - #1 * #5
13526   \exp_after:wN \__fp_pack_Bigg:NNNNNNw
13527   \int_use:N \__int_eval:w \c__fp_Bigg_middle_shift_int
13528     + #3 - #1 * #6
13529   \exp_after:wN \__fp_pack_Bigg:NNNNNNw
13530   \int_use:N \__int_eval:w \c__fp_Bigg_middle_shift_int
13531     + #4 - #1 * #7
13532   \exp_after:wN \__fp_pack_Bigg:NNNNNNw
13533   \int_use:N \__int_eval:w \c__fp_Bigg_trailing_shift_int
13534     - #1 * #8 ;

```

```

13535     {#5}{#6}{#7}{#8}
13536   }

```

(End definition for `_fp_div_significand_calc:wwnnnnnnn`.)

```

\_fp_div_significand_ii:wwn    \_fp_div_significand_ii:wnn <y> ; <B1> ; {<B2>} {<B3>} {<B4>} {<Z1>}
                                {<Z2>} {<Z3>} {<Z4>} <continuations> <sign>

```

Compute Q_B by evaluating $\langle B_1 \rangle \langle B_2 \rangle 0 / y - 1$. The result 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 .

```

13537 \cs_new:Npn \_fp_div_significand_ii:wwn #1; #2;#3
13538 {
13539   \exp_after:wN \_fp_div_significand_pack:NNN
13540   \int_use:N \_int_eval:w
13541   \exp_after:wN \_fp_div_significand_calc:wwnnnnnnn
13542   \int_use:N \_int_eval:w 999999 + #2 #3 0 / #1 ; #2 #3 ;
13543 }

```

(End definition for `_fp_div_significand_ii:wwn`.)

```

\_fp_div_significand_iii:wwnnnnn  \_fp_div_significand_iii:wwnnnnn <y> ; <E1> ; {<E2>} {<E3>} {<E4>}
                                   {<Z1>} {<Z2>} {<Z3>} {<Z4>} <sign>

```

We compute $P \simeq 2E/Z$ by rounding $2E_1E_2/Z_1Z_2$. Note the first 0, which multiplies Q_D by 10: we 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 .

```

13544 \cs_new:Npn \_fp_div_significand_iii:wwnnnnn #1; #2;#3#4#5 #6#7
13545 {
13546   0
13547   \exp_after:wN \_fp_div_significand_iv:wwnnnnnnn
13548   \int_use:N \_int_eval:w (\c_two * #2 #3) / #6 #7 ; % <- P
13549   #2 ; {#3} {#4} {#5}
13550   {#6} {#7}
13551 }

```

(End definition for `_fp_div_significand_iii:wwnnnnn`.)

```

\_fp_div_significand_iv:wwnnnnnnn  \_fp_div_significand_iv:wwnnnnnnn <P> ; <E1> ; {<E2>} {<E3>} {<E4>}
\_fp_div_significand_v:NNw         {<Z1>} {<Z2>} {<Z3>} {<Z4>} <sign>
\_fp_div_significand_vi:Nw

```

This adds to the current expression $(10^7 + 10 \cdot Q_D)$ a contribution of $5 \cdot P + \text{sign}(T)$ with $T = 2E - PZ$. This amounts to adding $P/2$ to Q_D , with an extra *rounding* digit. This *rounding* digit is 0 or 5 if T does not contribute, *i.e.*, if $0 = T = 2E - PZ$, in other words if $10^{16}A/Z$ is an integer or half-integer. Otherwise it is in the appropriate range, $[1, 4]$ or $[6, 9]$. This is precise enough for rounding purposes (in any mode).

It seems an overkill to compute T exactly as I do here, but I see no faster way right now.

Once more, we need to be careful and show that the calculation `#1 · #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: $T = 0$.

```

13552 \cs_new:Npn \__fp_div_significand_iv:wnnnnnnnn #1; #2;#3#4#5 #6#7#8#9
13553 {
13554   + \c_five * #1
13555   \exp_after:wN \__fp_div_significand_vi:Nw
13556   \int_use:N \__int_eval:w -20 + 2*#2#3 - #1*#6#7 +
13557   \exp_after:wN \__fp_div_significand_v:NN
13558   \int_use:N \__int_eval:w 199980 + 2*#4 - #1*#8 +
13559   \exp_after:wN \__fp_div_significand_v:NN
13560   \int_use:N \__int_eval:w 200000 + 2*#5 - #1*#9 ;
13561 }
13562 \cs_new:Npn \__fp_div_significand_v:NN #1#2 { #1#2 \__int_eval_end: + }
13563 \cs_new:Npn \__fp_div_significand_vi:Nw #1#2;
13564 {
13565   \if_meaning:w 0 #1
13566   \if_int_compare:w \__int_eval:w #2 > \c_zero + \c_one \fi:
13567   \else:
13568   \if_meaning:w - #1 - \else: + \fi: \c_one
13569   \fi:
13570   ;
13571 }

```

(End definition for `__fp_div_significand_iv:wnnnnnnnn`, `__fp_div_significand_v:NNw`, and `__fp_div_significand_vi:Nw`.)

`__fp_div_significand_pack:NNN` At this stage, we are in the following situation: \TeX is in the process of expanding several integer expressions, thus functions at the bottom expand before those above.

```

\__fp_div_significand_test_o:w 106 + QA \__fp_div_significand_
pack:NNN 106 + QB \__fp_div_significand_pack:NNN 106 + QC \__fp_
div_significand_pack:NNN 107 + 10 · QD + 5 · P + ε ; <sign>

```

Here, $\varepsilon = \text{sign}(T)$ is 0 in case $2E = PZ$, 1 in case $2E > PZ$, which means that P was the correct value, but not with an exact quotient, and -1 if $2E < PZ$, i.e., P was an

overestimate. The packing function we define now does nothing special: it removes the 10^6 and carries two digits (for the 10^5 's and the 10^4 's).

```
13572 \cs_new:Npn \__fp_div_significand_pack:NNN 1 #1 #2 { + #1 #2 ; }
```

(End definition for `__fp_div_significand_pack:NNN`.)

```
\__fp_div_significand_test_o:w \__fp_div_significand_test_o:w 1 0 <5d> ; <4d> ; <4d> ; <5d> ; <sign>
```

The reason we know that the first two digits are 1 and 0 is that the final result is known to be between 0.1 (inclusive) and 10, hence \widetilde{Q}_A (the tilde denoting the contribution from the other Q_i) is at most 99999, and $10^6 + \widetilde{Q}_A = 10 \dots$.

It is now time to round. This depends on how many digits the final result will have.

```
13573 \cs_new:Npn \__fp_div_significand_test_o:w 10 #1
13574 {
13575   \if_meaning:w 0 #1
13576     \exp_after:wN \__fp_div_significand_small_o:wwwNNNNwN
13577   \else:
13578     \exp_after:wN \__fp_div_significand_large_o:wwwNNNNwN
13579   \fi:
13580   #1
13581 }
```

(End definition for `__fp_div_significand_test_o:w`.)

```
\__fp_div_significand_small_o:wwwNNNNwN \__fp_div_significand_small_o:wwwNNNNwN 0 <4d> ; <4d> ; <4d> ; <5d>
; <final sign>
```

Standard use of the functions `__fp_basics_pack_low:NNNNw` and `__fp_basics_pack_high:NNNNw`. We finally get to use the `<final sign>` which has been sitting there for a while.

```
13582 \cs_new:Npn \__fp_div_significand_small_o:wwwNNNNwN
13583   0 #1; #2; #3; #4#5#6#7#8; #9
13584 {
13585   \exp_after:wN \__fp_basics_pack_high:NNNNw
13586   \int_use:N \__int_eval:w 1 #1#2
13587   \exp_after:wN \__fp_basics_pack_low:NNNNw
13588   \int_use:N \__int_eval:w 1 #3#4#5#6#7
13589   + \__fp_round:NNN #9 #7 #8
13590   \exp_after:wN ;
13591 }
```

(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.

```
13592 \cs_new:Npn \__fp_div_significand_large_o:wwwNNNNwN
13593   #1; #2; #3; #4#5#6#7#8; #9
13594 {
```

```

13595     + \c_one
13596     \exp_after:wN \_fp_basics_pack_weird_high:NNNNNNNNw
13597     \int_use:N \_int_eval:w 1 #1 #2
13598     \exp_after:wN \_fp_basics_pack_weird_low:NNNNw
13599     \int_use:N \_int_eval:w 1 #3 #4 #5 #6 +
13600     \exp_after:wN \_fp_round:NNN
13601     \exp_after:wN #9
13602     \exp_after:wN #6
13603     \_int_value:w \_fp_round_digit:Nw #7 #8 ;
13604     \exp_after:wN ;
13605 }

```

(End definition for `_fp_div_significand_large_o:wwwNNNNwN.`)

27.5 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.

```

13606 \cs_new:Npn \_fp_sqrt_o:w #1 \s_fp \_fp_chk:w #2#3#4; @
13607 {
13608     \if_meaning:w 0 #2 \_fp_case_return_same_o:w \fi:
13609     \if_meaning:w 2 #3
13610         \_fp_case_use:nw { \_fp_invalid_operation_o:nw { sqrt } }
13611     \fi:
13612     \if_meaning:w 1 #2 \else: \_fp_case_return_same_o:w \fi:
13613     \_fp_sqrt_npos_o:w
13614     \s_fp \_fp_chk:w #2 #3 #4;
13615 }

```

(End definition for `_fp_sqrt_o:w.`)

```

\_fp_sqrt_npos_o:w
\_fp_sqrt_npos_auxi_o:wwwNN
\_fp_sqrt_npos_auxii_o:wwwNNNNNNN

```

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 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 $17782794 \simeq 10^{7.25}$.

```

13616 \cs_new:Npn \_fp_sqrt_npos_o:w \s_fp \_fp_chk:w 1 0 #1#2#3#4#5;
13617 {
13618     \exp_after:wN \_fp_sanitize:Nw
13619     \exp_after:wN 0
13620     \int_use:N \_int_eval:w
13621     \if_int_odd:w #1 \exp_stop_f:
13622         \exp_after:wN \_fp_sqrt_npos_auxi_o:wwwNN
13623     \fi:
13624     #1 / \c_two
13625     \_fp_sqrt_Newton_o:wwn 56234133; 0; {#2#3} {#4#5} 0
13626 }

```

```

13627 \cs_new:Npn \__fp_sqrt_npos_auxi_o:wwnnN #1 / \c_two #2; 0; #3#4#5
13628 {
13629   ( #1 + \c_one ) / \c_two
13630   \__fp_pack_eight:wNNNNNNNN
13631   \__fp_sqrt_npos_auxii_o:wNNNNNNNN
13632   ;
13633   0 #3 #4
13634 }
13635 \cs_new:Npn \__fp_sqrt_npos_auxii_o:wNNNNNNNN #1; #2#3#4#5#6#7#8#9
13636 { \__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_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

$$x' = \left\lceil \frac{x + [10^8 a_1/x]}{2} \right\rceil \geq \frac{x + 10^8 a_1/x - 1/2 + 1/(2x)}{2} \geq \sqrt{10^8 a_1} - \frac{1}{4} + \frac{1}{4x}.$$

After any step of iteration, we thus have $\delta = x - \sqrt{10^8 a_1} \geq -0.25 + 0.25 \cdot 10^{-8}$. The new difference $\delta' = x' - \sqrt{10^8 a_1}$ after one step is bounded above as

$$x' - \sqrt{10^8 a_1} \leq \frac{x + 10^8 a_1/x + 1/2}{2} + \frac{1}{2} - \sqrt{10^8 a_1} \leq \frac{\delta}{2} \frac{\delta}{\sqrt{10^8 a_1} + \delta} + \frac{3}{4}.$$

For $\delta > 3/2$, this last expression is $\leq \delta/2 + 3/4 < \delta$, hence δ decreases at each step: since all x are integers, δ must reach a value $-1/4 < \delta \leq 3/2$. In this range of values, we get $\delta' \leq \frac{3}{4} \frac{3}{2\sqrt{10^8 a_1}} + \frac{3}{4} \leq 0.75 + 1.125 \cdot 10^{-7}$. We deduce that the difference $\delta = x - \sqrt{10^8 a_1}$ eventually reaches a value in the interval $[-0.25 + 0.25 \cdot 10^{-8}, 0.75 + 11.25 \cdot 10^{-8}]$, whose width is $1 + 11 \cdot 10^{-8}$. The corresponding interval for x may contain two integers, hence x might oscillate between those two values.

However, the fact that $x \mapsto x - 1$ and $x - 1 \mapsto x$ puts stronger constraints, which are not compatible: the first implies

$$x + [10^8 a_1/x] \leq 2x - 2$$

hence $10^8 a_1/x \leq x - 3/2$, while the second implies

$$x - 1 + [10^8 a_1/(x-1)] \geq 2x - 1$$

hence $10^8 a_1/(x-1) \geq x - 1/2$. Combining the two inequalities yields $x^2 - 3x/2 \geq 10^8 a_1 \geq x - 3x/2 + 1/2$, which cannot hold. Therefore, the iteration always converges to a single

integer x . To stop the iteration when two consecutive results are equal, the function `_fp_sqrt_Newton_o:wnn` receives the newly computed result as `#1`, the previous result as `#2`, and a_1 as `#3`. Note that ε -TeX combines the computation of a multiplication and a following division, thus avoiding overflow in `#3 * 100000000 / #1`. In any case, the result is within $[10^7, 10^8]$.

```

13637 \cs_new:Npn \_fp_sqrt_Newton_o:wnn #1; #2; #3
13638 {
13639   \if_int_compare:w #1 = #2 \exp_stop_f:
13640     \exp_after:wN \_fp_sqrt_auxi_o:NNNNwnnN
13641     \int_use:N \_int_eval:w 9999 9999 +
13642     \exp_after:wN \_fp_use_none_until_s:w
13643   \fi:
13644   \exp_after:wN \_fp_sqrt_Newton_o:wnn
13645   \int_use:N \_int_eval:w (#1 + #3 * 1 0000 0000 / #1) / \c_two ;
13646   #1; {#3}
13647 }

```

(End definition for `_fp_sqrt_Newton_o:wnn`.)

`_fp_sqrt_auxi_o:NNNNwnnN` This function is followed by $10^8 + x - 1$, which has 9 digits starting with 1, then ; $\{ \langle a_1 \rangle \} \{ \langle a_2 \rangle \} \langle a' \rangle$. Here, $x \simeq \sqrt{10^8 a_1}$ and we want to estimate the square root of $a = 10^{-8}a_1 + 10^{-16}a_2 + 10^{-17}a'$. We set up an initial underestimate

$$y = (x - 1)10^{-8} + 0.2499998875 \cdot 10^{-8} \lesssim \sqrt{a}.$$

From the inequalities shown earlier, we know that $y \leq \sqrt{10^{-8}a_1} \leq \sqrt{a}$ and that $\sqrt{10^{-8}a_1} \leq y + 10^{-8} + 11 \cdot 10^{-16}$ hence (using $0.1 \leq y \leq \sqrt{a} \leq 1$)

$$a - y^2 \leq 10^{-8}a_1 + 10^{-8} - y^2 \leq (y + 10^{-8} + 11 \cdot 10^{-16})^2 - y^2 + 10^{-8} < 3.2 \cdot 10^{-8},$$

and $\sqrt{a} - y = (a - y^2)/(\sqrt{a} + y) \leq 16 \cdot 10^{-8}$. Next, `_fp_sqrt_auxii_o:NnnnnnnnnN` 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, $y < 1$.

```

13648 \cs_new:Npn \_fp_sqrt_auxi_o:NNNNwnnN 1 #1#2#3#4#5;
13649 {
13650   \_fp_sqrt_auxii_o:NnnnnnnnnN
13651   \_fp_sqrt_auxiii_o:wnnnnnnnnn
13652   {#1#2#3#4} {#5} {2499} {9988} {7500}
13653 }

```

(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)

$$10^{4j}z = \left[([10^{4j}(a - y^2)] - 257) \cdot (0.5 \cdot 10^8) / [10^8 y + 1] \right].$$

The choice of j ensures that $10^{4j}z < 2 \cdot 10^8 \cdot 0.5 \cdot 10^8/10^7 = 10^9$, thus $10^9 + 10^{4j}z$ has exactly 10 digits, does not overflow $\text{T}_{\text{E}}\text{X}$'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 $\varepsilon\text{-T}_{\text{E}}\text{X}$'s integer division obeys $\lfloor b/c \rfloor \leq b/c + 1/2$, we deduce that $10^{4j}z \leq 10^{4j}(\sqrt{a} - y)$, hence $y + z \leq \sqrt{a}$ is an underestimate of \sqrt{a} , as claimed. One implementation detail: because the computation involves $\text{-}\#4\text{*}\#4 - 2\text{*}\#3\text{*}\#5 - 2\text{*}\#2\text{*}\#6$ which may be as low as $-5 \cdot 10^8$, we need to use the `pack_big` functions, and the big shifts.

```

13654 \cs_new:Npn \__fp_sqrt_auxii_o:NnnnnnnnN #1 #2#3#4#5#6 #7#8#9
13655 {
13656   \exp_after:wN #1
13657   \int_use:N \__int_eval:w \c__fp_big_leading_shift_int
13658     + #7 - #2 * #2
13659   \exp_after:wN \__fp_pack_big:NNNNNNw
13660   \int_use:N \__int_eval:w \c__fp_big_middle_shift_int
13661     - 2 * #2 * #3
13662   \exp_after:wN \__fp_pack_big:NNNNNNw
13663   \int_use:N \__int_eval:w \c__fp_big_middle_shift_int
13664     + #8 - #3 * #3 - 2 * #2 * #4
13665   \exp_after:wN \__fp_pack_big:NNNNNNw
13666   \int_use:N \__int_eval:w \c__fp_big_middle_shift_int
13667     - 2 * #3 * #4 - 2 * #2 * #5
13668   \exp_after:wN \__fp_pack_big:NNNNNNw
13669   \int_use:N \__int_eval:w \c__fp_big_middle_shift_int
13670     + #9 000 0000 - #4 * #4 - 2 * #3 * #5 - 2 * #2 * #6
13671   \exp_after:wN \__fp_pack_big:NNNNNNw
13672   \int_use:N \__int_eval:w \c__fp_big_middle_shift_int
13673     - 2 * #4 * #5 - 2 * #3 * #6
13674   \exp_after:wN \__fp_pack_big:NNNNNNw
13675   \int_use:N \__int_eval:w \c__fp_big_middle_shift_int
13676     - #5 * #5 - 2 * #4 * #6
13677   \exp_after:wN \__fp_pack_big:NNNNNNw
13678   \int_use:N \__int_eval:w
13679     \c__fp_big_middle_shift_int
13680     - 2 * #5 * #6
13681   \exp_after:wN \__fp_pack_big:NNNNNNw
13682   \int_use:N \__int_eval:w
13683     \c__fp_big_trailing_shift_int

```

```

13684         - #6 * #6 ;
13685     % (
13686     - 257 ) * 5000 0000 / (#2#3 + 1) + 10 0000 0000 ;
13687     {#2}{#3}{#4}{#5}{#6} {#7}{#8}#9
13688 }

```

(End definition for _fp_sqrt_auxii_o:NnnnnnnnN.)

```

\_fp_sqrt_auxiii_o:wnnnnnnnn
\_fp_sqrt_auxiv_o:NNNNNw
\_fp_sqrt_auxv_o:NNNNNw
\_fp_sqrt_auxvi_o:NNNNNw
\_fp_sqrt_auxvii_o:NNNNNw

```

We receive here the difference $a - y^2 = d = \sum_i d_i \cdot 10^{-4i}$, as $\langle d_2 \rangle$; $\{\langle d_3 \rangle\} \dots \{\langle d_{10} \rangle\}$, where each block has 4 digits, except $\langle d_2 \rangle$. This function finds the largest $j \leq 6$ such that $10^{4j}(a - y^2) < 2 \cdot 10^8$, then leaves an open parenthesis and the integer $\lfloor 10^{4j}(a - y^2) \rfloor$ in an integer expression. The closing parenthesis is provided by the caller _fp_sqrt_auxii_o:NnnnnnnnN, which completes the expression

$$10^{4j}z = \left[(\lfloor 10^{4j}(a - y^2) \rfloor - 257) \cdot (0.5 \cdot 10^8) / \lfloor 10^8 y + 1 \rfloor \right]$$

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 $10^4 d_2 + d_3 \geq 2$, $j = 4$ 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

$$10^{4j}z \geq \frac{(20000 - 257)(0.5 \cdot 10^8)}{\lfloor 10^8 y + 1 \rfloor} \geq (20000 - 257) \cdot 0.5 > 0.$$

```

13689 \cs_new:Npn \_fp_sqrt_auxiii_o:wnnnnnnnn
13690 #1; #2#3#4#5#6#7#8#9
13691 {
13692   \if_int_compare:w #1 > \c_one
13693     \exp_after:wN \_fp_sqrt_auxiv_o:NNNNNw
13694     \int_use:N \_int_eval:w (#1#2 %)
13695   \else:
13696     \if_int_compare:w #1#2 > \c_one
13697       \exp_after:wN \_fp_sqrt_auxv_o:NNNNNw
13698       \int_use:N \_int_eval:w (#1#2#3 %)
13699     \else:
13700       \if_int_compare:w #1#2#3 > \c_one
13701         \exp_after:wN \_fp_sqrt_auxvi_o:NNNNNw
13702         \int_use:N \_int_eval:w (#1#2#3#4 %)
13703       \else:
13704         \exp_after:wN \_fp_sqrt_auxvii_o:NNNNNw
13705         \int_use:N \_int_eval:w (#1#2#3#4#5 %)
13706       \fi:
13707     \fi:
13708   \fi:

```

```

13709   }
13710   \cs_new:Npn \__fp_sqrt_auxiv_o:NNNNNw 1#1#2#3#4#5#6;
13711   { \__fp_sqrt_auxviii_o:nnnnnnnn {#1#2#3#4#5#6} {00000000} }
13712   \cs_new:Npn \__fp_sqrt_auxv_o:NNNNNw 1#1#2#3#4#5#6;
13713   { \__fp_sqrt_auxviii_o:nnnnnnnn {000#1#2#3#4#5} {#60000} }
13714   \cs_new:Npn \__fp_sqrt_auxvi_o:NNNNNw 1#1#2#3#4#5#6;
13715   { \__fp_sqrt_auxviii_o:nnnnnnnn {0000000#1} {#2#3#4#5#6} }
13716   \cs_new:Npn \__fp_sqrt_auxvii_o:NNNNNw 1#1#2#3#4#5#6;
13717   {
13718     \if_int_compare:w #1#2 = \c_zero
13719       \exp_after:wN \__fp_sqrt_auxx_o:Nnnnnnnnn
13720     \fi:
13721     \__fp_sqrt_auxviii_o:nnnnnnnn {00000000} {000#1#2#3#4#5}
13722   }

```

(End definition for `__fp_sqrt_auxiii_o:wnnnnnnnnn` and others.)

`__fp_sqrt_auxviii_o:nnnnnnnn`
`__fp_sqrt_auxix_o:wnwnw`

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 $y'^2 = (y + z)^2$.

```

13723   \cs_new:Npn \__fp_sqrt_auxviii_o:nnnnnnnn #1#2 #3#4#5#6#7
13724   {
13725     \exp_after:wN \__fp_sqrt_auxix_o:wnwnw
13726     \int_use:N \__int_eval:w #3
13727     \exp_after:wN \__fp_basics_pack_low:NNNNNw
13728     \int_use:N \__int_eval:w #1 + 1#4#5
13729     \exp_after:wN \__fp_basics_pack_low:NNNNNw
13730     \int_use:N \__int_eval:w #2 + 1#6#7 ;
13731   }
13732   \cs_new:Npn \__fp_sqrt_auxix_o:wnwnw #1; #2#3; #4#5;
13733   {
13734     \__fp_sqrt_auxii_o:NnnnnnnnnN
13735     \__fp_sqrt_auxiii_o:wnnnnnnnnn {#1}{#2}{#3}{#4}{#5}
13736   }

```

(End definition for `__fp_sqrt_auxviii_o:nnnnnnnn` and `__fp_sqrt_auxix_o:wnwnw`.)

`__fp_sqrt_auxx_o:Nnnnnnnnn`
`__fp_sqrt_auxxi_o:wnnnN`

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 $a \leq 1 - 10^{-16}$.

```

13737 \cs_new:Npn \__fp_sqrt_auxx_o:Nnnnnnnn #1#2#3 #4#5#6#7#8
13738 {
13739   \exp_after:wN \__fp_sqrt_auxxi_o:wwnnN
13740   \int_use:N \__int_eval:w
13741     (#8 + 2499) / 5000 * 5000 ;
13742   {#4} {#5} {#6} {#7} ;
13743 }
13744 \cs_new:Npn \__fp_sqrt_auxxi_o:wwnnN #1; #2; #3#4#5
13745 {
13746   \__fp_sqrt_auxii_o:NnnnnnnnN
13747   \__fp_sqrt_auxxii_o:nnnnnnnnw
13748   #2 {#1}
13749   {#3} { #4 + \c_one } #5
13750 }

```

(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.

```

13751 \cs_new:Npn \__fp_sqrt_auxxii_o:nnnnnnnnw 0; #1#2#3#4#5#6#7#8 #9;
13752 {
13753   \if_int_compare:w #1#2 > \c_zero
13754     \if_int_compare:w #1#2 = \c_one
13755       \if_int_compare:w #3#4 = \c_zero
13756         \if_int_compare:w #5#6 = \c_zero
13757           \if_int_compare:w #7#8 = \c_zero
13758             \__fp_sqrt_auxxiii_o:w
13759           \fi:
13760         \fi:
13761       \fi:
13762     \fi:
13763     \exp_after:wN \__fp_sqrt_auxxiv_o:wnnnnnnnN
13764     \__int_value:w 9998
13765   \else:
13766     \exp_after:wN \__fp_sqrt_auxxiv_o:wnnnnnnnN
13767     \__int_value:w 10000
13768   \fi:
13769   ;
13770 }

```

```

13771 \cs_new:Npn \__fp_sqrt_auxxiii_o:w \fi: \fi: \fi: \fi: #1 \fi: ;
13772 {
13773   \fi: \fi: \fi: \fi: \fi:
13774   \__fp_sqrt_auxxiv_o:wnnnnnnnnN 9999 ;
13775 }

```

(End definition for __fp_sqrt_auxxii_o:nnnnnnnnw and __fp_sqrt_auxxiii_o:w.)

__fp_sqrt_auxxiv_o:wnnnnnnnnN

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.

```

13776 \cs_new:Npn \__fp_sqrt_auxxiv_o:wnnnnnnnnN #1; #2#3#4#5#6 #7#8#9
13777 {
13778   \exp_after:wN \__fp_basics_pack_high:NNNNNw
13779   \int_use:N \__int_eval:w 1 0000 0000 + #2#3
13780   \exp_after:wN \__fp_basics_pack_low:NNNNNw
13781   \int_use:N \__int_eval:w 1 0000 0000
13782   + #4#5
13783   \if_int_compare:w #6 > #1 \exp_stop_f: + \c_one \fi:
13784   + \exp_after:wN \__fp_round:NNN
13785   \exp_after:wN 0
13786   \exp_after:wN 0
13787   \__int_value:w
13788   \exp_after:wN \use_i:nn
13789   \exp_after:wN \__fp_round_digit:Nw
13790   \int_use:N \__int_eval:w #6 + 19999 - #1 ;
13791   \exp_after:wN ;
13792 }

```

(End definition for __fp_sqrt_auxxiv_o:wnnnnnnnnN.)

27.6 Setting the sign

__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.

```

13793 \cs_new:Npn \__fp_set_sign_o:w #1 \s__fp \__fp_chk:w #2#3#4; @

```

```

13794 {
13795   \exp_after:wN \_fp_exp_after_o:w
13796   \exp_after:wN \s__fp
13797   \exp_after:wN \_fp_chk:w
13798   \exp_after:wN #2
13799   \_int_value:w
13800   \if_case:w #3 \exp_stop_f: #1 \or: 1 \or: 0 \fi: \exp_stop_f:
13801   #4;
13802 }

```

(End definition for _fp_set_sign_o:w.)

```

13803 </initex | package>

```

28 l3fp-extended implementation

```

13804 <*initex | package>

```

```

13805 <@@=fp>

```

28.1 Description of fixed point numbers

This module provides a few functions to manipulate positive floating point numbers with extended precision (24 digits), but mostly provides functions for fixed-point numbers with this precision (24 digits). Those are used in the computation of Taylor series for the logarithm, exponential, and trigonometric functions. Since we eventually only care about the 16 first digits of the final result, some of the calculations are not performed with the full 24-digit precision. In other words, the last two blocks of each fixed point number may be wrong as long as the error is small enough to be rounded away when converting back to a floating point number. The fixed point numbers are expressed as

$$\{\langle a_1 \rangle\} \{\langle a_2 \rangle\} \{\langle a_3 \rangle\} \{\langle a_4 \rangle\} \{\langle a_5 \rangle\} \{\langle a_6 \rangle\} ;$$

where each $\langle a_i \rangle$ is exactly 4 digits (ranging from 0000 to 9999), except $\langle a_1 \rangle$, which may be any “not-too-large” non-negative integer, with or without leading zeros. Here, “not-too-large” depends on the specific function (see the corresponding comments for details). Checking for overflow is the responsibility of the code calling those functions. The fixed point number a corresponding to the representation above is $a = \sum_{i=1}^6 \langle a_i \rangle \cdot 10^{-4i}$.

Most functions we define here have the form They perform the $\langle calculation \rangle$ on the two $\langle operands \rangle$, then feed the result (6 brace groups followed by a semicolon) to the $\langle continuation \rangle$, responsible for the next step of the calculation. Some functions only accept an N-type $\langle continuation \rangle$. This allows constructions such as

```

\_fp_fixed_add:wwn \langle X_1 \rangle ; \langle X_2 \rangle ;
\_fp_fixed_mul:wwn \langle X_3 \rangle ;
\_fp_fixed_add:wwn \langle X_4 \rangle ;

```

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: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.

28.2 Helpers for numbers with extended precision

`\c__fp_one_fixed_tl` The fixed-point number 1, used in `l3fp-expo`.

```
13806 \tl_const:Nn \c__fp_one_fixed_tl
13807 { {10000} {0000} {0000} {0000} {0000} {0000} }
```

(End definition for `\c__fp_one_fixed_tl`.)

`__fp_fixed_continue:wn` This function does nothing. Of course, there is no bound on a_1 (except T_EX's own $2^{31} - 1$).

```
13808 \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 *continuation*. This requires $a_1 \leq 2^{31} - 10001$.

```
13809 \cs_new:Npn \__fp_fixed_add_one:wN #1#2; #3
13810 {
13811   \exp_after:wN #3 \exp_after:wN
13812   { \int_use:N \__int_eval:w \c_ten_thousand + #1 } #2 ;
13813 }
```

(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 $[0, 5 \cdot 10^8 - 1]$.

```
13814 \cs_new:Npn \__fp_fixed_div_myriad:wn #1#2#3#4#5#6;
13815 {
13816   \exp_after:wN \__fp_fixed_mul_after:wnn
13817   \int_use:N \__int_eval:w \c__fp_leading_shift_int
13818   \exp_after:wN \__fp_pack:NNNNNw
13819   \int_use:N \__int_eval:w \c__fp_trailing_shift_int
13820   + #1 ; {#2}{#3}{#4}{#5};
13821 }
```

(End definition for `__fp_fixed_div_myriad:wn`.)

`__fp_fixed_mul_after:wnn` The fixed point operations which involve multiplication end by calling this auxiliary. It braces the last block of digits, and places the *continuation* `#2` in front. The *continuation* was brought up through the expansions by the packing functions.

```
13822 \cs_new:Npn \__fp_fixed_mul_after:wnn #1; #2; #3 { #3 {#1} #2; }
```

(End definition for `__fp_fixed_mul_after:wnn`.)

28.3 Multiplying a fixed point number by a short one

`_fp_fixed_mul_short:wnn` Computes the product $c = ab$ of $a = \sum_i \langle a_i \rangle 10^{-4i}$ and $b = \sum_i \langle b_i \rangle 10^{-4i}$, rounds it to the closest multiple of 10^{-24} , and leaves $\langle continuation \rangle \{ \langle c_1 \rangle \} \dots \{ \langle c_6 \rangle \}$; in the input stream, where each of the $\langle c_i \rangle$ are blocks of 4 digits, except $\langle c_1 \rangle$, which is any TeX integer. Note that indices for $\langle b \rangle$ start at 0: a second operand of $\{0001\}\{0000\}\{0000\}$ will leave the first operand unchanged (rather than dividing it by 10^4 , as `_fp_fixed_mul:wnn` would).

```

13823 \cs_new:Npn \_fp_fixed_mul_short:wnn #1#2#3#4#5#6; #7#8#9;
13824 {
13825   \exp_after:wN \_fp_fixed_mul_after:wnn
13826   \int_use:N \_int_eval:w \c\_fp_leading_shift_int
13827   + #1*#7
13828   \exp_after:wN \_fp_pack:NNNNNw
13829   \int_use:N \_int_eval:w \c\_fp_middle_shift_int
13830   + #1*#8 + #2*#7
13831   \exp_after:wN \_fp_pack:NNNNNw
13832   \int_use:N \_int_eval:w \c\_fp_middle_shift_int
13833   + #1*#9 + #2*#8 + #3*#7
13834   \exp_after:wN \_fp_pack:NNNNNw
13835   \int_use:N \_int_eval:w \c\_fp_middle_shift_int
13836   + #2*#9 + #3*#8 + #4*#7
13837   \exp_after:wN \_fp_pack:NNNNNw
13838   \int_use:N \_int_eval:w \c\_fp_middle_shift_int
13839   + #3*#9 + #4*#8 + #5*#7
13840   \exp_after:wN \_fp_pack:NNNNNw
13841   \int_use:N \_int_eval:w \c\_fp_trailing_shift_int
13842   + #4*#9 + #5*#8 + #6*#7
13843   + ( #5*#9 + #6*#8 + #6*#9 / \c_ten_thousand )
13844   / \c_ten_thousand ; ;
13845 }
```

(End definition for `_fp_fixed_mul_short:wnn`.)

28.4 Dividing a fixed point number by a small integer

`_fp_fixed_div_int:wnN` Divides the fixed point number $\langle a \rangle$ by the (small) integer $0 < \langle n \rangle < 10^4$ and feeds the result to the $\langle continuation \rangle$. There is no bound on a_1 .

`_fp_fixed_div_int:wnN` 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 .

`_fp_fixed_div_int_auxi:wnn` 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 \langle continuation \rangle
-1 + Q_1
```

```

\__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 $\langle continuation \rangle$ as appropriate.

```

13846 \cs_new:Npn \__fp_fixed_div_int:wnN #1#2#3#4#5#6 ; #7 ; #8
13847 {
13848   \exp_after:wN \__fp_fixed_div_int_after:Nw
13849   \exp_after:wN #8
13850   \int_use:N \__int_eval:w \c_minus_one
13851   \__fp_fixed_div_int:wnN
13852   #1; {#7} \__fp_fixed_div_int_auxi:wnn
13853   #2; {#7} \__fp_fixed_div_int_auxi:wnn
13854   #3; {#7} \__fp_fixed_div_int_auxi:wnn
13855   #4; {#7} \__fp_fixed_div_int_auxi:wnn
13856   #5; {#7} \__fp_fixed_div_int_auxi:wnn
13857   #6; {#7} \__fp_fixed_div_int_auxii:wnn ;
13858 }
13859 \cs_new:Npn \__fp_fixed_div_int:wnN #1; #2 #3
13860 {
13861   \exp_after:wN #3
13862   \int_use:N \__int_eval:w #1 / #2 - \c_one ;
13863   {#2}
13864   {#1}
13865 }
13866 \cs_new:Npn \__fp_fixed_div_int_auxi:wnn #1; #2 #3
13867 {
13868   + #1
13869   \exp_after:wN \__fp_fixed_div_int_pack:Nw
13870   \int_use:N \__int_eval:w 9999
13871   \exp_after:wN \__fp_fixed_div_int:wnN
13872   \int_use:N \__int_eval:w #3 - #1*#2 \__int_eval_end:
13873 }
13874 \cs_new:Npn \__fp_fixed_div_int_auxii:wnn #1; #2 #3 { + #1 + \c_two ; }
13875 \cs_new:Npn \__fp_fixed_div_int_pack:Nw #1 #2; { + #1; {#2} }
13876 \cs_new:Npn \__fp_fixed_div_int_after:Nw #1 #2; { #1 {#2} }

```

(End definition for `__fp_fixed_div_int:wnN`.)

28.5 Adding and subtracting fixed points

`__fp_fixed_add:wwn` Computes $a + b$ (resp. $a - b$) and feeds the result to the $\langle continuation \rangle$. This function requires $0 \leq a_1, b_1 \leq 114748$, its result must be positive (this happens automatically for addition) and its first group must have at most 5 digits: $(a \pm b)_1 < 100000$. The two functions only differ by a sign, hence use a common auxiliary. It would be nice to grab the 12 brace groups in one go; only 9 parameters are allowed. Start by grabbing the sign, a_1, \dots, a_4 , the rest of a , and b_1 and b_2 . The second auxiliary receives the rest of a , the sign multiplying b , the rest of b , and the $\langle continuation \rangle$ as arguments. After going down through the various level, we go back up, packing digits and bringing the $\langle continuation \rangle$ (#8, then #7) from the end of the argument list to its start.

```

13877 \cs_new_nopar:Npn \__fp_fixed_add:wwn { \__fp_fixed_add:Nnnnnwnn + }
13878 \cs_new_nopar:Npn \__fp_fixed_sub:wwn { \__fp_fixed_add:Nnnnnwnn - }
13879 \cs_new:Npn \__fp_fixed_add:Nnnnnwnn #1 #2#3#4#5 #6; #7#8
13880 {
13881   \exp_after:wN \__fp_fixed_add_after:NNNNNwn
13882   \int_use:N \__int_eval:w 9 9999 9998 + #2#3 #1 #7#8
13883   \exp_after:wN \__fp_fixed_add_pack:NNNNNwn
13884   \int_use:N \__int_eval:w 1 9999 9998 + #4#5
13885   \__fp_fixed_add:nnNnnnwn #6 #1
13886 }
13887 \cs_new:Npn \__fp_fixed_add:nnNnnnwn #1#2 #3 #4#5 #6#7 ; #8
13888 {
13889   #3 #4#5
13890   \exp_after:wN \__fp_fixed_add_pack:NNNNNwn
13891   \int_use:N \__int_eval:w 2 0000 0000 #3 #6#7 + #1#2 ; {#8} ;
13892 }
13893 \cs_new:Npn \__fp_fixed_add_pack:NNNNNwn #1 #2#3#4#5 #6; #7
13894 { + #1 ; {#7} {#2#3#4#5} {#6} }
13895 \cs_new:Npn \__fp_fixed_add_after:NNNNNwn 1 #1 #2#3#4#5 #6; #7
13896 { #7 {#1#2#3#4#5} {#6} }
  
```

(End definition for `__fp_fixed_add:wwn` and `__fp_fixed_sub:wwn`.)

28.6 Multiplying fixed points

`__fp_fixed_mul:wwn` Computes $a \times b$ and feeds the result to $\langle continuation \rangle$. This function requires $0 \leq a_1, b_1 < 10000$. Once more, we need to play around the limit of 9 arguments for \TeX macros. Note that we don't need to obtain an exact rounding, contrarily to the `*` operator, so

things could be harder. We wish to perform carries in

$$\begin{aligned}
a \times b = & a_1 \cdot b_1 \cdot 10^{-8} \\
& + (a_1 \cdot b_2 + a_2 \cdot b_1) \cdot 10^{-12} \\
& + (a_1 \cdot b_3 + a_2 \cdot b_2 + a_3 \cdot b_1) \cdot 10^{-16} \\
& + (a_1 \cdot b_4 + a_2 \cdot b_3 + a_3 \cdot b_2 + a_4 \cdot b_1) \cdot 10^{-20} \\
& + \left(a_2 \cdot b_4 + a_3 \cdot b_3 + a_4 \cdot b_2 \right. \\
& \quad \left. + \frac{a_3 \cdot b_4 + a_4 \cdot b_3 + a_1 \cdot b_6 + a_2 \cdot b_5 + a_5 \cdot b_2 + a_6 \cdot b_1}{10^4} \right. \\
& \quad \left. + a_1 \cdot b_5 + a_5 \cdot b_1 \right) \cdot 10^{-24} + O(10^{-24}),
\end{aligned}$$

where the $O(10^{-24})$ stands for terms which are at most $5 \cdot 10^{-24}$; ignoring those leads to an error of at most 5 ulp. Note how the first 15 terms only depend on a_1, \dots, a_4 and b_1, \dots, b_4 , while the last 6 terms only depend on a_1, a_2, a_5, a_6 , and the corresponding parts of b . Hence, the first function grabs a_1, \dots, a_4 , the rest of a , and b_1, \dots, b_4 , and writes the 15 first terms of the expression, including a left parenthesis for the fraction. The `i` auxiliary receives $a_5, a_6, b_1, b_2, a_1, a_2, b_5, b_6$ and finally the $\langle continuation \rangle$ as arguments. It writes the end of the expression, including the right parenthesis and the denominator of the fraction. The $\langle continuation \rangle$ is finally placed in front of the 6 brace groups by `_fp_fixed_mul_after:wwn`.

```

13897 \cs_new:Npn \_fp\_fixed\_mul:wwn #1#2#3#4 #5; #6#7#8#9
13898 {
13899   \exp\_after:wN \_fp\_fixed\_mul\_after:wwn
13900   \int\_use:N \_int\_eval:w \c\_fp\_leading\_shift\_int
13901   \exp\_after:wN \_fp\_pack:NNNNNw
13902   \int\_use:N \_int\_eval:w \c\_fp\_middle\_shift\_int
13903   + #1*#6
13904   \exp\_after:wN \_fp\_pack:NNNNNw
13905   \int\_use:N \_int\_eval:w \c\_fp\_middle\_shift\_int
13906   + #1*#7 + #2*#6
13907   \exp\_after:wN \_fp\_pack:NNNNNw
13908   \int\_use:N \_int\_eval:w \c\_fp\_middle\_shift\_int
13909   + #1*#8 + #2*#7 + #3*#6
13910   \exp\_after:wN \_fp\_pack:NNNNNw
13911   \int\_use:N \_int\_eval:w \c\_fp\_middle\_shift\_int
13912   + #1*#9 + #2*#8 + #3*#7 + #4*#6
13913   \exp\_after:wN \_fp\_pack:NNNNNw
13914   \int\_use:N \_int\_eval:w \c\_fp\_trailing\_shift\_int
13915   + #2*#9 + #3*#8 + #4*#7
13916   + ( #3*#9 + #4*#8
13917     + \_fp\_fixed\_mul:nnnnnnnw #5 {#6}{#7} {#1}{#2}
13918   )
13919 \cs\_new:Npn \_fp\_fixed\_mul:nnnnnnnw #1#2 #3#4 #5#6 #7#8 ;
13920 {
13921   #1*#4 + #2*#3 + #5*#8 + #6*#7 ) / \c\_ten\_thousand

```

```

13922     + #1*#3 + #5*#7 ; ;
13923 }

```

(End definition for _fp_fixed_mul:wnn.)

28.7 Combining product and sum of fixed points

_fp_fixed_mul_add:wnn Compute $a \times b + c$, $c - a \times b$, and $1 - a \times b$ and feed the result to the $\langle continuation \rangle$.
 _fp_fixed_mul_sub_back:wnn Those functions require $0 \leq a_1, b_1, c_1 \leq 10000$. Since those functions are at the heart of
 _fp_fixed_mul_one_minus_mul:wnn 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 $+ c_5 c_6$; $\{\langle continuation \rangle\}$; $;$. The $+ c_5 c_6$ piece, which is omitted for `_fp_fixed_one_minus_mul:wnn`, will be taken in the integer expression for the 10^{-24} level.

```

13924 \cs_new:Npn \_fp_fixed_mul_add:wnn #1; #2; #3#4#5#6#7#8;
13925 {
13926   \exp_after:wN \_fp_fixed_mul_after:wnn
13927   \int_use:N \_int_eval:w \c\_fp_big_leading_shift_int
13928   \exp_after:wN \_fp_pack_big:NNNNNNw
13929   \int_use:N \_int_eval:w \c\_fp_big_middle_shift_int + #3 #4
13930   \_fp_fixed_mul_add:Nwnnnwnnn +
13931     + #5 #6 ; #2 ; #1 ; #2 ; +
13932     + #7 #8 ; ;
13933 }
13934 \cs_new:Npn \_fp_fixed_mul_sub_back:wnn #1; #2; #3#4#5#6#7#8;
13935 {
13936   \exp_after:wN \_fp_fixed_mul_after:wnn
13937   \int_use:N \_int_eval:w \c\_fp_big_leading_shift_int
13938   \exp_after:wN \_fp_pack_big:NNNNNNw

```

```

13939      \int_use:N \__int_eval:w \c__fp_big_middle_shift_int + #3 #4
13940      \__fp_fixed_mul_add:Nwnnnwnnn -
13941      + #5 #6 ; #2 ; #1 ; #2 ; -
13942      + #7 #8 ; ;
13943    }
13944    \cs_new:Npn \__fp_fixed_one_minus_mul:wnn #1; #2;
13945    {
13946      \exp_after:wN \__fp_fixed_mul_after:wnn
13947      \int_use:N \__int_eval:w \c__fp_big_leading_shift_int
13948      \exp_after:wN \__fp_pack_big:NNNNNNw
13949      \int_use:N \__int_eval:w \c__fp_big_middle_shift_int + 1 0000 0000
13950      \__fp_fixed_mul_add:Nwnnnwnnn -
13951      ; #2 ; #1 ; #2 ; -
13952      ; ;
13953    }

```

(End definition for `__fp_fixed_mul_add:wnnn`, `__fp_fixed_mul_sub_back:wnnn`, and `__fp_fixed_mul_one_minus_mul:wnn`.)

`__fp_fixed_mul_add:Nwnnnwnnn`

Here, $\langle op \rangle$ is either + or -. Arguments #3, #4, #5 are $\langle b_1 \rangle$, $\langle b_2 \rangle$, $\langle b_3 \rangle$; arguments #7, #8, #9 are $\langle a_1 \rangle$, $\langle a_2 \rangle$, $\langle a_3 \rangle$. We can build three levels: $a_1 \cdot b_1$ for 10^{-8} , $(a_1 \cdot b_2 + a_2 \cdot b_1)$ for 10^{-12} , and $(a_1 \cdot b_3 + a_2 \cdot b_2 + a_3 \cdot b_1 + c_3 c_4)$ for 10^{-16} . The $a-b$ products use the sign #1. Note that #2 is empty for `__fp_fixed_one_minus_mul:wnn`. 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.

```

13954    \cs_new:Npn \__fp_fixed_mul_add:Nwnnnwnnn #1 #2; #3#4#5#6; #7#8#9
13955    {
13956      #1 #7*#3
13957      \exp_after:wN \__fp_pack_big:NNNNNNw
13958      \int_use:N \__int_eval:w \c__fp_big_middle_shift_int
13959      #1 #7*#4 #1 #8*#3
13960      \exp_after:wN \__fp_pack_big:NNNNNNw
13961      \int_use:N \__int_eval:w \c__fp_big_middle_shift_int
13962      #1 #7*#5 #1 #8*#4 #1 #9*#3 #2
13963      \exp_after:wN \__fp_pack_big:NNNNNNw
13964      \int_use:N \__int_eval:w \c__fp_big_middle_shift_int
13965      #1 \__fp_fixed_mul_add:nnnnwnnn {#7}{#8}{#9}
13966    }

```

(End definition for `__fp_fixed_mul_add:Nwnnnwnnn`.)

`__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 $\langle b \rangle$.

```

13967 \cs_new:Npn \__fp_fixed_mul_add:nnnnwnnnn #1#2#3#4#5; #6#7#8#9
13968 {
13969   ( #1*#9 + #2*#8 + #3*#7 + #4*#6 )
13970   \exp_after:wN \__fp_pack_big:NNNNNNw
13971   \int_use:N \__int_eval:w \c__fp_big_trailing_shift_int
13972   \__fp_fixed_mul_add:nnnnwnnnwN
13973   { #6 + #4*#7 + #3*#8 + #2*#9 + #1 }
13974   { #7 + #4*#8 + #3*#9 + #2 }
13975   {#1} #5;
13976   {#6}
13977 }

```

(End definition for `__fp_fixed_mul_add:nnnnwnnnn`.)

`_fp_fixed_mul_add:nnnnwnnnwN`

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.

```

13978 \cs_new:Npn \__fp_fixed_mul_add:nnnnwnnnwN #1#2 #3#4#5; #6#7#8; #9
13979 {
13980   #9 (#4* #1 *#7)
13981   #9 (#5*#6+#4* #2 *#7+#3*#8) / \c_ten_thousand
13982 }

```

(End definition for `__fp_fixed_mul_add:nnnnwnnnwN`.)

28.8 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.

`__fp_ep_to_fixed:wwn`
`__fp_ep_to_fixed_auxi:www`
`_fp_ep_to_fixed_auxii:nnnnnnnnwn`

Converts an extended-precision number with an exponent at most 4 to a fixed point number whose first block will have 12 digits, most often starting with many zeros.

```

13983 \cs_new:Npn \__fp_ep_to_fixed:wwn #1,#2
13984 {
13985   \exp_after:wN \__fp_ep_to_fixed_auxi:www
13986   \int_use:N \__int_eval:w 1 0000 0000 + #2 \exp_after:wN ;
13987   \exp:w \exp_end_continue_f:w
13988   \prg_replicate:nn { \c_four - \int_max:nn {#1} { -32 } } { 0 } ;

```

```

13989 }
13990 \cs_new:Npn \__fp_ep_to_fixed_auxi:www #1; #2; #3#4#5#6#7;
13991 {
13992   \__fp_pack_eight:wNNNNNNNN
13993   \__fp_pack_twice_four:wNNNNNNNN
13994   \__fp_pack_twice_four:wNNNNNNNN
13995   \__fp_pack_twice_four:wNNNNNNNN
13996   \__fp_ep_to_fixed_auxii:nnnnnnwn ;
13997   #2 #1#3#4#5#6#7 0000 !
13998 }
13999 \cs_new:Npn \__fp_ep_to_fixed_auxii:nnnnnnwn #1#2#3#4#5#6#7; #8! #9
14000 { #9 {#1#2}{#3}{#4}{#5}{#6}{#7}; }

```

(End definition for __fp_ep_to_fixed:wnn.)

```

\__fp_ep_to_ep:wwN
\__fp_ep_to_ep_loop:N
\__fp_ep_to_ep_end:www
\__fp_ep_to_ep_zero:ww

```

Normalize an extended-precision number. More precisely, leading zeros are removed from the mantissa of the argument, decreasing its exponent as appropriate. Then the digits are packed into 6 groups of 4 (discarding any remaining digit, not rounding). Finally, the continuation #8 is placed before the resulting exponent–mantissa pair. The input exponent may in fact be given as an integer expression. The `loop` auxiliary grabs a digit: if it is 0, decrement the exponent and continue looping, and otherwise call the `end` auxiliary, which places all digits in the right order (the digit that was not 0, and any remaining digits), followed by some 0, then packs them up neatly in $3 \times 2 = 6$ blocks of four. At the end of the day, remove with `__fp_use_i:ww` any digit that did not make it in the final mantissa (typically only zeros, unless the original first block has more than 4 digits).

```

14001 \cs_new:Npn \__fp_ep_to_ep:wwN #1,#2#3#4#5#6#7; #8
14002 {
14003   \exp_after:wN #8
14004   \int_use:N \__int_eval:w #1 + \c_four
14005   \exp_after:wN \use_i:nn
14006   \exp_after:wN \__fp_ep_to_ep_loop:N
14007   \int_use:N \__int_eval:w 1 0000 0000 + #2 \__int_eval_end:
14008   #3#4#5#6#7 ; ; !
14009 }
14010 \cs_new:Npn \__fp_ep_to_ep_loop:N #1
14011 {
14012   \if_meaning:w 0 #1
14013   - \c_one
14014   \else:
14015     \__fp_ep_to_ep_end:www #1
14016     \fi:
14017     \__fp_ep_to_ep_loop:N
14018   }
14019 \cs_new:Npn \__fp_ep_to_ep_end:www
14020 #1 \fi: \__fp_ep_to_ep_loop:N #2; #3!
14021 {
14022   \fi:
14023   \if_meaning:w ; #1

```

```

14024         - \c_two * \c__fp_max_exponent_int
14025         \__fp_ep_to_ep_zero:ww
14026     \fi:
14027     \__fp_pack_twice_four:wNNNNNNNN
14028     \__fp_pack_twice_four:wNNNNNNNN
14029     \__fp_pack_twice_four:wNNNNNNNN
14030     \__fp_use_i:ww , ;
14031     #1 #2 0000 0000 0000 0000 0000 0000 ;
14032 }
14033 \cs_new:Npn \__fp_ep_to_ep_zero:ww \fi: #1; #2; #3;
14034 { \fi: , {1000}{0000}{0000}{0000}{0000}{0000} ; }

```

(End definition for __fp_ep_to_ep:wwN.)

__fp_ep_compare:wwwN 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].

```

14035 \cs_new:Npn \__fp_ep_compare:wwwN #1,#2#3#4#5#6#7;
14036 { \__fp_ep_compare_aux:wwwN {#1}{#2}{#3}{#4}{#5}; #6#7; }
14037 \cs_new:Npn \__fp_ep_compare_aux:wwwN #1,#2;#3,#4#5#6#7#8#9;
14038 {
14039     \if_case:w
14040         \__fp_compare_npos:wnw #1; {#3}{#4}{#5}{#6}{#7}; \exp_stop_f:
14041         \if_int_compare:w #2 = #8#9 \exp_stop_f:
14042             0
14043         \else:
14044             \if_int_compare:w #2 < #8#9 - \fi: 1
14045         \fi:
14046     \or: 1
14047     \else: -1
14048     \fi:
14049 }

```

(End definition for __fp_ep_compare:wwwN.)

__fp_ep_mul: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].

```

14050 \cs_new:Npn \__fp_ep_mul:wwwN #1,#2; #3,#4;
14051 {
14052     \__fp_ep_to_ep:wwN #3,#4;
14053     \__fp_fixed_continue:wn
14054     {
14055         \__fp_ep_to_ep:wwN #1,#2;
14056         \__fp_ep_mul_raw:wwwN
14057     }
14058     \__fp_fixed_continue:wn
14059 }
14060 \cs_new:Npn \__fp_ep_mul_raw:wwwN #1,#2; #3,#4; #5

```

```

14061 {
14062   \_fp_fixed_mul:wwn #2; #4;
14063   { \exp_after:wN #5 \int_use:N \_int_eval:w #1 + #3 , }
14064 }

```

(End definition for _fp_ep_mul:wwwn and _fp_ep_mul_raw:wwwn.)

28.9 Dividing extended-precision numbers

Divisions of extended-precision numbers are difficult to perform with exact rounding: the technique used in `l3fp-basics` for 16-digit floating point numbers does not generalize easily to 24-digit numbers. Thankfully, there is no need for exact rounding.

Let us call $\langle n \rangle$ the numerator and $\langle d \rangle$ the denominator. After a simple normalization step, we can assume that $\langle n \rangle \in [0.1, 1)$ and $\langle d \rangle \in [0.1, 1)$, and compute $\langle n \rangle / (10 \langle d \rangle) \in (0.01, 1)$. In terms of the 6 blocks of digits $\langle n_1 \rangle \cdots \langle n_6 \rangle$ and the 6 blocks $\langle d_1 \rangle \cdots \langle d_6 \rangle$, the condition translates to $\langle n_1 \rangle, \langle d_1 \rangle \in [1000, 9999]$.

We will first find an integer estimate $a \simeq 10^8 / \langle d \rangle$ by computing

$$\begin{aligned} \alpha &= \left\lceil \frac{10^9}{\langle d_1 \rangle + 1} \right\rceil \\ \beta &= \left\lfloor \frac{10^9}{\langle d_1 \rangle} \right\rfloor \\ a &= 10^3 \alpha + (\beta - \alpha) \cdot \left(10^3 - \left\lfloor \frac{\langle d_2 \rangle}{10} \right\rfloor \right) - 1250, \end{aligned}$$

where $\left\lceil \cdot \right\rceil$ 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 $\left\lfloor \frac{\langle d_2 \rangle}{10} \right\rfloor$ 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,

$$10^7 \langle d \rangle a < \left(10^3 \langle d_1 \rangle + \left\lfloor \frac{\langle d_2 \rangle}{10} \right\rfloor + \frac{1}{2} \right) \left(\left(10^3 - \left\lfloor \frac{\langle d_2 \rangle}{10} \right\rfloor \right) \beta + \left\lfloor \frac{\langle d_2 \rangle}{10} \right\rfloor \alpha - 1250 \right) \quad (1)$$

$$< \left(10^3 \langle d_1 \rangle + \left\lfloor \frac{\langle d_2 \rangle}{10} \right\rfloor + \frac{1}{2} \right) \quad (2)$$

$$\left(\left(10^3 - \left\lfloor \frac{\langle d_2 \rangle}{10} \right\rfloor \right) \left(\frac{10^9}{\langle d_1 \rangle} + \frac{1}{2} \right) + \left\lfloor \frac{\langle d_2 \rangle}{10} \right\rfloor \left(\frac{10^9}{\langle d_1 \rangle + 1} + \frac{1}{2} \right) - 1250 \right) \quad (3)$$

$$< \left(10^3 \langle d_1 \rangle + \left\lfloor \frac{\langle d_2 \rangle}{10} \right\rfloor + \frac{1}{2} \right) \left(\frac{10^{12}}{\langle d_1 \rangle} - \left\lfloor \frac{\langle d_2 \rangle}{10} \right\rfloor \frac{10^9}{\langle d_1 \rangle (\langle d_1 \rangle + 1)} - 750 \right) \quad (4)$$

We recognize a quadratic polynomial in $[\langle d_2 \rangle / 10]$ with a negative leading coefficient: this polynomial is bounded above, according to $([\langle d_2 \rangle / 10] + a)(b - c[\langle d_2 \rangle / 10]) \leq (b + ca)^2 / (4c)$. Hence,

$$10^7 \langle d \rangle a < \frac{10^{15}}{\langle d_1 \rangle (\langle d_1 \rangle + 1)} \left(\langle d_1 \rangle + \frac{1}{2} + \frac{1}{4} 10^{-3} - \frac{3}{8} \cdot 10^{-9} \langle d_1 \rangle (\langle d_1 \rangle + 1) \right)^2$$

Since $\langle d_1 \rangle$ takes integer values within $[1000, 9999]$, it is a simple programming exercise to check that the squared expression is always less than $\langle d_1 \rangle (\langle d_1 \rangle + 1)$, hence $10^7 \langle d \rangle a < 10^{15}$. The upper bound is proven. We also find that $\frac{3}{8}$ can be replaced by slightly smaller numbers, but nothing less than $0.374563\dots$, and going back through the derivation of the upper bound, we find that 1250 is as small a shift as we can obtain without breaking the bound.

Now, the lower bound. The same computation as for the upper bound implies

$$10^7 \langle d \rangle a > \left(10^3 \langle d_1 \rangle + \left\lfloor \frac{\langle d_2 \rangle}{10} \right\rfloor - \frac{1}{2} \right) \left(\frac{10^{12}}{\langle d_1 \rangle} - \left\lfloor \frac{\langle d_2 \rangle}{10} \right\rfloor \frac{10^9}{\langle d_1 \rangle (\langle d_1 \rangle + 1)} - 1750 \right)$$

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 $\text{T}_{\text{E}}\text{X}$ integer, and even add 10^9 to it to ease grabbing of all the digits. The lower bound implies $10^8 - 1755 < a$, which we do not care about.

`_fp_ep_div:wwwn` Compute the ratio of two extended-precision numbers. The result is an extended-precision number whose first block lies in the range $[100, 9999]$, and is placed after the $\langle continuation \rangle$ once we are done. First normalize the inputs so that both first block lie in $[1000, 9999]$, then call `_fp_ep_div_esti:wwwn $\langle denominator \rangle$ $\langle numerator \rangle$` , responsible for estimating the inverse of the denominator.

```

14065 \cs_new:Npn \_fp_ep_div:wwwn #1,#2; #3,#4;
14066 {
14067   \_fp_ep_to_ep:wwN #1,#2;
14068   \_fp_fixed_continue:wn

```

```

14069     {
14070         \__fp_ep_to_ep:wwN #3,#4;
14071         \__fp_ep_div_esti:wwwn
14072     }
14073 }

```

(End definition for __fp_ep_div:wwwn.)

```

\__fp_ep_div_esti:wwwn
\__fp_ep_div_estii:wwnnwwn
\__fp_ep_div_estiii:NNNNwwn

```

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_epsilon:wnNNNNn**, which computes $10^{-9}a/(1 - \epsilon)$, that is, $1/(10 \langle d \rangle)$ and we finally multiply this by the numerator #8.

```

14074 \cs_new:Npn \__fp_ep_div_esti:wwwn #1,#2#3; #4,
14075 {
14076     \exp_after:wN \__fp_ep_div_estii:wwnnwwn
14077     \int_use:N \__int_eval:w 10 0000 0000 / ( #2 + \c_one )
14078     \exp_after:wN ;
14079     \int_use:N \__int_eval:w #4 - #1 + \c_one ,
14080     {#2} #3;
14081 }
14082 \cs_new:Npn \__fp_ep_div_estii:wwnnwwn #1; #2,#3#4#5; #6; #7
14083 {
14084     \exp_after:wN \__fp_ep_div_estiii:NNNNwwn
14085     \int_use:N \__int_eval:w 10 0000 0000 - 1750
14086     + #1 000 + (10 0000 0000 / #3 - #1) * (1000 - #4 / 10) ;
14087     {#3}{#4}#5; #6; { #7 #2, }
14088 }
14089 \cs_new:Npn \__fp_ep_div_estiii:NNNNwwn 1#1#2#3#4#5#6; #7;
14090 {
14091     \__fp_fixed_mul_short:wnn #7; {#1}{#2#3#4#5}{#6};
14092     \__fp_ep_div_epsilon:wnNNNNn {#1#2#3#4}#5#6
14093     \__fp_fixed_mul:wnn
14094 }

```

(End definition for __fp_ep_div_esti:wwwn, __fp_ep_div_estii:wwnnwwn, and __fp_ep_div_estiii:NNNNwwn.)

```

\__fp_ep_div_epsilon:wnNNNNn
\__fp_ep_div_eps_pack:NNNNw
\__fp_ep_div_epsii:wnNNNNn

```

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 **epsii** 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`.

```

14095 \cs_new:Npn \__fp_ep_div_epsilon:wnNNNNNn #1#2#3#4#5#6;
14096 {
14097   \exp_after:wN \__fp_ep_div_epsilon:wnNNNNNn
14098   \int_use:N \__int_eval:w 1 9998 - #2
14099   \exp_after:wN \__fp_ep_div_epsilon_pack:NNNNNw
14100   \int_use:N \__int_eval:w 1 9999 9998 - #3#4
14101   \exp_after:wN \__fp_ep_div_epsilon_pack:NNNNNw
14102   \int_use:N \__int_eval:w 2 0000 0000 - #5#6 ; ;
14103 }
14104 \cs_new:Npn \__fp_ep_div_epsilon_pack:NNNNNw #1#2#3#4#5#6;
14105 { + #1 ; {#2#3#4#5} {#6} }
14106 \cs_new:Npn \__fp_ep_div_epsilon:wnNNNNNn 1#1; #2; #3#4#5#6#7#8
14107 {
14108   \__fp_fixed_mul:wnn {0000}{#1}#2; {0000}{#1}#2;
14109   \__fp_fixed_add_one:wN
14110   \__fp_fixed_mul:wnn {10000} {#1} #2 ;
14111   {
14112     \__fp_fixed_mul_short:wnn {0000}{#1}#2; {#3}{#4#5#6#7}{#8000};
14113     \__fp_fixed_div_myriad:wn
14114     \__fp_fixed_mul:wnn
14115   }
14116   \__fp_fixed_add:wnn {#3}{#4#5#6#7}{#8000}{0000}{0000}{0000};
14117 }

```

(End definition for `__fp_ep_div_epsilon:wnNNNNNn`, `__fp_ep_div_epsilon_pack:NNNNNw`, and `__fp_ep_div_epsilon:wnNNNNNn`.)

28.10 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 $r^2 x \geq 10^4$. This also causes $r \in [101, 1003]$. Another correction to the above is that the input is actually normalized to $[0.1, 1)$, and we use either 10^8 or 10^9 in the Newton method, depending on the parity of the exponent. Skipping those technical hurdles, once we have the approximation r , we set $y = 10^{-4} r^2 x$ (or rather, the correct power of 10 to get $y \simeq 1$) and compute $y^{-1/2}$ through another application of Newton's method. This time, the starting value is $z = 1$, each step maps $z \mapsto z(1.5 - 0.5yz^2)$, and we perform a fixed number of steps. Our final result combines r with $y^{-1/2}$ as $x^{-1/2} = 10^{-2} r y^{-1/2}$.

```

\__fp_ep_isqrt:wnn First normalize the input, then check the parity of the exponent #1. If it is even, the
\__fp_ep_isqrt_aux:wnn result's exponent will be  $-\#1/2$ , otherwise it will be  $(\#1 - 1)/2$  (except in the case
\__fp_ep_isqrt_auxii:wnnnwnn 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.

```

14118 \cs_new:Npn \__fp_ep_isqrt:wnn #1,#2;
14119 {
14120   \__fp_ep_to_ep:wwN #1,#2;
14121   \__fp_ep_isqrt_auxi:wnn
14122 }
14123 \cs_new:Npn \__fp_ep_isqrt_auxi:wnn #1,
14124 {
14125   \exp_after:wN \__fp_ep_isqrt_auxii:wwnnwn
14126   \int_use:N \__int_eval:w
14127   \int_if_odd:nTF {#1}
14128     { (\c_one - #1) / \c_two , 535 , { 0 } { } }
14129     { \c_one - #1 / \c_two , 168 , { } { 0 } }
14130 }
14131 \cs_new:Npn \__fp_ep_isqrt_auxii:wwnnwn #1, #2, #3#4 #5#6; #7
14132 {
14133   \__fp_ep_isqrt_esti:wwnnwn #2, 0, #5, {#3} {#4}
14134   {#5} #6 ; { #7 #1 , }
14135 }

```

(End definition for `__fp_ep_isqrt:wnn`.)

```

\__fp_ep_isqrt_esti:wwnnwn
\__fp_ep_isqrt_estii:wwnnwn
\__fp_ep_isqrt_estiii:NNNNwwnn

```

If the last two approximations gave the same result, we are done: call the `estii` function to clean up. Otherwise, evaluate $(\langle prev \rangle + 1.005 \cdot 10^8 \text{ or } 9 / (\langle prev \rangle \cdot x)) / 2$, as the next approximation: omitting the 1.005 factor, this would be Newton's method. We can check by brute force that if #4 is empty (the original exponent was even), the process computes an integer slightly larger than $100/\sqrt{x}$, while if #4 is 0 (the original exponent was odd), the result is an integer slightly larger than $100/\sqrt{x/10}$. Once we are done, we evaluate $100r^2/2$ or $10r^2/2$ (when the exponent is even or odd, respectively) and feed that to `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_epsilon:wN`, which computes $1/\sqrt{y}$, and we finally multiply the result by r .

```

14136 \cs_new:Npn \__fp_ep_isqrt_esti:wwnnwn #1, #2, #3, #4
14137 {
14138   \if_int_compare:w #1 = #2 \exp_stop_f:
14139   \exp_after:wN \__fp_ep_isqrt_estii:wwnnwn
14140   \fi:
14141   \exp_after:wN \__fp_ep_isqrt_esti:wwnnwn
14142   \int_use:N \__int_eval:w
14143   (#1 + 1 0050 0000 #4 / (#1 * #3)) / \c_two ,

```

```

14144     #1, #3, {#4}
14145   }
14146   \cs_new:Npn \__fp_ep_isqrt_estii:wwnnwn #1, #2, #3, #4#5
14147   {
14148     \exp_after:wN \__fp_ep_isqrt_estiii:NNNNNwwnn
14149     \int_use:N \__int_eval:w 1000 0000 + #2 * #2 #5 * \c_five
14150     \exp_after:wN , \int_use:N \__int_eval:w 10000 + #2 ;
14151   }
14152   \cs_new:Npn \__fp_ep_isqrt_estiii:NNNNNwwnn 1#1#2#3#4#5#6, 1#7#8; #9;
14153   {
14154     \__fp_fixed_mul_short:wwn #9; {#1} {#2#3#4#5} {#600} ;
14155     \__fp_ep_isqrt_epsilon:wN
14156     \__fp_fixed_mul_short:wwn {#7} {#80} {0000} ;
14157   }

```

(End definition for __fp_ep_isqrt_esti:wwnnwn, __fp_ep_isqrt_estii:wwnnwn, and __fp_ep_isqrt_estiii:NNNNNwwnn.)

__fp_ep_isqrt_epsilon:wN
 __fp_ep_isqrt_epsilonii: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`.

```

14158   \cs_new:Npn \__fp_ep_isqrt_epsilon:wN #1;
14159   {
14160     \__fp_fixed_sub:wwn {15000}{0000}{0000}{0000}{0000}{0000}; #1;
14161     \__fp_ep_isqrt_epsilonii:wwN #1;
14162     \__fp_ep_isqrt_epsilonii:wwN #1;
14163     \__fp_ep_isqrt_epsilonii:wwN #1;
14164   }
14165   \cs_new:Npn \__fp_ep_isqrt_epsilonii:wwN #1; #2;
14166   {
14167     \__fp_fixed_mul:wwn #1; #1;
14168     \__fp_fixed_mul_sub_back:wwnn #2;
14169     {15000}{0000}{0000}{0000}{0000}{0000};
14170     \__fp_fixed_mul:wwn #1;
14171   }

```

(End definition for __fp_ep_isqrt_epsilon:wN and __fp_ep_isqrt_epsilonii:wwN.)

28.11 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:wwN
 __fp_ep_inv_to_float: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.

```

14172   \cs_new:Npn \__fp_ep_to_float:wwN #1,

```

```

14173 { + \__int_eval:w #1 \__fp_fixed_to_float:wN }
14174 \cs_new:Npn \__fp_ep_inv_to_float:wwN #1,#2;
14175 {
14176   \__fp_ep_div:wwwN 1,{1000}{0000}{0000}{0000}{0000}{0000}; #1,#2;
14177   \__fp_ep_to_float:wwN
14178 }

```

(End definition for __fp_ep_to_float:wwN and __fp_ep_inv_to_float:wwN.)

__fp_fixed_inv_to_float:wN Another function which reduces to converting an extended precision number to a float.

```

14179 \cs_new:Npn \__fp_fixed_inv_to_float:wN
14180 { \__fp_ep_inv_to_float:wwN 0, }

```

(End definition for __fp_fixed_inv_to_float:wN.)

__fp_fixed_to_float_rad:wN Converts the fixed point number #1 from degrees to radians then to a floating point number. This could perhaps remain in l3fp-trig.

```

14181 \cs_new:Npn \__fp_fixed_to_float_rad:wN #1;
14182 {
14183   \__fp_fixed_mul:wwN #1; {5729}{5779}{5130}{8232}{0876}{7981};
14184   { \__fp_ep_to_float:wwN 2, }
14185 }

```

(End definition for __fp_fixed_to_float_rad:wN.)

__fp_fixed_to_float:wN yields

__fp_fixed_to_float:Nw $\langle exponent' \rangle ; \{ \langle a'_1 \rangle \} \{ \langle a'_2 \rangle \} \{ \langle a'_3 \rangle \} \{ \langle a'_4 \rangle \} ;$

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 .¹⁰

```

14186 \cs_new:Npn \__fp_fixed_to_float:Nw #1#2; { \__fp_fixed_to_float:wN #2; #1 }
14187 \cs_new:Npn \__fp_fixed_to_float:wN #1#2#3#4#5#6; #7
14188 {
14189   + \__int_eval:w \c_four % for the 8-digit-at-the-start thing.
14190   \exp_after:wN \exp_after:wN
14191   \exp_after:wN \__fp_fixed_to_loop:N
14192   \exp_after:wN \use_none:n
14193   \int_use:N \__int_eval:w
14194   1 0000 0000 + #1 \exp_after:wN \__fp_use_none_stop_f:n
14195   \__int_value:w 1#2 \exp_after:wN \__fp_use_none_stop_f:n
14196   \__int_value:w 1#3#4 \exp_after:wN \__fp_use_none_stop_f:n
14197   \__int_value:w 1#5#6
14198   \exp_after:wN ;
14199   \exp_after:wN ;
14200 }
14201 \cs_new:Npn \__fp_fixed_to_loop:N #1
14202 {

```

¹⁰Bruno: I must double check this assumption.

```

14203     \if_meaning:w 0 #1
14204     - \c_one
14205     \exp_after:wN \__fp_fixed_to_loop:N
14206   \else:
14207     \exp_after:wN \__fp_fixed_to_loop_end:w
14208     \exp_after:wN #1
14209   \fi:
14210 }
14211 \cs_new:Npn \__fp_fixed_to_loop_end:w #1 #2 ;
14212 {
14213   \if_meaning:w ; #1
14214   \exp_after:wN \__fp_fixed_to_float_zero:w
14215   \else:
14216     \exp_after:wN \__fp_pack_twice_four:wNNNNNNNN
14217     \exp_after:wN \__fp_pack_twice_four:wNNNNNNNN
14218     \exp_after:wN \__fp_fixed_to_float_pack:ww
14219     \exp_after:wN ;
14220   \fi:
14221   #1 #2 0000 0000 0000 0000 ;
14222 }
14223 \cs_new:Npn \__fp_fixed_to_float_zero:w ; 0000 0000 0000 0000 ;
14224 {
14225   - \c_two * \c__fp_max_exponent_int ;
14226   {0000} {0000} {0000} {0000} ;
14227 }
14228 \cs_new:Npn \__fp_fixed_to_float_pack:ww #1 ; #2#3 ; ;
14229 {
14230   \if_int_compare:w #2 > \c_four
14231     \exp_after:wN \__fp_fixed_to_float_round_up:wnnnnw
14232   \fi:
14233   ; #1 ;
14234 }
14235 \cs_new:Npn \__fp_fixed_to_float_round_up:wnnnnw ; #1#2#3#4 ;
14236 {
14237   \exp_after:wN \__fp_basics_pack_high:NNNNNw
14238   \int_use:N \__int_eval:w 1 #1#2
14239   \exp_after:wN \__fp_basics_pack_low:NNNNNw
14240   \int_use:N \__int_eval:w 1 #3#4 + \c_one ;
14241 }

```

(End definition for __fp_fixed_to_float:wN and __fp_fixed_to_float:Nw.)

```

14242 </initex | package>

```

29 l3fp-expo implementation

```

14243 <*initex | package>
14244 <@@=fp>

```

29.1 Logarithm

29.1.1 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.

29.1.2 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 $\ln(5)$.

<code>\c_fp_ln_i_fixed_t1</code>	14245	<code>\tl_const:Nn \c_fp_ln_i_fixed_t1</code>	{ {0000}{0000}{0000}{0000}{0000} }
<code>\c_fp_ln_ii_fixed_t1</code>	14246	<code>\tl_const:Nn \c_fp_ln_ii_fixed_t1</code>	{ {6931}{4718}{0559}{9453}{0941}{7232} }
<code>\c_fp_ln_iii_fixed_t1</code>	14247	<code>\tl_const:Nn \c_fp_ln_iii_fixed_t1</code>	{ {10986}{1228}{8668}{1096}{9139}{5245} }
<code>\c_fp_ln_iv_fixed_t1</code>	14248	<code>\tl_const:Nn \c_fp_ln_iv_fixed_t1</code>	{ {13862}{9436}{1119}{8906}{1883}{4464} }
<code>\c_fp_ln_vii_fixed_t1</code>	14249	<code>\tl_const:Nn \c_fp_ln_vi_fixed_t1</code>	{ {17917}{5946}{9228}{0550}{0081}{2477} }
<code>\c_fp_ln_viii_fixed_t1</code>	14250	<code>\tl_const:Nn \c_fp_ln_vii_fixed_t1</code>	{ {19459}{1014}{9055}{3133}{0510}{5353} }
<code>\c_fp_ln_ix_fixed_t1</code>	14251	<code>\tl_const:Nn \c_fp_ln_viii_fixed_t1</code>	{ {20794}{4154}{1679}{8359}{2825}{1696} }
<code>\c_fp_ln_x_fixed_t1</code>	14252	<code>\tl_const:Nn \c_fp_ln_ix_fixed_t1</code>	{ {21972}{2457}{7336}{2193}{8279}{0490} }
	14253	<code>\tl_const:Nn \c_fp_ln_x_fixed_t1</code>	{ {23025}{8509}{2994}{0456}{8401}{7991} }

(End definition for `\c_fp_ln_i_fixed_t1` and others.)

29.1.3 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`.

```

14254 \cs_new:Npn \__fp_ln_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
14255 {
14256   \if_meaning:w 2 #3
14257     \__fp_case_use:nw { \__fp_invalid_operation_o:nw { ln } }
14258   \fi:
14259   \if_case:w #2 \exp_stop_f:
14260     \__fp_case_use:nw
14261     { \__fp_division_by_zero_o:Nnw \c_minus_inf_fp { ln } }
14262   \or:
14263   \else:
14264     \__fp_case_return_same_o:w
14265   \fi:
14266   \__fp_ln_npos_o:w \s__fp \__fp_chk:w #2#3#4;
14267 }

```

(End definition for __fp_ln_o:w.)

29.1.4 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.

```

14268 \cs_new:Npn \__fp_ln_npos_o:w \s__fp \__fp_chk:w 10#1#2#3;
14269 { %^A todo: ln(1) should be "exact zero", not "underflow"
14270   \exp_after:wN \__fp_sanitizewN
14271   \__int_value:w % for the overall sign
14272   \if_int_compare:w #1 < \c_one
14273     2
14274   \else:
14275     0
14276   \fi:
14277   \exp_after:wN \exp_stop_f:
14278   \int_use:N \__int_eval:w % for the exponent
14279   \__fp_ln_significand:NNNNnnnnN #2#3
14280   \__fp_ln_exponent:wn {#1}
14281 }

```

(End definition for __fp_ln_npos_o:w.)

__fp_ln_significand:NNNNnnnnN __fp_ln_significand:NNNNnnnnN $\langle X_1 \rangle \{ \langle X_2 \rangle \} \{ \langle X_3 \rangle \} \{ \langle X_4 \rangle \} \langle continuation \rangle$

This function expands to

$\langle continuation \rangle \{ \langle Y_1 \rangle \} \{ \langle Y_2 \rangle \} \{ \langle Y_3 \rangle \} \{ \langle Y_4 \rangle \} \{ \langle Y_5 \rangle \} \{ \langle Y_6 \rangle \} ;$

where $Y = -\ln(X)$ as an extended fixed point.

```

14282 \cs_new:Npn \__fp_ln_significand:NNNNnnnnN #1#2#3#4
14283 {
14284   \exp_after:wN \__fp_ln_x_ii:wnnnn
14285   \__int_value:w
14286   \if_case:w #1 \exp_stop_f:
14287   \or:

```

```

14288         \if_int_compare:w #2 < \c_four
14289         \__int_eval:w \c_ten - #2
14290         \else:
14291             6
14292         \fi:
14293         \or: 4
14294         \or: 3
14295         \or: 2
14296         \or: 2
14297         \or: 2
14298         \else: 1
14299         \fi:
14300     ; { #1 #2 #3 #4 }
14301 }

```

(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 $1 + x = 1 + ac \in [1.7, 2.4)$.

```

14302 \cs_new:Npn \__fp_ln_x_ii:wnnnn #1; #2#3#4#5
14303 {
14304     \exp_after:wN \__fp_ln_div_after:Nw
14305     \cs:w c__fp_ln_ \__int_to_roman:w #1 _fixed_tl \exp_after:wN \cs_end:
14306     \__int_value:w
14307     \exp_after:wN \__fp_ln_x_iv:wnnnnnnnn
14308     \int_use:N \__int_eval:w
14309     \exp_after:wN \__fp_ln_x_iii_var:NNNNNw
14310     \int_use:N \__int_eval:w 9999 9990 + #1*#2#3 +
14311     \exp_after:wN \__fp_ln_x_iii:NNNNNNw
14312     \int_use:N \__int_eval:w 10 0000 0000 + #1*#4#5 ;
14313     {20000} {0000} {0000} {0000}
14314 } %^A todo: reoptimize (a generalization attempt failed).
14315 \cs_new:Npn \__fp_ln_x_iii:NNNNNNw #1#2 #3#4#5#6 #7;
14316 { #1#2; {#3#4#5#6} {#7} }
14317 \cs_new:Npn \__fp_ln_x_iii_var:NNNNNw #1 #2#3#4#5 #6;
14318 {
14319     #1#2#3#4#5 + \c_one ;
14320     {#1#2#3#4#5} {#6}
14321 }

```

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 $2.147 \dots$, 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 $B = 10^4 A - Q_1 Z$. Then

$$\begin{aligned} 10^4 B &\leq A_1 A_2 \cdot A_3 A_4 - \left(\frac{A_1 A_2}{y} - \frac{3}{2} \right) 10^4 Z \\ &\leq A_1 A_2 \left(1 - \frac{10^4 Z}{y} \right) + 1 + \frac{3}{2} y \\ &\leq 10^8 \frac{A}{y} + 1 + \frac{3}{2} y \end{aligned}$$

In the same way, and using $1.7 \cdot 10^4 \leq y \leq 2.4 \cdot 10^4$, and convexity, we get

$$\begin{aligned} 10^4 A &= 2 \cdot 10^4 \\ 10^4 B &\leq 10^8 \frac{A}{y} + 1.6y \leq 4.7 \cdot 10^4 \\ 10^4 C &\leq 10^8 \frac{B}{y} + 1.6y \leq 5.8 \cdot 10^4 \\ 10^4 D &\leq 10^8 \frac{C}{y} + 1.6y \leq 6.3 \cdot 10^4 \\ 10^4 E &\leq 10^8 \frac{D}{y} + 1.6y \leq 6.5 \cdot 10^4 \\ 10^4 F &\leq 10^8 \frac{E}{y} + 1.6y \leq 6.6 \cdot 10^4 \end{aligned}$$

Note that we compute more steps than for division: since t is not the end result, we need to know it with more accuracy (on the other hand, the ending is much simpler, as we don't need an exact rounding for transcendental functions, but just a faithful rounding).¹¹

`_fp_ln_x_iv:wnnnnnnnnn <1 or 2> <8d> ; {\<4d>} {\<4d>} <fixed-tl>`

The number is x . Compute y by adding 1 to the five first digits.

14322 `\cs_new:Npn _fp_ln_x_iv:wnnnnnnnnn #1; #2#3#4#5 #6#7#8#9`
14323 `{`

¹¹Bruno: to be completed.

```

14324 \exp_after:wN \_fp_div_significand_pack:NNN
14325 \int_use:N \_int_eval:w
14326 \_fp_ln_div_i:w #1 ;
14327 #6 #7 ; {#8} {#9}
14328 {#2} {#3} {#4} {#5}
14329 { \exp_after:wN \_fp_ln_div_ii:wwn \_int_value:w #1 }
14330 { \exp_after:wN \_fp_ln_div_ii:wwn \_int_value:w #1 }
14331 { \exp_after:wN \_fp_ln_div_ii:wwn \_int_value:w #1 }
14332 { \exp_after:wN \_fp_ln_div_ii:wwn \_int_value:w #1 }
14333 { \exp_after:wN \_fp_ln_div_vi:wwn \_int_value:w #1 }
14334 }
14335 \cs_new:Npn \_fp_ln_div_i:w #1;
14336 {
14337 \exp_after:wN \_fp_div_significand_calc:wwnnnnnnn
14338 \int_use:N \_int_eval:w 999999 + 2 0000 0000 / #1 ; % Q1
14339 }
14340 \cs_new:Npn \_fp_ln_div_ii:wwn #1; #2;#3 % y; B1;B2 <- for k=1
14341 {
14342 \exp_after:wN \_fp_div_significand_pack:NNN
14343 \int_use:N \_int_eval:w
14344 \exp_after:wN \_fp_div_significand_calc:wwnnnnnnn
14345 \int_use:N \_int_eval:w 999999 + #2 #3 / #1 ; % Q2
14346 #2 #3 ;
14347 }
14348 \cs_new:Npn \_fp_ln_div_vi:wwn #1; #2;#3#4#5 #6#7#8#9 %y;F1;F2F3F4x1x2x3x4
14349 {
14350 \exp_after:wN \_fp_div_significand_pack:NNN
14351 \int_use:N \_int_eval:w 1000000 + #2 #3 / #1 ; % Q6
14352 }

```

We now have essentially¹²

$$\begin{aligned} & _fp_ln_div_after:Nw \langle fixed\ tl \rangle _fp_div_significand_pack:NNN 10^6 + \\ & Q_1 _fp_div_significand_pack:NNN 10^6 + Q_2 _fp_div_significand_ \\ & pack:NNN 10^6 + Q_3 _fp_div_significand_pack:NNN 10^6 + Q_4 _fp_ \\ & div_significand_pack:NNN 10^6 + Q_5 _fp_div_significand_pack:NNN \\ & 10^6 + Q_6 ; \langle exponent \rangle ; \langle continuation \rangle \end{aligned}$$

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

$$\begin{aligned} & _fp_ln_div_after:Nw \langle fixed\ tl \rangle \langle 1d \rangle ; \langle 4d \rangle ; \langle 4d \rangle ; \langle 4d \rangle ; \langle 4d \rangle ; \langle 4d \rangle ; \\ & \langle 4d \rangle ; \langle exponent \rangle ; \end{aligned}$$

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.

¹²Bruno: add a mention that the error on Q_6 is bounded by 10 (probably 6.7), and thus corresponds to an error of 10^{-23} on the final result, small enough in all cases.

```

14353 \cs_new:Npn \__fp_ln_div_after:Nw #1#2;
14354 {
14355   \if_meaning:w 0 #2
14356   \exp_after:wN \__fp_ln_t_small:Nw
14357   \else:
14358     \exp_after:wN \__fp_ln_t_large:NNw
14359     \exp_after:wN -
14360   \fi:
14361   #1
14362 }
14363 \cs_new:Npn \__fp_ln_t_small:Nw #1 #2; #3; #4; #5; #6; #7;
14364 {
14365   \exp_after:wN \__fp_ln_t_large:NNw
14366   \exp_after:wN + % <sign>
14367   \exp_after:wN #1
14368   \int_use:N \__int_eval:w 9999 - #2 \exp_after:wN ;
14369   \int_use:N \__int_eval:w 9999 - #3 \exp_after:wN ;
14370   \int_use:N \__int_eval:w 9999 - #4 \exp_after:wN ;
14371   \int_use:N \__int_eval:w 9999 - #5 \exp_after:wN ;
14372   \int_use:N \__int_eval:w 9999 - #6 \exp_after:wN ;
14373   \int_use:N \__int_eval:w 1 0000 - #7 ;
14374 }

\__fp_ln_t_large:NNw <sign><fixed tl> <t123456

```

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.

```

14375 \cs_new:Npn \__fp_ln_t_large:NNw #1 #2 #3; #4; #5; #6; #7; #8;
14376 {
14377   \exp_after:wN \__fp_ln_square_t_after:w
14378   \int_use:N \__int_eval:w 9999 0000 + #3*#3
14379   \exp_after:wN \__fp_ln_square_t_pack:NNNNNw
14380   \int_use:N \__int_eval:w 9999 0000 + 2*#3*#4
14381   \exp_after:wN \__fp_ln_square_t_pack:NNNNNw
14382   \int_use:N \__int_eval:w 9999 0000 + 2*#3*#5 + #4*#4
14383   \exp_after:wN \__fp_ln_square_t_pack:NNNNNw
14384   \int_use:N \__int_eval:w 9999 0000 + 2*#3*#6 + 2*#4*#5
14385   \exp_after:wN \__fp_ln_square_t_pack:NNNNNw
14386   \int_use:N \__int_eval:w 1 0000 0000 + 2*#3*#7 + 2*#4*#6 + #5*#5
14387   + (2*#3*#8 + 2*#4*#7 + 2*#5*#6) / 1 0000
14388   % ; ; ;
14389   \exp_after:wN \__fp_ln_twice_t_after:w
14390   \int_use:N \__int_eval:w -1 + 2*#3
14391   \exp_after:wN \__fp_ln_twice_t_pack:Nw
14392   \int_use:N \__int_eval:w 9999 + 2*#4
14393   \exp_after:wN \__fp_ln_twice_t_pack:Nw
14394   \int_use:N \__int_eval:w 9999 + 2*#5

```

```

14395         \exp_after:wN \__fp_ln_twice_t_pack:Nw
14396         \int_use:N \__int_eval:w 9999 + 2*#6
14397         \exp_after:wN \__fp_ln_twice_t_pack:Nw
14398         \int_use:N \__int_eval:w 9999 + 2*#7
14399         \exp_after:wN \__fp_ln_twice_t_pack:Nw
14400         \int_use:N \__int_eval:w 10000 + 2*#8 ; ;
14401     { \__fp_ln_c:NwNw #1 }
14402     #2
14403 }
14404 \cs_new:Npn \__fp_ln_twice_t_pack:Nw #1 #2; { + #1 ; {#2} }
14405 \cs_new:Npn \__fp_ln_twice_t_after:w #1; { ; ; ; {#1} }
14406 \cs_new:Npn \__fp_ln_square_t_pack:NNNNw #1 #2#3#4#5 #6;
14407     { + #1#2#3#4#5 ; {#6} }
14408 \cs_new:Npn \__fp_ln_square_t_after:w 1 0 #1#2#3 #4;
14409     { \__fp_ln_Taylor:wwNw {0#1#2#3} {#4} }

```

(End definition for __fp_ln_x_ii:wnnnn.)

__fp_ln_Taylor:wwNw Denoting $T = t^2$, we get

```

\__fp_ln_Taylor:wwNw {\langle T_1 \rangle} {\langle T_2 \rangle} {\langle T_3 \rangle} {\langle T_4 \rangle} {\langle T_5 \rangle} {\langle T_6 \rangle} ; ;
{\langle (2t)_1 \rangle} {\langle (2t)_2 \rangle} {\langle (2t)_3 \rangle} {\langle (2t)_4 \rangle} {\langle (2t)_5 \rangle} {\langle (2t)_6 \rangle} ; { \__fp_ln_
c:NwNn \langle sign \rangle } \langle fixed tl \rangle \langle exponent \rangle ; \langle continuation \rangle

```

And we want to compute

$$\ln\left(\frac{1+t}{1-t}\right) = 2t \left(1 + T \left(\frac{1}{3} + T \left(\frac{1}{5} + T \left(\frac{1}{7} + T \left(\frac{1}{9} + \dots\right)\right)\right)\right)\right)$$

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;

```

13

This uses the routine for dividing a number by a small integer ($< 10^4$).

```

14410 \cs_new:Npn \__fp_ln_Taylor:wwNw
14411     { \__fp_ln_Taylor_loop:www 21 ; {0000}{0000}{0000}{0000}{0000}{0000} ; }
14412 \cs_new:Npn \__fp_ln_Taylor_loop:www #1; #2; #3;
14413     {
14414         \if_int_compare:w #1 = \c_one
14415             \__fp_ln_Taylor_break:w
14416         \fi:
14417         \exp_after:wN \__fp_fixed_div_int:wwN \c__fp_one_fixed_tl ; #1;
14418         \__fp_fixed_add:wwN #2;
14419         \__fp_fixed_mul:wwN #3;

```

¹³Bruno: add explanations.

```

14420 {
14421   \exp_after:wN \_fp_ln_Taylor_loop:www
14422   \int_use:N \_int_eval:w #1 - \c_two ;
14423 }
14424 #3;
14425 }
14426 \cs_new:Npn \_fp_ln_Taylor_break:w \fi: #1 \_fp_fixed_add:wwn #2#3; #4 ;;
14427 {
14428   \fi:
14429   \exp_after:wN \_fp_fixed_mul:wwn
14430   \exp_after:wN { \int_use:N \_int_eval:w 10000 + #2 } #3;
14431 }

```

(End definition for `_fp_ln_Taylor:wwNw`. This function is documented on page ??.)

```

\_fp_ln_c:NwNw \_fp_ln_c:NwNw <sign> {\langle r_1 \rangle} {\langle r_2 \rangle} {\langle r_3 \rangle} {\langle r_4 \rangle} {\langle r_5 \rangle} {\langle r_6 \rangle} ; <fixed 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 $\ln(10)$ 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 $\ln(10/7) \simeq 0.35$.¹⁴

```

14432 \cs_new:Npn \_fp_ln_c:NwNw #1 #2; #3
14433 {
14434   \if_meaning:w + #1
14435     \exp_after:wN \exp_after:wN \exp_after:wN \_fp_fixed_sub:wwn
14436   \else:
14437     \exp_after:wN \exp_after:wN \exp_after:wN \_fp_fixed_add:wwn
14438   \fi:
14439   #3 ; #2 ;
14440 }

```

¹⁵

(End definition for `_fp_ln_c:NwNw`. This function is documented on page ??.)

```

\_fp_ln_exponent:wn \_fp_ln_exponent:wn {\langle s_1 \rangle} {\langle s_2 \rangle} {\langle s_3 \rangle} {\langle s_4 \rangle} {\langle s_5 \rangle} {\langle s_6 \rangle} ;
{\langle exponent \rangle}

```

Compute $\langle exponent \rangle$ times $\ln(10)$. Apart from the cases where $\langle exponent \rangle$ is 0 or 1, the result 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.

```

14441 \cs_new:Npn \_fp_ln_exponent:wn #1; #2
14442 {

```

¹⁴Bruno: that was wrong at some point, I must check.

¹⁵Bruno: this *must* be updated with correct values!

```

14443 \if_case:w #2 \exp_stop_f:
14444 \c_zero \__fp_case_return:nw { \__fp_fixed_to_float:Nw 2 }
14445 \or:
14446 \exp_after:wN \__fp_ln_exponent_one:ww \__int_value:w
14447 \else:
14448 \if_int_compare:w #2 > \c_zero
14449 \exp_after:wN \__fp_ln_exponent_small:NNww
14450 \exp_after:wN 0
14451 \exp_after:wN \__fp_fixed_sub:wwn \__int_value:w
14452 \else:
14453 \exp_after:wN \__fp_ln_exponent_small:NNww
14454 \exp_after:wN 2
14455 \exp_after:wN \__fp_fixed_add:wwn \__int_value:w -
14456 \fi:
14457 \fi:
14458 #2; #1;
14459 }

```

Now we painfully write all the cases.¹⁶ No overflow nor underflow can happen, except when computing $\ln(1)$.

```

14460 \cs_new:Npn \__fp_ln_exponent_one:ww #1; #1;
14461 {
14462 \c_zero
14463 \exp_after:wN \__fp_fixed_sub:wwn \c__fp_ln_x_fixed_t1 ; #1;
14464 \__fp_fixed_to_float:wN 0
14465 }

```

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.

```

14466 \cs_new:Npn \__fp_ln_exponent_small:NNww #1#2#3; #4#5#6#7#8#9;
14467 {
14468 \c_four
14469 \exp_after:wN \__fp_fixed_mul:wwn
14470 \c__fp_ln_x_fixed_t1 ;
14471 {#3}{0000}{0000}{0000}{0000}{0000} ;
14472 #2
14473 {0000}{#4}{#5}{#6}{#7}{#8};
14474 \__fp_fixed_to_float:wN #1
14475 }

```

(End definition for `__fp_ln_exponent:wn`. This function is documented on page ??.)

29.2 Exponential

29.2.1 Sign, exponent, and special numbers

`__fp_exp_o:w`

```

14476 \cs_new:Npn \__fp_exp_o:w #1 \s__fp \__fp_chk:w #2#3#4; @

```

¹⁶Bruno: do rounding.

```

14477 {
14478     \if_case:w #2 \exp_stop_f:
14479     \__fp_case_return_o:Nw \c_one_fp
14480 \or:
14481     \exp_after:wN \__fp_exp_normal:w
14482 \or:
14483     \if_meaning:w 0 #3
14484     \exp_after:wN \__fp_case_return_o:Nw
14485     \exp_after:wN \c_inf_fp
14486 \else:
14487     \exp_after:wN \__fp_case_return_o:Nw
14488     \exp_after:wN \c_zero_fp
14489 \fi:
14490 \or:
14491     \__fp_case_return_same_o:w
14492 \fi:
14493 \s__fp \__fp_chk:w #2#3#4;
14494 }

```

(End definition for __fp_exp_o:w.)

__fp_exp_normal:w
__fp_exp_pos:Nnwnw

```

14495 \cs_new:Npn \__fp_exp_normal:w \s__fp \__fp_chk:w 1#1
14496 {
14497     \if_meaning:w 0 #1
14498     \__fp_exp_pos:Nnwnw + \__fp_fixed_to_float:wN
14499 \else:
14500     \__fp_exp_pos:Nnwnw - \__fp_fixed_inv_to_float:wN
14501 \fi:
14502 }
14503 \cs_new:Npn \__fp_exp_pos:Nnwnw #1#2#3 \fi: #4#5;
14504 {
14505     \fi:
14506     \exp_after:wN \__fp_sanitizew
14507     \exp_after:wN 0
14508     \__int_value:w #1 \__int_eval:w
14509     \if_int_compare:w #4 < - \c_eight
14510     \c_one
14511     \exp_after:wN \__fp_add_big_i_o:wNww
14512     \int_use:N \__int_eval:w \c_one - #4 ;
14513     0 {1000}{0000}{0000}{0000} ; #5;
14514     \exp:w
14515 \else:
14516     \if_int_compare:w #4 > \c_five % cf \c__fp_max_exponent_int
14517     \exp_after:wN \__fp_exp_overflow:
14518     \exp:w
14519 \else:
14520     \if_int_compare:w #4 < \c_zero
14521     \exp_after:wN \use_i:nn
14522 \else:

```

```

14523         \exp_after:wN \use_ii:nn
14524     \fi:
14525     {
14526         \c_zero
14527         \__fp_decimate:nNnnnn { - #4 }
14528         \__fp_exp_Taylor:Nnnwn
14529     }
14530     {
14531         \__fp_decimate:nNnnnn { \c_sixteen - #4 }
14532         \__fp_exp_pos_large:NnnNwn
14533     }
14534     #5
14535     {#4}
14536     #1 #2 0
14537     \exp:w
14538     \fi:
14539     \fi:
14540     \exp_after:wN \c_zero
14541 }
14542 \cs_new:Npn \__fp_exp_overflow:
14543 { + \c_two * \c__fp_max_exponent_int ; {1000} {0000} {0000} {0000} ; }

```

(End definition for __fp_exp_normal:w and __fp_exp_pos:Nnnwnw.)

```

\__fp_exp_Taylor:Nnnwn
\__fp_exp_Taylor_loop:www
\__fp_exp_Taylor_break:Nww

```

This function is called for numbers in the range $[10^{-9}, 10^{-1})$. Our only task is to compute 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.

```

14544 \cs_new:Npn \__fp_exp_Taylor:Nnnwn #1#2#3 #4; #5 #6
14545 {
14546     #6
14547     \__fp_pack_twice_four:wNNNNNNNN
14548     \__fp_pack_twice_four:wNNNNNNNN
14549     \__fp_pack_twice_four:wNNNNNNNN
14550     \__fp_exp_Taylor_ii:ww
14551     ; #2#3#4 0000 0000 ;
14552 }
14553 \cs_new:Npn \__fp_exp_Taylor_ii:ww #1; #2;
14554 { \__fp_exp_Taylor_loop:www 10 ; #1 ; #1 ; \s_stop }
14555 \cs_new:Npn \__fp_exp_Taylor_loop:www #1; #2; #3;
14556 {
14557     \if_int_compare:w #1 = \c_one
14558     \exp_after:wN \__fp_exp_Taylor_break:Nww
14559     \fi:
14560     \__fp_fixed_div_int:wwN #3 ; #1 ;
14561     \__fp_fixed_add_one:wN
14562     \__fp_fixed_mul:wwn #2 ;
14563     {
14564         \exp_after:wN \__fp_exp_Taylor_loop:www
14565         \int_use:N \__int_eval:w #1 - 1 ;

```

```

14566         #2 ;
14567     }
14568 }
14569 \cs_new:Npn \__fp_exp_Taylor_break:Nww #1 #2; #3 \s_stop
14570 { \__fp_fixed_add_one:wN #2 ; }

```

(End definition for __fp_exp_Taylor:Nnnwn.)

```

\__fp_exp_pos_large:NnnNwn
\__fp_exp_large_after:wnn
  \__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: T_EX jumps to the appropriate case, but we then close the \if_case:w “by hand”, using \or: and \fi: as delimiters.

```

14571 \cs_new:Npn \__fp_exp_pos_large:NnnNwn #1#2#3 #4#5; #6
14572 {
14573   \exp_after:wN \exp_after:wN
14574   \cs:w \__fp_exp_large_ \__int_to_roman:w #6 :wN \exp_after:wN \cs_end:
14575   \exp_after:wN \c__fp_one_fixed_tl
14576   \exp_after:wN ;
14577   \__int_value:w #3 #4 \exp_stop_f:
14578   #5 00000 ;
14579 }
14580 \cs_new:Npn \__fp_exp_large:w #1 \or: #2 \fi:
14581 { \fi: \__fp_fixed_mul:wnn #1; }
14582 \cs_new:Npn \__fp_exp_large_v:wN #1; #2
14583 {
14584   \if_case:w #2 ~ \exp_after:wN \__fp_fixed_continue:wn \or:
14585   + 4343 \__fp_exp_large:w {8806}{8182}{2566}{2921}{5872}{6150} \or:
14586   + 8686 \__fp_exp_large:w {7756}{0047}{2598}{6861}{0458}{3204} \or:
14587   + 13029 \__fp_exp_large:w {6830}{5723}{7791}{4884}{1932}{7351} \or:
14588   + 17372 \__fp_exp_large:w {6015}{5609}{3095}{3052}{3494}{7574} \or:
14589   + 21715 \__fp_exp_large:w {5297}{7951}{6443}{0315}{3251}{3576} \or:
14590   + 26058 \__fp_exp_large:w {4665}{6719}{0099}{3379}{5527}{2929} \or:
14591   + 30401 \__fp_exp_large:w {4108}{9724}{3326}{3186}{5271}{5665} \or:
14592   + 34744 \__fp_exp_large:w {3618}{6973}{3140}{0875}{3856}{4102} \or:
14593   + 39087 \__fp_exp_large:w {3186}{9209}{6113}{3900}{6705}{9685} \or:
14594   \fi:
14595   #1;
14596   \__fp_exp_large_iv:wN
14597 }
14598 \cs_new:Npn \__fp_exp_large_iv:wN #1; #2
14599 {
14600   \if_case:w #2 ~ \exp_after:wN \__fp_fixed_continue:wn \or:

```

```

14601 + 435 \__fp_exp_large:w {1970}{0711}{1401}{7046}{9938}{8888} \or:
14602 + 869 \__fp_exp_large:w {3881}{1801}{9428}{4368}{5764}{8232} \or:
14603 + 1303 \__fp_exp_large:w {7646}{2009}{8905}{4704}{8893}{1073} \or:
14604 + 1738 \__fp_exp_large:w {1506}{3559}{7005}{0524}{9009}{7592} \or:
14605 + 2172 \__fp_exp_large:w {2967}{6283}{8402}{3667}{0689}{6630} \or:
14606 + 2606 \__fp_exp_large:w {5846}{4389}{5650}{2114}{7278}{5046} \or:
14607 + 3041 \__fp_exp_large:w {1151}{7900}{5080}{6878}{2914}{4154} \or:
14608 + 3475 \__fp_exp_large:w {2269}{1083}{0850}{6857}{8724}{4002} \or:
14609 + 3909 \__fp_exp_large:w {4470}{3047}{3316}{5442}{6408}{6591} \or:
14610 \fi:
14611 #1;
14612 \__fp_exp_large_iii:wN
14613 }
14614 \cs_new:Npn \__fp_exp_large_iii:wN #1; #2
14615 {
14616 \if_case:w #2 ~ \exp_after:wN \__fp_fixed_continue:wn \or:
14617 + 44 \__fp_exp_large:w {2688}{1171}{4181}{6135}{4484}{1263} \or:
14618 + 87 \__fp_exp_large:w {7225}{9737}{6812}{5749}{2581}{7748} \or:
14619 + 131 \__fp_exp_large:w {1942}{4263}{9524}{1255}{9365}{8421} \or:
14620 + 174 \__fp_exp_large:w {5221}{4696}{8976}{4143}{9505}{8876} \or:
14621 + 218 \__fp_exp_large:w {1403}{5922}{1785}{2837}{4107}{3977} \or:
14622 + 261 \__fp_exp_large:w {3773}{0203}{0092}{9939}{8234}{0143} \or:
14623 + 305 \__fp_exp_large:w {1014}{2320}{5473}{5004}{5094}{5533} \or:
14624 + 348 \__fp_exp_large:w {2726}{3745}{7211}{2566}{5673}{6478} \or:
14625 + 391 \__fp_exp_large:w {7328}{8142}{2230}{7421}{7051}{8866} \or:
14626 \fi:
14627 #1;
14628 \__fp_exp_large_ii:wN
14629 }
14630 \cs_new:Npn \__fp_exp_large_ii:wN #1; #2
14631 {
14632 \if_case:w #2 ~ \exp_after:wN \__fp_fixed_continue:wn \or:
14633 + 5 \__fp_exp_large:w {2202}{6465}{7948}{0671}{6516}{9579} \or:
14634 + 9 \__fp_exp_large:w {4851}{6519}{5409}{7902}{7796}{9107} \or:
14635 + 14 \__fp_exp_large:w {1068}{6474}{5815}{2446}{2146}{9905} \or:
14636 + 18 \__fp_exp_large:w {2353}{8526}{6837}{0199}{8540}{7900} \or:
14637 + 22 \__fp_exp_large:w {5184}{7055}{2858}{7072}{4640}{8745} \or:
14638 + 27 \__fp_exp_large:w {1142}{0073}{8981}{5684}{2836}{6296} \or:
14639 + 31 \__fp_exp_large:w {2515}{4386}{7091}{9167}{0062}{6578} \or:
14640 + 35 \__fp_exp_large:w {5540}{6223}{8439}{3510}{0525}{7117} \or:
14641 + 40 \__fp_exp_large:w {1220}{4032}{9431}{7840}{8020}{0271} \or:
14642 \fi:
14643 #1;
14644 \__fp_exp_large_i:wN
14645 }
14646 \cs_new:Npn \__fp_exp_large_i:wN #1; #2
14647 {
14648 \if_case:w #2 ~ \exp_after:wN \__fp_fixed_continue:wn \or:
14649 + 1 \__fp_exp_large:w {2718}{2818}{2845}{9045}{2353}{6029} \or:
14650 + 1 \__fp_exp_large:w {7389}{0560}{9893}{0650}{2272}{3043} \or:

```

```

14651     + 2 \__fp_exp_large:w {2008}{5536}{9231}{8766}{7740}{9285} \or:
14652     + 2 \__fp_exp_large:w {5459}{8150}{0331}{4423}{9078}{1103} \or:
14653     + 3 \__fp_exp_large:w {1484}{1315}{9102}{5766}{0342}{1116} \or:
14654     + 3 \__fp_exp_large:w {4034}{2879}{3492}{7351}{2260}{8387} \or:
14655     + 4 \__fp_exp_large:w {1096}{6331}{5842}{8458}{5992}{6372} \or:
14656     + 4 \__fp_exp_large:w {2980}{9579}{8704}{1728}{2747}{4359} \or:
14657     + 4 \__fp_exp_large:w {8103}{0839}{2757}{5384}{0077}{1000} \or:
14658     \fi:
14659     #1;
14660     \__fp_exp_large_:wN
14661   }
14662   \cs_new:Npn \__fp_exp_large_:wN #1; #2
14663   {
14664     \if_case:w #2 ~      \exp_after:wN \__fp_fixed_continue:wn \or:
14665     + 1 \__fp_exp_large:w {1105}{1709}{1807}{5647}{6248}{1171} \or:
14666     + 1 \__fp_exp_large:w {1221}{4027}{5816}{0169}{8339}{2107} \or:
14667     + 1 \__fp_exp_large:w {1349}{8588}{0757}{6003}{1039}{8374} \or:
14668     + 1 \__fp_exp_large:w {1491}{8246}{9764}{1270}{3178}{2485} \or:
14669     + 1 \__fp_exp_large:w {1648}{7212}{7070}{0128}{1468}{4865} \or:
14670     + 1 \__fp_exp_large:w {1822}{1188}{0039}{0508}{9748}{7537} \or:
14671     + 1 \__fp_exp_large:w {2013}{7527}{0747}{0476}{5216}{2455} \or:
14672     + 1 \__fp_exp_large:w {2225}{5409}{2849}{2467}{6045}{7954} \or:
14673     + 1 \__fp_exp_large:w {2459}{6031}{1115}{6949}{6638}{0013} \or:
14674     \fi:
14675     #1;
14676     \__fp_exp_large_after:wwn
14677   }
14678   \cs_new:Npn \__fp_exp_large_after:wwn #1; #2; #3
14679   {
14680     \__fp_exp_Taylor:Nnnwn ? { } { } 0 #2; {} #3
14681     \__fp_fixed_mul:wwn #1;
14682   }

```

(End definition for __fp_exp_pos_large:NnnNwn and others.)

29.3 Power

Raising a number a to a power b leads to many distinct situations.

a^b	$-\infty$	$-y$	$-n$	± 0	$+n$	$+y$	$+\infty$	NaN
$+\infty$	+0	+0	+0	+1	$+\infty$	$+\infty$	$+\infty$	NaN
$1 < x$	+0	$+x^{-y}$	$+x^{-n}$	+1	$+x^n$	$+x^y$	$+\infty$	NaN
+1	+1	+1	+1	+1	+1	+1	+1	+1
$0 < x < 1$	$+\infty$	$+x^{-y}$	$+x^{-n}$	+1	$+x^n$	$+x^y$	+0	NaN
+0	$+\infty$	$+\infty$	$+\infty$	+1	+0	+0	+0	NaN
-0	NaN	NaN	$\pm\infty$	+1	± 0	+0	+0	NaN
$-1 < -x < 0$	NaN	NaN	$\pm x^{-n}$	+1	$\pm x^n$	NaN	+0	NaN
-1	NaN	NaN	± 1	+1	± 1	NaN	NaN	NaN
$-x < -1$	+0	NaN	$\pm x^{-n}$	+1	$\pm x^n$	NaN	NaN	NaN
$-\infty$	+0	+0	± 0	+1	$\pm\infty$	NaN	NaN	NaN
NaN	NaN	NaN	NaN	+1	NaN	NaN	NaN	NaN

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 a 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: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: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.

```

14683 \cs_new:cpn { __fp_ \iow_char:N \^ _o:ww }
14684 \s__fp \__fp_chk:w #1#2#3; \s__fp \__fp_chk:w #4#5#6;
14685 {
14686   \if_meaning:w 0 #4
14687     \__fp_case_return_o:Nw \c_one_fp
14688   \fi:
14689   \if_case:w #2 \exp_stop_f:
14690     \exp_after:wN \use_i:nn
14691   \or:
14692     \__fp_case_return_o:Nw \c_nan_fp
14693   \else:
14694     \exp_after:wN \__fp_pow_neg:www
14695     \exp:w \exp_end_continue_f:w \exp_after:wN \use:nn
14696   \fi:
14697   {
14698     \if_meaning:w 1 #1
14699     \exp_after:wN \__fp_pow_normal:ww
14700   \else:

```

```

14701         \exp_after:wN \__fp_pow_zero_or_inf:ww
14702         \fi:
14703         \s__fp \__fp_chk:w #1#2#3;
14704     }
14705     { \s__fp \__fp_chk:w #4#5#6; \s__fp \__fp_chk:w #1#2#3; }
14706     \s__fp \__fp_chk:w #4#5#6;
14707 }

```

(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 $a = \pm 0$ with $b < 0$ and we have a division by zero, or $a = \pm \infty$ and $b > 0$ and the result is also $+\infty$, but without any exception.

```

14708 \cs_new:Npn \__fp_pow_zero_or_inf:ww
14709     \s__fp \__fp_chk:w #1#2; \s__fp \__fp_chk:w #3#4
14710 {
14711     \if_meaning:w 1 #4
14712         \__fp_case_return_same_o:w
14713     \fi:
14714     \if_meaning:w #1 #4
14715         \__fp_case_return_o:Nw \c_zero_fp
14716     \fi:
14717     \if_meaning:w 0 #1
14718         \__fp_case_use:nw
14719         {
14720             \__fp_division_by_zero_o:NNww \c_inf_fp ^
14721             \s__fp \__fp_chk:w #1 #2 ;
14722         }
14723     \else:
14724         \__fp_case_return_o:Nw \c_inf_fp
14725     \fi:
14726     \s__fp \__fp_chk:w #3#4
14727 }

```

(End definition for $__fp_pow_zero_or_inf:ww$.)

$__fp_pow_normal:ww$

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 $__fp_pow_npos:ww$.

2 Return $+\infty$ or $+0$ depending on the sign of b and whether the exponent of a is positive or not.

3 Return b .

```

14728 \cs_new:Npn \__fp_pow_normal:ww
14729   \s__fp \__fp_chk:w 1 #1#2#3; \s__fp \__fp_chk:w #4#5
14730   {
14731     \if_int_compare:w \__str_if_eq_x:nn { #2 #3 }
14732       { 1 {1000} {0000} {0000} {0000} } = \c_zero
14733       \if_int_compare:w #4 #1 = 32 \exp_stop_f:
14734       \exp_after:wN \__fp_case_return_ii_o:ww
14735       \fi:
14736       \__fp_case_return_o:Nww \c_one_fp
14737     \fi:
14738     \if_case:w #4 \exp_stop_f:
14739     \or:
14740       \exp_after:wN \__fp_pow_npos:Nww
14741       \exp_after:wN #5
14742     \or:
14743       \if_meaning:w 2 #5 \exp_after:wN \reverse_if:N \fi:
14744       \if_int_compare:w #2 > \c_zero
14745         \exp_after:wN \__fp_case_return_o:Nww
14746         \exp_after:wN \c_inf_fp
14747       \else:
14748         \exp_after:wN \__fp_case_return_o:Nww
14749         \exp_after:wN \c_zero_fp
14750       \fi:
14751     \or:
14752       \__fp_case_return_ii_o:ww
14753     \fi:
14754     \s__fp \__fp_chk:w 1 #1 {#2} #3 ;
14755     \s__fp \__fp_chk:w #4 #5
14756   }

```

(End definition for __fp_pow_normal:ww.)

__fp_pow_npos: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.

```

14757 \cs_new:Npn \__fp_pow_npos:Nww #1 \s__fp \__fp_chk:w 1#2#3
14758   {
14759     \exp_after:wN \__fp_sanitizew:Nw
14760     \exp_after:wN 0
14761     \__int_value:w
14762     \if:w #1 \if_int_compare:w #3 > \c_zero 0 \else: 2 \fi:
14763     \exp_after:wN \__fp_pow_npos_aux:NNww
14764     \exp_after:wN +
14765     \exp_after:wN \__fp_fixed_to_float:wN
14766     \else:

```

```

14767         \exp_after:wN \__fp_pow_npos_aux:NNnww
14768         \exp_after:wN -
14769         \exp_after:wN \__fp_fixed_inv_to_float:wN
14770     \fi:
14771     {#3}
14772 }

```

(End definition for __fp_pow_npos: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 $-\ln(x)$.

```

14773 \cs_new:Npn \__fp_pow_npos_aux:NNnww #1#2#3#4#5; \s__fp \__fp_chk:w 1#6#7#8;
14774 {
14775     #1
14776     \__int_eval:w
14777     \__fp_ln_significand:NNNNnnnN #4#5
14778     \__fp_pow_exponent:wnN {#3}
14779     \__fp_fixed_mul:wwn #8 {0000}{0000} ;
14780     \__fp_pow_B:wwN #7;
14781     #1 #2 0 % fixed_to_float:wN
14782 }
14783 \cs_new:Npn \__fp_pow_exponent:wnN #1; #2
14784 {
14785     \if_int_compare:w #2 > \c_zero
14786         \exp_after:wN \__fp_pow_exponent:Nwnnnnnw % n\ln(10) - (-\ln(x))
14787         \exp_after:wN +
14788     \else:
14789         \exp_after:wN \__fp_pow_exponent:Nwnnnnnw % -(|n|\ln(10) + (-\ln(x)))
14790         \exp_after:wN -
14791     \fi:
14792     #2; #1;
14793 }
14794 \cs_new:Npn \__fp_pow_exponent:Nwnnnnnw #1#2; #3#4#5#6#7#8;
14795 { %^A todo: use that in ln.
14796     \exp_after:wN \__fp_fixed_mul_after:wwn
14797     \int_use:N \__int_eval:w \c__fp_leading_shift_int
14798     \exp_after:wN \__fp_pack:NNNNNw
14799     \int_use:N \__int_eval:w \c__fp_middle_shift_int
14800     #1#2*23025 - #1 #3
14801     \exp_after:wN \__fp_pack:NNNNNw
14802     \int_use:N \__int_eval:w \c__fp_middle_shift_int
14803     #1 #2*8509 - #1 #4
14804     \exp_after:wN \__fp_pack:NNNNNw
14805     \int_use:N \__int_eval:w \c__fp_middle_shift_int
14806     #1 #2*2994 - #1 #5
14807     \exp_after:wN \__fp_pack:NNNNNw
14808     \int_use:N \__int_eval:w \c__fp_middle_shift_int
14809     #1 #2*0456 - #1 #6
14810     \exp_after:wN \__fp_pack:NNNNNw
14811     \int_use:N \__int_eval:w \c__fp_trailing_shift_int

```

```

14812             #1 #2*8401 - #1 #7
14813             #1 ( #2*7991 - #8 ) / 1 0000 ; ;
14814     }
14815 \cs_new:Npn \__fp_pow_B:wwN #1#2#3#4#5#6; #7;
14816 {
14817     \if_int_compare:w #7 < \c_zero
14818     \exp_after:wN \__fp_pow_C_neg:w \__int_value:w -
14819     \else:
14820     \if_int_compare:w #7 < 22 \exp_stop_f:
14821     \exp_after:wN \__fp_pow_C_pos:w \__int_value:w
14822     \else:
14823     \exp_after:wN \__fp_pow_C_overflow:w \__int_value:w
14824     \fi:
14825     \fi:
14826     #7 \exp_after:wN ;
14827     \int_use:N \__int_eval:w 10 0000 + #1 \__int_eval_end:
14828     #2#3#4#5#6 0000 0000 0000 0000 0000 0000 ; %^A todo: how many 0?
14829 }
14830 \cs_new:Npn \__fp_pow_C_overflow:w #1; #2; #3
14831 {
14832     + \c_two * \c_fp_max_exponent_int
14833     \exp_after:wN \__fp_fixed_continue:wn \c__fp_one_fixed_tl ;
14834 }
14835 \cs_new:Npn \__fp_pow_C_neg:w #1 ; 1
14836 {
14837     \exp_after:wN \exp_after:wN \exp_after:wN \__fp_pow_C_pack:w
14838     \prg_replicate:nn {#1} {0}
14839 }
14840 \cs_new:Npn \__fp_pow_C_pos:w #1; 1
14841 { \__fp_pow_C_pos_loop:wN #1; }
14842 \cs_new:Npn \__fp_pow_C_pos_loop:wN #1; #2
14843 {
14844     \if_meaning:w 0 #1
14845     \exp_after:wN \__fp_pow_C_pack:w
14846     \exp_after:wN #2
14847     \else:
14848     \if_meaning:w 0 #2
14849     \exp_after:wN \__fp_pow_C_pos_loop:wN \__int_value:w
14850     \else:
14851     \exp_after:wN \__fp_pow_C_overflow:w \__int_value:w
14852     \fi:
14853     \__int_eval:w #1 - \c_one \exp_after:wN ;
14854     \fi:
14855 }
14856 \cs_new:Npn \__fp_pow_C_pack:w
14857 { \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, $(-0.1)**(12345.6)$ will give $+0$ rather than complaining that the sign is not defined.

```

14858 \cs_new:Npn \__fp_pow_neg:www \s__fp \__fp_chk:w #1#2; #3; #4;
14859 {
14860   \if_case:w \__fp_pow_neg_case:w #4 ;
14861     \exp_after:wN \__fp_pow_neg_aux:wNN
14862   \or:
14863     \if_int_compare:w \__int_eval:w #1 / \c_two = \c_one
14864       \__fp_invalid_operation_o:Nww ^ #3; #4;
14865       \exp:w \exp_end_continue_f:w
14866       \exp_after:wN \exp_after:wN
14867       \exp_after:wN \__fp_use_none_until_s:w
14868     \fi:
14869   \fi:
14870   \__fp_exp_after_o:w
14871   \s__fp \__fp_chk:w #1#2;
14872 }
14873 \cs_new:Npn \__fp_pow_neg_aux:wNN #1 \s__fp \__fp_chk:w #2#3
14874 {
14875   \exp_after:wN \__fp_exp_after_o:w
14876   \exp_after:wN \s__fp
14877   \exp_after:wN \__fp_chk:w
14878   \exp_after:wN #2
14879   \int_use:N \__int_eval:w \c_two - #3 \__int_eval_end:
14880 }

```

(End definition for `__fp_pow_neg:www` and `__fp_pow_neg_aux:wNN`.)

```

\__fp_pow_neg_case:w
\__fp_pow_neg_case_aux:nnnnn
\__fp_pow_neg_case_aux:NNNNNNNNw

```

This function expects a floating point number, and “returns” -1 if it is an even integer, 0 if it is an odd integer, and 1 if it is not an integer. Zeros are even, $\pm\infty$ and `nan` are non-integers. The sign of normal numbers is irrelevant to parity. If the exponent is greater than sixteen, then the number is even. If the exponent is non-positive, the number cannot be an integer. We also separate the ranges of exponent $[1, 8]$ and $[9, 16]$. In the former case, check that the last 8 digits are zero (otherwise we don’t have an integer). In both cases, consider the appropriate 8 digits, either `#4#5` or `#2#3`, remove the first few: we are then left with $\langle digit \rangle \langle digits \rangle$; which would be the digits surrounding the decimal period. If the $\langle digits \rangle$ are non-zero, the number is not an integer. Otherwise, check the parity of the $\langle digit \rangle$ and return `\c_zero` or `\c_minus_one`.

```

14881 \cs_new:Npn \__fp_pow_neg_case:w \s__fp \__fp_chk:w #1#2#3;
14882 {
14883   \if_case:w #1 \exp_stop_f:
14884     \c_minus_one
14885   \or:   \__fp_pow_neg_case_aux:nnnnn #3
14886   \else: \c_one
14887   \fi:
14888 }

```

```

14889 \cs_new:Npn \__fp_pow_neg_case_aux:nnnnn #1#2#3#4#5
14890 {
14891   \if_int_compare:w #1 > \c_eight
14892     \if_int_compare:w #1 > \c_sixteen
14893       \c_minus_one
14894     \else:
14895       \exp_after:wN \exp_after:wN
14896       \exp_after:wN \__fp_pow_neg_case_aux:NNNNNNNNw
14897       \prg_replicate:nn { \c_sixteen - #1 } { 0 } #4#5 ;
14898     \fi:
14899   \else:
14900     \if_int_compare:w #1 > \c_zero
14901       \if_int_compare:w #4#5 = \c_zero
14902         \exp_after:wN \exp_after:wN
14903         \exp_after:wN \__fp_pow_neg_case_aux:NNNNNNNNw
14904         \prg_replicate:nn { \c_eight - #1 } { 0 } #2#3 ;
14905       \else:
14906         \c_one
14907       \fi:
14908     \else:
14909       \c_one
14910     \fi:
14911   \fi:
14912 }
14913 \cs_new:Npn \__fp_pow_neg_case_aux:NNNNNNNNw #1#2#3#4#5#6#7#8#9;
14914 {
14915   \if_int_compare:w 0 #9 = \c_zero
14916     \if_int_odd:w #8 \exp_stop_f:
14917       \c_zero
14918     \else:
14919       \c_minus_one
14920     \fi:
14921   \else:
14922     \c_one
14923   \fi:
14924 }

```

(End definition for __fp_pow_neg_case:w, __fp_pow_neg_case_aux:nnnnn, and __fp_pow_neg_case_aux:NNNNNNNNw.)

```

14925 </initex | package>

```

30 l3fp-trig Implementation

```

14926 <*initex | package>

```

```

14927 <@@=fp>

```

30.1 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.

30.1.1 Filtering special cases

`__fp_sin_o:w` This function, and its analogs for `cos`, `csc`, `sec`, `tan`, and `cot` instead of `sin`, are followed either by `\use_i:nn` and a float in radians or by `\use_ii:nn` and a float in degrees. The sine of ± 0 or NaN is the same float. The sine of $\pm \infty$ raises an invalid operation exception with the appropriate function name. Otherwise, call the `trig` function to perform argument reduction and if necessary convert the reduced argument to radians. Then, `__fp_sin_series_o:NNwww` 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:wwN` which converts the result of the series to a floating point directly rather than taking its inverse, since $\sin(x) = \#3 \sin|x|$.

```

14928 \cs_new:Npn \__fp_sin_o:w #1 \s_fp \__fp_chk:w #2#3#4; @
14929 {
14930   \if_case:w #2 \exp_stop_f:
14931     \__fp_case_return_same_o:w
14932   \or: \__fp_case_use:nw
14933     {
14934       \__fp_trig:NNNNwn #1 \__fp_sin_series_o:NNwww
14935       \__fp_ep_to_float:wwN #3 \c_zero
14936     }
14937   \or: \__fp_case_use:nw
14938     { \__fp_invalid_operation_o:fw { #1 { sin } { sind } } }
14939   \else: \__fp_case_return_same_o:w
14940   \fi:
14941   \s_fp \__fp_chk:w #2 #3 #4;
14942 }

```

(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 $\cos(x) = +\sin(\pi/2 + |x|)$.

```

14943 \cs_new:Npn \__fp_cos_o:w #1 \s__fp \__fp_chk:w #2#3; @
14944 {
14945   \if_case:w #2 \exp_stop_f:
14946     \__fp_case_return_o:Nw \c_one_fp
14947   \or:   \__fp_case_use:nw
14948     {
14949       \__fp_trig:NNNNNwn #1 \__fp_sin_series_o:NNwww
14950       \__fp_ep_to_float:wwN 0 \c_two
14951     }
14952   \or:   \__fp_case_use:nw
14953     { \__fp_invalid_operation_o:fw { #1 { cos } { cosd } } }
14954   \else: \__fp_case_return_same_o:w
14955   \fi:
14956   \s__fp \__fp_chk:w #2 #3;
14957 }

```

(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 $\csc(x) = \#3(\sin|x|)^{-1}$.

```

14958 \cs_new:Npn \__fp_csc_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
14959 {
14960   \if_case:w #2 \exp_stop_f:
14961     \__fp_cot_zero_o:Nfw #3 { #1 { csc } { cscd } }
14962   \or:   \__fp_case_use:nw
14963     {
14964       \__fp_trig:NNNNNwn #1 \__fp_sin_series_o:NNwww
14965       \__fp_ep_inv_to_float:wwN #3 \c_zero
14966     }
14967   \or:   \__fp_case_use:nw
14968     { \__fp_invalid_operation_o:fw { #1 { csc } { cscd } } }
14969   \else: \__fp_case_return_same_o:w
14970   \fi:
14971   \s__fp \__fp_chk:w #2 #3 #4;
14972 }

```

(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 $\sec(x) = +1/\sin(\pi/2 + |x|)$.

```

14973 \cs_new:Npn \__fp_sec_o:w #1 \s__fp \__fp_chk:w #2#3; @
14974 {
14975   \if_case:w #2 \exp_stop_f:
14976     \__fp_case_return_o:Nw \c_one_fp
14977   \or: \__fp_case_use:nw
14978     {
14979       \__fp_trig:NNNNNwn #1 \__fp_sin_series_o:NNwww
14980       \__fp_ep_inv_to_float:wwN 0 \c_two
14981     }
14982   \or: \__fp_case_use:nw
14983     { \__fp_invalid_operation_o:fw { #1 { sec } { secd } } }
14984   \else: \__fp_case_return_same_o:w
14985   \fi:
14986   \s__fp \__fp_chk:w #2 #3;
14987 }

```

(End definition for __fp_sec_o:w.)

__fp_tan_o:w The tangent of ± 0 or NaN is the same floating point number. The tangent of $\pm\infty$ raises an invalid operation exception. Once more, the `trig` function does the argument reduction step and conversion to radians before calling `__fp_tan_series_o:NNwww`, with a sign `#3` and an initial octant of 1 (this shift is somewhat arbitrary). See `__fp_cot_o:w` for an explanation of the 0 argument.

```

14988 \cs_new:Npn \__fp_tan_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
14989 {
14990   \if_case:w #2 \exp_stop_f:
14991     \__fp_case_return_same_o:w
14992   \or: \__fp_case_use:nw
14993     {
14994       \__fp_trig:NNNNNwn #1
14995       \__fp_tan_series_o:NNwww 0 #3 \c_one
14996     }
14997   \or: \__fp_case_use:nw
14998     { \__fp_invalid_operation_o:fw { #1 { tan } { tand } } }
14999   \else: \__fp_case_return_same_o:w
15000   \fi:
15001   \s__fp \__fp_chk:w #2 #3 #4;
15002 }

```

(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.

```

15003 \cs_new:Npn \__fp_cot_o:w #1 \s__fp \__fp_chk:w #2#3#4; @

```

```

15004 {
15005   \if_case:w #2 \exp_stop_f:
15006     \__fp_cot_zero_o:Nfw #3 { #1 { cot } { cotd } }
15007   \or:   \__fp_case_use:nw
15008     {
15009       \__fp_trig:NNNNNwn #1
15010       \__fp_tan_series_o:NNwww 2 #3 \c_three
15011     }
15012   \or:   \__fp_case_use:nw
15013     { \__fp_invalid_operation_o:fw { #1 { cot } { cotd } } }
15014   \else: \__fp_case_return_same_o:w
15015   \fi:
15016   \s__fp \__fp_chk:w #2 #3 #4;
15017 }
15018 \cs_new:Npn \__fp_cot_zero_o:Nfw #1#2#3 \fi:
15019 {
15020   \fi:
15021   \token_if_eq_meaning:NNTF 0 #1
15022   { \exp_args:Nnf \__fp_division_by_zero_o:Nnw \c_inf_fp }
15023   { \exp_args:Nnf \__fp_division_by_zero_o:Nnw \c_minus_inf_fp }
15024   {#2}
15025 }

```

(End definition for __fp_cot_o:w.)

30.1.2 Distinguishing small and large arguments

__fp_trig:NNNNNwn The first argument is \use_i:nn if the operand is in radians and \use_ii:nn if it is in degrees. Arguments #2 to #5 control what trigonometric function we compute, and #6 to #8 are pieces of a normal floating point number. Call the `_series` function #2, with arguments #3, either a conversion function (__fp_ep_to_float:wN or __fp_ep_inv_to_float: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.

```

15026 \cs_new:Npn \__fp_trig:NNNNNwn #1#2#3#4#5 \s__fp \__fp_chk:w 1#6#7#8;
15027 {
15028   \exp_after:wN #2
15029   \exp_after:wN #3
15030   \exp_after:wN #4

```

```

15031 \int_use:N \__int_eval:w #5
15032 \exp_after:wN \exp_after:wN \exp_after:wN \exp_after:wN
15033 \if_int_compare:w #7 > #1 \c_zero \c_one
15034 #1 \__fp_trig_large:ww \__fp_trigd_large:ww
15035 \else:
15036 #1 \__fp_trig_small:ww \__fp_trigd_small:ww
15037 \fi:
15038 #7,#8{0000}{0000};
15039 }

```

(End definition for __fp_trig:NNNNwn.)

30.1.3 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.

```

15040 \cs_new:Npn \__fp_trig_small:ww #1,#2;
15041 { \__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.

```

15042 \cs_new:Npn \__fp_trigd_small:ww #1,#2;
15043 {
15044 \__fp_ep_mul_raw:wwwN
15045 -1,{1745}{3292}{5199}{4329}{5769}{2369}; #1,#2;
15046 \__fp_trig_small:ww
15047 }

```

(End definition for __fp_trigd_small:ww.)

30.1.4 Argument reduction in degrees

__fp_trigd_large:ww Note that $25 \times 360 = 9000$, so $10^{k+1} \equiv 10^k \pmod{360}$ for $k \geq 3$. When the exponent #1 is very large, we can thus safely replace it by 22 (or even 19). We turn the floating point number into a fixed point number with two blocks of 8 digits followed by five blocks of 4 digits. The original float is $100 \times \langle block_1 \rangle \cdots \langle block_3 \rangle . \langle block_4 \rangle \cdots \langle block_7 \rangle$, or is equal to it modulo 360 if the exponent #1 is very large. The first auxiliary finds $\langle block_1 \rangle + \langle block_2 \rangle \pmod{9}$, a single digit, and prepends it to the 4 digits of $\langle block_3 \rangle$. It also unpacks $\langle block_4 \rangle$ and grabs the 4 digits of $\langle block_7 \rangle$. The second auxiliary grabs the $\langle block_3 \rangle$ plus any contribution from the first two blocks as #1, the first digit of $\langle block_4 \rangle$ (just after the decimal point in hundreds of degrees) as #2, and the three other digits as #3. It finds the quotient and remainder of #1#2 modulo 9, adds twice the quotient to the integer expression for the octant, and places the remainder (between 0 and 8) before #3 to form

a new $\langle block_4 \rangle$. The resulting fixed point number is $x \in [0, 0.9]$. If $x \geq 0.45$, we add 1 to the octant and feed $0.9 - x$ with an exponent of 2 (to compensate the fact that we are working in units of hundreds of degrees rather than degrees) to `_fp_trigd_small:ww`. Otherwise, we feed it x with an exponent of 2. The third auxiliary also discards digits which were not packed into the various $\langle blocks \rangle$. Since the original exponent #1 is at least 2, those are all 0 and no precision is lost (#6 and #7 are four 0 each).

```

15048 \cs_new:Npn \_fp_trigd_large:ww #1, #2#3#4#5#6#7;
15049 {
15050   \exp_after:wN \_fp_pack_eight:wNNNNNNNN
15051   \exp_after:wN \_fp_pack_eight:wNNNNNNNN
15052   \exp_after:wN \_fp_pack_twice_four:wNNNNNNNN
15053   \exp_after:wN \_fp_pack_twice_four:wNNNNNNNN
15054   \exp_after:wN \_fp_trigd_large_auxi:nnnnwNNNN
15055   \exp_after:wN ;
15056   \exp:w \exp_end_continue_f:w
15057   \prg_replicate:nn { \int_max:nn { 22 - #1 } { 0 } } { 0 }
15058   #2#3#4#5#6#7 0000 0000 0000 !
15059 }
15060 \cs_new:Npn \_fp_trigd_large_auxi:nnnnwNNNN #1#2#3#4#5; #6#7#8#9
15061 {
15062   \exp_after:wN \_fp_trigd_large_auxii:wNw
15063   \int_use:N \_int_eval:w #1 + #2
15064   - (#1 + #2 - \c_four) / \c_nine * \c_nine \_int_eval_end:
15065   #3;
15066   #4; #5{#6#7#8#9};
15067 }
15068 \cs_new:Npn \_fp_trigd_large_auxii:wNw #1; #2#3;
15069 {
15070   + (#1#2 - \c_four) / \c_nine * \c_two
15071   \exp_after:wN \_fp_trigd_large_auxiii:www
15072   \int_use:N \_int_eval:w #1#2
15073   - (#1#2 - \c_four) / \c_nine * \c_nine \_int_eval_end: #3 ;
15074 }
15075 \cs_new:Npn \_fp_trigd_large_auxiii:www #1; #2; #3!
15076 {
15077   \if_int_compare:w #1 < 4500 \exp_stop_f:
15078   \exp_after:wN \_fp_use_i_until_s:nw
15079   \exp_after:wN \_fp_fixed_continue:wn
15080   \else:
15081     + \c_one
15082   \fi:
15083   \_fp_fixed_sub:wwn {9000}{0000}{0000}{0000}{0000}{0000};
15084   {#1}#2{0000}{0000};
15085   { \_fp_trigd_small:ww 2, }
15086 }

```

(End definition for `_fp_trigd_large:ww` and others.)

30.1.5 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.

```

15087 \cs_new_nopar:Npx \_fp_trig_inverse_two_pi:
15088 {
15089   \exp_not:n { \exp_after:wN \use_none:n \token_to_str:N }
15090   \cs:w , , !
15091   0000000000000000159154943091895335768883763372514362034459645740 ~
15092   4564487476673440588967976342265350901138027662530859560728427267 ~
15093   5795803689291184611457865287796741073169983922923996693740907757 ~
15094   3077746396925307688717392896217397661693362390241723629011832380 ~
15095   1142226997557159404618900869026739561204894109369378440855287230 ~
15096   9994644340024867234773945961089832309678307490616698646280469944 ~
15097   8652187881574786566964241038995874139348609983868099199962442875 ~
15098   5851711788584311175187671605465475369880097394603647593337680593 ~
15099   0249449663530532715677550322032477781639716602294674811959816584 ~
15100   0606016803035998133911987498832786654435279755070016240677564388 ~
15101   8495713108801221993761476813777647378906330680464579784817613124 ~
15102   2731406996077502450029775985708905690279678513152521001631774602 ~
15103   0924811606240561456203146484089248459191435211575407556200871526 ~
15104   6068022171591407574745827225977462853998751553293908139817724093 ~
15105   5825479707332871904069997590765770784934703935898280871734256403 ~
15106   6689511662545705943327631268650026122717971153211259950438667945 ~
15107   0376255608363171169525975812822494162333431451061235368785631136 ~
15108   3669216714206974696012925057833605311960859450983955671870995474 ~

```

15109 6510431623815517580839442979970999505254387566129445883306846050 ~
15110 7852915151410404892988506388160776196993073410389995786918905980 ~
15111 9373777206187543222718930136625526123878038753888110681406765434 ~
15112 0828278526933426799556070790386060352738996245125995749276297023 ~
15113 5940955843011648296411855777124057544494570217897697924094903272 ~
15114 9477021664960356531815354400384068987471769158876319096650696440 ~
15115 4776970687683656778104779795450353395758301881838687937766124814 ~
15116 9530599655802190835987510351271290432315804987196868777594656634 ~
15117 6221034204440855497850379273869429353661937782928735937843470323 ~
15118 0237145837923557118636341929460183182291964165008783079331353497 ~
15119 7909974586492902674506098936890945883050337030538054731232158094 ~
15120 3197676032283131418980974982243833517435698984750103950068388003 ~
15121 9786723599608024002739010874954854787923568261139948903268997427 ~
15122 0834961149208289037767847430355045684560836714793084567233270354 ~
15123 8539255620208683932409956221175331839402097079357077496549880868 ~
15124 6066360968661967037474542102831219251846224834991161149566556037 ~
15125 9696761399312829960776082779901007830360023382729879085402387615 ~
15126 5744543092601191005433799838904654921248295160707285300522721023 ~
15127 6017523313173179759311050328155109373913639645305792607180083617 ~
15128 9548767246459804739772924481092009371257869183328958862839904358 ~
15129 6866663975673445140950363732719174311388066383072592302759734506 ~
15130 0548212778037065337783032170987734966568490800326988506741791464 ~
15131 6835082816168533143361607309951498531198197337584442098416559541 ~
15132 5225064339431286444038388356150879771645017064706751877456059160 ~
15133 8716857857939226234756331711132998655941596890719850688744230057 ~
15134 5191977056900382183925622033874235362568083541565172971088117217 ~
15135 9593683256488518749974870855311659830610139214454460161488452770 ~
15136 2511411070248521739745103866736403872860099674893173561812071174 ~
15137 0478899368886556923078485023057057144063638632023685201074100574 ~
15138 8592281115721968003978247595300166958522123034641877365043546764 ~
15139 6456565971901123084767099309708591283646669191776938791433315566 ~
15140 5066981321641521008957117286238426070678451760111345080069947684 ~
15141 2235698962488051577598095339708085475059753626564903439445420581 ~
15142 7886435683042000315095594743439252544850674914290864751442303321 ~
15143 3324569511634945677539394240360905438335528292434220349484366151 ~
15144 4663228602477666660495314065734357553014090827988091478669343492 ~
15145 2737602634997829957018161964321233140475762897484082891174097478 ~
15146 2637899181699939487497715198981872666294601830539583275209236350 ~
15147 6853889228468247259972528300766856937583659722919824429747406163 ~
15148 8183113958306744348516928597383237392662402434501997809940402189 ~
15149 6134834273613676449913827154166063424829363741850612261086132119 ~
15150 9863346284709941839942742955915628333990480382117501161211667205 ~
15151 1912579303552929241134403116134112495318385926958490443846807849 ~
15152 0973982808855297045153053991400988698840883654836652224668624087 ~
15153 2540140400911787421220452307533473972538149403884190586842311594 ~
15154 6322744339066125162393106283195323883392131534556381511752035108 ~
15155 7459558201123754359768155340187407394340363397803881721004531691 ~
15156 8295194879591767395417787924352761740724605939160273228287946819 ~
15157 3649128949714953432552723591659298072479985806126900733218844526 ~
15158 7943350455801952492566306204876616134365339920287545208555344144 ~

15159 0990512982727454659118132223284051166615650709837557433729548631 ~
15160 2041121716380915606161165732000083306114606181280326258695951602 ~
15161 4632166138576614804719932707771316441201594960110632830520759583 ~
15162 4850305079095584982982186740289838551383239570208076397550429225 ~
15163 9847647071016426974384504309165864528360324933604354657237557916 ~
15164 1366324120457809969715663402215880545794313282780055246132088901 ~
15165 8742121092448910410052154968097113720754005710963406643135745439 ~
15166 9159769435788920793425617783022237011486424925239248728713132021 ~
15167 7667360756645598272609574156602343787436291321097485897150713073 ~
15168 9104072643541417970572226547980381512759579124002534468048220261 ~
15169 7342299001020483062463033796474678190501811830375153802879523433 ~
15170 4195502135689770912905614317878792086205744999257897569018492103 ~
15171 2420647138519113881475640209760554895793785141404145305151583964 ~
15172 2823265406020603311891586570272086250269916393751527887360608114 ~
15173 5569484210322407772727421651364234366992716340309405307480652685 ~
15174 0930165892136921414312937134106157153714062039784761842650297807 ~
15175 8606266969960809184223476335047746719017450451446166382846208240 ~
15176 8673595102371302904443779408535034454426334130626307459513830310 ~
15177 2293146934466832851766328241515210179422644395718121717021756492 ~
15178 1964449396532222187658488244511909401340504432139858628621083179 ~
15179 3939608443898019147873897723310286310131486955212620518278063494 ~
15180 5711866277825659883100535155231665984394090221806314454521212978 ~
15181 9734471488741258268223860236027109981191520568823472398358013366 ~
15182 0683786328867928619732367253606685216856320119489780733958419190 ~
15183 6659583867852941241871821727987506103946064819585745620060892122 ~
15184 8416394373846549589932028481236433466119707324309545859073361878 ~
15185 6290631850165106267576851216357588696307451999220010776676830946 ~
15186 9814975622682434793671310841210219520899481912444048751171059184 ~
15187 4139907889455775184621619041530934543802808938628073237578615267 ~
15188 7971143323241969857805637630180884386640607175368321362629671224 ~
15189 2609428540110963218262765120117022552929289655594608204938409069 ~
15190 0760692003954646191640021567336017909631872891998634341086903200 ~
15191 5796637103128612356988817640364252540837098108148351903121318624 ~
15192 7228181050845123690190646632235938872454630737272808789830041018 ~
15193 9485913673742589418124056729191238003306344998219631580386381054 ~
15194 2457893450084553280313511884341007373060595654437362488771292628 ~
15195 9807423539074061786905784443105274262641767830058221486462289361 ~
15196 9296692992033046693328438158053564864073184440599549689353773183 ~
15197 6726613130108623588021288043289344562140479789454233736058506327 ~
15198 0439981932635916687341943656783901281912202816229500333012236091 ~
15199 8587559201959081224153679499095448881099758919890811581163538891 ~
15200 6339402923722049848375224236209100834097566791710084167957022331 ~
15201 7897107102928884897013099533995424415335060625843921452433864640 ~
15202 3432440657317477553405404481006177612569084746461432976543900008 ~
15203 3826521145210162366431119798731902751191441213616962045693602633 ~
15204 6102355962140467029012156796418735746835873172331004745963339773 ~
15205 2477044918885134415363760091537564267438450166221393719306748706 ~
15206 2881595464819775192207710236743289062690709117919412776212245117 ~
15207 2354677115640433357720616661564674474627305622913332030953340551 ~
15208 3841718194605321501426328000879551813296754972846701883657425342 ~

```

15209 5016994231069156343106626043412205213831587971115075454063290657 ~
15210 0248488648697402872037259869281149360627403842332874942332178578 ~
15211 7750735571857043787379693402336902911446961448649769719434527467 ~
15212 4429603089437192540526658890710662062575509930379976658367936112 ~
15213 8137451104971506153783743579555867972129358764463093757203221320 ~
15214 2460565661129971310275869112846043251843432691552928458573495971 ~
15215 5042565399302112184947232132380516549802909919676815118022483192 ~
15216 5127372199792134331067642187484426215985121676396779352982985195 ~
15217 8545392106957880586853123277545433229161989053189053725391582222 ~
15218 9232597278133427818256064882333760719681014481453198336237910767 ~
15219 1255017528826351836492103572587410356573894694875444694018175923 ~
15220 0609370828146501857425324969212764624247832210765473750568198834 ~
15221 5641035458027261252285503154325039591848918982630498759115406321 ~
15222 0354263890012837426155187877318375862355175378506956599570028011 ~
15223 5841258870150030170259167463020842412449128392380525772514737141 ~
15224 2310230172563968305553583262840383638157686828464330456805994018 ~
15225 7001071952092970177990583216417579868116586547147748964716547948 ~
15226 8312140431836079844314055731179349677763739898930227765607058530 ~
15227 4083747752640947435070395214524701683884070908706147194437225650 ~
15228 2823145872995869738316897126851939042297110721350756978037262545 ~
15229 8141095038270388987364516284820180468288205829135339013835649144 ~
15230 3004015706509887926715417450706686888783438055583501196745862340 ~
15231 8059532724727843829259395771584036885940989939255241688378793572 ~
15232 7967951654076673927031256418760962190243046993485989199060012977 ~
15233 7469214532970421677817261517850653008552559997940209969455431545 ~
15234 2745856704403686680428648404512881182309793496962721836492935516 ~
15235 2029872469583299481932978335803459023227052612542114437084359584 ~
15236 9443383638388317751841160881711251279233374577219339820819005406 ~
15237 3292937775306906607415304997682647124407768817248673421685881509 ~
15238 9133422075930947173855159340808957124410634720893194912880783576 ~
15239 3115829400549708918023366596077070927599010527028150868897828549 ~
15240 4340372642729262103487013992868853550062061514343078665396085995 ~
15241 0058714939141652065302070085265624074703660736605333805263766757 ~
15242 2018839497277047222153633851135483463624619855425993871933367482 ~
15243 0422097449956672702505446423243957506869591330193746919142980999 ~
15244 3424230550172665212092414559625960554427590951996824313084279693 ~
15245 7113207021049823238195747175985519501864630940297594363194450091 ~
15246 9150616049228764323192129703446093584259267276386814363309856853 ~
15247 2786024332141052330760658841495858718197071242995959226781172796 ~
15248 4438853796763139274314227953114500064922126500133268623021550837 ~
15249 \cs_end:
15250 }

```

(End definition for _fp_trig_inverse_two_pi:.)

```

\_fp_trig_large:ww
\_fp_trig_large_auxi:wwwww
\_fp_trig_large_auxii:ww
\_fp_trig_large_auxiii:wnnnnnnnn
\_fp_trig_large_auxiv:wN

```

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.

```

15251 \cs_new:Npn \__fp_trig_large:ww #1, #2#3#4#5#6;
15252 {
15253   \exp_after:wN \__fp_trig_large_auxi:wwwww
15254   \int_use:N \__int_eval:w (#1 - 32) / 64 \exp_after:wN ,
15255   \int_use:N \__int_eval:w (#1 - 4) / 8 \exp_after:wN ,
15256   \__int_value:w #1 \__fp_trig_inverse_two_pi: ;
15257   {#2}{#3}{#4}{#5} ;
15258 }
15259 \cs_new:Npn \__fp_trig_large_auxi:wwwww #1, #2, #3, #4!
15260 {
15261   \prg_replicate:nn {#1} { \__fp_trig_large_auxii:ww }
15262   \prg_replicate:nn { #2 - #1 * \c_eight }
15263   { \__fp_trig_large_auxiii:wNNNNNNNN }
15264   \prg_replicate:nn { #3 - #2 * \c_eight }
15265   { \__fp_trig_large_auxiv:wN }
15266   \prg_replicate:nn { \c_eight } { \__fp_pack_twice_four:wNNNNNNNN }
15267   \__fp_trig_large_auxv:www
15268   ;
15269 }
15270 \cs_new:Npn \__fp_trig_large_auxii:ww #1; #2 ~ { #1; }
15271 \cs_new:Npn \__fp_trig_large_auxiii:wNNNNNNNN
15272   #1; #2#3#4#5#6#7#8#9 { #1; }
15273 \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:NNNNw

```

First come the first 64 digits of the fractional part of $10^{*1-16}/(2\pi)$, arranged in 16 blocks of 4, and ending with a semicolon. Then 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.

```

15274 \cs_new:Npn \__fp_trig_large_auxv:www #1; #2; #3;
15275 {
15276   \exp_after:wN \__fp_use_i_until_s:nw
15277   \exp_after:wN \__fp_trig_large_auxvii:w
15278   \int_use:N \__int_eval:w \c__fp_leading_shift_int
15279   \prg_replicate:nn { \c_thirteen }
15280   { \__fp_trig_large_auxvi:wNNNNNNNN }
15281   + \c__fp_trailing_shift_int - \c__fp_middle_shift_int
15282   \__fp_use_i_until_s:nw

```

```

15283         ; #3 #1 ; ;
15284     }
15285 \cs_new:Npn \__fp_trig_large_auxvi:wnnnnnnnn #1; #2#3#4#5#6#7#8#9
15286 {
15287     \exp_after:wN \__fp_trig_large_pack:NNNNNw
15288     \int_use:N \__int_eval:w \c__fp_middle_shift_int
15289         + #2*#9 + #3*#8 + #4*#7 + #5*#6
15290         #1; {#2}{#3}{#4}{#5} {#7}{#8}{#9}
15291 }
15292 \cs_new:Npn \__fp_trig_large_pack:NNNNNw #1#2#3#4#5#6;
15293 { + #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:wNNNNN
\__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.

```

15294 \cs_new:Npn \__fp_trig_large_auxvii:w #1#2#3
15295 {
15296     \exp_after:wN \__fp_trig_large_auxviii:ww
15297     \int_use:N \__int_eval:w (#1#2#3 - 62) / 125 ;
15298     #1#2#3
15299 }
15300 \cs_new:Npn \__fp_trig_large_auxviii:ww #1;
15301 {
15302     + #1
15303     \if_int_odd:w #1 \exp_stop_f:
15304         \exp_after:wN \__fp_trig_large_auxix:Nw
15305         \exp_after:wN -
15306     \else:
15307         \exp_after:wN \__fp_trig_large_auxix:Nw
15308         \exp_after:wN +
15309     \fi:
15310 }
15311 \cs_new_nopar:Npn \__fp_trig_large_auxix:Nw
15312 {
15313     \exp_after:wN \__fp_use_i_until_s:nw
15314     \exp_after:wN \__fp_trig_large_auxxi:w
15315     \int_use:N \__int_eval:w \c__fp_leading_shift_int
15316     \prg_replicate:nn { \c_thirteen }
15317     { \__fp_trig_large_auxx:wNNNNN }

```

```

15318         + \c__fp_trailing_shift_int - \c__fp_middle_shift_int
15319     ;
15320 }
15321 \cs_new:Npn \__fp_trig_large_auxx:wNNNNN #1; #2 #3#4#5#6
15322 {
15323     \exp_after:wN \__fp_trig_large_pack:NNNNNw
15324     \int_use:N \__int_eval:w \c__fp_middle_shift_int
15325     #2 \c_eight * #3#4#5#6
15326     #1; #2
15327 }
15328 \cs_new:Npn \__fp_trig_large_auxxi:w #1;
15329 {
15330     \exp_after:wN \__fp_ep_mul_raw:wwwN
15331     \int_use:N \__int_eval:w \c_zero \__fp_ep_to_ep_loop:N #1 ; ; !
15332     0,{7853}{9816}{3397}{4483}{0961}{5661};
15333     \__fp_trig_small:ww
15334 }

```

(End definition for __fp_trig_large_auxvii:w and __fp_trig_large_auxviii:w.)

30.1.6 Computing the power series

```

\__fp_sin_series_o:NNwww
\__fp_sin_series_aux_o:NNwww

```

Here we receive a conversion function __fp_ep_to_float:wwN or __fp_ep_inv_to_float:wwN, a *sign* (0 or 2), a (non-negative) *octant* delimited by a dot, a *fixed point* number delimited by a semicolon, and an extended-precision number. The auxiliary receives:

- the conversion function #1;
- the final sign, which depends on the octant #3 and the sign #2;
- the octant #3, which 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, ...}, we are near an extremum of the function and we use the series

$$\cos(x) = 1 - x^2 \left(\frac{1}{2!} - x^2 \left(\frac{1}{4!} - x^2 \left(\dots \right) \right) \right).$$

Otherwise, the series

$$\sin(x) = x \left(1 - x^2 \left(\frac{1}{3!} - x^2 \left(\frac{1}{5!} - x^2 \left(\dots \right) \right) \right) \right)$$

is used. Finally, the extended-precision number is converted to a floating point number with the given sign, and __fp_sanitize:Nw checks for overflow and underflow.

```

15335 \cs_new:Npn \__fp_sin_series_o:NNwww #1#2#3. #4;

```

```

15336 {
15337   \__fp_fixed_mul:wwn #4; #4;
15338   {
15339     \exp_after:wN \__fp_sin_series_aux_o:NNnwww
15340     \exp_after:wN #1
15341     \__int_value:w
15342     \if_int_odd:w \__int_eval:w (#3 + \c_two) / \c_four \__int_eval_end:
15343       #2
15344     \else:
15345       \if_meaning:w #2 0 2 \else: 0 \fi:
15346     \fi:
15347     {#3}
15348   }
15349 }
15350 \cs_new:Npn \__fp_sin_series_aux_o:NNnwww #1#2#3 #4; #5,#6;
15351 {
15352   \if_int_odd:w \__int_eval:w #3 / \c_two \__int_eval_end:
15353   \exp_after:wN \use_i:nn
15354   \else:
15355     \exp_after:wN \use_ii:nn
15356   \fi:
15357   { % 1/18!
15358     \__fp_fixed_mul_sub_back:wwwn {0000}{0000}{0000}{0001}{5619}{2070};
15359     #4;{0000}{0000}{0000}{0477}{9477}{3324};
15360     \__fp_fixed_mul_sub_back:wwwn #4;{0000}{0000}{0011}{4707}{4559}{7730};
15361     \__fp_fixed_mul_sub_back:wwwn #4;{0000}{0000}{2087}{6756}{9878}{6810};
15362     \__fp_fixed_mul_sub_back:wwwn #4;{0000}{0027}{5573}{1922}{3985}{8907};
15363     \__fp_fixed_mul_sub_back:wwwn #4;{0000}{2480}{1587}{3015}{8730}{1587};
15364     \__fp_fixed_mul_sub_back:wwwn #4;{0013}{8888}{8888}{8888}{8888}{8889};
15365     \__fp_fixed_mul_sub_back:wwwn #4;{0416}{6666}{6666}{6666}{6666}{6667};
15366     \__fp_fixed_mul_sub_back:wwwn #4;{5000}{0000}{0000}{0000}{0000}{0000};
15367     \__fp_fixed_mul_sub_back:wwwn#4;{10000}{0000}{0000}{0000}{0000}{0000};
15368     { \__fp_fixed_continue:wn 0, }
15369   }
15370   { % 1/17!
15371     \__fp_fixed_mul_sub_back:wwwn {0000}{0000}{0000}{0028}{1145}{7254};
15372     #4;{0000}{0000}{0000}{7647}{1637}{3182};
15373     \__fp_fixed_mul_sub_back:wwwn #4;{0000}{0000}{0160}{5904}{3836}{8216};
15374     \__fp_fixed_mul_sub_back:wwwn #4;{0000}{0002}{5052}{1083}{8544}{1719};
15375     \__fp_fixed_mul_sub_back:wwwn #4;{0000}{0275}{5731}{9223}{9858}{9065};
15376     \__fp_fixed_mul_sub_back:wwwn #4;{0001}{9841}{2698}{4126}{9841}{2698};
15377     \__fp_fixed_mul_sub_back:wwwn #4;{0083}{3333}{3333}{3333}{3333}{3333};
15378     \__fp_fixed_mul_sub_back:wwwn #4;{1666}{6666}{6666}{6666}{6666}{6667};
15379     \__fp_fixed_mul_sub_back:wwwn#4;{10000}{0000}{0000}{0000}{0000}{0000};
15380     { \__fp_ep_mul:wwwn 0, } #5,#6;
15381   }
15382   {
15383     \exp_after:wN \__fp_sanitize:Nw
15384     \exp_after:wN #2
15385     \int_use:N \__int_eval:w #1

```

```

15386     }
15387     #2
15388 }

```

(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

$$\tan(x) \simeq \frac{x(1 - x^2(a_1 - x^2(a_2 - x^2(a_3 - x^2(a_4 - x^2 a_5))))))}{1 - x^2(b_1 - x^2(b_2 - x^2(b_3 - x^2(b_4 - x^2 b_5)))}.$$

The ratio is computed by `_fp_ep_div:wwwn`, then converted to a floating point number. For octants #3 (really, quadrants) next to a pole of the functions, the fixed point numerator and denominator are exchanged before computing the ratio. Note that this `\if_int_odd:w` test relies on the fact that the octant is at least 1.

```

15389 \cs_new:Npn \_fp_tan_series_o:NNwww #1#2#3. #4;
15390 {
15391   \_fp_fixed_mul:wn #4; #4;
15392   {
15393     \exp_after:wN \_fp_tan_series_aux_o:Nnwww
15394     \_int_value:w
15395     \if_int_odd:w \_int_eval:w #3 / \c_two \_int_eval_end:
15396     \exp_after:wN \reverse_if:N
15397     \fi:
15398     \if_meaning:w #1#2 2 \else: 0 \fi:
15399     {#3}
15400   }
15401 }
15402 \cs_new:Npn \_fp_tan_series_aux_o:Nnwww #1 #2 #3; #4,#5;
15403 {
15404   \_fp_fixed_mul_sub_back:wwn {0000}{0000}{1527}{3493}{0856}{7059};
15405   #3; {0000}{0159}{6080}{0274}{5257}{6472};
15406   \_fp_fixed_mul_sub_back:wwn #3; {0002}{4571}{2320}{0157}{2558}{8481};
15407   \_fp_fixed_mul_sub_back:wwn #3; {0115}{5830}{7533}{5397}{3168}{2147};
15408   \_fp_fixed_mul_sub_back:wwn #3; {1929}{8245}{6140}{3508}{7719}{2982};
15409   \_fp_fixed_mul_sub_back:wwn #3; {10000}{0000}{0000}{0000}{0000}{0000};
15410   { \_fp_ep_mul:wwwn 0, } #4,#5;
15411   {
15412     \_fp_fixed_mul_sub_back:wwn {0000}{0007}{0258}{0681}{9408}{4706};

```

```

15413                                     #3;{0000}{2343}{7175}{1399}{6151}{7670};
15414 \__fp_fixed_mul_sub_back:wwwn #3;{0019}{2638}{4588}{9232}{8861}{3691};
15415 \__fp_fixed_mul_sub_back:wwwn #3;{0536}{6357}{0691}{4344}{6852}{4252};
15416 \__fp_fixed_mul_sub_back:wwwn #3;{5263}{1578}{9473}{6842}{1052}{6315};
15417 \__fp_fixed_mul_sub_back:wwwn#3;{10000}{0000}{0000}{0000}{0000}{0000};
15418 {
15419     \reverse_if:N \if_int_odd:w
15420     \__int_eval:w (#2 - \c_one) / \c_two \__int_eval_end:
15421     \exp_after:wN \__fp_reverse_args:Nww
15422     \fi:
15423     \__fp_ep_div:wwwn 0,
15424 }
15425 }
15426 {
15427     \exp_after:wN \__fp_sanitize:Nw
15428     \exp_after:wN #1
15429     \int_use:N \__int_eval:w \__fp_ep_to_float:wwN
15430 }
15431 #1
15432 }

```

(End definition for `__fp_tan_series_o:NNwww` and `__fp_tan_series_aux_o:Nnwww`.)

30.2 Inverse trigonometric functions

All inverse trigonometric functions (arcsine, arccosine, arctangent, arccotangent, arcsecant, 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

$$\arccos x = \text{atan}(\sqrt{1 - x^2}, x) \quad (5)$$

$$\arcsin x = \text{atan}(x, \sqrt{1 - x^2}) \quad (6)$$

$$\text{asec } x = \text{atan}(\sqrt{x^2 - 1}, 1) \quad (7)$$

$$\text{acsc } x = \text{atan}(1, \sqrt{x^2 - 1}) \quad (8)$$

$$\text{atan } x = \text{atan}(x, 1) \quad (9)$$

$$\text{acot } x = \text{atan}(1, x). \quad (10)$$

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 $\operatorname{atan} \frac{|y|}{x}$ is given by a nicely convergent Taylor series;

1 $0 < 0.41421x < |y| < x$, then $\operatorname{atan} \frac{|y|}{x} = \frac{\pi}{4} - \operatorname{atan} \frac{x-|y|}{x+|y|}$;

2 $0 < 0.41421|y| < x < |y|$, then $\operatorname{atan} \frac{|y|}{x} = \frac{\pi}{4} + \operatorname{atan} \frac{-x+|y|}{x+|y|}$;

3 $0 < x < 0.41421|y|$, then $\operatorname{atan} \frac{|y|}{x} = \frac{\pi}{2} - \operatorname{atan} \frac{x}{|y|}$;

4 $0 < -x < 0.41421|y|$, then $\operatorname{atan} \frac{|y|}{x} = \frac{\pi}{2} + \operatorname{atan} \frac{-x}{|y|}$;

5 $0 < 0.41421|y| < -x < |y|$, then $\operatorname{atan} \frac{|y|}{x} = \frac{3\pi}{4} - \operatorname{atan} \frac{x+|y|}{-x+|y|}$;

6 $0 < -0.41421x < |y| < -x$, then $\operatorname{atan} \frac{|y|}{x} = \frac{3\pi}{4} + \operatorname{atan} \frac{-x-|y|}{-x+|y|}$;

7 $0 < |y| < -0.41421x$, then $\operatorname{atan} \frac{|y|}{x} = \pi - \operatorname{atan} \frac{|y|}{-x}$.

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.

30.2.1 Arctangent and arccotangent

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: $\operatorname{atan}(y) = \operatorname{atan}(y, 1) = \operatorname{acot}(1, y)$ and $\operatorname{acot}(x) = \operatorname{atan}(1, x) = \operatorname{acot}(x, 1)$.

`__fp_atan_o:Nw`
`__fp_acot_o:Nw`
`__fp_atan_dispatch_o:NNnNw`

```

15433 \cs_new_nopar:Npn \__fp_atan_o:Nw
15434 {
15435   \__fp_atan_dispatch_o:NNnNw
15436   \__fp_acotii_o:Nww \__fp_atanii_o:Nww { atan }
15437 }
15438 \cs_new_nopar:Npn \__fp_acot_o:Nw
15439 {
15440   \__fp_atan_dispatch_o:NNnNw
15441   \__fp_atanii_o:Nww \__fp_acotii_o:Nww { acot }
15442 }
15443 \cs_new:Npn \__fp_atan_dispatch_o:NNnNw #1#2#3#4#5@
15444 {
15445   \if_case:w
15446     \__int_eval:w \__fp_array_count:n {#5} - \c_one \__int_eval_end:

```

```

15447         \exp_after:wN #1 \exp_after:wN #4 \c_one_fp #5
15448         \exp:w
15449     \or: #2 #4 #5 \exp:w
15450 \else:
15451     \_msg_kernel_expandable_error:nnnnn
15452     { kernel } { fp-num-args } { #3() } { 1 } { 2 }
15453     \exp_after:wN \c_nan_fp \exp:w
15454 \fi:
15455 \exp_after:wN \c_zero
15456 }

```

(End definition for `_fp_atan_o:Nw` and `_fp_acot_o:Nw`.)

`_fp_atanii_o:Nww` If either operand is `nan`, we return it. If both are normal, we call `_fp_atan_normal_o:NNnwNnw`. If both are zero or both infinity, we call `_fp_atan_inf_o:NNNw` with argument 2, leading to a result among $\{\pm\pi/4, \pm3\pi/4\}$ (in degrees, $\{\pm45, \pm135\}$). Otherwise, one is much bigger than the other, and we call `_fp_atan_inf_o:NNNw` with either an argument of 4, leading to the values $\pm\pi/2$ (in degrees, ±90), or 0, leading to $\{\pm0, \pm\pi\}$ (in degrees, $\{\pm0, \pm180\}$). Since $\text{acot}(x, y) = \text{atan}(y, x)$, `_fp_acotii_o:ww` simply reverses its two arguments.

```

15457 \cs_new:Npn \_fp_atanii_o:Nww
15458     #1 \s__fp \_fp_chk:w #2#3#4; \s__fp \_fp_chk:w #5
15459 {
15460     \if_meaning:w 3 #2 \_fp_case_return_i_o:ww \fi:
15461     \if_meaning:w 3 #5 \_fp_case_return_ii_o:ww \fi:
15462     \if_case:w
15463         \if_meaning:w #2 #5
15464         \if_meaning:w 1 #2 \c_ten \else: \c_zero \fi:
15465     \else:
15466         \if_int_compare:w #2 > #5 \c_one \else: \c_two \fi:
15467     \fi:
15468     \_fp_case_return:nw { \_fp_atan_inf_o:NNNw #1 #3 \c_two }
15469     \or: \_fp_case_return:nw { \_fp_atan_inf_o:NNNw #1 #3 \c_four }
15470     \or: \_fp_case_return:nw { \_fp_atan_inf_o:NNNw #1 #3 \c_zero }
15471     \fi:
15472     \_fp_atan_normal_o:NNnwNnw #1
15473     \s__fp \_fp_chk:w #2#3#4;
15474     \s__fp \_fp_chk:w #5
15475 }
15476 \cs_new:Npn \_fp_acotii_o:Nww #1#2; #3;
15477 { \_fp_atanii_o:Nww #1#3; #2; }

```

(End definition for `_fp_atanii_o:Nww` and `_fp_acotii_o:Nww`.)

`_fp_atan_inf_o:NNNw` This auxiliary is called whenever one number is ± 0 or $\pm\infty$ (and neither is `NaN`). Then the result only depends on the signs, and its value is a multiple of $\pi/4$. We use the same auxiliary as for normal numbers, `_fp_atan_combine_o:NwwwwN`, with arguments the final sign `#2`; the octant `#3`; $\text{atan } z/z = 1$ as a fixed point number; $z = 0$ as a fixed point number; and $z = 0$ as an extended-precision number. Given the values we provide, $\text{atan } z$

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.

```

15478 \cs_new:Npn \__fp_atan_inf_o:NNNw #1#2#3 \s__fp \__fp_chk:w #4#5#6;
15479 {
15480   \exp_after:wN \__fp_atan_combine_o:NwwwwwN
15481   \exp_after:wN #2
15482   \int_use:N \__int_eval:w
15483   \if_meaning:w 2 #5 \c_seven - \fi: #3 \exp_after:wN ;
15484   \c__fp_one_fixed_tl ;
15485   {0000}{0000}{0000}{0000}{0000}{0000};
15486   0,{0000}{0000}{0000}{0000}{0000}{0000}; #1
15487 }

```

(End definition for __fp_atan_inf_o:NNNw.)

__fp_atan_normal_o:NNwNnw

Here we simply reorder the floating point data into a pair of signed extended-precision numbers, that is, a sign, an exponent ending with a comma, and a six-block mantissa ending with a semi-colon. This extended precision is required by other inverse trigonometric functions, to compute things like $\text{atan}(x, \sqrt{1 - x^2})$ without intermediate rounding errors.

```

15488 \cs_new_protected:Npn \__fp_atan_normal_o:NNwNnw
15489   #1 \s__fp \__fp_chk:w 1#2#3#4; \s__fp \__fp_chk:w 1#5#6#7;
15490 {
15491   \__fp_atan_test_o:NwNwNwN
15492   #2 #3, #4{0000}{0000};
15493   #5 #6, #7{0000}{0000}; #1
15494 }

```

(End definition for __fp_atan_normal_o:NNwNnw.)

__fp_atan_test_o:NwNwNwN

This receives: the sign #1 of y , its exponent #2, its 24 digits #3 in groups of 4, and similarly for x . We prepare to call __fp_atan_combine_o:NwwwwwN 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:wNwNw after the operands have been ordered.

```

15495 \cs_new:Npn \__fp_atan_test_o:NwNwNwN #1#2,#3; #4#5,#6;
15496 {
15497   \exp_after:wN \__fp_atan_combine_o:NwwwwwN
15498   \exp_after:wN #1
15499   \int_use:N \__int_eval:w
15500   \if_meaning:w 2 #4
15501   \c_seven - \__int_eval:w
15502   \fi:
15503   \if_int_compare:w

```

```

15504         \__fp_ep_compare:www #2,#3; #5,#6; > \c_zero
15505         \c_three -
15506         \exp_after:wN \__fp_reverse_args:Nww
15507     \fi:
15508     \__fp_atan_div:wnwnw #2,#3; #5,#6;
15509 }

```

(End definition for __fp_atan_test_o:NwwNwwN.)

```

\__fp_atan_div:wnwnw
\__fp_atan_near:wwn
\__fp_atan_near_aux:wn

```

This receives two positive numbers a and b (equal to $|x|$ and $|y|$ in some order), each as an exponent and 6 blocks of 4 digits, such that $0 < a < b$. If $0.41421b < a$, the two numbers are “near”, hence the point (y, x) that we started with is closer to the diagonals $\{|y| = |x|\}$ than to the axes $\{xy = 0\}$. In that case, the octant is 1 (possibly combined with the 7– and 3– inserted earlier) and we wish to compute $\operatorname{atan} \frac{b-a}{a+b}$. Otherwise, the octant is 0 (again, combined with earlier terms) and we wish to compute $\operatorname{atan} \frac{a}{b}$. In any case, call __fp_atan_auxi:ww followed by z , as a comma-delimited exponent and a fixed point number.

```

15510 \cs_new:Npn \__fp_atan_div:wnwnw #1,#2#3; #4,#5#6;
15511 {
15512     \if_int_compare:w
15513     \__int_eval:w 41421 * #5 < #2 000
15514     \if_case:w \__int_eval:w #4 - #1 \__int_eval_end: 00 \or: 0 \fi:
15515     \exp_stop_f:
15516     \exp_after:wN \__fp_atan_near:wwn
15517     \fi:
15518     \c_zero
15519     \__fp_ep_div:wwwn #1,{#2}#3; #4,{#5}#6;
15520     \__fp_atan_auxi:ww
15521 }
15522 \cs_new:Npn \__fp_atan_near:wwn
15523     \c_zero \__fp_ep_div:wwwn #1,#2; #3,
15524     {
15525     \c_one
15526     \__fp_ep_to_fixed:wn #1 - #3, #2;
15527     \__fp_atan_near_aux:wn
15528     }
15529 \cs_new:Npn \__fp_atan_near_aux:wn #1; #2;
15530 {
15531     \__fp_fixed_add:wn #1; #2;
15532     { \__fp_fixed_sub:wn #2; #1; { \__fp_ep_div:wwwn 0, } 0, }
15533 }

```

(End definition for __fp_atan_div:wnwnw and __fp_atan_near: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.

```

15534 \cs_new:Npn \__fp_atan_auxi:ww #1,#2;
15535 { \__fp_ep_to_fixed:wn #1,#2; \__fp_atan_auxii:w #1,#2; }

```

```

15536 \cs_new:Npn \__fp_atan_auxii:w #1;
15537 {
15538   \__fp_fixed_mul:wwn #1; #1;
15539   {
15540     \__fp_atan_Taylor_loop:www 39 ;
15541     {0000}{0000}{0000}{0000}{0000}{0000} ;
15542   }
15543   ! #1;
15544 }

```

(End definition for __fp_atan_auxi:ww and __fp_atan_auxii:w.)

```

\__fp_atan_Taylor_loop:www
\__fp_atan_Taylor_break:w

```

We compute the series of $(\operatorname{atan} z)/z$. A typical intermediate stage has $\#1 = 2k - 1$, $\#2 = \frac{1}{2k+1} - z^2(\frac{1}{2k+3} - z^2(\dots - z^2\frac{1}{39}))$, and $\#3 = z^2$. To go to the next step $k \rightarrow k - 1$, we compute $\frac{1}{2k-1}$, then subtract from it z^2 times $\#2$. The loop stops when $k = 0$: then $\#2$ is $(\operatorname{atan} z)/z$, and there is a need to clean up all the unnecessary data, end the integer expression computing the octant with a semicolon, and leave the result $\#2$ afterwards.

```

15545 \cs_new:Npn \__fp_atan_Taylor_loop:www #1; #2; #3;
15546 {
15547   \if_int_compare:w #1 = \c_minus_one
15548     \__fp_atan_Taylor_break:w
15549   \fi:
15550   \exp_after:wN \__fp_fixed_div_int:wwN \c__fp_one_fixed_tl ; #1;
15551   \__fp_rrot:www \__fp_fixed_mul_sub_back:wwwn #2; #3;
15552   {
15553     \exp_after:wN \__fp_atan_Taylor_loop:www
15554     \int_use:N \__int_eval:w #1 - \c_two ;
15555   }
15556   #3;
15557 }
15558 \cs_new:Npn \__fp_atan_Taylor_break:w
15559   \fi: #1 \__fp_fixed_mul_sub_back:wwwn #2; #3 !
15560 { \fi: ; #2 ; }

```

(End definition for __fp_atan_Taylor_loop:www and __fp_atan_Taylor_break:w.)

```

\__fp_atan_combine_o:NwwwN
\__fp_atan_combine_aux:ww

```

This receives a $\langle \text{sign} \rangle$, an $\langle \text{octant} \rangle$, a fixed point value of $(\operatorname{atan} z)/z$, a fixed point number z , and another representation of z , as an $\langle \text{exponent} \rangle$ and the fixed point number $10^{-\langle \text{exponent} \rangle} z$, 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

$$\langle \text{sign} \rangle \left(\left\lceil \frac{\langle \text{octant} \rangle}{2} \right\rceil \frac{\pi}{4} + (-1)^{\langle \text{octant} \rangle} \frac{\operatorname{atan} z}{z} \cdot z \right), \quad (11)$$

multiplied by $180/\pi$ if working in degrees, and using in any case the most appropriate representation of z . The floating point result is passed to `__fp_sanitize:Nw`, which checks for overflow or underflow. If the octant is 0, leave the exponent $\#5$ for `__fp_sanitize:Nw`, and multiply $\#3 = \frac{\operatorname{atan} z}{z}$ with $\#6$, the adjusted z . Otherwise, multiply $\#3 = \frac{\operatorname{atan} z}{z}$ with $\#4 = z$, then compute the appropriate multiple of $\frac{\pi}{4}$ and add or subtract

the product $\#3 \cdot \#4$. In both cases, convert to a floating point with `__fp_fixed_to_float:wN`.

```

15561 \cs_new:Npn \__fp_atan_combine_o:NwwwwN #1 #2; #3; #4; #5,#6; #7
15562 {
15563   \exp_after:wN \__fp_sanitize:Nw
15564   \exp_after:wN #1
15565   \int_use:N \__int_eval:w
15566   \if_meaning:w 0 #2
15567     \exp_after:wN \use_i:nn
15568   \else:
15569     \exp_after:wN \use_ii:nn
15570   \fi:
15571   { #5 \__fp_fixed_mul:wwn #3; #6; }
15572   {
15573     \__fp_fixed_mul:wwn #3; #4;
15574     {
15575       \exp_after:wN \__fp_atan_combine_aux:ww
15576       \int_use:N \__int_eval:w #2 / \c_two ; #2;
15577     }
15578   }
15579   { #7 \__fp_fixed_to_float:wN \__fp_fixed_to_float_rad:wN }
15580   #1
15581 }
15582 \cs_new:Npn \__fp_atan_combine_aux:ww #1; #2;
15583 {
15584   \__fp_fixed_mul_short:wwn
15585   {7853}{9816}{3397}{4483}{0961}{5661};
15586   {#1}{0000}{0000};
15587   {
15588     \if_int_odd:w #2 \exp_stop_f:
15589     \exp_after:wN \__fp_fixed_sub:wwn
15590   \else:
15591     \exp_after:wN \__fp_fixed_add:wwn
15592   \fi:
15593   }
15594 }

```

(End definition for `__fp_atan_combine_o:NwwwwN` and `__fp_atan_combine_aux:ww`.)

30.2.2 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).

```

15595 \cs_new:Npn \__fp_asin_o:w #1 \s__fp \__fp_chk:w #2#3; @
15596 {
15597   \if_case:w #2 \exp_stop_f:

```

```

15598     \__fp_case_return_same_o:w
15599 \or:
15600     \__fp_case_use:nw
15601     { \__fp_asin_normal_o:NfwNnnnnw #1 { #1 { asin } { asind } } }
15602 \or:
15603     \__fp_case_use:nw
15604     { \__fp_invalid_operation_o:fw { #1 { asin } { asind } } }
15605 \else:
15606     \__fp_case_return_same_o:w
15607 \fi:
15608 \s__fp \__fp_chk:w #2 #3;
15609 }

```

(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.

```

15610 \cs_new:Npn \__fp_acos_o:w #1 \s__fp \__fp_chk:w #2#3; @
15611 {
15612     \if_case:w #2 \exp_stop_f:
15613     \__fp_case_use:nw { \__fp_atan_inf_o:NNNw #1 0 \c_four }
15614 \or:
15615     \__fp_case_use:nw
15616     {
15617         \__fp_asin_normal_o:NfwNnnnnw #1 { #1 { acos } { acosd } }
15618         \__fp_reverse_args:Nww
15619     }
15620 \or:
15621     \__fp_case_use:nw
15622     { \__fp_invalid_operation_o:fw { #1 { acos } { acosd } } }
15623 \else:
15624     \__fp_case_return_same_o:w
15625 \fi:
15626 \s__fp \__fp_chk:w #2 #3;
15627 }

```

(End definition for __fp_acos_o:w.)

__fp_asin_normal_o:NfwNnnnnw If the exponent #5 is strictly less than 1, the operand lies within $(-1, 1)$ and the operation is permitted: call __fp_asin_auxi_o:nNww 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:nNww. 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.

```

15628 \cs_new:Npn \__fp_asin_normal_o:NfwNnnnnw
15629     #1#2#3 \s__fp \__fp_chk:w 1#4#5#6#7#8#9;
15630 {

```

```

15631 \if_int_compare:w #5 < \c_one
15632 \exp_after:wN \__fp_use_none_until_s:w
15633 \fi:
15634 \if_int_compare:w \__int_eval:w #5 + #6#7 + #8#9 = 1000 0001 ~
15635 \exp_after:wN \__fp_use_none_until_s:w
15636 \fi:
15637 \__fp_use_i:ww
15638 \__fp_invalid_operation_o:fw {#2}
15639 \s__fp \__fp_chk:w 1#4{#5}{#6}{#7}{#8}{#9};
15640 \__fp_asin_auxi_o:NnNww
15641 #1 {#3} #4 #5,{#6}{#7}{#8}{#9}{0000}{0000};
15642 }

```

(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`.

```

15643 \cs_new:Npn \__fp_asin_auxi_o:NnNww #1#2#3#4,#5;
15644 {
15645   \__fp_ep_to_fixed:wwn #4,#5;
15646   \__fp_asin_isqrt:wn
15647   \__fp_ep_mul:wwwwn #4,#5;
15648   \__fp_ep_to_ep:wwN
15649   \__fp_fixed_continue:wn
15650   { #2 \__fp_atan_test_o:NwwNwwN #3 }
15651   0 1,{1000}{0000}{0000}{0000}{0000}{0000}; #1
15652 }
15653 \cs_new:Npn \__fp_asin_isqrt:wn #1;
15654 {
15655   \exp_after:wN \__fp_fixed_sub:wwn \c__fp_one_fixed_tl ; #1;
15656   {
15657     \__fp_fixed_add_one:wN #1;
15658     \__fp_fixed_continue:wn { \__fp_ep_mul:wwwwn 0, } 0,
15659   }
15660   \__fp_ep_isqrt:wwn
15661 }

```

(End definition for __fp_asin_auxi_o:NnNww and __fp_asin_isqrt:wn.)

30.2.3 Arc cosecant 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 arc cosecant of ± 0 raises an invalid operation exception. The arc cosecant of $\pm\infty$ is ± 0 with the same sign. The arc cosecant of NaN is itself. Otherwise, `__fp_acsc_normal_o:NfwNnw` does some more tests, keeping the function name (acsc or acscd) as an argument for invalid operation exceptions.

```

15662 \cs_new:Npn \__fp_acsc_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
15663 {
15664   \if_case:w \if_meaning:w 2 #2 #3 \fi: #2 \exp_stop_f:
15665     \__fp_case_use:nw
15666     { \__fp_invalid_operation_o:fw { #1 { acsc } { acscd } } }
15667   \or: \__fp_case_use:nw
15668     { \__fp_acsc_normal_o:NfwNnw #1 { #1 { acsc } { acscd } } }
15669   \or: \__fp_case_return_o:Nw \c_zero_fp
15670   \or: \__fp_case_return_same_o:w
15671   \else: \__fp_case_return_o:Nw \c_minus_zero_fp
15672   \fi:
15673   \s__fp \__fp_chk:w #2 #3 #4;
15674 }

```

(End definition for `__fp_acsc_o:w`.)

`__fp_asec_o:w` The arcsecant of ± 0 raises an invalid operation exception. The arcsecant of $\pm\infty$ is $\pi/2$ (in degrees, 90). The arc cosecant of NaN is itself. Otherwise, do some more tests, keeping the function name asec (or asecd) as an argument for invalid operation exceptions, and a `__fp_reverse_args:Nww` following precisely that appearing in `__fp_acos_o:w`.

```

15675 \cs_new:Npn \__fp_asec_o:w #1 \s__fp \__fp_chk:w #2#3; @
15676 {
15677   \if_case:w #2 \exp_stop_f:
15678     \__fp_case_use:nw
15679     { \__fp_invalid_operation_o:fw { #1 { asec } { asecd } } }
15680   \or:
15681     \__fp_case_use:nw
15682     {
15683       \__fp_acsc_normal_o:NfwNnw #1 { #1 { asec } { asecd } }
15684       \__fp_reverse_args:Nww
15685     }
15686   \or: \__fp_case_use:nw { \__fp_atan_inf_o:NNnw #1 0 \c_four }
15687   \else: \__fp_case_return_same_o:w
15688   \fi:
15689   \s__fp \__fp_chk:w #2 #3;
15690 }

```

(End definition for `__fp_asec_o:w`.)

`__fp_acsc_normal_o:NfwNnw` If the exponent is non-positive, the operand is less than 1 in absolute value, which is always an invalid operation: complain. Otherwise, compute the inverse of the operand,

and feed it to `__fp_asin_auxi_o:nNww` (with all the appropriate arguments). This computes what we want thanks to $\operatorname{acsc}(x) = \operatorname{asin}(1/x)$ and $\operatorname{asec}(x) = \operatorname{acos}(1/x)$.

```

15691 \cs_new:Npn \__fp_acsc_normal_o:NfwNnw #1#2#3 \s__fp \__fp_chk:w 1#4#5#6;
15692 {
15693   \int_compare:nNnTF {#5} < \c_one
15694   {
15695     \__fp_invalid_operation_o:fw {#2}
15696     \s__fp \__fp_chk:w 1#4{#5}#6;
15697   }
15698   {
15699     \__fp_ep_div:wwwn
15700     1,{1000}{0000}{0000}{0000}{0000}{0000};
15701     #5,#6{0000}{0000};
15702     { \__fp_asin_auxi_o:NnNww #1 {#3} #4 }
15703   }
15704 }

```

(End definition for `__fp_acsc_normal_o:NfwNnw`.)

```

15705 </initex | package>

```

31 13fp-convert implementation

```

15706 <*initex | package>

```

```

15707 <@@=fp>

```

31.1 Trimming trailing zeros

`__fp_trim_zeros: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.

```

15708 \cs_new:Npn \__fp_trim_zeros:w #1 ;
15709 {
15710   \__fp_trim_zeros_loop:w #1
15711   ; \__fp_trim_zeros_loop:w 0; \__fp_trim_zeros_dot:w .; \s__stop
15712 }
15713 \cs_new:Npn \__fp_trim_zeros_loop:w #1 0; #2 { #2 #1 ; #2 }
15714 \cs_new:Npn \__fp_trim_zeros_dot:w #1 .; { \__fp_trim_zeros_end:w #1 ; }
15715 \cs_new:Npn \__fp_trim_zeros_end:w #1 ; #2 \s__stop { #1 }

```

(End definition for `__fp_trim_zeros:w`.)

31.2 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:c`

`\fp_to_scientific:n`

```

15716 \cs_new:Npn \fp_to_scientific:N #1
15717 { \exp_after:wN \__fp_to_scientific_dispatch:w #1 }

```

```

15718 \cs_generate_variant:Nn \fp_to_scientific:N { c }
15719 \cs_new_nopar:Npn \fp_to_scientific:n
15720 {
15721   \exp_after:wN \__fp_to_scientific_dispatch:w
15722   \exp:w \exp_end_continue_f:w \__fp_parse:n
15723 }

```

(End definition for `\fp_to_scientific:N`, `\fp_to_scientific:c`, and `\fp_to_scientific:n`. These functions are documented on page 195.)

```

\__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. Finally trim zeros. The whole construction is within a call to `\tl_to_lowercase:n`, responsible for creating `e` with category “other”.

```

15724 \group_begin:
15725 \char_set_catcode_other:N E
15726 \tl_to_lowercase:n
15727 {
15728   \group_end:
15729   \cs_new:Npn \__fp_to_scientific_dispatch:w \s__fp \__fp_chk:w #1#2
15730   {
15731     \if_meaning:w 2 #2 \exp_after:wN - \exp:w \exp_end_continue_f:w \fi:
15732     \if_case:w #1 \exp_stop_f:
15733       \__fp_case_return:nw { 0 }
15734     \or: \exp_after:wN \__fp_to_scientific_normal:wnnnnn
15735     \or:
15736       \__fp_case_use:nw
15737       {
15738         \__fp_invalid_operation:nnw
15739         {
15740           \exp_after:wN 1
15741           \exp_after:wN E
15742           \int_use:N \c__fp_max_exponent_int
15743         }
15744         { fp_to_scientific }
15745       }
15746     \or:
15747       \__fp_case_use:nw
15748       {
15749         \__fp_invalid_operation:nnw
15750         { 0 }
15751         { fp_to_scientific }
15752       }

```

```

15753         \fi:
15754         \s__fp \__fp_chk:w #1 #2
15755     }
15756     \cs_new:Npn \__fp_to_scientific_normal:wnnnnn
15757         \s__fp \__fp_chk:w 1 #1 #2 #3#4#5#6 ;
15758     {
15759         \if_int_compare:w #2 = \c_one
15760             \exp_after:wN \__fp_to_scientific_normal:wNw
15761         \else:
15762             \exp_after:wN \__fp_to_scientific_normal:wNw
15763             \exp_after:wN E
15764             \int_use:N \__int_eval:w #2 - \c_one
15765         \fi:
15766         ; #3 #4 #5 #6 ;
15767     }
15768 }
15769 \cs_new:Npn \__fp_to_scientific_normal:wNw #1 ; #2#3;
15770 { \__fp_trim_zeros:w #2.#3 ; #1 }

```

(End definition for `__fp_to_scientific_dispatch:w`, `__fp_to_scientific_normal:wnnnnn`, and `__fp_to_scientific_normal:wNw`.)

31.3 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`

```

15771 \cs_new:Npn \fp_to_decimal:N #1
15772 { \exp_after:wN \__fp_to_decimal_dispatch:w #1 }
15773 \cs_generate_variant:Nn \fp_to_decimal:N { c }
15774 \cs_new_nopar:Npn \fp_to_decimal:n
15775 {
15776     \exp_after:wN \__fp_to_decimal_dispatch:w
15777     \exp:w \exp_end_continue_f:w \__fp_parse:n
15778 }

```

(End definition for `\fp_to_decimal:N`, `\fp_to_decimal:c`, and `\fp_to_decimal:n`. These functions are documented on page 194.)

`__fp_to_decimal_dispatch:w`
`__fp_to_decimal_normal:wnnnnn`
`__fp_to_decimal_large:Nnnw`
`__fp_to_decimal_huge:wnnnn`

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 $0.\langle zeros \rangle \langle digits \rangle$, trimmed.

```

15779 \cs_new:Npn \__fp_to_decimal_dispatch:w \s__fp \__fp_chk:w #1#2
15780 {
15781     \if_meaning:w 2 #2 \exp_after:wN - \exp:w \exp_end_continue_f:w \fi:

```

```

15782 \if_case:w #1 \exp_stop_f:
15783     \__fp_case_return:nw { 0 }
15784 \or: \exp_after:wN \__fp_to_decimal_normal:wnnnnn
15785 \or:
15786     \__fp_case_use:nw
15787     {
15788         \__fp_invalid_operation:nnw
15789         {
15790             \exp_after:wN \exp_after:wN \exp_after:wN 1
15791             \prg_replicate:nn \c__fp_max_exponent_int 0
15792         }
15793         { fp_to_decimal }
15794     }
15795 \or:
15796     \__fp_case_use:nw
15797     {
15798         \__fp_invalid_operation:nnw
15799         { 0 }
15800         { fp_to_decimal }
15801     }
15802 \fi:
15803 \s__fp \__fp_chk:w #1 #2
15804 }
15805 \cs_new:Npn \__fp_to_decimal_normal:wnnnnn
15806 \s__fp \__fp_chk:w 1 #1 #2 #3#4#5#6 ;
15807 {
15808     \int_compare:nNnTF {#2} > \c_zero
15809     {
15810         \int_compare:nNnTF {#2} < \c_sixteen
15811         {
15812             \__fp_decimate:nNnnnn { \c_sixteen - #2 }
15813             \__fp_to_decimal_large:Nnnw
15814         }
15815         {
15816             \exp_after:wN \exp_after:wN
15817             \exp_after:wN \__fp_to_decimal_huge:wnnnn
15818             \prg_replicate:nn { #2 - \c_sixteen } { 0 } ;
15819         }
15820         {#3} {#4} {#5} {#6}
15821     }
15822     {
15823         \exp_after:wN \__fp_trim_zeros:w
15824         \exp_after:wN 0
15825         \exp_after:wN .
15826         \exp:w \exp_end_continue_f:w \prg_replicate:nn { - #2 } { 0 }
15827         #3#4#5#6 ;
15828     }
15829 }
15830 \cs_new:Npn \__fp_to_decimal_large:Nnnw #1#2#3#4;
15831 {

```

```

15832     \exp_after:wN \_fp_trim_zeros:w \_int_value:w
15833     \if_int_compare:w #2 > \c_zero
15834         #2
15835     \fi:
15836     \exp_stop_f:
15837     #3.#4 ;
15838 }
15839 \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.)

31.4 Token list representation

`\fp_to_tl:N` These three public functions evaluate their argument, then pass it to `_fp_to_tl_dispatch:w`.
`\fp_to_tl:c`
`\fp_to_tl:n`

```

15840 \cs_new:Npn \fp_to_tl:N #1 { \exp_after:wN \_fp_to_tl_dispatch:w #1 }
15841 \cs_generate_variant:Nn \fp_to_tl:N { c }
15842 \cs_new_nopar:Npn \fp_to_tl:n
15843 {
15844     \exp_after:wN \_fp_to_tl_dispatch:w
15845     \exp:w \exp_end_continue_f:w \_fp_parse:n
15846 }

```

(End definition for `\fp_to_tl:N`, `\fp_to_tl:c`, and `\fp_to_tl:n`. These functions are documented on page 195.)

`_fp_to_tl_dispatch:w` 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.

```

15847 \cs_new:Npn \_fp_to_tl_dispatch:w \s_fp \_fp_chk:w #1#2
15848 {
15849     \if_meaning:w 2 #2 \exp_after:wN - \exp:w \exp_end_continue_f:w \fi:
15850     \if_case:w #1 \exp_stop_f:
15851         \_fp_case_return:nw { 0 }
15852     \or: \exp_after:wN \_fp_to_tl_normal:nnnnn
15853     \or: \_fp_case_return:nw { \tl_to_str:n {inf} }
15854     \else: \_fp_case_return:nw { \tl_to_str:n {nan} }
15855     \fi:
15856 }
15857 \cs_new:Npn \_fp_to_tl_normal:nnnnn #1
15858 {
15859     \if_int_compare:w #1 > \c_sixteen
15860         \exp_after:wN \_fp_to_scientific_normal:wnnnnn
15861     \else:
15862         \if_int_compare:w #1 < - \c_two
15863             \exp_after:wN \exp_after:wN
15864             \exp_after:wN \_fp_to_scientific_normal:wnnnnn
15865         \else:
15866             \exp_after:wN \exp_after:wN

```

```

15867         \exp_after:wN \__fp_to_decimal_normal:wnnnnn
15868         \fi:
15869     \fi:
15870     \s__fp \__fp_chk:w 1 0 {#1}
15871 }

```

(End definition for __fp_to_tl_dispatch:w and __fp_to_tl_normal:nnnnn.)

31.5 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.

31.6 Convert to dimension or integer

\fp_to_dim:N These three public functions rely on \fp_to_decimal:n internally. We make sure to produce pt with category other.

\fp_to_dim:c

\fp_to_dim:n

```

15872 \cs_new:Npx \fp_to_dim:N #1
15873 { \exp_not:N \fp_to_decimal:N #1 \tl_to_str:n {pt} }
15874 \cs_generate_variant:Nn \fp_to_dim:N { c }
15875 \cs_new:Npx \fp_to_dim:n #1
15876 { \exp_not:N \fp_to_decimal:n {#1} \tl_to_str:n {pt} }

```

(End definition for \fp_to_dim:N, \fp_to_dim:c, and \fp_to_dim:n. These functions are documented on page 195.)

\fp_to_int:N These three public functions evaluate their argument, then pass it to \fp_to_int_dispatch:w.

\fp_to_int:c

\fp_to_int:n

```

15877 \cs_new:Npn \fp_to_int:N #1 { \exp_after:wN \__fp_to_int_dispatch:w #1 }
15878 \cs_generate_variant:Nn \fp_to_int:N { c }
15879 \cs_new_nopar:Npn \fp_to_int:n
15880 {
15881     \exp_after:wN \__fp_to_int_dispatch:w
15882     \exp:w \exp_end_continue_f:w \__fp_parse:n
15883 }

```

(End definition for \fp_to_int:N, \fp_to_int:c, and \fp_to_int:n. These functions are documented on page 195.)

__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.

```

15884 \cs_new:Npn \__fp_to_int_dispatch:w #1;
15885 {
15886     \exp_after:wN \__fp_to_decimal_dispatch:w \exp:w \exp_end_continue_f:w
15887     \__fp_round:Nwn \__fp_round_to_nearest:NNN #1; { 0 }
15888 }

```

(End definition for __fp_to_int_dispatch:w.)

31.7 Convert from a dimension

`\dim_to_fp:n` 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:` in `\dim_to_fp:n`.

```

15889 \cs_new:Npn \dim_to_fp:n #1
15890 {
15891   \exp_after:wN \__fp_from_dim_test:ww
15892   \exp_after:wN 0
15893   \exp_after:wN ,
15894   \__int_value:w \etex_glueexpr:D #1 ;
15895 }
15896 \cs_new:Npn \__fp_from_dim_test:ww #1, #2
15897 {
15898   \if_meaning:w 0 #2
15899     \__fp_case_return:nw { \exp_after:wN \c_zero_fp }
15900   \else:
15901     \exp_after:wN \__fp_from_dim:wNw
15902     \int_use:N \__int_eval:w #1 - \c_four
15903     \if_meaning:w - #2
15904       \exp_after:wN , \exp_after:wN 2 \__int_value:w
15905     \else:
15906       \exp_after:wN , \exp_after:wN 0 \__int_value:w #2
15907     \fi:
15908   \fi:
15909 }
15910 \cs_new:Npn \__fp_from_dim:wNw #1,#2#3;
15911 {
15912   \__fp_pack_twice_four:wNNNNNNNN \__fp_from_dim:wNNnnnnnn ;
15913   #3 000 0000 00 {10}987654321; #2 {#1}
15914 }
15915 \cs_new:Npn \__fp_from_dim:wNNnnnnnn #1; #2#3#4#5#6#7#8#9
15916 { \__fp_from_dim:wnnnnwNn #1 {#2#300} {0000} ; }
15917 \cs_new:Npn \__fp_from_dim:wnnnnwNn #1; #2#3#4#5#6; #7#8
15918 {
15919   \__fp_mul_npos_o:Nww #7
15920   \s__fp \__fp_chk:w 1 #7 {#5} #1 ;
15921   \s__fp \__fp_chk:w 1 0 {#8} {1525} {8789} {0625} {0000} ;
15922   \prg_do_nothing:
15923 }

```

(End definition for `\dim_to_fp:n`. This function is documented on page 86.)

31.8 Use and eval

\fp_use:N Those public functions are simple copies of the decimal conversions.

```
\fp_use:c 15924 \cs_new_eq:NN \fp_use:N \fp_to_decimal:N
\fp_eval:n 15925 \cs_generate_variant:Nn \fp_use:N { c }
15926 \cs_new_eq:NN \fp_eval:n \fp_to_decimal:n
```

(End definition for \fp_use:N, \fp_use:c, and \fp_eval:n. These functions are documented on page 195.)

\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.

```
15927 \cs_new:Npn \fp_abs:n #1
15928 { \fp_to_decimal:n { abs \__fp_parse:n {#1} } }
```

(End definition for \fp_abs:n. This function is documented on page 208.)

\fp_max:nn Similar to \fp_abs:n, for consistency with \int_max:nn, etc.

```
\fp_min:nn 15929 \cs_new:Npn \fp_max:nn #1#2
15930 { \fp_to_decimal:n { max ( \__fp_parse:n {#1} , \__fp_parse:n {#2} ) } }
15931 \cs_new:Npn \fp_min:nn #1#2
15932 { \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 209.)

31.9 Convert an array of floating points to a comma list

__fp_array_to_clist:n Converts an array of floating point numbers to a comma-list. If speed here ends up irrelevant, we can simplify the code for the auxiliary to become

```
\cs_new:Npn \__fp_array_to_clist_loop:Nw #1#2;
{
  \use_none:n #1
  { , ~ } \fp_to_tl:n { #1 #2 ; }
  \__fp_array_to_clist_loop:Nw
}
```

The \use_ii:nn function is expanded after __fp_expand:n is done, and it removes ,~ from the start of the representation.

```
15933 \cs_new:Npn \__fp_array_to_clist:n #1
15934 {
15935   \tl_if_empty:nF {#1}
15936   {
15937     \__fp_expand:n
15938     {
15939       { \use_ii:nn }
15940       \__fp_array_to_clist_loop:Nw #1 { ? \__prg_break: } ;
15941       \__prg_break_point:
15942     }
15943   }
```

```

15944 }
15945 \cs_new:Npx \__fp_array_to_clist_loop:Nw #1#2;
15946 {
15947   \exp_not:N \use_none:n #1
15948   \exp_not:N \exp_after:wN
15949   {
15950     \exp_not:N \exp_after:wN ,
15951     \exp_not:N \exp_after:wN \c_space_tl
15952     \exp_not:N \exp:w
15953     \exp_not:N \exp_end_continue_f:w
15954     \exp_not:N \__fp_to_tl_dispatch:w #1 #2 ;
15955   }
15956   \exp_not:N \__fp_array_to_clist_loop:Nw
15957 }

```

(End definition for __fp_array_to_clist:n.)

```

15958 \</initex | package>

```

32 l3fp-assign implementation

```

15959 \<*initex | package>
15960 \<@@=fp>

```

32.1 Assigning values

\fp_new:N Floating point variables are initialized to be +0.

```

15961 \cs_new_protected:Npn \fp_new:N #1
15962 { \cs_new_eq:NN #1 \c_zero_fp }
15963 \cs_generate_variant:Nn \fp_new:N {c}

```

(End definition for \fp_new:N. This function is documented on page 193.)

\fp_set:Nn Simply use __fp_parse:n within various f-expanding assignments.

\fp_set:cn 15964 \cs_new_protected:Npn \fp_set:Nn #1#2

\fp_gset:Nn 15965 { \tl_set:Nx #1 { \exp_not:f { __fp_parse:n {#2} } } }

\fp_gset:cn 15966 \cs_new_protected:Npn \fp_gset:Nn #1#2

\fp_const:Nn 15967 { \tl_gset:Nx #1 { \exp_not:f { __fp_parse:n {#2} } } }

\fp_const:cn 15968 \cs_new_protected:Npn \fp_const:Nn #1#2

15969 { \tl_const:Nx #1 { \exp_not:f { __fp_parse:n {#2} } } }

15970 \cs_generate_variant:Nn \fp_set:Nn {c}

15971 \cs_generate_variant:Nn \fp_gset:Nn {c}

15972 \cs_generate_variant:Nn \fp_const:Nn {c}

(End definition for \fp_set:Nn and others. These functions are documented on page 194.)

\fp_set_eq:NN Copying a floating point is the same as copying the underlying token list.

\fp_set_eq:cN 15973 \cs_new_eq:NN \fp_set_eq:NN \tl_set_eq:NN

\fp_set_eq:Nc 15974 \cs_new_eq:NN \fp_gset_eq:NN \tl_gset_eq:NN

\fp_set_eq:cc 15975 \cs_generate_variant:Nn \fp_set_eq:NN { c , Nc , cc }

\fp_gset_eq:NN 15976 \cs_generate_variant:Nn \fp_gset_eq:NN { c , Nc , cc }

\fp_gset_eq:cN

\fp_gset_eq:Nc

\fp_gset_eq:cc

(End definition for `\fp_set_eq:NN` and others. These functions are documented on page 194.)

```

\fp_zero:N Setting a floating point to zero: copy \c_zero_fp.
\fp_zero:c 15977 \cs_new_protected:Npn \fp_zero:N #1 { \fp_set_eq:NN #1 \c_zero_fp }
\fp_gzero:N 15978 \cs_new_protected:Npn \fp_gzero:N #1 { \fp_gset_eq:NN #1 \c_zero_fp }
\fp_gzero:c 15979 \cs_generate_variant:Nn \fp_zero:N { c }
15980 \cs_generate_variant:Nn \fp_gzero:N { c }

```

(End definition for `\fp_zero:N` and others. These functions are documented on page 193.)

```

\fp_zero_new:N Set the floating point to zero, or define it if needed.
\fp_zero_new:c 15981 \cs_new_protected:Npn \fp_zero_new:N #1
\fp_gzero_new:N 15982 { \fp_if_exist:NTF #1 { \fp_zero:N #1 } { \fp_new:N #1 } }
\fp_gzero_new:c 15983 \cs_new_protected:Npn \fp_gzero_new:N #1
15984 { \fp_if_exist:NTF #1 { \fp_gzero:N #1 } { \fp_new:N #1 } }
15985 \cs_generate_variant:Nn \fp_zero_new:N { c }
15986 \cs_generate_variant:Nn \fp_gzero_new:N { c }

```

(End definition for `\fp_zero_new:N` and others. These functions are documented on page 193.)

32.2 Updating values

These match the equivalent functions in `l3int` and `l3skip`.

```

\fp_add:Nn For the sake of error recovery we should not simply set #1 to #1±(#2): for instance, if #2
\fp_add:cn is 0)+2, the parsing error would be raised at the last closing parenthesis rather than at
\fp_gadd:Nn the closing parenthesis in the user argument. Thus we evaluate #2 instead of just putting
\fp_gadd:cn parentheses. As an optimization we use \__fp_parse:n rather than \fp_eval:n, which
\fp_sub:Nn would convert the result away from the internal representation and back.
\fp_sub:cn 15987 \cs_new_protected_nopar:Npn \fp_add:Nn { \__fp_add:NNNn \fp_set:Nn + }
\fp_gsub:Nn 15988 \cs_new_protected_nopar:Npn \fp_gadd:Nn { \__fp_add:NNNn \fp_gset:Nn + }
\fp_gsub:cn 15989 \cs_new_protected_nopar:Npn \fp_sub:Nn { \__fp_add:NNNn \fp_set:Nn - }
\__fp_add:NNNn 15990 \cs_new_protected_nopar:Npn \fp_gsub:Nn { \__fp_add:NNNn \fp_gset:Nn - }
15991 \cs_new_protected:Npn \__fp_add:NNNn #1#2#3#4
15992 { #1 #3 { #3 #2 \__fp_parse:n {#4} } }
15993 \cs_generate_variant:Nn \fp_add:Nn { c }
15994 \cs_generate_variant:Nn \fp_gadd:Nn { c }
15995 \cs_generate_variant:Nn \fp_sub:Nn { c }
15996 \cs_generate_variant:Nn \fp_gsub:Nn { c }

```

(End definition for `\fp_add:Nn` and others. These functions are documented on page 194.)

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

\fp_show:n

```

15997 \cs_new_protected:Npn \fp_show:N #1
15998 {
15999     \_msg_show_variable:NNNnn #1 \fp_if_exist:NTF ? { }
16000     { > ~ \token_to_str:N #1 = \fp_to_tl:N #1 }
16001 }
16002 \cs_new_protected_nopar:Npn \fp_show:n
16003 { \_msg_show_wrap:Nn \fp_to_tl:n }
16004 \cs_generate_variant:Nn \fp_show:N { c }

```

(End definition for `\fp_show:N`, `\fp_show:c`, and `\fp_show:n`. These functions are documented on page 201.)

32.4 Some useful constants and scratch variables

\c_one_fp Some constants.

\c_e_fp

```

16005 \fp_const:Nn \c_e_fp { 2.718 2818 2845 9045 }
16006 \fp_const:Nn \c_one_fp { 1 }

```

(End definition for `\c_one_fp` and `\c_e_fp`. These variables are documented on page 199.)

\c_pi_fp We simply round π to the closest multiple of 10^{-15} .

\c_one_degree_fp

```

16007 \fp_const:Nn \c_pi_fp { 3.141 5926 5358 9793 }
16008 \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 199.)

\l_tmpa_fp Scratch variables are simply initialized there.

\l_tmpb_fp

\g_tmpa_fp

\g_tmpb_fp

```

16009 \fp_new:N \l_tmpa_fp
16010 \fp_new:N \l_tmpb_fp
16011 \fp_new:N \g_tmpa_fp
16012 \fp_new:N \g_tmpb_fp

```

(End definition for `\l_tmpa_fp` and others. These variables are documented on page 199.)

16013 `</initex | package>`

33 l3candidates Implementation

16014 `<*initex | package>`

33.1 Additions to l3basics

16015 `<@@=cs>`

\cs_log:N Use `\cs_show:N` or `\cs_show:c` after calling `_msg_log_next:` to redirect their output 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`).

\cs_log:c

```

16016 \cs_new_protected_nopar:Npn \cs_log:N
16017 { \__msg_log_next: \cs_show:N }
16018 \cs_new_protected_nopar:Npn \cs_log:c
16019 { \__msg_log_next: \cs_show:c }

```

(End definition for \cs_log:N and \cs_log:c. These functions are documented on page 212.)

__kernel_register_log:N Redirect the output of __kernel_register_show:N to the log.

```

\__kernel_register_log:c
16020 \cs_new_protected_nopar:Npn \__kernel_register_log:N
16021 { \__msg_log_next: \__kernel_register_show:N }
16022 \cs_generate_variant:Nn \__kernel_register_log:N { c }

```

(End definition for __kernel_register_log:N and __kernel_register_log:c.)

33.2 Additions to l3box

```

16023 <@@=box>

```

33.3 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.

```

16024 \fp_new:N \l__box_angle_fp

```

(End definition for \l__box_angle_fp. This variable is documented on page 215.)

\l__box_cos_fp These are used to hold the calculated sine and cosine values while carrying out a rotation.

```

\l__box_sin_fp
16025 \fp_new:N \l__box_cos_fp
16026 \fp_new:N \l__box_sin_fp

```

(End definition for \l__box_cos_fp and \l__box_sin_fp. These variables are documented on page 215.)

\l__box_top_dim These are the positions of the four edges of a box before manipulation.

```

\l__box_bottom_dim
16027 \dim_new:N \l__box_top_dim
\l__box_left_dim
16028 \dim_new:N \l__box_bottom_dim
\l__box_right_dim
16029 \dim_new:N \l__box_left_dim
16030 \dim_new:N \l__box_right_dim

```

(End definition for \l__box_top_dim and others. These variables are documented on page ??.)

\l__box_top_new_dim These are the positions of the four edges of a box after manipulation.

```

\l__box_bottom_new_dim
16031 \dim_new:N \l__box_top_new_dim
\l__box_left_new_dim
16032 \dim_new:N \l__box_bottom_new_dim
\l__box_right_new_dim
16033 \dim_new:N \l__box_left_new_dim
16034 \dim_new:N \l__box_right_new_dim

```

(End definition for \l__box_top_new_dim and others. These variables are documented on page ??.)

\l__box_internal_box Scratch space, but also needed by some parts of the driver.

```

16035 \box_new:N \l__box_internal_box

```

(End definition for \l__box_internal_box. This variable is documented on page 216.)

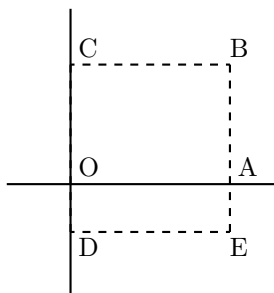


Figure 1: Co-ordinates of a box prior to rotation.

\box_rotate:Nn Rotation of a box starts with working out the relevant sine and cosine. The actual rotation is in an auxiliary to keep the flow slightly clearer

__box_rotate:N

__box_rotate_x:nnN

__box_rotate_y:nnN

__box_rotate_quadrant_one:

__box_rotate_quadrant_two:

_box_rotate_quadrant_three:

__box_rotate_quadrant_four:

```

16036 \cs_new_protected:Npn \box_rotate:Nn #1#2
16037 {
16038   \hbox_set:Nn #1
16039   {
16040     \group_begin:
16041     \fp_set:Nn \l__box_angle_fp {#2}
16042     \fp_set:Nn \l__box_sin_fp { sind ( \l__box_angle_fp ) }
16043     \fp_set:Nn \l__box_cos_fp { cosd ( \l__box_angle_fp ) }
16044     \__box_rotate:N #1
16045   \group_end:
16046   }
16047 }
```

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.

```

16048 \cs_new_protected:Npn \__box_rotate:N #1
16049 {
16050   \dim_set:Nn \l__box_top_dim { \box_ht:N #1 }
16051   \dim_set:Nn \l__box_bottom_dim { -\box_dp:N #1 }
16052   \dim_set:Nn \l__box_right_dim { \box_wd:N #1 }
16053   \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.)

```

16054 \fp_compare:nNnTF \l__box_sin_fp > \c_zero_fp
16055 {
16056   \fp_compare:nNnTF \l__box_cos_fp > \c_zero_fp
16057   { \__box_rotate_quadrant_one: }
16058   { \__box_rotate_quadrant_two: }
16059 }
16060 {
16061   \fp_compare:nNnTF \l__box_cos_fp < \c_zero_fp
16062   { \__box_rotate_quadrant_three: }
16063   { \__box_rotate_quadrant_four: }
16064 }

```

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.

```

16065 \hbox_set:Nn \l__box_internal_box { \box_use:N #1 }
16066 \hbox_set:Nn \l__box_internal_box
16067 {
16068   \tex_kern:D -\l__box_left_new_dim
16069   \hbox:n
16070   {
16071     \__driver_box_rotate_begin:
16072     \box_use:N \l__box_internal_box
16073     \__driver_box_rotate_end:
16074   }
16075 }

```

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.

```

16076 \box_set_ht:Nn \l__box_internal_box { \l__box_top_new_dim }
16077 \box_set_dp:Nn \l__box_internal_box { -\l__box_bottom_new_dim }
16078 \box_set_wd:Nn \l__box_internal_box
16079 { \l__box_right_new_dim - \l__box_left_new_dim }
16080 \box_use:N \l__box_internal_box
16081 }

```

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.

```

16082 \cs_new_protected:Npn \__box_rotate_x:nnN #1#2#3
16083 {
16084   \dim_set:Nn #3
16085   {

```

```

16086         \fp_to_dim:n
16087         {
16088             \l__box_cos_fp * \dim_to_fp:n {#1}
16089             - \l__box_sin_fp * \dim_to_fp:n {#2}
16090         }
16091     }
16092 }
16093 \cs_new_protected:Npn \__box_rotate_y:nnN #1#2#3
16094 {
16095     \dim_set:Nn #3
16096     {
16097         \fp_to_dim:n
16098         {
16099             \l__box_sin_fp * \dim_to_fp:n {#1}
16100             + \l__box_cos_fp * \dim_to_fp:n {#2}
16101         }
16102     }
16103 }

```

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.

```

16104 \cs_new_protected:Npn \__box_rotate_quadrant_one:
16105 {
16106     \__box_rotate_y:nnN \l__box_right_dim \l__box_top_dim
16107     \l__box_top_new_dim
16108     \__box_rotate_y:nnN \l__box_left_dim \l__box_bottom_dim
16109     \l__box_bottom_new_dim
16110     \__box_rotate_x:nnN \l__box_left_dim \l__box_top_dim
16111     \l__box_left_new_dim
16112     \__box_rotate_x:nnN \l__box_right_dim \l__box_bottom_dim
16113     \l__box_right_new_dim
16114 }
16115 \cs_new_protected:Npn \__box_rotate_quadrant_two:
16116 {
16117     \__box_rotate_y:nnN \l__box_right_dim \l__box_bottom_dim
16118     \l__box_top_new_dim
16119     \__box_rotate_y:nnN \l__box_left_dim \l__box_top_dim
16120     \l__box_bottom_new_dim
16121     \__box_rotate_x:nnN \l__box_right_dim \l__box_top_dim
16122     \l__box_left_new_dim
16123     \__box_rotate_x:nnN \l__box_left_dim \l__box_bottom_dim
16124     \l__box_right_new_dim
16125 }
16126 \cs_new_protected:Npn \__box_rotate_quadrant_three:
16127 {
16128     \__box_rotate_y:nnN \l__box_left_dim \l__box_bottom_dim
16129     \l__box_top_new_dim

```

```

16130 \l__box_rotate_y:nnN \l__box_right_dim \l__box_top_dim
16131 \l__box_bottom_new_dim
16132 \l__box_rotate_x:nnN \l__box_right_dim \l__box_bottom_dim
16133 \l__box_left_new_dim
16134 \l__box_rotate_x:nnN \l__box_left_dim \l__box_top_dim
16135 \l__box_right_new_dim
16136 }
16137 \cs_new_protected:Npn \l__box_rotate_quadrant_four:
16138 {
16139 \l__box_rotate_y:nnN \l__box_left_dim \l__box_top_dim
16140 \l__box_top_new_dim
16141 \l__box_rotate_y:nnN \l__box_right_dim \l__box_bottom_dim
16142 \l__box_bottom_new_dim
16143 \l__box_rotate_x:nnN \l__box_left_dim \l__box_bottom_dim
16144 \l__box_left_new_dim
16145 \l__box_rotate_x:nnN \l__box_right_dim \l__box_top_dim
16146 \l__box_right_new_dim
16147 }

```

(End definition for `\box_rotate:Nn`. This function is documented on page 214.)

`\l__box_scale_x_fp` Scaling is potentially-different in the two axes.
`\l__box_scale_y_fp`

```

16148 \fp_new:N \l__box_scale_x_fp
16149 \fp_new:N \l__box_scale_y_fp

```

(End definition for `\l__box_scale_x_fp` and `\l__box_scale_y_fp`. These variables are documented on page 216.)

`\box_resize:Nnn` Resizing a box starts by working out the various dimensions of the existing box.
`\box_resize:cnm`
`__box_resize_set_corners:N`
`__box_resize:N`
`__box_resize:NNN`

```

16150 \cs_new_protected:Npn \box_resize:Nnn #1#2#3
16151 {
16152 \hbox_set:Nn #1
16153 {
16154 \group_begin:
16155 \__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.

```

16156 \fp_set:Nn \l__box_scale_x_fp
16157 { \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.

```

16158 \fp_set:Nn \l__box_scale_y_fp
16159 {
16160 \dim_to_fp:n {#3}
16161 / \dim_to_fp:n { \l__box_top_dim - \l__box_bottom_dim }
16162 }

```

Hand off to the auxiliary which does the rest of the work.

```

16163 \__box_resize:N #1
16164 \group_end:

```

```

16165     }
16166   }
16167   \cs_generate_variant:Nn \box_resize:Nnn { c }
16168   \cs_new_protected:Npn \__box_resize_set_corners:N #1
16169   {
16170     \dim_set:Nn \l__box_top_dim { \box_ht:N #1 }
16171     \dim_set:Nn \l__box_bottom_dim { -\box_dp:N #1 }
16172     \dim_set:Nn \l__box_right_dim { \box_wd:N #1 }
16173     \dim_zero:N \l__box_left_dim
16174   }

```

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.

```

16175   \cs_new_protected:Npn \__box_resize:N #1
16176   {
16177     \__box_resize:NNN \l__box_right_new_dim
16178     \l__box_scale_x_fp \l__box_right_dim
16179     \__box_resize:NNN \l__box_bottom_new_dim
16180     \l__box_scale_y_fp \l__box_bottom_dim
16181     \__box_resize:NNN \l__box_top_new_dim
16182     \l__box_scale_y_fp \l__box_top_dim
16183     \__box_resize_common:N #1
16184   }
16185   \cs_new_protected:Npn \__box_resize:NNN #1#2#3
16186   {
16187     \dim_set:Nn #1
16188     { \fp_to_dim:n { \fp_abs:n { #2 } * \dim_to_fp:n { #3 } } }
16189   }

```

(End definition for `\box_resize:Nnn` and `\box_resize:cnn`. These functions are documented on page 213.)

`\box_resize_to_ht:Nn` 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).

```

\box_resize_to_ht:Nn 16190 \cs_new_protected:Npn \box_resize_to_ht:Nn #1#2
\box_resize_to_ht:cn 16191 {
\box_resize_to_ht_plus_dp:Nn 16192   \hbox_set:Nn #1
\box_resize_to_ht_plus_dp:cn 16193   {
\box_resize_to_wd:Nn 16194     \group_begin:
\box_resize_to_wd:cn 16195     \__box_resize_set_corners:N #1
\box_resize_to_wd_and_ht:Nnn 16196     \fp_set:Nn \l__box_scale_y_fp
\box_resize_to_wd_and_ht:cnn 16197     {
16198       \dim_to_fp:n {#2}
16199       / \dim_to_fp:n { \l__box_top_dim }
16200     }
16201     \fp_set_eq:NN \l__box_scale_x_fp \l__box_scale_y_fp

```

```

16202         \__box_resize:N #1
16203     \group_end:
16204 }
16205 }
16206 \cs_generate_variant:Nn \box_resize_to_ht:Nn { c }
16207 \cs_new_protected:Npn \box_resize_to_ht_plus_dp:Nn #1#2
16208 {
16209     \hbox_set:Nn #1
16210     {
16211         \group_begin:
16212         \__box_resize_set_corners:N #1
16213         \fp_set:Nn \l__box_scale_y_fp
16214         {
16215             \dim_to_fp:n {#2}
16216             / \dim_to_fp:n { \l__box_top_dim - \l__box_bottom_dim }
16217         }
16218         \fp_set_eq:NN \l__box_scale_x_fp \l__box_scale_y_fp
16219         \__box_resize:N #1
16220     \group_end:
16221 }
16222 }
16223 \cs_generate_variant:Nn \box_resize_to_ht_plus_dp:Nn { c }
16224 \cs_new_protected:Npn \box_resize_to_wd:Nn #1#2
16225 {
16226     \hbox_set:Nn #1
16227     {
16228         \group_begin:
16229         \__box_resize_set_corners:N #1
16230         \fp_set:Nn \l__box_scale_x_fp
16231         { \dim_to_fp:n {#2} / \dim_to_fp:n { \l__box_right_dim } }
16232         \fp_set_eq:NN \l__box_scale_y_fp \l__box_scale_x_fp
16233         \__box_resize:N #1
16234     \group_end:
16235 }
16236 }
16237 \cs_generate_variant:Nn \box_resize_to_wd:Nn { c }
16238 \cs_new_protected:Npn \box_resize_to_wd_and_ht:Nnn #1#2#3
16239 {
16240     \hbox_set:Nn #1
16241     {
16242         \group_begin:
16243         \__box_resize_set_corners:N #1
16244         \fp_set:Nn \l__box_scale_x_fp
16245         { \dim_to_fp:n {#2} / \dim_to_fp:n { \l__box_right_dim } }
16246         \fp_set:Nn \l__box_scale_y_fp
16247         {
16248             \dim_to_fp:n {#3}
16249             / \dim_to_fp:n { \l__box_top_dim }
16250         }
16251         \__box_resize:N #1

```

```

16252         \group_end:
16253     }
16254 }
16255 \cs_generate_variant:Nn \box_resize_to_wd_and_ht:Nnn { c }

```

(End definition for `\box_resize_to_ht:Nn` and `\box_resize_to_ht:cn`. These functions are documented on page 213.)

`\box_scale:Nnn` When scaling a box, setting the scaling itself is easy enough. The new dimensions are also relatively easy to find, allowing only for the need to keep them positive in all cases. Once that is done then after a check for the trivial scaling a hand-off can be made to the common code. The dimension scaling operations are carried out using the \TeX mechanism as it avoids needing to use too many `fp` operations.

`\box_scale:cn`

```

16256 \cs_new_protected:Npn \box_scale:Nnn #1#2#3
16257 {
16258     \hbox_set:Nn #1
16259     {
16260         \group_begin:
16261         \fp_set:Nn \l__box_scale_x_fp {#2}
16262         \fp_set:Nn \l__box_scale_y_fp {#3}
16263         \dim_set:Nn \l__box_top_dim { \box_ht:N #1 }
16264         \dim_set:Nn \l__box_bottom_dim { -\box_dp:N #1 }
16265         \dim_set:Nn \l__box_right_dim { \box_wd:N #1 }
16266         \dim_zero:N \l__box_left_dim
16267         \dim_set:Nn \l__box_top_new_dim
16268             { \fp_abs:n { \l__box_scale_y_fp } \l__box_top_dim }
16269         \dim_set:Nn \l__box_bottom_new_dim
16270             { \fp_abs:n { \l__box_scale_y_fp } \l__box_bottom_dim }
16271         \dim_set:Nn \l__box_right_new_dim
16272             { \fp_abs:n { \l__box_scale_x_fp } \l__box_right_dim }
16273         \__box_resize_common:N #1
16274     \group_end:
16275 }
16276 }
16277 \cs_generate_variant:Nn \box_scale:Nnn { c }

```

(End definition for `\box_scale:Nnn` and `\box_scale:cn`. These functions are documented on page 214.)

`__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.

```

16278 \cs_new_protected:Npn \__box_resize_common:N #1
16279 {
16280     \hbox_set:Nn \l__box_internal_box
16281     {
16282         \__driver_box_scale_begin:
16283         \hbox_overlap_right:n { \box_use:N #1 }
16284         \__driver_box_scale_end:
16285     }

```

The new height and depth can be applied directly.

```

16286 \fp_compare:nNnTF \l__box_scale_y_fp > \c_zero_fp
16287 {
16288   \box_set_ht:Nn \l__box_internal_box { \l__box_top_new_dim }
16289   \box_set_dp:Nn \l__box_internal_box { -\l__box_bottom_new_dim }
16290 }
16291 {
16292   \box_set_dp:Nn \l__box_internal_box { \l__box_top_new_dim }
16293   \box_set_ht:Nn \l__box_internal_box { -\l__box_bottom_new_dim }
16294 }

```

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.

```

16295 \fp_compare:nNnTF \l__box_scale_x_fp < \c_zero_fp
16296 {
16297   \hbox_to_wd:nn { \l__box_right_new_dim }
16298   {
16299     \tex_kern:D \l__box_right_new_dim
16300     \box_use:N \l__box_internal_box
16301     \tex_hss:D
16302   }
16303 }
16304 {
16305   \box_set_wd:Nn \l__box_internal_box { \l__box_right_new_dim }
16306   \hbox:n
16307   {
16308     \tex_kern:D \c_zero_dim
16309     \box_use:N \l__box_internal_box
16310     \tex_hss:D
16311   }
16312 }
16313 }

```

(End definition for __box_resize_common:N.)

33.4 Viewing part of a box

\box_clip:N A wrapper around the driver-dependent code.

```

\box_clip:c 16314 \cs_new_protected:Npn \box_clip:N #1
16315 { \hbox_set:Nn #1 { \__driver_box_use_clip:N #1 } }
16316 \cs_generate_variant:Nn \box_clip:N { c }

```

(End definition for \box_clip:N and \box_clip:c. These functions are documented on page 215.)

\box_trim:Nnnnn Trimming from the left- and right-hand edges of the box is easy: kern the appropriate parts off each side.

```

\box_trim:cnnnn 16317 \cs_new_protected:Npn \box_trim:Nnnnn #1#2#3#4#5
16318 {

```

```

16319 \hbox_set:Nn \l__box_internal_box
16320 {
16321   \tex_kern:D -\__dim_eval:w #2 \__dim_eval_end:
16322   \box_use:N #1
16323   \tex_kern:D -\__dim_eval:w #4 \__dim_eval_end:
16324 }

```

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`.

```

16325 \dim_compare:nNnTF { \box_dp:N #1 } > {#3}
16326 {
16327   \hbox_set:Nn \l__box_internal_box
16328   {
16329     \box_move_down:nn \c_zero_dim
16330     { \box_use:N \l__box_internal_box }
16331   }
16332   \box_set_dp:Nn \l__box_internal_box { \box_dp:N #1 - (#3) }
16333 }
16334 {
16335   \hbox_set:Nn \l__box_internal_box
16336   {
16337     \box_move_down:nn { #3 - \box_dp:N #1 }
16338     { \box_use:N \l__box_internal_box }
16339   }
16340   \box_set_dp:Nn \l__box_internal_box \c_zero_dim
16341 }

```

Same thing, this time from the top of the box.

```

16342 \dim_compare:nNnTF { \box_ht:N \l__box_internal_box } > {#5}
16343 {
16344   \hbox_set:Nn \l__box_internal_box
16345   {
16346     \box_move_up:nn \c_zero_dim
16347     { \box_use:N \l__box_internal_box }
16348   }
16349   \box_set_ht:Nn \l__box_internal_box
16350   { \box_ht:N \l__box_internal_box - (#5) }
16351 }
16352 {
16353   \hbox_set:Nn \l__box_internal_box
16354   {
16355     \box_move_up:nn { #5 - \box_ht:N \l__box_internal_box }
16356     { \box_use:N \l__box_internal_box }
16357   }
16358   \box_set_ht:Nn \l__box_internal_box \c_zero_dim
16359 }

```

```

16360 \box_set_eq:Nn #1 \l__box_internal_box
16361 }
16362 \cs_generate_variant:Nn \box_trim:Nnnnn { c }

```

(End definition for `\box_trim:Nnnnn` and `\box_trim:cnnnn`. These functions are documented on page 215.)

`\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.

`\box_viewport:cnnnn`

```

16363 \cs_new_protected:Npn \box_viewport:Nnnnn #1#2#3#4#5
16364 {
16365   \hbox_set:Nn \l__box_internal_box
16366   {
16367     \tex_kern:D -\dim_eval:w #2 \dim_eval_end:
16368     \box_use:N #1
16369     \tex_kern:D \dim_eval:w #4 - \box_wd:N #1 \dim_eval_end:
16370   }
16371   \dim_compare:nNnTF {#3} < \c_zero_dim
16372   {
16373     \hbox_set:Nn \l__box_internal_box
16374     {
16375       \box_move_down:nn \c_zero_dim
16376       { \box_use:N \l__box_internal_box }
16377     }
16378     \box_set_dp:Nn \l__box_internal_box { -\dim_eval:n {#3} }
16379   }
16380   {
16381     \hbox_set:Nn \l__box_internal_box
16382     { \box_move_down:nn {#3} { \box_use:N \l__box_internal_box } }
16383     \box_set_dp:Nn \l__box_internal_box \c_zero_dim
16384   }
16385   \dim_compare:nNnTF {#5} > \c_zero_dim
16386   {
16387     \hbox_set:Nn \l__box_internal_box
16388     {
16389       \box_move_up:nn \c_zero_dim
16390       { \box_use:N \l__box_internal_box }
16391     }
16392     \box_set_ht:Nn \l__box_internal_box
16393     {
16394       #5
16395       \dim_compare:nNnT {#3} > \c_zero_dim
16396       { - (#3) }
16397     }
16398   }
16399   {
16400     \hbox_set:Nn \l__box_internal_box
16401     {
16402       \box_move_up:nn { -\dim_eval:n {#5} }
16403       { \box_use:N \l__box_internal_box }

```

```

16404         }
16405         \box_set_ht:Nn \l__box_internal_box \c_zero_dim
16406     }
16407     \box_set_eq:NN #1 \l__box_internal_box
16408 }
16409 \cs_generate_variant:Nn \box_viewport:Nnnnn { c }

```

(End definition for `\box_viewport:Nnnnn` and `\box_viewport:cnnnn`. These functions are documented on page 215.)

33.5 Additions to `l3clist`

```

16410 <@@=clist>

\clist_log:N Redirect output of \clist_show:N to the log.
\clist_log:c 16411 \cs_new_protected_nopar:Npn \clist_log:N
\clist_log:n 16412 { \__msg_log_next: \clist_show:N }
16413 \cs_new_protected_nopar:Npn \clist_log:n
16414 { \__msg_log_next: \clist_show:n }
16415 \cs_generate_variant:Nn \clist_log:N { c }

```

(End definition for `\clist_log:N`, `\clist_log:c`, and `\clist_log:n`. These functions are documented on page 216.)

33.6 Additions to `l3coffins`

```

16416 <@@=coffin>

```

33.7 Rotating coffins

```

\l__coffin_sin_fp Used for rotations to get the sine and cosine values.
\l__coffin_cos_fp 16417 \fp_new:N \l__coffin_sin_fp
16418 \fp_new:N \l__coffin_cos_fp

```

(End definition for `\l__coffin_sin_fp`. This variable is documented on page ??.)

`\l__coffin_bounding_prop` A property list for the bounding box of a coffin. This is only needed during the rotation, so there is just the one.

```

16419 \prop_new:N \l__coffin_bounding_prop

```

(End definition for `\l__coffin_bounding_prop`. This variable is documented on page ??.)

`\l__coffin_bounding_shift_dim` The shift of the bounding box of a coffin from the real content.

```

16420 \dim_new:N \l__coffin_bounding_shift_dim

```

(End definition for `\l__coffin_bounding_shift_dim`. This variable is documented on page ??.)

`\l__coffin_left_corner_dim` These are used to hold maxima for the various corner values: these thus define the minimum size of the bounding box after rotation.

```

\l__coffin_right_corner_dim 16421 \dim_new:N \l__coffin_left_corner_dim
\l__coffin_bottom_corner_dim 16422 \dim_new:N \l__coffin_right_corner_dim
\l__coffin_top_corner_dim 16423 \dim_new:N \l__coffin_bottom_corner_dim
16424 \dim_new:N \l__coffin_top_corner_dim

```

(End definition for `\l__coffin_left_corner_dim`. This variable is documented on page ??.)

`\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.

```
16425 \cs_new_protected:Npn \coffin_rotate:Nn #1#2
16426 {
16427   \fp_set:Nn \l__coffin_sin_fp { sind ( #2 ) }
16428   \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.

```
16429   \prop_map_inline:cn { l__coffin_corners_ \__int_value:w #1 _prop }
16430   { \__coffin_rotate_corner:Nnnn #1 {##1} ##2 }
16431   \prop_map_inline:cn { l__coffin_poles_ \__int_value:w #1 _prop }
16432   { \__coffin_rotate_pole:Nnnnnn #1 {##1} ##2 }
```

The bounding box of the coffin needs to be rotated, and to do this the corners have to be found first. They are then rotated in the same way as the corners of the coffin material itself.

```
16433   \__coffin_set_bounding:N #1
16434   \prop_map_inline:Nn \l__coffin_bounding_prop
16435   { \__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.

```
16436   \__coffin_find_corner_maxima:N #1
16437   \__coffin_find_bounding_shift:
16438   \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.

```
16439   \hbox_set:Nn \l__coffin_internal_box
16440   {
16441     \tex_kern:D
16442     \__dim_eval:w
16443     \l__coffin_bounding_shift_dim - \l__coffin_left_corner_dim
16444     \__dim_eval_end:
16445     \box_move_down:nn { \l__coffin_bottom_corner_dim }
16446     { \box_use:N #1 }
16447   }
```

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.

```

16448 \box_set_ht:Nn \l__coffin_internal_box
16449 { \l__coffin_top_corner_dim - \l__coffin_bottom_corner_dim }
16450 \box_set_dp:Nn \l__coffin_internal_box { 0 pt }
16451 \box_set_wd:Nn \l__coffin_internal_box
16452 { \l__coffin_right_corner_dim - \l__coffin_left_corner_dim }
16453 \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.

```

16454 \prop_map_inline:cn { l__coffin_corners_ \__int_value:w #1 _prop }
16455 { \__coffin_shift_corner:Nnnn #1 {##1} ##2 }
16456 \prop_map_inline:cn { l__coffin_poles_ \__int_value:w #1 _prop }
16457 { \__coffin_shift_pole:Nnnnn #1 {##1} ##2 }
16458 }
16459 \cs_generate_variant:Nn \coffin_rotate:Nn { c }

```

(End definition for \coffin_rotate:Nn and \coffin_rotate:cn. These functions are documented on page 216.)

`__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.

```

16460 \cs_new_protected:Npn \__coffin_set_bounding:N #1
16461 {
16462   \prop_put:Nnx \l__coffin_bounding_prop { tl }
16463   { { 0 pt } { \dim_use:N \box_ht:N #1 } }
16464   \prop_put:Nnx \l__coffin_bounding_prop { tr }
16465   { { \dim_use:N \box_wd:N #1 } { \dim_use:N \box_ht:N #1 } }
16466   \dim_set:Nn \l__coffin_internal_dim { - \box_dp:N #1 }
16467   \prop_put:Nnx \l__coffin_bounding_prop { bl }
16468   { { 0 pt } { \dim_use:N \l__coffin_internal_dim } }
16469   \prop_put:Nnx \l__coffin_bounding_prop { br }
16470   { { \dim_use:N \box_wd:N #1 } { \dim_use:N \l__coffin_internal_dim } }
16471 }

```

(End definition for __coffin_set_bounding:N. This function is documented on page ??.)

`__coffin_rotate_bounding:nnn` Rotating the position of the corner of the coffin is just a case of treating this as a vector from the reference point. The same treatment is used for the corners of the material itself and the bounding box.

`__coffin_rotate_corner:Nnnn`

```

16472 \cs_new_protected:Npn \__coffin_rotate_bounding:nnn #1#2#3
16473 {
16474   \__coffin_rotate_vector:nnNN {#2} {#3} \l__coffin_x_dim \l__coffin_y_dim
16475   \prop_put:Nnx \l__coffin_bounding_prop {#1}
16476   { { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim } }
16477 }
16478 \cs_new_protected:Npn \__coffin_rotate_corner:Nnnn #1#2#3#4
16479 {
16480   \__coffin_rotate_vector:nnNN {#3} {#4} \l__coffin_x_dim \l__coffin_y_dim

```

```

16481     \prop_put:cnx { l__coffin_corners_ \__int_value:w #1 _prop } {#2}
16482     { { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim } }
16483 }

```

(End definition for __coffin_rotate_bounding:nnn. This function is documented on page ??.)

__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.

```

16484 \cs_new_protected:Npn \__coffin_rotate_pole:Nnnnnn #1#2#3#4#5#6
16485 {
16486     \__coffin_rotate_vector:nnNN {#3} {#4} \l__coffin_x_dim \l__coffin_y_dim
16487     \__coffin_rotate_vector:nnNN {#5} {#6}
16488     \l__coffin_x_prime_dim \l__coffin_y_prime_dim
16489     \__coffin_set_pole:Nnx #1 {#2}
16490     {
16491         { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim }
16492         { \dim_use:N \l__coffin_x_prime_dim }
16493         { \dim_use:N \l__coffin_y_prime_dim }
16494     }
16495 }

```

(End definition for __coffin_rotate_pole:Nnnnnn. This function is documented on page ??.)

__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.

```

16496 \cs_new_protected:Npn \__coffin_rotate_vector:nnNN #1#2#3#4
16497 {
16498     \dim_set:Nn #3
16499     {
16500         \fp_to_dim:n
16501         {
16502             \dim_to_fp:n {#1} * \l__coffin_cos_fp
16503             - \dim_to_fp:n {#2} * \l__coffin_sin_fp
16504         }
16505     }
16506     \dim_set:Nn #4
16507     {
16508         \fp_to_dim:n
16509         {
16510             \dim_to_fp:n {#1} * \l__coffin_sin_fp
16511             + \dim_to_fp:n {#2} * \l__coffin_cos_fp
16512         }
16513     }
16514 }

```

(End definition for __coffin_rotate_vector:nnNN. This function is documented on page ??.)

_coffin_find_corner_maxima:N
_coffin_find_corner_maxima_aux:nn

The idea here is to find the extremities of the content of the coffin. This is done by looking for the smallest values for the bottom and left corners, and the largest values for the top and right corners. The values start at the maximum dimensions so that the case where all are positive or all are negative works out correctly.

```

16515 \cs_new_protected:Npn \__coffin_find_corner_maxima:N #1
16516 {
16517   \dim_set:Nn \l__coffin_top_corner_dim { -\c_max_dim }
16518   \dim_set:Nn \l__coffin_right_corner_dim { -\c_max_dim }
16519   \dim_set:Nn \l__coffin_bottom_corner_dim { \c_max_dim }
16520   \dim_set:Nn \l__coffin_left_corner_dim { \c_max_dim }
16521   \prop_map_inline:cn { l__coffin_corners_ \__int_value:w #1 _prop }
16522     { \__coffin_find_corner_maxima_aux:nn ##2 }
16523 }
16524 \cs_new_protected:Npn \__coffin_find_corner_maxima_aux:nn #1#2
16525 {
16526   \dim_set:Nn \l__coffin_left_corner_dim
16527     { \dim_min:nn { \l__coffin_left_corner_dim } {#1} }
16528   \dim_set:Nn \l__coffin_right_corner_dim
16529     { \dim_max:nn { \l__coffin_right_corner_dim } {#1} }
16530   \dim_set:Nn \l__coffin_bottom_corner_dim
16531     { \dim_min:nn { \l__coffin_bottom_corner_dim } {#2} }
16532   \dim_set:Nn \l__coffin_top_corner_dim
16533     { \dim_max:nn { \l__coffin_top_corner_dim } {#2} }
16534 }

```

(End definition for __coffin_find_corner_maxima:N. This function is documented on page ??.)

_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.

```

16535 \cs_new_protected_nopar:Npn \__coffin_find_bounding_shift:
16536 {
16537   \dim_set:Nn \l__coffin_bounding_shift_dim { \c_max_dim }
16538   \prop_map_inline:Nn \l__coffin_bounding_prop
16539     { \__coffin_find_bounding_shift_aux:nn ##2 }
16540 }
16541 \cs_new_protected:Npn \__coffin_find_bounding_shift_aux:nn #1#2
16542 {
16543   \dim_set:Nn \l__coffin_bounding_shift_dim
16544     { \dim_min:nn { \l__coffin_bounding_shift_dim } {#1} }
16545 }

```

(End definition for __coffin_find_bounding_shift:. This function is documented on page ??.)

_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.

```

16546 \cs_new_protected:Npn \__coffin_shift_corner:Nnnn #1#2#3#4
16547 {

```

```

16548 \prop_put:cnx { l__coffin_corners_ \__int_value:w #1 _ prop } {#2}
16549 {
16550   { \dim_eval:n { #3 - \l__coffin_left_corner_dim } }
16551   { \dim_eval:n { #4 - \l__coffin_bottom_corner_dim } }
16552 }
16553 }
16554 \cs_new_protected:Npn \__coffin_shift_pole:Nnnnnn #1#2#3#4#5#6
16555 {
16556   \prop_put:cnx { l__coffin_poles_ \__int_value:w #1 _ prop } {#2}
16557   {
16558     { \dim_eval:n { #3 - \l__coffin_left_corner_dim } }
16559     { \dim_eval:n { #4 - \l__coffin_bottom_corner_dim } }
16560     {#5} {#6}
16561   }
16562 }

```

(End definition for __coffin_shift_corner:Nnnn. This function is documented on page ??.)

33.8 Resizing coffins

`\l__coffin_scale_x_fp` Storage for the scaling factors in x and y , respectively.
`\l__coffin_scale_y_fp`

```

16563 \fp_new:N \l__coffin_scale_x_fp
16564 \fp_new:N \l__coffin_scale_y_fp

```

(End definition for \l__coffin_scale_x_fp. This variable is documented on page ??.)

`\l__coffin_scaled_total_height_dim` When scaling, the values given have to be turned into absolute values.
`\l__coffin_scaled_width_dim`

```

16565 \dim_new:N \l__coffin_scaled_total_height_dim
16566 \dim_new:N \l__coffin_scaled_width_dim

```

(End definition for \l__coffin_scaled_total_height_dim. This variable is documented on page ??.)

`\coffin_resize:Nnn` Resizing a coffin begins by setting up the user-friendly names for the dimensions of the coffin box. The new sizes are then turned into scale factor. This is the same operation as takes place for the underlying box, but that operation is grouped and so the same calculation is done here.
`\coffin_resize:cnn`

```

16567 \cs_new_protected:Npn \coffin_resize:Nnn #1#2#3
16568 {
16569   \fp_set:Nn \l__coffin_scale_x_fp
16570   { \dim_to_fp:n {#2} / \dim_to_fp:n { \coffin_wd:N #1 } }
16571   \fp_set:Nn \l__coffin_scale_y_fp
16572   {
16573     \dim_to_fp:n {#3}
16574     / \dim_to_fp:n { \coffin_ht:N #1 + \coffin_dp:N #1 }
16575   }
16576   \box_resize:Nnn #1 {#2} {#3}
16577   \__coffin_resize_common:Nnn #1 {#2} {#3}
16578 }
16579 \cs_generate_variant:Nn \coffin_resize:Nnn { c }

```

(End definition for `\coffin_resize:Nnn` and `\coffin_resize:cnn`. These functions are documented on page 216.)

`__coffin_resize_common:Nnn` The poles and corners of the coffin are scaled to the appropriate places before actually resizing the underlying box.

```

16580 \cs_new_protected:Npn \__coffin_resize_common:Nnn #1#2#3
16581 {
16582   \prop_map_inline:cn { l__coffin_corners_ \__int_value:w #1 _prop }
16583   { \__coffin_scale_corner:Nnnn #1 {##1} ##2 }
16584   \prop_map_inline:cn { l__coffin_poles_ \__int_value:w #1 _prop }
16585   { \__coffin_scale_pole:Nnnnnn #1 {##1} ##2 }

```

Negative x -scaling values will place the poles in the wrong location: this is corrected here.

```

16586   \fp_compare:nNnT \l__coffin_scale_x_fp < \c_zero_fp
16587   {
16588     \prop_map_inline:cn { l__coffin_corners_ \__int_value:w #1 _prop }
16589     { \__coffin_x_shift_corner:Nnnn #1 {##1} ##2 }
16590     \prop_map_inline:cn { l__coffin_poles_ \__int_value:w #1 _prop }
16591     { \__coffin_x_shift_pole:Nnnnnn #1 {##1} ##2 }
16592   }
16593 }

```

(End definition for `__coffin_resize_common:Nnn`. This function is documented on page ??.)

`\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 T_EX way as this works properly with floating point values without needing to use the `fp` module.

```

16594 \cs_new_protected:Npn \coffin_scale:Nnn #1#2#3
16595 {
16596   \fp_set:Nn \l__coffin_scale_x_fp {#2}
16597   \fp_set:Nn \l__coffin_scale_y_fp {#3}
16598   \box_scale:Nnn #1 { \l__coffin_scale_x_fp } { \l__coffin_scale_y_fp }
16599   \dim_set:Nn \l__coffin_internal_dim
16600   { \coffin_ht:N #1 + \coffin_dp:N #1 }
16601   \dim_set:Nn \l__coffin_scaled_total_height_dim
16602   { \fp_abs:n { \l__coffin_scale_y_fp } \l__coffin_internal_dim }
16603   \dim_set:Nn \l__coffin_scaled_width_dim
16604   { -\fp_abs:n { \l__coffin_scale_x_fp } \coffin_wd:N #1 }
16605   \__coffin_resize_common:Nnn #1
16606   { \l__coffin_scaled_width_dim } { \l__coffin_scaled_total_height_dim }
16607 }
16608 \cs_generate_variant:Nn \coffin_scale:Nnn { c }

```

(End definition for `\coffin_scale:Nnn` and `\coffin_scale:cnn`. These functions are documented on page 216.)

`_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.

```

16609 \cs_new_protected:Npn \_coffin_scale_vector:nnNN #1#2#3#4
16610 {
16611   \dim_set:Nn #3
16612     { \fp_to_dim:n { \dim_to_fp:n {#1} * \l__coffin_scale_x_fp } }
16613   \dim_set:Nn #4
16614     { \fp_to_dim:n { \dim_to_fp:n {#2} * \l__coffin_scale_y_fp } }
16615 }

```

(End definition for `_coffin_scale_vector:nnNN`. This function is documented on page ??.)

`_coffin_scale_corner:Nnnn` `_coffin_scale_pole:Nnnnnn` Scaling both corners and poles is a simple calculation using the preceding vector scaling.

```

16616 \cs_new_protected:Npn \_coffin_scale_corner:Nnnn #1#2#3#4
16617 {
16618   \_coffin_scale_vector:nnNN {#3} {#4} \l__coffin_x_dim \l__coffin_y_dim
16619   \prop_put:cnx { l__coffin_corners_ \_int_value:w #1 _prop } {#2}
16620     { { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim } }
16621 }
16622 \cs_new_protected:Npn \_coffin_scale_pole:Nnnnnn #1#2#3#4#5#6
16623 {
16624   \_coffin_scale_vector:nnNN {#3} {#4} \l__coffin_x_dim \l__coffin_y_dim
16625   \_coffin_set_pole:Nnx #1 {#2}
16626   {
16627     { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim }
16628     {#5} {#6}
16629   }
16630 }

```

(End definition for `_coffin_scale_corner:Nnnn`. This function is documented on page ??.)

`_coffin_x_shift_corner:Nnnn` `_coffin_x_shift_pole:Nnnnnn` These functions correct for the x displacement that takes place with a negative horizontal scaling.

```

16631 \cs_new_protected:Npn \_coffin_x_shift_corner:Nnnn #1#2#3#4
16632 {
16633   \prop_put:cnx { l__coffin_corners_ \_int_value:w #1 _prop } {#2}
16634     {
16635       { \dim_eval:n { #3 + \box_wd:N #1 } } {#4}
16636     }
16637 }
16638 \cs_new_protected:Npn \_coffin_x_shift_pole:Nnnnnn #1#2#3#4#5#6
16639 {
16640   \prop_put:cnx { l__coffin_poles_ \_int_value:w #1 _prop } {#2}
16641     {
16642       { \dim_eval:n { #3 + \box_wd:N #1 } } {#4}
16643       {#5} {#6}
16644     }
16645 }

```

(End definition for `_coffin_x_shift_corner:Nnnn`. This function is documented on page ??.)

33.9 Coffin diagnostics

\coffin_log_structure:N Redirect output of \coffin_show_structure:N to the log.

```
\coffin_log_structure:c 16646 \cs_new_protected_nopar:Npn \coffin_log_structure:N
16647 { \__msg_log_next: \coffin_show_structure:N }
16648 \cs_generate_variant:Nn \coffin_log_structure:N { c }
```

(End definition for \coffin_log_structure:N and \coffin_log_structure:c. These functions are documented on page 216.)

33.10 Additions to l3file

```
16649 <@@=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.

```
16650 \cs_new_protected:Npn \file_if_exist_input:n #1
16651 {
16652   \file_if_exist:nT {#1}
16653   { \__file_input:V \l__file_internal_name_tl }
16654 }
16655 \cs_new_protected:Npn \file_if_exist_input:nT #1#2
16656 {
16657   \file_if_exist:nT {#1}
16658   {
16659     #2
16660     \__file_input:V \l__file_internal_name_tl
16661   }
16662 }
16663 \cs_new_protected:Npn \file_if_exist_input:nF #1
16664 {
16665   \file_if_exist:nTF {#1}
16666   { \__file_input:V \l__file_internal_name_tl }
16667 }
16668 \cs_new_protected:Npn \file_if_exist_input:nTF #1#2
16669 {
16670   \file_if_exist:nTF {#1}
16671   {
16672     #2
16673     \__file_input:V \l__file_internal_name_tl
16674   }
16675 }
```

(End definition for \file_if_exist_input:nTF. This function is documented on page 217.)

```
16676 <@@=ior>
```

\ior_map_break: Usual map breaking functions. Those are not yet in l3kernel proper since the mapping below is the first of its kind.

```
\ior_map_break:n 16677 \cs_new_nopar:Npn \ior_map_break:
16678 { \__prg_map_break:Nn \ior_map_break: { } }
```

```

16679 \cs_new_nopar:Npn \ior_map_break:n
16680 { \__prg_map_break:Nn \ior_map_break: }

```

(End definition for \ior_map_break: and \ior_map_break:n. These functions are documented on page 217.)

\ior_map_inline:Nn Mapping to an input stream can be done on either a token or a string basis, hence the
\ior_str_map_inline:Nn set up. Within that, there is a check to avoid reading past the end of a file, hence the
 __ior_map_inline:NNn two applications of \ior_if_eof:N. This mapping cannot be nested as the stream has
 __ior_map_inline:NNNn only one “current line”.
 __ior_map_inline_loop:NNN
 \l_ior_internal_tl

```

16681 \cs_new_protected_nopar:Npn \ior_map_inline:Nn
16682 { \__ior_map_inline:NNn \ior_get:NN }
16683 \cs_new_protected_nopar:Npn \ior_str_map_inline:Nn
16684 { \__ior_map_inline:NNn \ior_get_str:NN }
16685 \cs_new_protected_nopar:Npn \__ior_map_inline:NNn
16686 {
16687   \int_gincr:N \g__prg_map_int
16688   \exp_args:Nc \__ior_map_inline:NNNn
16689   { __prg_map_ \int_use:N \g__prg_map_int :n }
16690 }
16691 \cs_new_protected:Npn \__ior_map_inline:NNNn #1#2#3#4
16692 {
16693   \cs_set:Npn #1 ##1 {#4}
16694   \ior_if_eof:NF #3 { \__ior_map_inline_loop:NNN #1#2#3 }
16695   \__prg_break_point:Nn \ior_map_break:
16696   { \int_gdecr:N \g__prg_map_int }
16697 }
16698 \cs_new_protected:Npn \__ior_map_inline_loop:NNN #1#2#3
16699 {
16700   #2 #3 \l_ior_internal_tl
16701   \ior_if_eof:NF #3
16702   {
16703     \exp_args:No #1 \l_ior_internal_tl
16704     \__ior_map_inline_loop:NNN #1#2#3
16705   }
16706 }
16707 \tl_new:N \l_ior_internal_tl

```

(End definition for \ior_map_inline:Nn and \ior_str_map_inline:Nn. These functions are documented on page 217.)

\ior_log_streams: Redirect output of \ior_list_streams: to the log.

```

16708 \cs_new_protected_nopar:Npn \ior_log_streams:
16709 { \__msg_log_next: \ior_list_streams: }

```

(End definition for \ior_log_streams:. This function is documented on page 218.)

```

16710 <@@=iow>

```

\iow_log_streams: Redirect output of \iow_list_streams: to the log.

```

16711 \cs_new_protected_nopar:Npn \iow_log_streams:
16712 { \__msg_log_next: \iow_list_streams: }

```

(End definition for `\iow_log_streams:`. This function is documented on page 218.)

33.11 Additions to l3fp-assign

16713 `<@@=fp>`

`\fp_log:N` Redirect output of `\fp_show:N` to the log.

`\fp_log:c` 16714 `\cs_new_protected_nopar:Npn \fp_log:N`

`\fp_log:n` 16715 `{ __msg_log_next: \fp_show:N }`

16716 `\cs_new_protected_nopar:Npn \fp_log:n`

16717 `{ __msg_log_next: \fp_show:n }`

16718 `\cs_generate_variant:Nn \fp_log:N { c }`

(End definition for `\fp_log:N`, `\fp_log:c`, and `\fp_log:n`. These functions are documented on page 218.)

33.12 Additions to l3int

`\int_log:N` Redirect output of `\int_show:N` to the log. This is not just a copy of `__kernel_-`

`\int_log:c` `register_log:N` because of subtleties involving `\currentgrouplevel` and `\currentgrouptype`. See `\int_show:N` for details.

16719 `\cs_new_protected_nopar:Npn \int_log:N`

16720 `{ __msg_log_next: \int_show:N }`

16721 `\cs_generate_variant:Nn \int_log:N { c }`

(End definition for `\int_log:N` and `\int_log:c`. These functions are documented on page 218.)

`\int_log:n` Redirect output of `\int_show:n` to the log.

16722 `\cs_new_protected_nopar:Npn \int_log:n`

16723 `{ __msg_log_next: \int_show:n }`

(End definition for `\int_log:n`. This function is documented on page 219.)

33.13 Additions to l3keys

16724 `<@@=keys>`

`\keys_log:nn` Redirect output of `\keys_show:nn` to the log.

16725 `\cs_new_protected_nopar:Npn \keys_log:nn`

16726 `{ __msg_log_next: \keys_show:nn }`

(End definition for `\keys_log:nn`. This function is documented on page 219.)

33.14 Additions to l3msg

16727 <@@=msg>

Pass to an auxiliary the message to display and the module name

```

\msg_expandable_error:nnnnnn
\msg_expandable_error:nnnnnn
\msg_expandable_error:nnnn
\msg_expandable_error:nnn
\msg_expandable_error:nn
\msg_expandable_error:nnfff
\msg_expandable_error:nnfff
\msg_expandable_error:nnff
\msg_expandable_error:nnf
  \_msg_expandable_error_module:nn
16728 \cs_new:Npn \msg_expandable_error:nnnnnn #1#2#3#4#5#6
16729 {
16730   \exp_args:Nf \_msg_expandable_error_module:nn
16731   {
16732     \exp_args:Nf \tl_to_str:n
16733     { \use:c { \c__msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6} }
16734   }
16735   {#1}
16736 }
16737 \cs_new:Npn \msg_expandable_error:nnnnn #1#2#3#4#5
16738 { \msg_expandable_error:nnnnnn {#1} {#2} {#3} {#4} {#5} { } }
16739 \cs_new:Npn \msg_expandable_error:nnnn #1#2#3#4
16740 { \msg_expandable_error:nnnnnn {#1} {#2} {#3} {#4} { } { } }
16741 \cs_new:Npn \msg_expandable_error:nnn #1#2#3
16742 { \msg_expandable_error:nnnnnn {#1} {#2} {#3} { } { } { } }
16743 \cs_new:Npn \msg_expandable_error:nn #1#2
16744 { \msg_expandable_error:nnnnnn {#1} {#2} { } { } { } { } }
16745 \cs_generate_variant:Nn \msg_expandable_error:nnnnnn { nnffff }
16746 \cs_generate_variant:Nn \msg_expandable_error:nnnnnn { nnfff }
16747 \cs_generate_variant:Nn \msg_expandable_error:nnnn { nnff }
16748 \cs_generate_variant:Nn \msg_expandable_error:nnn { nnf }
16749 \cs_new:Npn \_msg_expandable_error_module:nn #1#2
16750 {
16751   \exp_after:wN \exp_after:wN
16752   \exp_after:wN \use_none_delimit_by_q_stop:w
16753   \use:n { \::error ! ~ #2 : ~ #1 } \q_stop
16754 }

```

(End definition for `\msg_expandable_error:nnnnnn` and others. These functions are documented on page 219.)

33.15 Additions to l3prg

16755 <@@=bool>

`\bool_log:N` Redirect output of `\bool_show:N` to the log.

```

\bool_log:c 16756 \cs_new_protected_nopar:Npn \bool_log:N
\bool_log:n 16757 { \_msg_log_next: \bool_show:N }
16758 \cs_new_protected_nopar:Npn \bool_log:n
16759 { \_msg_log_next: \bool_show:n }
16760 \cs_generate_variant:Nn \bool_log:N { c }

```

(End definition for `\bool_log:N`, `\bool_log:c`, and `\bool_log:n`. These functions are documented on page 220.)

33.16 Additions to l3prop

16761 <@@=prop>

\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.

```
16762 \cs_new:Npn \prop_map_tokens:Nn #1#2
16763 {
16764   \exp_last_unbraced:Nno \__prop_map_tokens:nwwn {#2} #1
16765   \__prop_pair:wn \q_recursion_tail \s__prop { }
16766   \__prg_break_point:Nn \prop_map_break: { }
16767 }
16768 \cs_new:Npn \__prop_map_tokens:nwwn #1#2 \__prop_pair:wn #3 \s__prop #4
16769 {
16770   \if_meaning:w \q_recursion_tail #3
16771   \exp_after:wN \prop_map_break:
16772   \fi:
16773   \use:n {#1} {#3} {#4}
16774   \__prop_map_tokens:nwwn {#1}
16775 }
16776 \cs_generate_variant:Nn \prop_map_tokens:Nn { c }
```

(End definition for `\prop_map_tokens:Nn` and `\prop_map_tokens:cn`. These functions are documented on page 220.)

\prop_log:N
\prop_log:c

Redirect output of `\prop_show:N` to the log.

```
16777 \cs_new_protected_nopar:Npn \prop_log:N
16778 { \__msg_log_next: \prop_show:N }
16779 \cs_generate_variant:Nn \prop_log:N { c }
```

(End definition for `\prop_log:N` and `\prop_log:c`. These functions are documented on page 220.)

33.17 Additions to l3seq

16780 <@@=seq>

\seq_mapthread_function:NNN
\seq_mapthread_function:NcN
\seq_mapthread_function:cNN
\seq_mapthread_function:ccN
__seq_mapthread_function:wNN
__seq_mapthread_function:wNw
__seq_mapthread_function:Nnnwnn

The idea is to first expand both sequences, adding the usual `{ ? __prg_break: } { }` to the end of each one. This is most conveniently done in two steps using an auxiliary function. The mapping then throws away the first tokens of #2 and #5, which for items in 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.

```
16781 \cs_new:Npn \seq_mapthread_function:NNN #1#2#3
16782 { \exp_after:wN \__seq_mapthread_function:wNN #2 \q_stop #1 #3 }
16783 \cs_new:Npn \__seq_mapthread_function:wNN \s__seq #1 \q_stop #2#3
16784 {
```

```

16785     \exp_after:wN \_seq_mapthread_function:wNw #2 \q_stop #3
16786     #1 { ? \_prg_break: } { }
16787     \_prg_break_point:
16788   }
16789 \cs_new:Npn \_seq_mapthread_function:wNw \s__seq #1 \q_stop #2
16790 {
16791   \_seq_mapthread_function:Nnnwnn #2
16792   #1 { ? \_prg_break: } { }
16793   \q_stop
16794 }
16795 \cs_new:Npn \_seq_mapthread_function:Nnnwnn #1#2#3#4 \q_stop #5#6
16796 {
16797   \use_none:n #2
16798   \use_none:n #5
16799   #1 {#3} {#6}
16800   \_seq_mapthread_function:Nnnwnn #1 #4 \q_stop
16801 }
16802 \cs_generate_variant:Nn \seq_mapthread_function:NNN { Nc }
16803 \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 220.)

`\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.

```

16804 \cs_new_protected_nopar:Npn \seq_set_filter:NNn
16805 { \_seq_set_filter:NNNn \tl_set:Nx }
16806 \cs_new_protected_nopar:Npn \seq_gset_filter:NNn
16807 { \_seq_set_filter:NNNn \tl_gset:Nx }
16808 \cs_new_protected:Npn \_seq_set_filter:NNNn #1#2#3#4
16809 {
16810   \_seq_push_item_def:n { \bool_if:nT {#4} { \_seq_wrap_item:n {##1} } }
16811   #1 #2 { #3 }
16812   \_seq_pop_item_def:
16813 }

```

(End definition for `\seq_set_filter:NNn` and `\seq_gset_filter:NNn`. These functions are documented on page 221.)

`\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.

```

16814 \cs_new_protected_nopar:Npn \seq_set_map:NNn
16815 { \_seq_set_map:NNNn \tl_set:Nx }
16816 \cs_new_protected_nopar:Npn \seq_gset_map:NNn
16817 { \_seq_set_map:NNNn \tl_gset:Nx }
16818 \cs_new_protected:Npn \_seq_set_map:NNNn #1#2#3#4
16819 {
16820   \_seq_push_item_def:n { \exp_not:N \_seq_item:n {#4} }

```

```

16821     #1 #2 { #3 }
16822     \__seq_pop_item_def:
16823 }

```

(End definition for `\seq_set_map:NNn` and `\seq_gset_map:NNn`. These functions are documented on page 221.)

`\seq_log:N` Redirect output of `\seq_show:N` to the log.

```

\seq_log:c 16824 \cs_new_protected_nopar:Npn \seq_log:N
16825 { \__msg_log_next: \seq_show:N }
16826 \cs_generate_variant:Nn \seq_log:N { c }

```

(End definition for `\seq_log:N` and `\seq_log:c`. These functions are documented on page 221.)

33.18 Additions to l3skip

```

16827 <@@=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.

```

16828 \cs_new:Npn \skip_split_finite_else_action:nnNN #1#2#3#4
16829 {
16830   \skip_if_finite:nTF {#1}
16831   {
16832     #3 = \etex_gluestretch:D #1 \scan_stop:
16833     #4 = \etex_glueshrink:D #1 \scan_stop:
16834   }
16835   {
16836     #3 = \c_zero_skip
16837     #4 = \c_zero_skip
16838     #2
16839   }
16840 }

```

(End definition for `\skip_split_finite_else_action:nnNN`. This function is documented on page 221.)

`\dim_log:N` Diagnostics. Redirect output of `\dim_show:n` to the log.

```

\dim_log:c 16841 \cs_new_eq:NN \dim_log:N \__kernel_register_log:N
\dim_log:n 16842 \cs_new_eq:NN \dim_log:c \__kernel_register_log:c
16843 \cs_new_protected_nopar:Npn \dim_log:n
16844 { \__msg_log_next: \dim_show:n }

```

(End definition for `\dim_log:N`, `\dim_log:c`, and `\dim_log:n`. These functions are documented on page 221.)

`\skip_log:N` Diagnostics. Redirect output of `\skip_show:n` to the log.

```

\skip_log:c 16845 \cs_new_eq:NN \skip_log:N \__kernel_register_log:N
\skip_log:n 16846 \cs_new_eq:NN \skip_log:c \__kernel_register_log:c
16847 \cs_new_protected_nopar:Npn \skip_log:n
16848 { \__msg_log_next: \skip_show:n }

```

(End definition for `\skip_log:N`, `\skip_log:c`, and `\skip_log:n`. These functions are documented on page 222.)

```

\muskip_log:N Diagnostics. Redirect output of \muskip_show:n to the log.
\muskip_log:c 16849 \cs_new_eq:NN \muskip_log:N \__kernel_register_log:N
\muskip_log:n 16850 \cs_new_eq:NN \muskip_log:c \__kernel_register_log:c
16851 \cs_new_protected_nopar:Npn \muskip_log:n
16852 { \__msg_log_next: \muskip_show:n }

```

(End definition for `\muskip_log:N`, `\muskip_log:c`, and `\muskip_log:n`. These functions are documented on page 222.)

33.19 Additions to l3tl

```
16853 <@@=tl>
```

`\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.

```

16854 \prg_new_conditional:Npnn \tl_if_single_token:n #1 { p , T , F , TF }
16855 {
16856   \tl_if_head_is_N_type:nTF {#1}
16857   { \__tl_if_empty_return:o { \use_none:n #1 } }
16858   {
16859     \tl_if_empty:nTF {#1}
16860     { \prg_return_false: }
16861     { \__tl_if_empty_return:o { \exp:w \exp_end_continue_f:w #1 } }
16862   }
16863 }

```

(End definition for `\tl_if_single_token:nTF`. This function is documented on page 222.)

`\tl_reverse_tokens:n` The same as `\tl_reverse:n` but with recursion within brace groups.
`__tl_reverse_group:nn`

```

16864 \cs_new:Npn \tl_reverse_tokens:n #1
16865 {
16866   \etex_unexpanded:D \exp_after:wN
16867   {
16868     \exp:w
16869     \__tl_act:NNNnn
16870     \__tl_reverse_normal:nN
16871     \__tl_reverse_group:nn
16872     \__tl_reverse_space:n
16873     { }
16874     {#1}
16875   }
16876 }
16877 \cs_new:Npn \__tl_reverse_group:nn #1
16878 {

```

```

16879     \_tl_act_group_recurse:Nnn
16880     \_tl_act_reverse_output:n
16881     { \tl_reverse_tokens:n }
16882 }

```

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.

`_tl_act_group_recurse:Nnn`

```

16883 \cs_new:Npn \_tl_act_group_recurse:Nnn #1#2#3
16884 {
16885     \exp_args:Nf #1
16886     { \exp_after:wN \exp_after:wN \exp_after:wN { #2 {#3} } }
16887 }

```

(End definition for `\tl_reverse_tokens:n`. This function is documented on page 222.)

`\tl_count_tokens:n`
`_tl_act_count_normal:nN`
`_tl_act_count_group:nn`
`_tl_act_count_space:n`

The token count is computed through an `\int_eval:n` construction. Each `1+` is output to the *left*, into the integer expression, and the sum is ended by the `\exp_end:` inserted by `_tl_act_end:wn` (which is technically implemented as `\c_zero`). Somewhat a hack!

```

16888 \cs_new:Npn \tl_count_tokens:n #1
16889 {
16890     \int_eval:n
16891     {
16892         \_tl_act:NNNnn
16893         \_tl_act_count_normal:nN
16894         \_tl_act_count_group:nn
16895         \_tl_act_count_space:n
16896         { }
16897         {#1}
16898     }
16899 }
16900 \cs_new:Npn \_tl_act_count_normal:nN #1 #2 { 1 + }
16901 \cs_new:Npn \_tl_act_count_space:n #1 { 1 + }
16902 \cs_new:Npn \_tl_act_count_group:nn #1 #2
16903 { 2 + \tl_count_tokens:n {#2} + }

```

(End definition for `\tl_count_tokens:n`. This function is documented on page 222.)

`\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.

```

16904 \cs_new_protected_nopar:Npn \tl_set_from_file:Nnn
16905 { \_tl_set_from_file:NNnn \tl_set:Nn }
16906 \cs_new_protected_nopar:Npn \tl_gset_from_file:Nnn
16907 { \_tl_set_from_file:NNnn \tl_gset:Nn }
16908 \cs_generate_variant:Nn \tl_set_from_file:Nnn { c }
16909 \cs_generate_variant:Nn \tl_gset_from_file:Nnn { c }
16910 \cs_new_protected:Npn \_tl_set_from_file:NNnn #1#2#3#4

```

```

16911 {
16912   \_file_if_exist:nT {#4}
16913   {
16914     \group_begin:
16915     \exp_args:No \etex_everyeof:D
16916       { \c__tl_rescan_marker_tl \exp_not:N }
16917     #3 \scan_stop:
16918     \exp_after:wN \_tl_from_file_do:w
16919     \exp_after:wN \prg_do_nothing:
16920     \tex_input:D \l__file_internal_name_tl \scan_stop:
16921     \exp_args:NNNo \group_end:
16922     #1 #2 \l__tl_internal_a_tl
16923   }
16924 }
16925 \exp_args:Nno \use:nn
16926 { \cs_set_protected:Npn \_tl_from_file_do:w #1 }
16927 { \c__tl_rescan_marker_tl }
16928 { \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 225.)

`\tl_set_from_file_x:Nnn` 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.

```

\__tl_set_from_file_x:NNnn
16929 \cs_new_protected_nopar:Npn \tl_set_from_file_x:Nnn
16930   { \_tl_set_from_file_x:NNnn \tl_set:Nn }
16931 \cs_new_protected_nopar:Npn \tl_gset_from_file_x:Nnn
16932   { \_tl_set_from_file_x:NNnn \tl_gset:Nn }
16933 \cs_generate_variant:Nn \tl_set_from_file_x:Nnn { c }
16934 \cs_generate_variant:Nn \tl_gset_from_file_x:Nnn { c }
16935 \cs_new_protected:Npn \_tl_set_from_file_x:NNnn #1#2#3#4
16936   {
16937     \_file_if_exist:nT {#4}
16938     {
16939       \group_begin:
16940       \etex_everyeof:D { \exp_not:N }
16941       #3 \scan_stop:
16942       \tl_set:Nx \l__tl_internal_a_tl
16943         { \tex_input:D \l__file_internal_name_tl \c_space_token }
16944       \exp_args:NNNo \group_end:
16945       #1 #2 \l__tl_internal_a_tl
16946     }
16947   }

```

(End definition for `\tl_set_from_file_x:Nnn` and others. These functions are documented on page 226.)

33.19.1 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), which is covered by l3unicode-data.

`\tl_if_head_eq_catcode:oNTF`

Extra variants.

```
16948 \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 ??.)

```
\tl_lower_case:n
\tl_upper_case:n
\tl_mixed_case:n
\tl_lower_case:nn
\tl_upper_case:nn
\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.

```
16949 \cs_new_nopar:Npn \tl_lower_case:n { \tl_change_case:nnn { lower } { } }
16950 \cs_new_nopar:Npn \tl_upper_case:n { \tl_change_case:nnn { upper } { } }
16951 \cs_new_nopar:Npn \tl_mixed_case:n { \tl_mixed_case:nn { } }
16952 \cs_new_nopar:Npn \tl_lower_case:nn { \tl_change_case:nnn { lower } }
16953 \cs_new_nopar:Npn \tl_upper_case:nn { \tl_change_case:nnn { upper } }
16954 \cs_new_nopar:Npn \tl_mixed_case:nn { \tl_mixed_case:nn }
```

(End definition for `\tl_lower_case:n`, `\tl_upper_case:n`, and `\tl_mixed_case:n`. These functions are documented on page 223.)

```
\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:fwn
\tl_change_case_end:wn
\tl_change_case_group:nwnn
\tl_change_case_space:wnn
\tl_change_case_N_type:Nwnn
\tl_change_case_N_type:NNNnnn
\tl_change_case_math:NNNnnn
\tl_change_case_math_loop:wNNnn
\tl_change_case_math:NwNNnn
\tl_change_case_math_group:nwNNnn
\tl_change_case_math_space:wNNnn
\tl_change_case_N_type:Nnnn
\tl_change_case_char:Nnn
\tl_change_case_char:Nn
\tl_change_case_char:NNNNNNnn
\tl_change_case_cs:Nnnn
\tl_change_case_cs:nNNnn
\tl_change_case_cs_three:NNNw
\tl_change_case_cs_four:NNNNw
\tl_change_case_cs_cyr:NnNNNw
\tl_change_case_cs_type:Nnnnn
\tl_change_case_cs_type:nnn
\tl_change_case_cs:N
\tl_change_case_cs:NN
\tl_change_case_cs:NNn
\tl_change_case_if_expandable:NTF
\tl_change_case_cs_expand:Nnw
```

The mechanism for the core conversion of case is based on the idea that we can use a loop to grab the entire token list plus a quark: the latter is used as an end marker and to avoid any brace stripping. Depending on the nature of the first item in the grabbed argument, it can either be processed as a single token, treated as a group or treated as a space. These different cases all work by re-reading #1 in the appropriate way, hence the repetition of #1 `\q_recursion_stop`.

```
16955 \cs_new:Npn \tl_change_case:nnn #1#2#3
16956 {
16957   \etex_unexpanded:D \exp_after:wN
16958   {
16959     \exp:w
16960     \tl_change_case_aux:nnn {#1} {#2} {#3}
16961   }
16962 }
16963 \cs_new:Npn \tl_change_case_aux:nnn #1#2#3
16964 {
16965   \group_align_safe_begin:
16966   \tl_change_case_loop:wnn
16967   #3 \q_recursion_tail \q_recursion_stop {#1} {#2}
16968   \tl_change_case_result:n { }
16969 }
16970 \cs_new:Npn \tl_change_case_loop:wnn #1 \q_recursion_stop
16971 {
16972   \tl_if_head_is_N_type:nTF {#1}
16973   { \tl_change_case_N_type:Nwnn }
```

```

16974     {
16975         \tl_if_head_is_group:nTF {#1}
16976         { \__tl_change_case_group:nwnn }
16977         { \__tl_change_case_space:wnn }
16978     }
16979     #1 \q_recursion_stop
16980 }

```

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.

```

16981 \cs_new:Npn \__tl_change_case_output:nwn #1#2 \__tl_change_case_result:n #3
16982 { #2 \__tl_change_case_result:n { #3 #1 } }
16983 \cs_generate_variant:Nn \__tl_change_case_output:nwn { V , o , f }
16984 \cs_new:Npn \__tl_change_case_end:wn #1 \__tl_change_case_result:n #2
16985 {
16986     \group_align_safe_end:
16987     \exp_end:
16988     #2
16989 }

```

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.

```

16990 \cs_new:Npn \__tl_change_case_group:nwnn #1#2 \q_recursion_stop #3#4
16991 {
16992     \__tl_change_case_output:own
16993     {
16994         \exp_after:wN
16995         {
16996             \exp:w
16997             \__tl_change_case_aux:nnn {#3} {#4} {#1}
16998         }
16999     }
17000     \__tl_change_case_loop:wnn #2 \q_recursion_stop {#3} {#4}
17001 }
17002 \exp_last_unbraced:NNo \cs_new:Npn \__tl_change_case_space:wnn \c_space_tl
17003 {
17004     \__tl_change_case_output:nwn { ~ }
17005     \__tl_change_case_loop:wnn
17006 }

```

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.

```

17007 \cs_new:Npn \__tl_change_case_N_type:Nwnn #1#2 \q_recursion_stop
17008 {
17009   \quark_if_recursion_tail_stop_do:Nn #1
17010   { \__tl_change_case_end:wn }
17011   \exp_after:wN \__tl_change_case_N_type:NNNnnn
17012   \exp_after:wN #1 \l_tl_change_case_math_tl
17013   \q_recursion_tail ? \q_recursion_stop {#2}
17014 }

```

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).

```

17015 \cs_new:Npn \__tl_change_case_N_type:NNNnnn #1#2#3
17016 {
17017   \quark_if_recursion_tail_stop_do:Nn #2
17018   { \__tl_change_case_N_type:Nnnn #1 }
17019   \token_if_eq_meaning:NNTF #1 #2
17020   {
17021     \use_i_delimit_by_q_recursion_stop:nw
17022     {
17023       \__tl_change_case_math:NNNnnn
17024       #1 #3 \__tl_change_case_loop:wnn
17025     }
17026   }
17027   { \__tl_change_case_N_type:NNNnnn #1 }
17028 }
17029 \cs_new:Npn \__tl_change_case_math:NNNnnn #1#2#3#4
17030 {
17031   \__tl_change_case_output:nwn {#1}
17032   \__tl_change_case_math_loop:wNNnn #4 \q_recursion_stop #2 #3
17033 }
17034 \cs_new:Npn \__tl_change_case_math_loop:wNNnn #1 \q_recursion_stop
17035 {
17036   \tl_if_head_is_N_type:nTF {#1}
17037   { \__tl_change_case_math:NwNNnn }
17038   {
17039     \tl_if_head_is_group:nTF {#1}
17040     { \__tl_change_case_math_group:nwNNnn }
17041     { \__tl_change_case_math_space:wNNnn }
17042   }

```

```

17043     #1 \q_recursion_stop
17044   }
17045 \cs_new:Npn \__tl_change_case_math:NwNNnn #1#2 \q_recursion_stop #3#4
17046 {
17047   \token_if_eq_meaning:NNTF \q_recursion_tail #1
17048   { \__tl_change_case_end:wn }
17049   {
17050     \__tl_change_case_output:nwn {#1}
17051     \token_if_eq_meaning:NNTF #1 #3
17052     { #4 #2 \q_recursion_stop }
17053     { \__tl_change_case_math_loop:wNNnn #2 \q_recursion_stop #3#4 }
17054   }
17055 }
17056 \cs_new:Npn \__tl_change_case_math_group:nwNNnn #1#2 \q_recursion_stop
17057 {
17058   \__tl_change_case_output:nwn { {#1} }
17059   \__tl_change_case_math_loop:wNNnn #2 \q_recursion_stop
17060 }
17061 \exp_last_unbraced:NNo
17062 \cs_new:Npn \__tl_change_case_math_space:wNNnn \c_space_tl
17063 {
17064   \__tl_change_case_output:nwn { ~ }
17065   \__tl_change_case_math_loop:wNNnn
17066 }

```

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.

```

17067 \cs_new:Npn \__tl_change_case_N_type:Nnnn #1#2#3#4
17068 {
17069   \token_if_cs:NNTF #1
17070   <*initex>
17071   { \__tl_change_case_cs:N #1 }
17072   </initex>
17073   <*package>
17074   {
17075     \__tl_change_case_cs:Nnnn #1 {#3}
17076     { }
17077     { \__tl_change_case_cs:N #1 }
17078   }
17079   </package>
17080   { \__tl_change_case_char:Nnn #1 {#3} {#4} }
17081   \__tl_change_case_loop:wnn #2 \q_recursion_stop {#3} {#4}
17082 }

```

For character tokens there are a couple of potential special cases to handle then the core idea of the loop: a lookup table. The latter uses the character code to spilt what would otherwise be a very long list into 100 manageable blocks (this is a balance between hash

table usage and performance). 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.

```

17083 \cs_new:Npn \__tl_change_case_char:Nnn #1#2#3
17084 {
17085   \cs_if_exist_use:cF { __tl_change_case_ #2 _ #3 :Nnw }
17086   { \use_ii:nn }
17087   #1
17088   {
17089     \use:c { __tl_change_case_ #2 _ sigma:Nnw } #1
17090     { \__tl_change_case_char:Nn #1 {#2} }
17091   }
17092 }
17093 \cs_new:Npn \__tl_change_case_char:Nn #1#2
17094 {
17095   \__tl_change_case_output:fwn
17096   {
17097     \str_case:nvF #1 { c__unicode_ #2 _exceptions_tl }
17098     {
17099       \exp_after:wN \__tl_change_case_char:NNNNNNNn
17100       \int_use:N \__int_eval:w 1000000 + ‘#1 \__int_eval_end:
17101       #1 {#2}
17102     }
17103   }
17104 }
17105 \cs_new:Npn \__tl_change_case_char:NNNNNNNn #1#2#3#4#5#6#7#8#9
17106 {
17107   \str_case:nvF #8
17108   { c__unicode_ #9 _ #6 _X_ #7 _tl }
17109   { \exp_stop_f: #8 }
17110 }

```

If a control sequence has been given as the argument and it is not on the list of those with an argument to examine, the other possibility is that it is a character represented as a command such as `\aa`. To deal with that there is a need again to balance performance against name use. For $\text{\LaTeX} 2_{\epsilon}$ the list of possible special cases is quite long (there are around 100 for Cyrillic alone) so a single long `\str_case:nvF` is inefficient. On the other hand, using one `tl` per special case would use a lot of names. The balance here is to split up the special cases by type and for Cyrillic to further subdivide. This works as there are various cases:

- Short (one or two letter) names for special Latin characters
- Cyrillic names, all starting `cyr`
- Greek names, all starting `text`
- Greek accents, all starting `acc` and only applicable for upper casing
- Other cases (“bits and pieces”)

Thus a split can be carried out by converting the control sequence to a string then examining the first four characters. For Cyrillic, the fourth character can be used for a second split based on the character code.

Note that as this is dependent on L^AT_EX 2_ε, in format mode the code goes straight to the final phase of handling control sequences.

```

17111 <*package>
17112 \cs_new:Npn \__tl_change_case_cs:Nnnn #1#2
17113 {
17114   \exp_args:Nf \__tl_change_case_cs:nNnnn
17115   { \cs_to_str:N #1 } #1 {#2}
17116 }
17117 \cs_new:Npn \__tl_change_case_cs:nNnnn #1#2#3
17118 {
17119   \tl_if_head_eq_catcode:oNTF { \use_none:nnn #1 a a a a } a
17120   { \__tl_change_case_cs_type:Nnnnn #2 { latin } {#3} }
17121   {
17122     \str_if_eq_x:nnTF
17123     { \__tl_change_case_cs_three:NNNw #1 \q_nil }
17124     { \str_if_eq:nnTF {#3} { lower } { CYR } { cyr } }
17125     { \__tl_change_case_cs_cyr:NnNNNNw #2 {#3} #1 \q_stop }
17126     {
17127       \str_if_eq_x:nnTF
17128       { \__tl_change_case_cs_three:NNNw #1 \q_nil }
17129       { acc }
17130       { \__tl_change_case_cs_type:Nnnnn #2 { acc } {#3} }
17131       {
17132         \str_if_eq_x:nnTF
17133         { \__tl_change_case_cs_four:NNNNw #1 \q_nil }
17134         { text }
17135         { \__tl_change_case_cs_type:Nnnnn #2 { greek } {#3} }
17136         { \__tl_change_case_cs_type:Nnnnn #2 { misc } {#3} }
17137       }
17138     }
17139   }
17140 }
17141 \cs_new:Npn \__tl_change_case_cs_three:NNNw #1#2#3#4 \q_nil { #1#2#3 }
17142 \cs_new:Npn \__tl_change_case_cs_four:NNNNw #1#2#3#4#5 \q_nil { #1#2#3#4 }
17143 \cs_new:Npn \__tl_change_case_cs_cyr:NnNNNNw #1#2#3#4#5#6#7 \q_stop
17144 {
17145   \__tl_change_case_cs_type:Nnnnn #1
17146   { cyrillic }
17147   {
17148     #2 _
17149     \int_to_roman:n
17150     {
17151       1 +
17152       \int_div_truncate:nn
17153       {
17154         '#6 - \str_if_eq:nnTF {#2} { lower } { 'A } { 'a }

```

```

17155         }
17156         { 7 }
17157     }
17158 }
17159 }
17160 \cs_new:Npn \__tl_change_case_cs_type:Nnnnn #1#2#3
17161 {
17162     \exp_args:Nf \__tl_change_case_cs_type:nnn
17163     {
17164         \str_case:nvF #1
17165         { c__tl_change_case_ #2 _ #3 _ t1 }
17166         { \exp_stop_f: }
17167     }
17168 }
17169 \cs_new:Npn \__tl_change_case_cs_type:nnn #1#2#3
17170 {
17171     \tl_if_blank:nTF {#1}
17172     {#3}
17173     {
17174         \__tl_change_case_output:nwn {#1}
17175         #2
17176     }
17177 }
17178 </package>

```

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.

```

17179 \cs_new:Npn \__tl_change_case_cs:N #1
17180 {
17181     \exp_after:wN \__tl_change_case_cs:NN
17182     \exp_after:wN #1 \l_tl_change_case_exclude_tl
17183     \q_recursion_tail \q_recursion_stop
17184 }
17185 \cs_new:Npn \__tl_change_case_cs:NN #1#2
17186 {
17187     \quark_if_recursion_tail_stop_do:Nn #2
17188     {
17189         \__tl_change_case_cs_expand:Nnw #1
17190         { \__tl_change_case_output:nwn {#1} }
17191     }
17192     \str_if_eq:nnTF {#1} {#2}
17193     {
17194         \use_i_delimit_by_q_recursion_stop:nw
17195         { \__tl_change_case_cs:NNn #1 }
17196     }
17197     { \__tl_change_case_cs:NN #1 }
17198 }
17199 \cs_new:Npn \__tl_change_case_cs:NNn #1#2#3

```

```

17200 {
17201   \_tl_change_case_output:nwn { #1 {#3} }
17202   #2
17203 }

```

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 (!

```

17204 \cs_new:Npn \_tl_change_case_if_expandable:NTF #1
17205 {
17206   \token_if_expandable:NTF #1
17207   {
17208     \bool_if:nTF
17209     {
17210       \token_if_protected_macro_p:N #1
17211       || \token_if_protected_long_macro_p:N #1
17212       || \token_if_eq_meaning_p:NN \q_recursion_tail #1
17213     }
17214     { \use_ii:nn }
17215     { \use_i:nn }
17216   }
17217   { \use_ii:nn }
17218 }
17219 \cs_new:Npn \_tl_change_case_cs_expand:Nnw #1#2
17220 {
17221   \_tl_change_case_if_expandable:NTF #1
17222   { \_tl_change_case_cs_expand:NN #1 }
17223   { #2 }
17224 }
17225 \cs_new:Npn \_tl_change_case_cs_expand:NN #1#2
17226 { \exp_after:wN #2 #1 }

```

(End definition for _tl_change_case:nnn.)

```

\_tl_change_case_lower_sigma:Nnw
\_tl_change_case_lower_sigma:w
\_tl_change_case_lower_sigma:Nw
\_tl_change_case_upper_sigma:Nnw

```

If the current char is an upper case sigma, the a check is made on the next item in the input. If it is N-type and not a control sequence then there is a look-ahead phase.

```

17227 \cs_new:Npn \_tl_change_case_lower_sigma:Nnw #1#2#3#4 \q_recursion_stop
17228 {
17229   \int_compare:nNnTF { '#1 } = { "03A3 }
17230   {
17231     \_tl_change_case_output:fwn
17232     { \_tl_change_case_lower_sigma:w #4 \q_recursion_stop }
17233   }
17234   {#2}
17235   #3 #4 \q_recursion_stop
17236 }
17237 \cs_new:Npn \_tl_change_case_lower_sigma:w #1 \q_recursion_stop
17238 {

```

```

17239 \tl_if_head_is_N_type:nTF {#1}
17240 { \_tl_change_case_lower_sigma:Nw #1 \q_recursion_stop }
17241 { \c_unicode_final_sigma_tl }
17242 }
17243 \cs_new:Npn \_tl_change_case_lower_sigma:Nw #1#2 \q_recursion_stop
17244 {
17245   \_tl_change_case_if_expandable:NTF #1
17246   {
17247     \exp_after:wN \_tl_change_case_lower_sigma:w #1
17248     #2 \q_recursion_stop
17249   }
17250   {
17251     \token_if_letter:NTF #1
17252     { \c_unicode_std_sigma_tl }
17253     { \c_unicode_final_sigma_tl }
17254   }
17255 }

```

Simply skip to the final step for upper casing.

```

17256 \cs_new_eq:NN \_tl_change_case_upper_sigma:Nnw \use_ii:nn

```

(End definition for `_tl_change_case_lower_sigma:Nnw`.)

```

\_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

```

The Turkic languages need special treatment for dotted-i and dotless-i. The lower casing rule can be expressed in terms of searching first for either a dotless-I or a dotted-I. In the latter case the mapping is easy, but in the former there is a second stage search.

```

17257 \cs_new:Npn \_tl_change_case_lower_tr:Nnw #1#2
17258 {
17259   \int_compare:nNnTF { '#1 } = { "0049 }
17260   { \_tl_change_case_lower_tr_auxi:Nw }
17261   {
17262     \int_compare:nNnTF { '#1 } = { "0130 }
17263     { \_tl_change_case_output:nwn { i } }
17264     { #2 }
17265   }
17266 }

```

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_ii:nn` (it grabs `_tl_change_case_loop:wn` and the dot-above char and discards the latter).

```

17267 \cs_new:Npn \_tl_change_case_lower_tr_auxi:Nw #1#2 \q_recursion_stop
17268 {
17269   \tl_if_head_is_N_type:nTF {#2}
17270   { \_tl_change_case_lower_tr_auxii:Nw #2 \q_recursion_stop }
17271   { \_tl_change_case_output:Vwn \c_unicode_dotless_i_tl }
17272   #1 #2 \q_recursion_stop
17273 }
17274 \cs_new:Npn \_tl_change_case_lower_tr_auxii:Nw #1#2 \q_recursion_stop

```

```

17275 {
17276   \_tl_change_case_if_expandable:NTF #1
17277   {
17278     \exp_after:wN \_tl_change_case_lower_tr_auxi:Nw #1
17279     #2 \q_recursion_stop
17280   }
17281   {
17282     \bool_if:nTF
17283     {
17284       \token_if_cs_p:N #1
17285       || ! ( \int_compare_p:nNn { '#1 } = { "0307 } )
17286     }
17287     { \_tl_change_case_output:Vwn \c__unicode_dotless_i_tl }
17288     {
17289       \_tl_change_case_output:nwn { i }
17290       \use_i:nn
17291     }
17292   }
17293 }

```

Upper casing is easier: just one exception with no context.

```

17294 \cs_new:Npn \_tl_change_case_upper_tr:Nnw #1#2
17295 {
17296   \int_compare:nNnTF { '#1 } = { "0069 }
17297   { \_tl_change_case_output:Vwn \c__unicode_dotted_I_tl }
17298   {#2}
17299 }

```

Straight copies.

```

17300 \cs_new_eq:NN \_tl_change_case_lower_az:Nnw \_tl_change_case_lower_tr:Nnw
17301 \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.)

```

\_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.

```

17302 \cs_new:Npn \_tl_change_case_lower_lt:Nnw #1
17303 {
17304   \exp_args:Nf \_tl_change_case_lower_lt:nNnw
17305   { \str_case:nVF #1 \c__unicode_accents_lt_tl \exp_stop_f: }
17306   #1
17307 }
17308 \cs_new:Npn \_tl_change_case_lower_lt:nNnw #1#2
17309 {
17310   \tl_if_blank:nTF {#1}

```

```

17311 {
17312     \exp_args:Nf \__tl_change_case_lower_lt:nnw
17313     {
17314         \int_case:nnF {'#2}
17315         {
17316             { "0049 } i
17317             { "004A } j
17318             { "012E } \c__unicode_i_ogonek_tl
17319         }
17320         \exp_stop_f:
17321     }
17322 }
17323 {
17324     \__tl_change_case_output:wnw {#1}
17325     \use_none:n
17326 }
17327 }
17328 \cs_new:Npn \__tl_change_case_lower_lt:nnw #1#2
17329 {
17330     \tl_if_blank:nTF {#1}
17331     {#2}
17332     {
17333         \__tl_change_case_output:wnw {#1}
17334         \__tl_change_case_lower_lt:Nw
17335     }
17336 }

```

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.

```

17337 \cs_new:Npn \__tl_change_case_lower_lt:Nw #1#2 \q_recursion_stop
17338 {
17339     \tl_if_head_is_N_type:nT {#2}
17340     { \__tl_change_case_lower_lt:NNw }
17341     #1 #2 \q_recursion_stop
17342 }
17343 \cs_new:Npn \__tl_change_case_lower_lt:NNw #1#2#3 \q_recursion_stop
17344 {
17345     \__tl_change_case_if_expandable:NTF #2
17346     {
17347         \exp_after:wN \__tl_change_case_lower_lt:Nw \exp_after:wN #1 #2
17348         #3 \q_recursion_stop
17349     }
17350     {
17351         \bool_if:nT
17352         {
17353             ! \token_if_cs_p:N #2
17354             &&
17355             (
17356                 \int_compare_p:nNn { '#2 } = { "0300 }
17357                 || \int_compare_p:nNn { '#2 } = { "0301 }

```

```

17358         || \int_compare_p:nNn { '#2 } = { "0303 }
17359     )
17360 }
17361 { \__tl_change_case_output:Vwn \c__unicode_dot_above_tl }
17362 #1 #2#3 \q_recursion_stop
17363 }
17364 }

```

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.

```

17365 \cs_new:Npn \__tl_change_case_upper_lt:Nnw #1
17366 {
17367     \exp_args:Nf \__tl_change_case_upper_lt:nnw
17368     {
17369         \int_case:nnF { '#1 }
17370         {
17371             { "0069 } I
17372             { "006A } J
17373             { "012F } \c__unicode_I_ogonek_tl
17374         }
17375         \exp_stop_f:
17376     }
17377 }
17378 \cs_new:Npn \__tl_change_case_upper_lt:nnw #1#2
17379 {
17380     \tl_if_blank:nTF {#1}
17381     {#2}
17382     {
17383         \__tl_change_case_output:wnw {#1}
17384         \__tl_change_case_upper_lt:Nw
17385     }
17386 }
17387 \cs_new:Npn \__tl_change_case_upper_lt:Nw #1#2 \q_recursion_stop
17388 {
17389     \tl_if_head_is_N_type:nT {#2}
17390     { \__tl_change_case_upper_lt:NNw }
17391     #1 #2 \q_recursion_stop
17392 }
17393 \cs_new:Npn \__tl_change_case_upper_lt:NNw #1#2#3 \q_recursion_stop
17394 {
17395     \__tl_change_case_if_expandable:NTF #2
17396     {
17397         \exp_after:wN \__tl_change_case_upper_lt:Nw \exp_after:wN #1 #2
17398         #3 \q_recursion_stop
17399     }
17400     {
17401         \bool_if:nTF
17402         {
17403             ! \token_if_cs_p:N #2

```

```

17404         && \int_compare_p:nNn { '#2 } = { "0307 }
17405     }
17406     { #1 }
17407     { #1 #2 }
17408     #3 \q_recursion_stop
17409 }
17410 }

```

(End definition for _tl_change_case_lower_lt:Nnw.)

_tl_change_case_upper_de-alt:Nnw A simple alternative version for German.

```

17411 \cs_new:cpn { \_tl_change_case_upper_de-alt:Nnw } #1#2
17412 {
17413     \int_compare:nNnTF { '#1 } = { 223 }
17414     { \_tl_change_case_output:Vwn \c__unicode_upper_Eszett_tl }
17415     {#2}
17416 }

```

(End definition for _tl_change_case_upper_de-alt:Nnw. This function is documented on page ??.)

```

\_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_char:Nn
\_tl_mixed_case_skip:N
\_tl_mixed_case_skip:NN
\_tl_mixed_case_skip_tidy:Nwn
\_tl_mixed_case_char:nN

```

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.

```

17417 \cs_new:Npn \_tl_mixed_case:nn #1#2
17418 {
17419     \etex_unexpanded:D \exp_after:wN
17420     {
17421         \exp:w
17422         \_tl_mixed_case_aux:nn {#1} {#2}
17423     }
17424 }
17425 \cs_new:Npn \_tl_mixed_case_aux:nn #1#2
17426 {
17427     \group_align_safe_begin:
17428     \_tl_mixed_case_loop:wn
17429     #2 \q_recursion_tail \q_recursion_stop {#1}
17430     \_tl_change_case_result:n { }
17431 }
17432 \cs_new:Npn \_tl_mixed_case_loop:wn #1 \q_recursion_stop

```

```

17433 {
17434   \tl_if_head_is_N_type:nTF {#1}
17435   { \_tl_mixed_case_N_type:Nwn }
17436   {
17437     \tl_if_head_is_group:nTF {#1}
17438     { \_tl_mixed_case_group:nwn }
17439     { \_tl_mixed_case_space:wn }
17440   }
17441   #1 \q_recursion_stop
17442 }
17443 \cs_new:Npn \_tl_mixed_case_group:nwn #1#2 \q_recursion_stop #3
17444 {
17445   \_tl_change_case_output:own
17446   {
17447     \exp_after:wN
17448     {
17449       \exp:w
17450       \_tl_mixed_case_aux:nn {#3} {#1}
17451     }
17452   }
17453   \_tl_change_case_loop:wnn #2 \q_recursion_stop { lower } {#3}
17454 }
17455 \exp_last_unbraced:NNo \cs_new:Npn \_tl_mixed_case_space:wn \c_space_tl
17456 {
17457   \_tl_change_case_output:nwn { ~ }
17458   \_tl_mixed_case_loop:wn
17459 }
17460 \cs_new:Npn \_tl_mixed_case_N_type:Nwn #1#2 \q_recursion_stop
17461 {
17462   \quark_if_recursion_tail_stop_do:Nn #1
17463   { \_tl_change_case_end:wn }
17464   \exp_after:wN \_tl_mixed_case_N_type:NNNnn
17465   \exp_after:wN #1 \l_tl_case_change_math_tl
17466   \q_recursion_tail ? \q_recursion_stop {#2}
17467 }
17468 \cs_new:Npn \_tl_mixed_case_N_type:NNNnn #1#2#3
17469 {
17470   \quark_if_recursion_tail_stop_do:Nn #2
17471   { \_tl_mixed_case_N_type:Nnn #1 }
17472   \token_if_eq_meaning:NNTF #1 #2
17473   {
17474     \use_i_delimit_by_q_recursion_stop:nw
17475     {
17476       \_tl_change_case_math:NNNnnn
17477       #1 #3 \_tl_mixed_case_loop:wn
17478     }
17479   }
17480   { \_tl_mixed_case_N_type:NNNnn #1 }
17481 }

```

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.

```

17482 \cs_new:Npn \__tl_mixed_case_N_type:Nnn #1#2#3
17483 {
17484   \token_if_cs:NTF #1
17485   <*initex>
17486   {
17487     \__tl_change_case_cs:N #1
17488     \__tl_mixed_case_loop:wn #2 \q_recursion_stop {#3}
17489   }
17490 </initex>
17491 <*package>
17492 {
17493   \__tl_change_case_cs:Nnnn #1 { upper }
17494   {
17495     \__tl_change_case_loop:wnn
17496     #2 \q_recursion_stop { lower } {#3}
17497   }
17498   {
17499     \__tl_change_case_cs:N #1
17500     \__tl_mixed_case_loop:wn #2 \q_recursion_stop {#3}
17501   }
17502 }
17503 </package>
17504 {
17505   \__tl_mixed_case_char:Nn #1 {#3}
17506   \__tl_change_case_loop:wnn #2 \q_recursion_stop { lower } {#3}
17507 }
17508 }

```

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.

```

17509 \cs_new:Npn \__tl_mixed_case_char:Nn #1#2
17510 {
17511   \cs_if_exist_use:cF { __tl_change_case_mixed_ #2 :Nnw }
17512   {
17513     \cs_if_exist_use:cF { __tl_change_case_upper_ #2 :Nnw }
17514     { \use_ii:nn }
17515   }
17516   #1
17517   { \__tl_mixed_case_skip:N #1 }
17518 }
17519 \cs_new:Npn \__tl_mixed_case_skip:N #1
17520 {
17521   \exp_after:wN \__tl_mixed_case_skip:NN
17522   \exp_after:wN #1 \l_tl_mixed_case_ignore_tl
17523   \q_recursion_tail \q_recursion_stop
17524 }
17525 \cs_new:Npn \__tl_mixed_case_skip:NN #1#2
17526 {

```

```

17527 \quark_if_recursion_tail_stop_do:nn {#2}
17528 {
17529     \exp_args:Nf \__tl_mixed_case_char:nN
17530     { \str_case:nVF #1 \c__unicode_mixed_exceptions_tl \exp_stop_f: }
17531     #1
17532 }
17533 \int_compare:nNnT { '#1 } = { '#2 }
17534 {
17535     \use_i_delimit_by_q_recursion_stop:nw
17536     {
17537         \__tl_change_case_output:nwn {#1}
17538         \__tl_mixed_case_skip_tidy:Nwn
17539     }
17540 }
17541 \__tl_mixed_case_skip:NN #1
17542 }
17543 \cs_new:Npn \__tl_mixed_case_skip_tidy:Nwn #1#2 \q_recursion_stop #3
17544 {
17545     \__tl_mixed_case_loop:wn #2 \q_recursion_stop
17546 }
17547 \cs_new:Npn \__tl_mixed_case_char:nN #1#2
17548 {
17549     \tl_if_blank:nTF {#1}
17550     { \__tl_change_case_char:Nn #2 { upper } }
17551     { \__tl_change_case_output:nwn {#1} }
17552 }

```

(End definition for __tl_mixed_case:nn.)

```

\__tl_change_case_mixed_n1:Nnw
\__tl_change_case_mixed_n1:Nw
\__tl_change_case_mixed_n1: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.

```

17553 \cs_new:Npn \__tl_change_case_mixed_n1:Nnw #1
17554 {
17555     \bool_if:nTF
17556     {
17557         \int_compare_p:nNn { '#1 } = { 'i }
17558         || \int_compare_p:nNn { '#1 } = { 'I }
17559     }
17560     {
17561         \__tl_change_case_output:nwn { I }
17562         \__tl_change_case_mixed_n1:Nw
17563     }
17564 }
17565 \cs_new:Npn \__tl_change_case_mixed_n1:Nw #1#2 \q_recursion_stop
17566 {
17567     \tl_if_head_is_N_type:nT {#2}
17568     { \__tl_change_case_mixed_n1:NNw }
17569     #1 #2 \q_recursion_stop
17570 }
17571 \cs_new:Npn \__tl_change_case_mixed_n1:NNw #1#2#3 \q_recursion_stop

```

```

17572 {
17573   \_tl_change_case_if_expandable:NTF #2
17574   {
17575     \exp_after:wN \_tl_change_case_mixed_n1:Nw \exp_after:wN #1 #2
17576     #3 \q_recursion_stop
17577   }
17578   {
17579     \bool_if:nTF
17580     {
17581       ! ( \token_if_cs_p:N #2 )
17582       &&
17583       (
17584         \int_compare_p:nNn { '#2 } = { 'j }
17585         || \int_compare_p:nNn { '#2 } = { 'J }
17586       )
17587     }
17588     {
17589       \_tl_change_case_output:nwn { J }
17590       #1
17591     }
17592     { #1 #2 }
17593     #3 \q_recursion_stop
17594   }
17595 }

```

(End definition for `_tl_change_case_mixed_n1:Nnw`.)

`\l_tl_case_change_math_tl` The list of token pairs which are treated as math mode and so not case changed.

```

17596 \tl_new:N \l_tl_case_change_math_tl
17597 <*package>
17598 \tl_set:Nn \l_tl_case_change_math_tl
17599 { $ $ \ ( \ ) }
17600 </package>

```

(End definition for `\l_tl_case_change_math_tl`. This variable is documented on page 223.)

`\l_tl_case_change_exclude_tl` The list of commands for which an argument is not case changed.

```

17601 \tl_new:N \l_tl_case_change_exclude_tl
17602 <*package>
17603 \tl_set:Nn \l_tl_case_change_exclude_tl
17604 { \cite \ensuremath \label \ref }
17605 </package>

```

(End definition for `\l_tl_case_change_exclude_tl`. This variable is documented on page 224.)

`\l_tl_mixed_case_ignore_tl` Characters to skip over when finding the first letter in a word to be mixed cased.

```

17606 \tl_new:N \l_tl_mixed_case_ignore_tl
17607 \tl_set:Nx \l_tl_mixed_case_ignore_tl
17608 {
17609   ( % )

```

```

17610     [ % ]
17611     \cs_to_str:N \{ % \}
17612     ,
17613     -
17614   }

```

(End definition for \l_tl_mixed_case_ignore_tl. This variable is documented on page 225.)

The data for case changing control sequences representing characters is stored such that further expansion is inhibited. This is required as many of the L^AT_EX 2_ε commands are fragile and so will fail inside an x-type expansion without this precaution. As such, there is a bit of set up to deal with the various requirements here: upper and lower case mappings are not identical so special end cases are needed to get everything set up correctly with minimal repetition.

```

17615 (*package)
17616 \group_begin:
17617   \cs_set_protected:Npn \__tl_change_case_setup:nnnn #1#2#3#4
17618   {
17619     \tl_const:cx { c__tl_change_case_ #1 _upper #2 _tl }
17620     {
17621       \__tl_change_case_map:NN
17622       #3 \q_recursion_tail ? \q_recursion_stop
17623     }
17624     \tl_const:cx { c__tl_change_case_ #1 _lower #2 _tl }
17625     {
17626       \__tl_change_case_map:NNN #4
17627       #3 \q_recursion_tail ? \q_recursion_stop
17628     }
17629   }
17630   \cs_set:Npn \__tl_change_case_map:NN #1#2
17631   {
17632     \quark_if_recursion_tail_stop:N #1
17633     \exp_not:N #1 \exp_not:n { { \exp_stop_f: #2 } }
17634     \__tl_change_case_map:NN
17635   }
17636   \cs_set:Npn \__tl_change_case_map:NNN #1#2#3
17637   {
17638     \str_if_eq:nnT {#1} {#2}
17639     { \use_none_delimit_by_q_recursion_stop:w }
17640     \exp_not:N #3 \exp_not:n { { \exp_stop_f: #2 } }
17641     \__tl_change_case_map:NNN #1
17642   }
17643   \__tl_change_case_setup:nnnn
17644   { latin }
17645   { }
17646   {
17647     \aa \AA
17648     \ae \AE
17649     \dh \DH
17650     \dj \DJ

```

```

17651      \l  \L
17652      \ng \NG
17653      \o  \O
17654      \oe \OE
17655      \ss \SS
17656      \th \TH
17657      \i  \I
17658      \j  \J
17659      }
17660      { \i }
17661      \_tl_change_case_setup:nnnn
17662      { cyrillic }
17663      { _i }
17664      {
17665          \cyra      \CYRA
17666          \cyrabhch  \CYRABHCH
17667          \cyrabhchdsc \CYRABHCHDSC
17668          \cyrabhdze \CYRABHDZE
17669          \cyrabhha  \CYRABHHA
17670          \cyrae     \CYRAE
17671          \cyrb      \CYRB
17672          \cyrbyus   \CYRBYUS
17673          \cyr       \CYRC
17674          \cyrch     \CYRCH
17675          \cyrchldsc \CYRCHLDSC
17676          \cyrchrdsc \CYRCHRDSC
17677          \cyrchvcrs \CYRCHVCRS
17678          \cyrd      \CYRD
17679          \cyrdelta  \CYRDELTA
17680          \cyrdje    \CYRDJE
17681          \cyrdze    \CYRDZE
17682          \cyrdzhe   \CYRDZHE
17683          \cyre      \CYRE
17684          \cyreps    \CYREPS
17685          \cyrerev   \CYREREV
17686          \cyrery    \CYRERY
17687          \cyrf      \CYRF
17688          \cyrfita   \CYRFITA
17689          \cyr       \CYRG
17690          \cyrgdsc   \CYRGDSC
17691          \cyrgdschcrs \CYRGDSCHCRS
17692          \cyrghcrs  \CYRGHCRS
17693          \cyrghk    \CYRGHK
17694          \cyrgup    \CYRGUP
17695      }
17696      { \q_recursion_tail }
17697      \_tl_change_case_setup:nnnn
17698      { cyrillic }
17699      { _ii }
17700      {

```

17701	\cyrh	\CYRH
17702	\cyrhdsc	\CYRHDSC
17703	\cyrhhcrs	\CYRHHCRS
17704	\cyrhhk	\CYRHHK
17705	\cyrhrdsn	\CYRHRDSN
17706	\cyri	\CYRI
17707	\cyrie	\CYRIE
17708	\cyrii	\CYRII
17709	\cyrishrt	\CYRISHRT
17710	\cyrishrtdsc	\CYRISHRTDSC
17711	\cyrizh	\CYRIZH
17712	\cyrje	\CYRJE
17713	\cyrk	\CYRK
17714	\cyrkbeak	\CYRKBEAK
17715	\cyrkdsc	\CYRKDSC
17716	\cyrkhcrs	\CYRKHCRS
17717	\cyrkhk	\CYRKHK
17718	\cyrkvcrs	\CYRKVCRS
17719	\cyrl	\CYRL
17720	\cyrl dsc	\CYRLDSC
17721	\cyrlhk	\CYRLHK
17722	\cyrlje	\CYRLJE
17723	\cyr m	\CYRM
17724	\cyrmdsc	\CYRMDSC
17725	\cyrmhk	\CYRMHK
17726	\cyrn	\CYRN
17727	\cyrndsc	\CYRNDSC
17728	\cyrng	\CYRNG
17729	\cyrnhk	\CYRNHK
17730	\cyrnje	\CYRNJE
17731	\cyrnlhk	\CYRNLHK
17732	}	
17733	{ \q_recursion_tail }	
17734	_tl_change_case_setup:nnnn	
17735	{ cyrillic }	
17736	{ _iii }	
17737	{	
17738	\cyro	\CYRO
17739	\cyrotld	\CYROTLD
17740	\cyrp	\CYRP
17741	\cyrphk	\CYRPHK
17742	\cyrq	\CYRQ
17743	\cyrr	\CYRR
17744	\cyrrdsc	\CYRRDSC
17745	\cyrrhk	\CYRRHK
17746	\cyrrtick	\CYRRTICK
17747	\cyr s	\CYRS
17748	\cyr sacrs	\CYRSACRS
17749	\cyr schwa	\CYRSCHWA
17750	\cyr sdsc	\CYRSDSC

```

17751      \cyrsemisftsn \CYRSEMISFTSN
17752      \cyrstfn      \CYRSFTSN
17753      \cyrsh         \CYRSH
17754      \cyrshch       \CYRSHCH
17755      \cyrshha       \CYRSHHA
17756      \cyrst         \CYRT
17757      \cyrtdsc        \CYRTDSC
17758      \cyrstetse     \CYRTETSE
17759      \cyrstshe      \CYRTSHE
17760      \cyrst         \CYRU
17761      \cyrsthrst     \CYRUSHRT
17762      }
17763      { \q_recursion_tail }
17764      \_tl_change_case_setup:nnnn
17765      { cyrillic }
17766      { _iv }
17767      {
17768          \cyrst         \CYRV
17769          \cyrst         \CYRW
17770          \cyrst         \CYRY
17771          \cyrst         \CYRYA
17772          \cyrst         \CYRYAT
17773          \cyrst         \CYRYHCRS
17774          \cyrst         \CYRYI
17775          \cyrst         \CYRYO
17776          \cyrst         \CYRYU
17777          \cyrst         \CYRZ
17778          \cyrst         \CYRZDSC
17779          \cyrst         \CYRZH
17780          \cyrst         \CYRZHDSC
17781      }
17782      { \q_recursion_tail }
17783      \_tl_change_case_setup:nnnn
17784      { greek }
17785      { }
17786      {
17787          \textalpha      \textAlpha
17788          \textbeta       \textBeta
17789          \textchi         \textChi
17790          \textdelta       \textDelta
17791          \textdigamma     \textDigamma
17792          \texteta         \textEta
17793          \textepsilon     \textEpsilon
17794          \textgamma       \textGamma
17795          \textiota        \textIota
17796          \textkappa       \textKappa
17797          \textlambda      \textLambda
17798          \textmu          \textMu
17799          \textnu          \textNu
17800          \textomega       \textOmega

```

```

17801      \textomicron      \textOmicron
17802      \textphi        \textPhi
17803      \textpi          \textPi
17804      \textpsi         \textPsi
17805      \textqoppa       \textQoppa
17806      \textrho         \textRho
17807      \textsampi       \textSampi
17808      \textautosigma   \textSigma
17809      \textstigma      \textStigma
17810      \texttheta       \textTheta
17811      \texttau         \textTau
17812      \textupsilon     \textUpsilon
17813      \textxi          \textXi
17814      \textzeta        \textZeta
17815      \textsigma       \textSigma
17816      \textvarsigma    \textSigma
17817      \textvarstigma   \textStigma
17818    }
17819    { \textsigma }
17820    \tl_const:Nn \c__tl_change_case_acc_upper_tl
17821    {
17822      \accdasia          { \exp_stop_f: \LGR@accdropped }
17823      \accdasiaoxia      { \exp_stop_f: \LGR@hiatus }
17824      \accdasiaavaria    { \exp_stop_f: \LGR@accdropped }
17825      \accdasiaperispomeni { \exp_stop_f: \LGR@accdropped }
17826      \accpsili          { \exp_stop_f: \LGR@hiatus }
17827      \accpsilioxia      { \exp_stop_f: \LGR@hiatus }
17828      \accpsilivaria     { \exp_stop_f: \LGR@hiatus }
17829      \accpsiliperispomeni { \exp_stop_f: \LGR@accdropped }
17830      \acctonos          { \exp_stop_f: \LGR@hiatus }
17831      \accvaria          { \exp_stop_f: \LGR@accdropped }
17832      \accdialytikatonos { \exp_stop_f: \LGR@accDialytika }
17833      \accdialytikavaria { \exp_stop_f: \LGR@accDialytika }
17834      \accdialytikaperispomeni { \exp_stop_f: \LGR@accDialytika }
17835      \accperispomeni    { \exp_stop_f: \LGR@accdropped }
17836    }
17837    \tl_const:Nn \c__tl_change_case_acc_lower_tl { }
17838    \tl_const:Nn \c__tl_change_case_misc_upper_tl
17839    {
17840      \ypogegrammeni { \exp_stop_f: \prosgegrammeni }
17841      \abreve         { \exp_stop_f: \Abreve }
17842      \acircumflex    { \exp_stop_f: \Acircumflex }
17843      \ecircumflex    { \exp_stop_f: \Ecircumflex }
17844      \ocircumflex    { \exp_stop_f: \Ocircumflex }
17845      \ohorn          { \exp_stop_f: \Ohorn }
17846      \uhorn          { \exp_stop_f: \Uhorn }
17847    }
17848    \tl_const:Nn \c__tl_change_case_misc_lower_tl
17849    {
17850      \prosgegrammeni { \exp_stop_f: \ypogegrammeni }

```

```

17851 \Abreve { \exp_stop_f: \abreve }
17852 \Acircumflex { \exp_stop_f: \acircumflex }
17853 \Ecircumflex { \exp_stop_f: \ecircumflex }
17854 \Ocircumflex { \exp_stop_f: \ocircumflex }
17855 \Ohorn { \exp_stop_f: \ohorn }
17856 \Uhorn { \exp_stop_f: \uhorn }
17857 \ABREVE { \exp_stop_f: \abreve }
17858 \ACIRCUMFLEX { \exp_stop_f: \acircumflex }
17859 \ECIRCUMFLEX { \exp_stop_f: \ecircumflex }
17860 \OCIRCUMFLEX { \exp_stop_f: \ocircumflex }
17861 \OHORN { \exp_stop_f: \ohorn }
17862 \UHORN { \exp_stop_f: \uhorn }
17863 }
17864 \group_end:
17865 \</package>

```

(End definition for `\c__tl_change_case_latin_upper_tl` and others. These variables are documented on page ??.)

`\tl_log:N` Redirect output of `\tl_show:N` to the log.

```

\tl_log:c 17866 \cs_new_protected_nopar:Npn \tl_log:N
          17867 { \_msg_log_next: \tl_show:N }
          17868 \cs_generate_variant:Nn \tl_log:N { c }

```

(End definition for `\tl_log:N` and `\tl_log:c`. These functions are documented on page 226.)

`\tl_log:n` Redirect output of `\tl_show:n` to the log.

```

17869 \cs_new_protected_nopar:Npn \tl_log:n
17870 { \_msg_log_next: \tl_show:n }

```

(End definition for `\tl_log:n`. This function is documented on page 226.)

33.20 Additions to l3tokens

```

17871 \@@=char

```

`\char_set_active_eq:NN` Four simple functions with very similar definitions, so set up using an auxiliary.

```

\char_gset_active_eq:NN 17872 \group_begin:
\char_set_active_eq:nN 17873 \char_set_catcode_active:N \^^@
\char_gset_active_eq:nN 17874 \cs_set_protected:Npn \_char_tmp:nN #1#2
17875 {
17876   \cs_new_protected:cpn { #1 :nN } ##1
17877   {
17878     \group_begin:
17879     \char_set_catcode_active:n { ##1 }
17880     \char_set_lccode:nn { '\^^@ } { ##1 }
17881     \tex_lowercase:D { \group_end: #2 ^^@ }
17882   }
17883   \cs_new_protected:cpn { #1 :NN } ##1
17884   { \exp_not:c { #1 : nN } { '##1 } }
17885 }

```

```

17886 \__char_tmp:nN { char_set_active_eq } \cs_set_eq:NN
17887 \__char_tmp:nN { char_gset_active_eq } \cs_gset_eq:NN
17888 \group_end:

```

(End definition for `\char_set_active_eq:NN` and others. These functions are documented on page 226.)

```

\char_generate:nn
\__char_generate_aux:nn
\__char_generate_aux:nnw
\l__char_tmp_tl
\c__char_max_int
\_char_generate_invalid_catcode:

```

The aim here is to generate characters of (broadly) arbitrary category code. Where possible, that is done using engine support (Xe_{La}TeX, Lua_{TeX}). 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.

```

17889 \cs_new:Npn \char_generate:nn #1#2
17890 {
17891   \exp:w \exp_after:wN \__char_generate_aux:w
17892   \int_use:N \__int_eval:w #1 \exp_after:wN ;
17893   \int_use:N \__int_eval:w #2 ;
17894 }

```

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 Lua_{TeX} too. Spaces are also banned here as Lua_{TeX} emulation only makes normal (charcode 32 spaces). However, `^^@` 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.

```

17895 \cs_new:Npn \__char_generate_aux:w #1 ; #2 ;
17896 {
17897   \if_int_compare:w #2 = \c_thirteen
17898     \__msg_kernel_expandable_error:nn { kernel } { char-active }
17899   \else:
17900     \if_int_compare:w #2 = \c_ten
17901       \if_int_compare:w #1 = \c_zero
17902         \__msg_kernel_expandable_error:nn { kernel } { char-null-space }
17903       \else:
17904         \__msg_kernel_expandable_error:nn { kernel } { char-space }
17905       \fi:
17906     \else:
17907       \if_int_odd:w 0
17908         \if_int_compare:w #2 < \c_one      1 \fi:
17909         \if_int_compare:w #2 = \c_five    1 \fi:
17910         \if_int_compare:w #2 = \c_nine    1 \fi:
17911         \if_int_compare:w #2 > \c_thirteen 1 \fi: \exp_stop_f:
17912         \__msg_kernel_expandable_error:nn { kernel }
17913         { char-invalid-catcode }
17914       \else:
17915         \if_int_odd:w 0
17916           \if_int_compare:w #1 < \c_zero      1 \fi:
17917           \if_int_compare:w #1 > \c__char_max_int 1 \fi: \exp_stop_f:
17918           \__msg_kernel_expandable_error:nn { kernel }
17919           { char-out-of-range }
17920         \else:
17921           \__char_generate_aux:nnw {#1} {#2}

```

```

17922         \fi:
17923     \fi:
17924     \fi:
17925     \fi:
17926     \exp_end:
17927 }
17928 \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.

```

17929 \group_begin:
17930 <*package>
17931     \cs_set_nopar:Npn ^^L { }
17932 </package>
17933     \char_set_catcode_other:n { 0 }
17934     \if_int_odd:w 0
17935         \cs_if_exist:NT \luatex_directlua:D { 1 }
17936         \cs_if_exist:NT \utex_charcat:D { 1 } \exp_stop_f:
17937         \int_const:Nn \c__char_max_int { 1114111 }
17938         \cs_if_exist:NTF \luatex_directlua:D
17939         {
17940             \cs_new:Npn \__char_generate_aux:nnw #1#2#3 \exp_end:
17941             {
17942                 #3
17943                 \exp_after:wN \exp_end:
17944                 \luatex_directlua:D { l3kernel.charcat(#1, #2) }
17945             }
17946         }
17947         {
17948             \cs_new:Npn \__char_generate_aux:nnw #1#2#3 \exp_end:
17949             {
17950                 #3
17951                 \exp_after:wN \exp_end:
17952                 \utex_charcat:D #1 ~ #2 ~
17953             }
17954         }
17955     \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 here and will later be `x`-type expanded into the desired form. 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).

```

17956 \int_const:Nn \c__char_max_int { 255 }
17957 \tl_set:Nn \l__char_tmp_tl { \exp_not:N \or: }
17958 \char_set_catcode_group_begin:n { 0 } % {
17959 \tl_put_right:Nn \l__char_tmp_tl { ^^@ \if_false: } }
17960 \char_set_catcode_group_end:n { 0 }
17961 \tl_put_right:Nn \l__char_tmp_tl { { \fi: \exp_not:N \or: ^^@ } % }
17962 \tl_set:Nx \l__char_tmp_tl { \l__char_tmp_tl }
17963 \char_set_catcode_math_toggle:n { 0 }
17964 \tl_put_right:Nn \l__char_tmp_tl { \or: ^^@ }
17965 \char_set_catcode_alignment:n { 0 }
17966 \tl_put_right:Nn \l__char_tmp_tl { \or: ^^@ }
17967 \tl_put_right:Nn \l__char_tmp_tl { \or: }
17968 \char_set_catcode_parameter:n { 0 }
17969 \tl_put_right:Nn \l__char_tmp_tl { \or: ^^@ }
17970 \char_set_catcode_math_superscript:n { 0 }
17971 \tl_put_right:Nn \l__char_tmp_tl { \or: ^^@ }
17972 \char_set_catcode_math_subscript:n { 0 }
17973 \tl_put_right:Nn \l__char_tmp_tl { \or: ^^@ }
17974 \tl_put_right:Nn \l__char_tmp_tl { \or: }
17975 \char_set_catcode_space:n { 0 }
17976 \tl_put_right:No \l__char_tmp_tl { \use:n { \or: } ^^@ }
17977 \char_set_catcode_letter:n { 0 }
17978 \tl_put_right:Nn \l__char_tmp_tl { \or: ^^@ }
17979 \char_set_catcode_other:n { 0 }
17980 \tl_put_right:Nn \l__char_tmp_tl { \or: ^^@ }
17981 \char_set_catcode_active:n { 0 }
17982 \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 deals with the \if_false: stuff introduced earlier. This is done in three parts as ^^L is awkward. Notice that at this stage ^^@ is active. In format mode this is not needed.

```

17983 \cs_set_protected:Npn \__char_tmp:n #1
17984 {
17985 \char_set_lccode:nn { 0 } {#1}
17986 \char_set_lccode:nn { 32 } {#1}
17987 \exp_args:Nx \tex_lowercase:D
17988 {
17989 \tl_const:cn { c__char_ \__int_to_roman:w #1 _tl }
17990 { \exp_not:o \l__char_tmp_tl }
17991 }
17992 }
17993 <*package>
17994 \int_step_function:nnnN { 0 } { 1 } { 11 } \__char_tmp:n
17995 \group_begin:
17996 \tl_replace_once:Nnn \l__char_tmp_tl { ^^@ } { \ERROR }
17997 \__char_tmp:n { 12 }
17998 \group_end:

```

```

17999      \int_step_function:nnnN { 13 } { 1 } { 255 } \__char_tmp:n
18000 </package>
18001 <*initex>
18002      \int_step_function:nnnN { 0 } { 1 } { 255 } \__char_tmp:n
18003 </initex>
18004      \cs_new:Npn \__char_generate_aux:nnw #1#2#3 \exp_end:
18005      {
18006          #3
18007          \exp_after:wN \exp_after:wN
18008          \exp_after:wN \exp_end:
18009          \exp_after:wN \exp_after:wN
18010          \if_case:w #2
18011              \exp_last_unbraced:Nv \exp_stop_f:
18012              { c__char_ \__int_to_roman:w #1 _tl }
18013          \fi:
18014      }
18015      \fi:
18016 \group_end:

```

Job done, set up a few messages.

```

18017 \__msg_kernel_new:nnn { kernel } { char-active }
18018 { Cannot~generate~active~chars. }
18019 \__msg_kernel_new:nnn { kernel } { char-invalid-catcode }
18020 { Invalid~catcode~for~char~generation. }
18021 \__msg_kernel_new:nnn { kernel } { char-null-space }
18022 { Cannot~generate~null~char~as~a~space. }
18023 \__msg_kernel_new:nnn { kernel } { char-out-of-range }
18024 { Charcode~requested~out~of~engine~range. }
18025 \__msg_kernel_new:nnn { kernel } { char-space }
18026 { Cannot~generate~space~chars. }

```

(End definition for \char_generate:nn. This function is documented on page 227.)

```

18027 <@@=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.

```

18028 \group_begin:
18029 \cs_set_protected:Npn \__peek_tmp:w #1 \q_stop
18030 {
18031   \cs_new_protected_nopar:Npn \__peek_execute_branches_N_type:
18032   {
18033     \if_int_odd:w
18034       \if_catcode:w \exp_not:N \l_peek_token { \c_two \fi:
18035       \if_catcode:w \exp_not:N \l_peek_token } \c_two \fi:
18036       \if_meaning:w \l_peek_token \c_space_token \c_two \fi:
18037       \c_one
18038       \exp_after:wN \__peek_N_type:w
18039       \token_to_meaning:N \l_peek_token
18040       \q_mark \__peek_N_type_aux:nnw
18041       #1 \q_mark \use_none_delimit_by_q_stop:w
18042       \q_stop
18043       \exp_after:wN \__peek_true:w
18044     \else:
18045       \exp_after:wN \__peek_false:w
18046     \fi:
18047   }
18048   \cs_new_protected:Npn \__peek_N_type:w ##1 #1 ##2 \q_mark ##3
18049   { ##3 {##1} {##2} }
18050 }
18051 \exp_after:wN \__peek_tmp:w \tl_to_str:n { outer } \q_stop
18052 \group_end:
18053 \cs_new_protected:Npn \__peek_N_type_aux:nnw #1 #2 #3 \fi:
18054 {
18055   \fi:
18056   \tl_if_in:noTF {#1} { \tl_to_str:n {ma} }
18057   { \__peek_true:w }
18058   { \tl_if_empty:nTF {#2} { \__peek_true:w } { \__peek_false:w } }
18059 }
18060 \cs_new_protected_nopar:Npn \peek_N_type:TF
18061 { \__peek_token_generic:NNTF \__peek_execute_branches_N_type: \scan_stop: }
18062 \cs_new_protected_nopar:Npn \peek_N_type:T
18063 { \__peek_token_generic:NNT \__peek_execute_branches_N_type: \scan_stop: }
18064 \cs_new_protected_nopar:Npn \peek_N_type:F
18065 { \__peek_token_generic:NNTF \__peek_execute_branches_N_type: \scan_stop: }

(End definition for \peek_N_type:TF. This function is documented on page 227.)

18066 \</initex | package>

```

34 l3sys implementation

```

18067 \<*initex | package>

```

34.1 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.

```

18068 <*initex>
18069 \tex_everyjob:D \exp_after:wN
18070 {
18071   \tex_the:D \tex_everyjob:D
18072   \str_const:Nx \c_sys_jobname_str { \tex_jobname:D }
18073 }
18074 </initex>
18075 <*package>
18076 \str_const:Nx \c_sys_jobname_str { \tex_jobname:D }
18077 </package>

```

(End definition for `\c_sys_jobname_str`. This variable is documented on page 228.)

34.2 Time and date

`\c_sys_minute_int` Copies of the information provided by T_EX

```

\c_sys_hour_int      18078 \int_const:Nn \c_sys_minute_int
\c_sys_day_int       18079 { \int_mod:nn { \tex_time:D } { 60 } }
\c_sys_month_int     18080 \int_const:Nn \c_sys_hour_int
\c_sys_year_int      18081 { \int_div_truncate:nn { \tex_time:D } { 60 } }
18082 \int_const:Nn \c_sys_day_int { \tex_day:D }
18083 \int_const:Nn \c_sys_month_int { \tex_month:D }
18084 \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 228.)

34.3 Detecting the engine

`\sys_if_engine luatex_p:` Set up the engine tests on the basis exactly one test should be true. Mainly a case of
`\sys_if_engine pdftex_p:` looking for the appropriate marker primitive. For up_l^EX, there is a complexity in that
`\sys_if_engine ptex_p:` setting `-kanji-internal=sjis` or `-kanji-internal=euc` effective makes it more like
`\sys_if_engine uptex_p:` p_l^EX. In those cases we therefore report p_l^EX rather than up_l^EX.
`\sys_if_engine xetex_p:`

```

18085 \clist_map_inline:nn { lua , pdf , p , up , xe }
\sys_if_engine luatex:TF 18086 {
\sys_if_engine pdftex:TF 18087   \cs_new_eq:cN { sys_if_engine_ #1 tex:T } \use_none:n
\sys_if_engine ptex:TF   18088   \cs_new_eq:cN { sys_if_engine_ #1 tex:F } \use:n
\sys_if_engine uptex:TF   18089   \cs_new_eq:cN { sys_if_engine_ #1 tex:TF } \use_ii:nn
\sys_if_engine xetex:TF   18090   \cs_new_eq:cN { sys_if_engine_ #1 tex_p: } \c_false_bool
\c_sys_engine_str         18091 }
18092 \cs_if_exist:NT \luatex luatexversion:D
18093 {
18094   \cs_gset_eq:NN \sys_if_engine luatex:T \use:n
18095   \cs_gset_eq:NN \sys_if_engine luatex:F \use_none:n
18096   \cs_gset_eq:NN \sys_if_engine luatex:TF \use_i:nn
18097   \cs_gset_eq:NN \sys_if_engine luatex_p: \c_true_bool

```

```

18098     \str_const:Nn \c_sys_engine_str { luatex }
18099   }
18100   \cs_if_exist:NT \pdfTeX_pdfTeXversion:D
18101   {
18102     \cs_gset_eq:NN \sys_if_engine_pdfTeX:T \use:n
18103     \cs_gset_eq:NN \sys_if_engine_pdfTeX:F \use_none:n
18104     \cs_gset_eq:NN \sys_if_engine_pdfTeX:TF \use_i:nn
18105     \cs_gset_eq:NN \sys_if_engine_pdfTeX_p: \c_true_bool
18106     \str_const:Nn \c_sys_engine_str { pdfTeX }
18107   }
18108   \cs_if_exist:NT \ptex_kanjiskip:D
18109   {
18110     \bool_if:nTF
18111     {
18112       \cs_if_exist_p:N \uptex_disablecjktoken:D &&
18113       \int_compare_p:nNn { \ptex_jis:D "2121 } = { "3000 }
18114     }
18115     {
18116       \cs_gset_eq:NN \sys_if_engine_uptex:T \use:n
18117       \cs_gset_eq:NN \sys_if_engine_uptex:F \use_none:n
18118       \cs_gset_eq:NN \sys_if_engine_uptex:TF \use_i:nn
18119       \cs_gset_eq:NN \sys_if_engine_uptex_p: \c_true_bool
18120       \str_const:Nn \c_sys_engine_str { uptex }
18121     }
18122     {
18123       \cs_gset_eq:NN \sys_if_engine_ptex:T \use:n
18124       \cs_gset_eq:NN \sys_if_engine_ptex:F \use_none:n
18125       \cs_gset_eq:NN \sys_if_engine_ptex:TF \use_i:nn
18126       \cs_gset_eq:NN \sys_if_engine_ptex_p: \c_true_bool
18127       \str_const:Nn \c_sys_engine_str { ptex }
18128     }
18129   }
18130   \cs_if_exist:NT \xetex_XeTeXversion:D
18131   {
18132     \cs_gset_eq:NN \sys_if_engine_xetex:T \use:n
18133     \cs_gset_eq:NN \sys_if_engine_xetex:F \use_none:n
18134     \cs_gset_eq:NN \sys_if_engine_xetex:TF \use_i:nn
18135     \cs_gset_eq:NN \sys_if_engine_xetex_p: \c_true_bool
18136     \str_const:Nn \c_sys_engine_str { xetex }
18137   }

```

(End definition for `\sys_if_engine_luatex:TF` and others. These functions are documented on page 228.)

34.4 Detecting the output

`\sys_if_output_dvi_p:` This is a simple enough concept: the two views here are complementary.

`\sys_if_output_pdf_p:` 18138 `\bool_if:nTF`

`\sys_if_output_dvi:TF` 18139 `{`

`\sys_if_output_pdf:TF` 18140 `\cs_if_exist_p:N \pdfTeX_pdfoutput:D`

`\c_sys_output_str`

```

18141    && \int_compare_p:nNn \pdfTeX_pdfoutput:D > \c_zero
18142  }
18143  {
18144    \cs_new_eq:NN \sys_if_output_dvi:T \use_none:n
18145    \cs_new_eq:NN \sys_if_output_dvi:F \use:n
18146    \cs_new_eq:NN \sys_if_output_dvi:TF \use_ii:nn
18147    \cs_new_eq:NN \sys_if_output_dvi_p: \c_false_bool
18148    \cs_new_eq:NN \sys_if_output_pdf:T \use:n
18149    \cs_new_eq:NN \sys_if_output_pdf:F \use_none:n
18150    \cs_new_eq:NN \sys_if_output_pdf:TF \use_i:nn
18151    \cs_new_eq:NN \sys_if_output_pdf_p: \c_true_bool
18152    \str_const:Nn \c_sys_output_str { pdf }
18153  }
18154  {
18155    \cs_new_eq:NN \sys_if_output_dvi:T \use:n
18156    \cs_new_eq:NN \sys_if_output_dvi:F \use_none:n
18157    \cs_new_eq:NN \sys_if_output_dvi:TF \use_i:nn
18158    \cs_new_eq:NN \sys_if_output_dvi_p: \c_true_bool
18159    \cs_new_eq:NN \sys_if_output_pdf:T \use_none:n
18160    \cs_new_eq:NN \sys_if_output_pdf:F \use:n
18161    \cs_new_eq:NN \sys_if_output_pdf:TF \use_ii:nn
18162    \cs_new_eq:NN \sys_if_output_pdf_p: \c_false_bool
18163    \str_const:Nn \c_sys_output_str { dvi }
18164  }

```

(End definition for `\sys_if_output_dvi:TF` and `\sys_if_output_pdf:TF`. These functions are documented on page 229.)

34.5 Deprecated functions

Deprecated 2015-09-07 for removal after 2016-12-31. The older logic supported only three engines so that has to be allowed for.

```

18165 \prg_new_eq_conditional:NNn \luatex_if_engine: \sys_if_engine luatex:
18166 { T , F , TF , p }
18167 \prg_new_eq_conditional:NNn \xetex_if_engine: \sys_if_engine xetex:
18168 { T , F , TF , p }
18169 \bool_if:nTF
18170 {
18171   \sys_if_engine luatex_p: ||
18172   \sys_if_engine xetex_p:
18173 }
18174 {
18175   \cs_new_eq:NN \pdfTeX_if_engine:T \use_none:n
18176   \cs_new_eq:NN \pdfTeX_if_engine:F \use:n
18177   \cs_new_eq:NN \pdfTeX_if_engine:TF \use_ii:nn
18178   \cs_new_eq:NN \pdfTeX_if_engine_p: \c_false_bool
18179 }
18180 {
18181   \cs_new_eq:NN \pdfTeX_if_engine:T \use:n
18182   \cs_new_eq:NN \pdfTeX_if_engine:F \use_none:n

```

```

18183 \cs_new_eq:NN \pdfTeX_if_engine:TF \use_i:nn
18184 \cs_new_eq:NN \pdfTeX_if_engine_p: \c_true_bool
18185 }
    Deprecated 2015-09-19 for removal after 2016-12-31.
18186 \cs_set_eq:NN \c_job_name_tl \c_sys_jobname_str
18187 </initex | package>

```

35 l3luatex implementation

```

18188 <*initex | package>

```

35.0.1 Breaking out to Lua

`\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/

```

18189 \cs_new:Npn \lua_now_x:n #1 { \luatex_directlua:D {#1} }
18190 \cs_new:Npn \lua_now:n #1 { \lua_now_x:n { \exp_not:n {#1} } }
18191 \cs_new_protected:Npn \lua_shipout_x:n #1 { \luatex_latelua:D {#1} }
18192 \cs_new_protected:Npn \lua_shipout:n #1
18193 { \lua_shipout_x:n { \exp_not:n {#1} } }
18194 \cs_new:Npn \lua_escape_x:n #1 { \luatex_luaescapestring:D {#1} }
18195 \cs_new:Npn \lua_escape:n #1 { \lua_escape_x:n { \exp_not:n {#1} } }
18196 \sys_if_engine luatex:F
18197 {
18198   \clist_map_inline:nn
18199   { \lua_now_x:n , \lua_now:n , \lua_escape_x:n , \lua_escape:n }
18200   {
18201     \cs_set:Npn #1 ##1
18202     {
18203       \__msg_kernel_expandable_error:nnn
18204       { kernel } { luatex-required } { #1 }
18205     }
18206   }
18207   \clist_map_inline:nn
18208   { \lua_shipout_x :n , \lua_shipout:n }
18209   {
18210     \cs_set_protected:Npn #1 ##1
18211     {
18212       \__msg_kernel_error:nnn
18213       { kernel } { luatex-required } { #1 }
18214     }
18215   }
18216 }

```

(End definition for `\lua_now_x:n` and `\lua_now:n`. These functions are documented on page 230.)

35.1 Messages

```

18217 \_msg_kernel_new:nnnn { kernel } { luatex-required }
18218 { LuaTeX~engine~not~in~use!~Ignoring~#1. }
18219 {
18220   The~feature~you~are~using~is~only~available~
18221   with~the~LuaTeX~engine.~LaTeX3~ignored~'~#1'~.
18222 }
18223 </initex | package>

```

36 l3drivers Implementation

```

18224 <*initex | package>
18225 <@@=driver>
18226 <*package>
18227 \ProvidesExplFile
18228 <*dvipdfmx>
18229 {l3dvidpfmx.def}{\ExplFileDate}{\ExplFileVersion}
18230 {L3 Experimental driver: dvipdfmx}
18231 </dvipdfmx>
18232 <*dvips>
18233 {l3dvips.def}{\ExplFileDate}{\ExplFileVersion}
18234 {L3 Experimental driver: dvips}
18235 </dvips>
18236 <*pdfmode>
18237 {l3pdfmode.def}{\ExplFileDate}{\ExplFileVersion}
18238 {L3 Experimental driver: PDF mode}
18239 </pdfmode>
18240 <*xdvipdfmx>
18241 {l3xdvidpfmx.def}{\ExplFileDate}{\ExplFileVersion}
18242 {L3 Experimental driver: xdvipdfmx}
18243 </xdvipdfmx>
18244 </package>

```

36.1 Settings for direct PDF output

If the driver loaded is pdfmode then direct PDF output is required. (This may of course alter: it might be that the driver is picked based on the value of `\pdftex_pdfoutput:D`.)

```

18245 <*initex>
18246 <*pdfmode>
18247 \pdftex_pdfoutput:D = 1 \scan_stop:
18248 </pdfmode>
18249 </initex>

```

Set up the driver for direct PDF output to set the PDF origin equal to TeX's standard origin. The other settings make use of PDF 1.5, which is standard in TeX Live 2011 and should be a reasonable baseline for the future.

```

18250 <*initex>
18251 <*pdfmode>
18252 \pdftex_pdfhorigin:D = 1 true in \scan_stop:

```

```

18253 \pdfutex_pdfvorigin:D          = 1 true in \scan_stop:
18254 \pdfutex_pdfdecimaldigits:D   = 3          \scan_stop:
18255 \pdfutex_pdfpkresolution:D     = 600        \scan_stop:
18256 \pdfutex_pdfminorversion:D    = 5          \scan_stop:
18257 \pdfutex_pdfcompresslevel:D   = 9          \scan_stop:
18258 \pdfutex_pdfobjcompresslevel:D = 2          \scan_stop:
18259 </pdfmode>
18260 </initex>

```

36.2 Driver utility functions

`__driver_state_save:` All of the drivers have a stack for saving the graphic state. These have slightly different interfaces. For both dvips and (x)dvipdfmx this is done using an appropriate special. Note that here and later, the dvipdfmx documentation does not cover the `literal` key word but that this appears to behave in the same way as pdfTeX's `\pdfliteral` (making life easier all-round).

```

18261 <!*pdfmode>
18262 \cs_new_protected_nopar:Npn \__driver_state_save:
18263 <*dvips>
18264 { \tex_special:D { ps:gsave } }
18265 </dvips>
18266 <*dvipdfmx | xdvipdfmx>
18267 { \tex_special:D { pdf:literal~q } }
18268 </dvipdfmx | xdvipdfmx>
18269 \cs_new_protected_nopar:Npn \__driver_state_restore:
18270 <*dvips>
18271 { \tex_special:D { ps:grestore } }
18272 </dvips>
18273 <*dvipdfmx | xdvipdfmx>
18274 { \tex_special:D { pdf:literal~Q } }
18275 </dvipdfmx | xdvipdfmx>
18276 <!/pdfmode>

```

For direct PDF output there is also a need to worry about the version of pdfTeX in use: the `\pdfsave` primitive was only introduced in version 1.40.0.

```

18277 <*pdfmode>
18278 \cs_if_exist:NTF \pdfutex_pdfsave:D
18279 {
18280   \cs_new_eq:NN \__driver_state_save: \pdfutex_pdfsave:D
18281   \cs_new_eq:NN \__driver_state_restore: \pdfutex_pdfrestore:D
18282 }
18283 {
18284   \cs_new_protected_nopar:Npn \__driver_state_save:
18285   { \pdfutex_pdfliteral:D { q } }
18286   \cs_new_protected_nopar:Npn \__driver_state_restore:
18287   { \pdfutex_pdfliteral:D { Q } }
18288 }
18289 </pdfmode>

```

(End definition for `_driver_state_save:` and `_driver_state_restore:`. These functions are documented on page ??.)

`_driver_literal:n` The driver code needs to pass on a lot of “raw” information to the underlying binary. The exact command is driver-dependent but the concept is general enough to use a single function. However, it is important to remember this is a convenient shortcut: the arguments will be driver-specific. Note that these functions set the transformation matrix to the current position: contrast with `_driver_literal_direct:n`.

```
18290 \cs_new_protected:Npn \_driver\_literal:n #1
18291 { *dvipdfmx | xdvipdfmx }
18292 { \tex\_special:D { pdf:literal~ #1 } }
18293 { /dvipdfmx | xdvipdfmx }
```

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.

```
18294 { *dvips }
18295 {
18296   \tex\_special:D
18297   {
18298     ps:
18299     currentpoint~
18300     currentpoint~translate~
18301     #1 ~
18302     neg~exch~neg~exch~translate
18303   }
18304 }
18305 { /dvips }
18306 { *pdfmode }
18307 { \pdf\_tex\_pdfliteral:D {#1} }
18308 { /pdfmode }
```

(End definition for `_driver_literal:n`.)

`_driver_literal_direct:n` Even “lower level” than `_driver_literal:n`, these commands do not set the transformation matrix but simply dump the driver code directly into the output. In the `(x)dvipdfmx` case this two-part keyword is documented (*cf.* `literal` alone).

```
18309 \cs_new_protected:Npn \_driver\_literal\_direct:n #1
18310 { *dvipdfmx | xdvipdfmx }
18311 { \tex\_special:D { pdf:literal~direct~ #1 } }
18312 { /dvipdfmx | xdvipdfmx }
18313 { *dvips }
18314 { \tex\_special:D { ps:: #1 } }
18315 { /dvips }
18316 { *pdfmode }
18317 { \pdf\_tex\_pdfliteral:D direct {#1} }
18318 { /pdfmode }
```

(End definition for `_driver_literal_direct:n`.)

`_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`.

```

18319 <*dvips>
18320 \cs_new:Npn \_driver_absolute_lengths:n #1
18321 {
18322     /savedmatrix~matrix~currentmatrix~def~
18323     Resolution~72~div~VResolution~72~div~scale~
18324     DVImag~dup~scale~
18325     #1 ~
18326     savedmatrix~setmatrix
18327 }
18328 </dvips>

```

(End definition for `_driver_absolute_lengths:n`)

`_driver_matrix:n` Here the appropriate function is set up to insert an affine matrix into the PDF. With a new enough pdf \TeX (version 1.40.0 or later) there is a primitive for this, which only needs the rotation/scaling/skew part. With an older pdf \TeX or with (x)dvipdfmx the matrix also has to include a translation part: that is always zero and so is built in here.

```

18329 <*pdfmode>
18330 \cs_if_exist:NTF \pdfTeX_pdfsetmatrix:D
18331 {
18332     \cs_new_protected:Npn \_driver_matrix:n #1
18333     { \pdfTeX_pdfsetmatrix:D {#1} }
18334 }
18335 {
18336     \cs_new_protected:Npn \_driver_matrix:n #1
18337     { \_driver_literal:n { #1 \c_space_tl 0~0~cm } }
18338 }
18339 </pdfmode>
18340 <*dvipdfmx|xvipdfmx>
18341 \cs_new_protected:Npn \_driver_matrix:n #1
18342 { \_driver_literal:n { #1 \c_space_tl 0~0~cm } }
18343 </dvipdfmx|xvipdfmx>

```

(End definition for `_driver_matrix:n`)

36.3 Box clipping

`_driver_box_use_clip:N` The overall logic to clipping a box is the same in all cases. 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 three cases.

```

18344 \cs_new_protected:Npn \_driver_box_use_clip:N #1
18345 {
18346     \_driver_state_save:

```

```

18347 <*dvips>
18348   \_driver_literal:n
18349   {
18350     \_driver_absolute_lengths:n
18351     {
18352       0~
18353       \dim_to_decimal_in_bp:n { \box_dp:N #1 } ~
18354       \dim_to_decimal_in_bp:n { \box_wd:N #1 } ~
18355       \dim_to_decimal_in_bp:n { -\box_ht:N #1 - \box_dp:N #1 } ~
18356       rectclip
18357     }
18358   }
18359 </dvips>
18360 <*dvipdfmx | pdfmode | xdvipdfmx>
18361   \_driver_literal:n
18362   {
18363     0~
18364     \dim_to_decimal_in_bp:n { -\box_dp:N #1 } ~
18365     \dim_to_decimal_in_bp:n { \box_wd:N #1 } ~
18366     \dim_to_decimal_in_bp:n { \box_ht:N #1 + \box_dp:N #1 } ~
18367     re~W~n
18368   }
18369 </dvipdfmx | pdfmode | xdvipdfmx>

```

Insert the material in a box of no width, restore the graphic state and then insert the necessary width.

```

18370   \hbox_overlap_right:n { \box_use:N #1 }
18371   \_driver_state_restore:
18372   \skip_horizontal:n { \box_wd:N #1 }
18373 }

```

(End definition for `_driver_box_use_clip:N`. This function is documented on page 232.)

36.4 Box rotation and scaling

`_driver_box_rotate_begin:` The driver for `dvips` works with a simple rotation angle. In PDF mode, an affine matrix is used instead. The transformation for `(x)dvipdfmx` can be done either way: the affine approach is chosen here as where possible we pick the PDF-style route.

In both cases, some rounding code is included to limit the floating point values to five decimal places. There is no point using any more as \TeX 's dimensions are of that precision, and the extra figures will simply bloat the PDF and make values harder to trace. In the case where the sine and cosine are used, 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.

```

18374 \cs_new_protected_nopar:Npn \_driver_box_rotate_begin:
18375 {
18376   \_driver_state_save:
18377   <*dvipdfmx | pdfmode | xdvipdfmx>

```

```

18378 \box_set_wd:Nn \l__box_internal_box \c_zero_dim
18379 \fp_set:Nn \l__box_cos_fp { round ( \l__box_cos_fp , 5 ) }
18380 \fp_compare:nNnTF \l__box_cos_fp = \c_zero_fp
18381 { \fp_zero:N \l__box_cos_fp }
18382 \fp_set:Nn \l__box_sin_fp { round ( \l__box_sin_fp , 5 ) }
18383 \__driver_matrix:n
18384 {
18385   \fp_use:N \l__box_cos_fp \c_space_tl
18386   \fp_compare:nNnTF \l__box_sin_fp = \c_zero_fp
18387   { 0-0 }
18388   {
18389     \fp_use:N \l__box_sin_fp
18390     \c_space_tl
18391     \fp_eval:n { -\l__box_sin_fp }
18392   }
18393   \c_space_tl
18394   \fp_use:N \l__box_cos_fp
18395 }
18396 </dvipdfmx | pdfmode | xdvipdfmx>
18397 <*dvips>
18398 \fp_set:Nn \l__box_angle_fp { round ( \l__box_angle_fp , 5 ) }
18399 \__driver_literal:n
18400 {
18401   \fp_compare:nNnTF \l__box_angle_fp = \c_zero_fp
18402   { 0 }
18403   { \fp_eval:n { -\l__box_angle_fp } }
18404   \c_space_tl
18405   rotate
18406 }
18407 </dvips>
18408 }

```

The end of a rotation means tidying up the output grouping.

```

18409 \cs_new_eq:NN \__driver_box_rotate_end: \__driver_state_restore:

```

(End definition for `__driver_box_rotate_begin:` and `__driver_box_rotate_end:`. These functions are documented on page 233.)

`__driver_box_scale_begin:` Scaling is not dissimilar to rotation, but the calculations are somewhat less complex.

```

\__driver_box_scale_end:
18410 \cs_new_protected_nopar:Npn \__driver_box_scale_begin:
18411 {
18412   \__driver_state_save:
18413   \fp_set:Nn \l__box_scale_x_fp { round ( \l__box_scale_x_fp , 5 ) }
18414   \fp_set:Nn \l__box_scale_y_fp { round ( \l__box_scale_y_fp , 5 ) }
18415 <*dvips>
18416   \__driver_literal:n
18417   {
18418     \fp_use:N \l__box_scale_x_fp \c_space_tl
18419     \fp_use:N \l__box_scale_y_fp \c_space_tl
18420     scale

```

```

18421     }
18422   </dvips>
18423   <*dvipdfmx | pdfmode | xdvipdfmx>
18424     \__driver_matrix:n
18425     {
18426       \fp_use:N \l__box_scale_x_fp \c_space_tl
18427       0~0~
18428       \fp_use:N \l__box_scale_y_fp
18429     }
18430   </dvipdfmx | pdfmode | xdvipdfmx>
18431   }
18432   \cs_new_eq:NN \__driver_box_scale_end: \__driver_state_restore:

```

(End definition for `__driver_box_scale_begin:` and `__driver_box_scale_end:`. These functions are documented on page 233.)

36.5 Color support

`\l__driver_current_color_tl` The current color is needed by all of the engines, but the way this is stored varies.

```

18433   \tl_new:N \l__driver_current_color_tl
18434   <*dvipdfmx | dvips | xdvipdfmx>
18435   \tl_set:Nn \l__driver_current_color_tl { gray~0 }
18436   </dvipdfmx | dvips | xdvipdfmx>
18437   <*pdfmode>
18438   \tl_set:Nn \l__driver_current_color_tl { 0~g~0~G }
18439   </pdfmode>

```

(End definition for `\l__driver_current_color_tl`. This variable is documented on page ??.)

`\l__driver_color_stack_int` pdfTeX (version 1.40.0 or later) and LuaTeX have multiple stacks available, and the color stack therefore needs a number when in PDF mode.

```

18440   <*pdfmode>
18441   \int_new:N \l__driver_color_stack_int
18442   </pdfmode>

```

(End definition for `\l__driver_color_stack_int`. This variable is documented on page ??.)

`__driver_color_ensure_current:` `__driver_color_reset:` Setting the current color depends on the nature of the color stack available. In all cases there is a need to reset the color after the current group.

```

18443   <*dvipdfmx | dvips | xdvipdfmx>
18444   \cs_new_protected_nopar:Npn \__driver_color_ensure_current:
18445   {
18446     \tex_special:D { color~push~\l__driver_current_color_tl }
18447     \group_insert_after:N \__driver_color_reset:
18448   }
18449   \cs_new_protected_nopar:Npn \__driver_color_reset:
18450   { \tex_special:D { color~pop } }
18451   </dvipdfmx | dvips | xdvipdfmx>

```

Once again there is a version switch for pdfTeX, as the `\pdfcolorstack` primitive was introduced in version 1.40.0.

```

18452 <*pdfmode>
18453 \cs_if_exist:NTF \pdfTEX_pdfcolorstack:D
18454 {
18455   \cs_new_protected_nopar:Npn \__driver_color_ensure_current:
18456   {
18457     \pdfTEX_pdfcolorstack:D \l__driver_color_stack_int push
18458     { \l__driver_current_color_tl }
18459     \group_insert_after:N \__driver_color_reset:
18460   }
18461   \cs_new_protected_nopar:Npn \__driver_color_reset:
18462   { \pdfTEX_pdfcolorstack:D \l__driver_color_stack_int pop \scan_stop: }
18463 }
18464 {
18465   \cs_new_protected_nopar:Npn \__driver_color_ensure_current:
18466   {
18467     \__driver_literal:n { \l__driver_current_color_tl }
18468     \group_insert_after:N \__driver_color_reset:
18469   }
18470   \cs_new_protected_nopar:Npn \__driver_color_reset:
18471   { \__driver_literal:n { \l__driver_current_color_tl } }
18472 }
18473 </pdfmode>

```

(End definition for `__driver_color_ensure_current:`. This function is documented on page 233.)

```

18474 </initex | package>

```

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	
\!	36, 289, 289, 294, 295, 295, 295, 295,
\"	295, 295, 295, 295, 295, 295, 295, 295,
\#	295, 295, 295, 295, 295, 295, 295, 296
\\$	\::o_unbraced . 295, 295, 296, 296, 296, 296
\%	\::p . 36, 289, 289, 289
\&	\::v . 36, 290, 290
&&	\::v_unbraced . 295, 295
\(\::x . 36, 290, 290, 294,
\(pdf)strcmp	295, 295, 295, 295, 295, 295, 295,
\)	295, 295, 295, 295, 295, 295, 295, 295
*	\::x_unbraced . 295, 296, 296
*	:N . 635
**	< . 204
+	= . 204
\+	? . 204
,	\? . 638
-	? commands:
\-	? : . 203
.. commands:	\
\..._new	501, 501, 501, 504, 504, 505, 505,
/	505, 505, 505, 505, 506, 506, 506,
\/	506, 506, 506, 506, 506, 506, 506,
\:	508, 508, 510, 510, 511, 511, 517,
\::	517, 517, 517, 518, 518, 518, 518,
36, 289, 289, 289, 289, 289, 289, 289,	518, 518, 519, 519, 519, 519, 519,
289, 289, 289, 289, 289, 289, 290,	525, 525, 525, 549, 549, 549, 549,
290, 290, 290, 294, 294, 294, 294,	549, 549, 550, 550, 550, 567, 571,
294, 295, 295, 295, 295, 295, 295,	571, 591, 591, 592, 592, 592, 592, 592,
295, 295, 295, 295, 295, 295, 295,	\{ . 4, 2978, 6135, 8690, 9147,
295, 295, 295, 295, 295, 295, 295,	9349, 9353, 9354, 10681, 10681, 17611
295, 295, 295, 295, 295, 295, 295,	\} . 5, 2978, 6136, 8691, 9147,
295, 296, 296, 296, 296, 296, 296, 305	9349, 9353, 9354, 10683, 10683, 17611
\::N	... commands:
295, 295, 295, 295, 295, 295, 295, 296	\l... . 481
\::V	\..._open:Nn . 185
36, 290, 290, 290, 295	(name) commands:
\::V_unbraced	\(name):(arg spec) . 37, 37, 37, 37
295, 295	\(name):(arg spec)F . 38
\::c	\(name):(arg spec)T . 38
36, 289,	\(name):(arg spec)TF . 38
289, 294, 295, 295, 295, 295, 295, 295	\(name)_p:(arg spec) . 38
\::error	(type) commands:
219, 782	\(type)_map_break: . 45, 45, 45, 45, 49, 49, 49
\::f	
36, 290, 290, 294, 294, 294, 294, 295, 296	
\::f_unbraced	
295, 295	
\::n	
36, 289, 289, 294, 294,	
295, 295, 295, 295, 295, 295, 295,	
295, 295, 295, 295, 295, 295, 296, 296	

$\backslash\langle type \rangle_map_break:n$	45	acot	207
$\backslash\langle type \rangle_use:N$	192	acotd	207
$\backslash^$	234, 234, 238, 278, 298, 298, 327, 327, 391, 427, 507, 521, 564, 566, 566, 566, 566, 566, 566, 566, 566, 635, 717, 811, 811	acsc	206
\wedge	204	acscd	207
$\backslash_$	235, 236, 327, 327, 427	$\backslash\operatorname{adjdemerits}$	242
$\backslash\sim$	327, 327, 392, 427, 567, 567	$\backslash\operatorname{advance}$	242
$\backslash\sqcup$	242, 274, 327, 334, 334, 335, 335, 336, 336, 336, 336, 418, 507, 507, 517, 518, 518, 518, 519, 519, 519, 519, 521, 525, 525, 525, 525, 525, 525, 525, 525, 525, 525, 525, 525, 525, 566, 567	$\backslash\operatorname{AE}$	806
A		$\backslash\operatorname{ae}$	806
$\backslash\operatorname{A}$	418, 418	$\backslash\operatorname{afterassignment}$	242
$\backslash\operatorname{AA}$	806	$\backslash\operatorname{aftergroup}$	242
$\backslash\operatorname{aa}$	806	$\backslash\operatorname{alignmark}$	253
$\backslash\operatorname{above}$	242	alignment commands:	
$\backslash\operatorname{abovedisplayshortskip}$	242	$\backslash\operatorname{c_alignment_token}$	56, 328, 328, 329, 329
$\backslash\operatorname{abovedisplayskip}$	242	$\backslash\operatorname{aligntab}$	253
$\backslash\operatorname{abovewithdelims}$	242	ampersand commands:	
$\backslash\operatorname{ABREVE}$	811	$\backslash\operatorname{c_ampersand_str}$	117, 426, 426
$\backslash\operatorname{Abreve}$	810, 810	asec	206
$\backslash\operatorname{abreve}$	810, 810, 811	asecd	207
abs	204	asin	206
$\backslash\operatorname{accdasia}$	810	asind	207
$\backslash\operatorname{accdasiaoxia}$	810	atan	207
$\backslash\operatorname{accdasiaperispomeni}$	810	atand	207
$\backslash\operatorname{accdasiavaria}$	810	$\backslash\operatorname{AtBeginDocument}$	502, 555, 556
$\backslash\operatorname{accdialytikaperispomeni}$	810	$\backslash\operatorname{atop}$	242
$\backslash\operatorname{accdialytikatonos}$	810	$\backslash\operatorname{atopwithdelims}$	242
$\backslash\operatorname{accdialytikavaria}$	810	atsign commands:	
$\backslash\operatorname{accent}$	242	$\backslash\operatorname{c_atsign_str}$	117, 426, 426
$\backslash\operatorname{accperispomeni}$	810	$\backslash\operatorname{attribute}$	253
$\backslash\operatorname{accpsili}$	810	$\backslash\operatorname{attributedef}$	253
$\backslash\operatorname{accpsilioxia}$	810	$\backslash\operatorname{autospaceing}$	257
$\backslash\operatorname{accpsiliperispomeni}$	810	$\backslash\operatorname{autoxspacing}$	257
$\backslash\operatorname{accpsilivaria}$	810	B	
$\backslash\operatorname{acctonos}$	810	backslash commands:	
$\backslash\operatorname{accvaria}$	810	$\backslash\operatorname{c_backslash_str}$	117, 426, 427
$\backslash\operatorname{ACIRCUMFLEX}$	811	$\backslash\operatorname{badness}$	242
$\backslash\operatorname{Acircumflex}$	810, 810	$\backslash\operatorname{baselineskip}$	242
$\backslash\operatorname{acircumflex}$	810, 810, 811	$\backslash\operatorname{batchmode}$	242
acos	206	$\backslash\operatorname{begingroup}$	234, 234, 235, 235, 236, 236, 237, 239, 241, 242
acosd	207	$\backslash\operatorname{beginL}$	248
		$\backslash\operatorname{beginR}$	248
		$\backslash\operatorname{belowdisplayshortskip}$	242
		$\backslash\operatorname{belowdisplayskip}$	242
		$\backslash\operatorname{binoppenalty}$	242
		$\backslash\operatorname{bodydir}$	254
		bool commands:	
		$\backslash_bool_0:w$	312
		$\backslash_bool_1:w$	312

- _bool_(:Nw [311](#)
- _bool_)_0:w [312](#)
- _bool_)_1:w [312](#)
- _bool_):Nw [311](#)
- _bool_choose:NNN . [311](#), [311](#), [311](#), [311](#)
- \bool_do_until:cn [314](#)
- \bool_do_until:Nn
..... [42](#), [42](#), [314](#), [314](#), [314](#), [314](#)
- \bool_do_until:nn [43](#), [43](#), [314](#), [315](#), [315](#)
- \bool_do_while:cn [314](#)
- \bool_do_while:Nn
..... [42](#), [42](#), [314](#), [314](#), [314](#), [314](#)
- \bool_do_while:nn [43](#), [43](#), [314](#), [314](#), [315](#)
- _bool_eval_skip_to_end_auxi:Nw
..... [312](#), [312](#), [312](#), [313](#), [313](#)
- _bool_eval_skip_to_end_auxii:Nw [312](#), [313](#), [313](#)
- _bool_eval_skip_to_end_auxiii:Nw [312](#), [313](#), [313](#)
- _bool_get_next:NN
.... [310](#), [310](#), [310](#), [311](#), [311](#), [312](#), [312](#)
- .bool_gset:c [173](#), [538](#)
- \bool_gset:cn [306](#)
- .bool_gset:N [173](#), [538](#)
- \bool_gset:Nn . . [40](#), [306](#), [306](#), [306](#), [307](#)
- \bool_gset_eq:cc [306](#), [306](#)
- \bool_gset_eq:cN [306](#), [306](#)
- \bool_gset_eq:Nc [306](#), [306](#)
- \bool_gset_eq:NN . . . [40](#), [306](#), [306](#), [307](#)
- \bool_gset_false:c [306](#)
- \bool_gset_false:N
..... [40](#), [306](#), [306](#), [306](#), [307](#), [524](#), [525](#)
- .bool_gset_inverse:c [173](#), [538](#)
- .bool_gset_inverse:N [173](#), [538](#)
- \bool_gset_true:c [306](#)
- \bool_gset_true:N
..... [40](#), [306](#), [306](#), [306](#), [307](#), [522](#)
- \bool_if:cTF [307](#)
- \bool_if:N [307](#)
- \bool_if:n [309](#)
- \bool_if:n(TF) [40](#)
- \bool_if:NF [240](#), [307](#), [314](#), [314](#), [499](#), [547](#)
- \bool_if:nF [315](#), [315](#)
- \bool_if:NT . . . [307](#), [314](#), [314](#), [488](#), [519](#)
- \bool_if:nT [314](#), [315](#), [784](#), [799](#)
- \bool_if:NTF [40](#), [40](#),
[285](#), [307](#), [307](#), [523](#), [525](#), [532](#), [544](#),
[545](#), [545](#), [545](#), [546](#), [546](#), [546](#), [547](#), [569](#)
- \bool_if:nTF [42](#), [42](#), [43](#), [43](#),
[43](#), [43](#), [308](#), [308](#), [544](#), [544](#), [796](#), [796](#),
[798](#), [800](#), [804](#), [805](#), [815](#), [818](#), [818](#), [819](#)
- \bool_if_exist:c [308](#)
- \bool_if_exist:cTF [308](#)
- \bool_if_exist:N [308](#)
- \bool_if_exist:NF [533](#), [533](#)
- \bool_if_exist:NTF . . [40](#), [40](#), [308](#), [308](#)
- \bool_if_exist_p:c [308](#)
- \bool_if_exist_p:N [40](#), [40](#), [308](#)
- _bool_if_left_parentheses:wwwn
..... [309](#), [310](#), [310](#), [310](#)
- _bool_if_or:wwwn . [309](#), [310](#), [310](#), [310](#)
- \bool_if_p:c [307](#)
- \bool_if_p:N [40](#), [40](#), [307](#), [307](#)
- \bool_if_p:n
..... [42](#), [42](#), [306](#), [306](#), [307](#), [307](#),
[308](#), [309](#), [309](#), [310](#), [313](#), [313](#), [314](#), [314](#)
- _bool_if_parse:NNnw
..... [310](#), [310](#), [310](#), [310](#)
- _bool_if_right_parentheses:wwwn
..... [309](#), [310](#), [310](#), [310](#)
- \bool_log:c [782](#)
- \bool_log:N . . . [220](#), [220](#), [782](#), [782](#), [782](#)
- \bool_log:n [220](#), [220](#), [782](#), [782](#)
- \bool_new:c [305](#)
- \bool_new:N [39](#), [39](#), [305](#), [305](#),
[305](#), [308](#), [308](#), [308](#), [308](#), [479](#), [522](#),
[530](#), [530](#), [530](#), [530](#), [530](#), [533](#), [533](#), [566](#)
- \bool_not_p:n [42](#), [42](#), [313](#), [313](#)
- _bool_p:Nw [311](#)
- _bool_S_0:w [312](#)
- _bool_S_1:w [312](#)
- .bool_set:c [173](#), [538](#)
- \bool_set:cn [306](#)
- .bool_set:N [173](#), [538](#)
- \bool_set:Nn [40](#), [40](#), [306](#), [306](#), [306](#), [307](#)
- \bool_set_eq:cc [306](#), [306](#)
- \bool_set_eq:cN [306](#), [306](#)
- \bool_set_eq:Nc [306](#), [306](#)
- \bool_set_eq:NN . [40](#), [40](#), [306](#), [306](#), [307](#)
- \bool_set_false:c [306](#)
- \bool_set_false:N [40](#), [40](#),
[240](#), [306](#), [306](#), [306](#), [306](#), [487](#), [499](#),
[531](#), [543](#), [543](#), [543](#), [543](#), [544](#), [545](#), [569](#)
- .bool_set_inverse:c [173](#), [538](#)
- .bool_set_inverse:N [173](#), [538](#)
- \bool_set_true:c [306](#)

- \bool_set_true:N . 40, 40, 241, 306,
306, 306, 306, 488, 489, 489, 531,
542, 543, 543, 543, 543, 545, 568, 570
- \bool_show:c 308
- \bool_show:N
..... 40, 40, 308, 308, 308, 782, 782
- \bool_show:n 40, 40, 308, 308, 782
- _bool_to_str:n ... 308, 308, 308, 308
- \bool_until_do:cn 314
- \bool_until_do:Nn
..... 43, 43, 314, 314, 314, 314
- \bool_until_do:nn 43, 43, 314, 315, 315
- \bool_while_do:cn 314
- \bool_while_do:Nn
..... 43, 43, 314, 314, 314, 314
- \bool_while_do:nn 43, 43, 314, 314, 314
- \bool_xor_p:nn 42, 42, 314, 314
- \botmark 242
- \botmarks 248
- \box 242
- box commands:
- \box_(g)clear:N 147
- \l__box_angle_fp ... 215, 233, 760,
760, 761, 761, 761, 826, 826, 826, 826
- \l__box_bottom_dim 760, 760,
761, 763, 763, 763, 763, 763, 764,
764, 764, 764, 765, 765, 766, 767, 767
- \l__box_bottom_new_dim
..... 760, 760, 762,
763, 763, 764, 764, 765, 767, 768, 768
- \box_clear:c 471
- \box_clear:N 147,
147, 471, 471, 471, 471, 480, 482, 483
- \box_clear_new:c 471
- \box_clear_new:N 147, 147, 471, 471, 471
- \box_clip:c 768
- \box_clip:N
..... 215, 215, 215, 215, 768, 768, 768
- \l__box_cos_fp
215, 233, 760, 760, 761, 762, 762,
763, 763, 825, 825, 825, 825, 825, 826
- \box_dp:c 472, 484
- \box_dp:N 148, 148, 472,
472, 472, 472, 484, 486, 486, 487,
487, 492, 492, 500, 761, 765, 767,
769, 769, 769, 773, 824, 824, 825, 825
- \box_gclear:c 471
- \box_gclear:N . 147, 471, 471, 471, 471
- \box_gclear_new:c 471
- \box_gclear_new:N .. 147, 471, 471, 471
- \box_gset_eq:cc 471
- \box_gset_eq:cN 471
- \box_gset_eq:Nc 471
- \box_gset_eq:NN 147, 471, 471, 471, 471
- \box_gset_eq_clear:cc 472
- \box_gset_eq_clear:cN 472
- \box_gset_eq_clear:Nc 472
- \box_gset_eq_clear:NN
..... 147, 147, 472, 472, 472
- \box_gset_to_last:c 473
- \box_gset_to_last:N 150, 473, 473, 473
- \box_ht:c 472, 484
- \box_ht:N 149, 149, 472, 472, 472, 472,
482, 482, 483, 483, 484, 486, 486,
487, 487, 492, 492, 500, 761, 765,
767, 769, 769, 769, 773, 773, 824, 825
- \box_if_empty:cTF 473
- \box_if_empty:N 473
- \box_if_empty:NF 473
- \box_if_empty:NT 473
- \box_if_empty:NTF .. 149, 149, 473, 473
- \box_if_empty_p:c 473
- \box_if_empty_p:N .. 149, 149, 473, 473
- \box_if_exist:c 472
- \box_if_exist:cTF 472
- \box_if_exist:N 472
- \box_if_exist:NTF
..... 148, 148, 471, 471, 472, 475
- \box_if_exist_p:c 472
- \box_if_exist_p:N 148, 148, 472
- \box_if_horizontal:cTF 473
- \box_if_horizontal:N 473
- \box_if_horizontal:NF 473
- \box_if_horizontal:NT 473
- \box_if_horizontal:NTF
..... 149, 149, 473, 473
- \box_if_horizontal_p:c 473
- \box_if_horizontal_p:N
..... 149, 149, 473, 473
- \box_if_vertical:cTF 473
- \box_if_vertical:N 473
- \box_if_vertical:NF 473
- \box_if_vertical:NT 473
- \box_if_vertical:NTF 149, 149, 473, 473
- \box_if_vertical_p:c 473
- \box_if_vertical_p:N 149, 149, 473, 473
- \l__box_internal_box .. 216, 233,
233, 760, 760, 762, 762, 762, 762,
762, 762, 762, 767, 768, 768, 768,
768, 768, 768, 768, 769, 769, 769,

- 769, 769, 769, 769, 769, 769, 769,
 769, 769, 769, 769, 769, 769, 770,
 770, 770, 770, 770, 770, 770, 770,
 770, 770, 770, 770, 771, 771, 825
 \l__box_left_dim
 760, 760, 761, 763, 763,
 763, 763, 763, 764, 764, 764, 765, 767
 \l__box_left_new_dim
 760, 760, 762, 762, 763, 763, 764, 764
 \box_log:c 474
 \box_log:cnn 474
 \box_log:N 150, 150, 474, 474, 474
 \box_log:Nnn 151, 151, 474, 474, 474, 474
 \box_move_down:nn 148,
 472, 473, 769, 769, 769, 770, 770, 772
 \box_move_left:nn 148, 472, 472
 \box_move_right:nn . 148, 148, 472, 473
 \box_move_up:nn 148, 148,
 472, 473, 493, 500, 769, 769, 770, 770
 \box_new:c 471
 \box_new:N 147,
 147, 147, 471, 471, 471, 471, 471,
 474, 474, 474, 474, 474, 478, 481, 760
 \box_resize:cnn 764
 __box_resize:N
 764, 764, 765, 766, 766, 766, 766
 __box_resize:NNN
 764, 765, 765, 765, 765
 \box_resize:Nnn
 213, 213, 216, 764, 764, 765, 776
 __box_resize_common:N
 765, 767, 767, 767
 __box_resize_set_corners:N
 764, 764, 765, 765, 766, 766, 766
 \box_resize_to_ht:cn 765
 \box_resize_to_ht:Nn
 213, 213, 765, 765, 766
 \box_resize_to_ht_plus_dp:cn .. 765
 \box_resize_to_ht_plus_dp:Nn ...
 213, 213, 765, 766, 766
 \box_resize_to_wd:cn 765
 \box_resize_to_wd:Nn
 214, 214, 765, 766, 766
 \box_resize_to_wd_and_ht:cnn .. 765
 \box_resize_to_wd_and_ht:Nnn ...
 214, 214, 765, 766, 767
 \l__box_right_dim .. 760, 760, 761,
 763, 763, 763, 763, 764, 764, 764,
 764, 764, 765, 765, 766, 766, 767, 767
 \l__box_right_new_dim
 760, 760, 762, 763,
 763, 764, 764, 765, 767, 768, 768, 768
 __box_rotate:N 761, 761, 761
 \box_rotate:Nn
 214, 214, 215, 215, 216, 761, 761, 772
 __box_rotate_quadrant_four: ...
 761, 762, 764
 __box_rotate_quadrant_one:
 761, 762, 763
 __box_rotate_quadrant_three: ...
 761, 762, 763
 __box_rotate_quadrant_two:
 761, 762, 763
 __box_rotate_x:nnN 761, 762,
 763, 763, 763, 763, 764, 764, 764, 764
 __box_rotate_y:nnN 761, 763,
 763, 763, 763, 763, 763, 764, 764, 764
 \box_scale:cnn 767
 \box_scale:Nnn
 214, 214, 216, 767, 767, 767, 777
 \l__box_scale_x_fp . 216, 233, 764,
 764, 764, 765, 765, 766, 766, 766,
 766, 767, 767, 768, 826, 826, 826, 826
 \l__box_scale_y_fp
 216, 233, 764, 764, 764, 765,
 765, 765, 765, 766, 766, 766, 766,
 767, 767, 767, 768, 826, 826, 826, 826
 \box_set_dp:cn 472
 \box_set_dp:Nn 149, 149,
 472, 472, 472, 492, 492, 500, 762,
 768, 768, 769, 769, 769, 770, 770, 773
 \box_set_eq:cc 471
 \box_set_eq:cN 471
 \box_set_eq:Nc 471
 \box_set_eq:NN 147, 147, 471, 471,
 471, 471, 471, 484, 492, 500, 770, 771
 \box_set_eq_clear:cc 472
 \box_set_eq_clear:cN 472
 \box_set_eq_clear:Nc 472
 \box_set_eq_clear:NN
 147, 147, 472, 472, 472, 472
 \box_set_ht:cn 472
 \box_set_ht:Nn 149,
 149, 472, 472, 472, 492, 492, 500,
 762, 768, 768, 769, 769, 770, 771, 773
 \box_set_to_last:c 473
 \box_set_to_last:N
 150, 150, 473, 473, 473, 473
 \box_set_wd:cn 472


```

\char_set_catcode_comment:N . . . .
    . . . . . 52, 325, 325
\char_set_catcode_comment:n . . . .
    . . . . . 53, 325, 326
\char_set_catcode_end_line:N . . .
    . . . . . 52, 325, 325
\char_set_catcode_end_line:n . . .
    . . . . . 53, 325, 326
\char_set_catcode_escape:N . . . .
    . . . . . 52, 325, 325
\char_set_catcode_escape:n . . . .
    . . . . . 53, 325, 325
\char_set_catcode_group_begin:N .
    . . . . . 52, 325, 325
\char_set_catcode_group_begin:n .
    . . . . . 53, 325, 325, 813
\char_set_catcode_group_end:N . . .
    . . . . . 52, 325, 325
\char_set_catcode_group_end:n . . .
    . . . . . 53, 325, 325, 814
\char_set_catcode_ignore:N . . . .
    . . . . . 52, 325, 325
\char_set_catcode_ignore:n . . . .
    . . . . . 53, 240, 241, 325, 326
\char_set_catcode_invalid:N . . . .
    . . . . . 52, 325, 325
\char_set_catcode_invalid:n . . . .
    . . . . . 53, 325, 326
\char_set_catcode_letter:N . . . .
    . . . . . 52, 52, 325, 325, 611,
    . . . . . 628, 628, 633, 634, 635, 635, 635,
    . . . . . 635, 636, 637, 637, 637, 638, 650, 650
\char_set_catcode_letter:n . . . .
    . . . . . 53, 53, 241, 241, 325, 326, 814
\char_set_catcode_math_subscript:N
    . . . . . 52, 325, 325, 328
\char_set_catcode_math_subscript:n
    . . . . . 53, 325, 326, 814
\char_set_catcode_math_superscript:N
    . . . . . 52, 325, 325, 521
\char_set_catcode_math_superscript:n
    . . . . . 53, 241, 325, 326, 814
\char_set_catcode_math_toggle:N .
    . . . . . 52, 325, 325, 328
\char_set_catcode_math_toggle:n .
    . . . . . 53, 325, 325, 814
\char_set_catcode_other:N . . . .
    . . . . . 52, 325, 325, 566, 635, 750
\char_set_catcode_other:n . . . .
    . . . . . 53, 241, 241, 325, 326, 813, 814

\char_set_catcode_parameter:N . . .
    . . . . . 52, 325, 325
\char_set_catcode_parameter:n . . .
    . . . . . 53, 325, 326, 814
\char_set_catcode_space:N 52, 325, 325
\char_set_catcode_space:n . . . .
    . . . . . 53, 241, 325, 326, 814
\char_set_lccode:nn . . . . 54, 98,
    . . . . . 323, 324, 326, 326, 333, 333, 333,
    . . . . . 333, 333, 391, 392, 507, 507, 507,
    . . . . . 521, 521, 521, 521, 566, 811, 814, 814
\char_set_lcode:nn . . . . . 54
\char_set_mathcode:nn 54, 54, 326, 326
\char_set_sfcode:nn . 55, 55, 326, 327
\char_set_uccode:nn 54, 54, 98, 326, 326
\char_show_value_catcode:n . . . .
    . . . . . 53, 53, 322, 324
\char_show_value_lccode:n . . . .
    . . . . . 54, 54, 326, 326
\char_show_value_mathcode:n . . . .
    . . . . . 55, 55, 326, 326
\char_show_value_sfcode:n . . . .
    . . . . . 55, 55, 326, 327
\char_show_value_uccode:n . . . .
    . . . . . 54, 54, 326, 327
\l_char_special_seq . . . . . 52,
    . . . . . 53, 53, 55, 322, 324, 327, 327, 327
\__char_tmp:n . 814, 814, 814, 814, 814
\__char_tmp:nN . . . . . 811, 811, 811
\l__char_tmp_tl . . . . .
    . . . . . 812, 813, 813, 814, 814,
    . . . . . 814, 814, 814, 814, 814, 814, 814,
    . . . . . 814, 814, 814, 814, 814, 814, 814
\char_value_catcode:n . . . . 53,
    . . . . . 53, 240, 240, 240, 240, 240, 240,
    . . . . . 240, 240, 240, 322, 324, 325, 392, 392
\char_value_lccode:n . . . . .
    . . . . . 54, 54, 326, 326, 326
\char_value_mathcode:n . . . . .
    . . . . . 55, 55, 326, 326, 326
\char_value_sfcode:n . . . . .
    . . . . . 55, 55, 326, 327, 327
\char_value_uccode:n . . . . .
    . . . . . 54, 54, 326, 327, 327
\chardef . . . . . 236, 236, 240, 240, 242
chk commands:
  \__chk_if_exist_cs:c . . . . .
    . . . . . 272, 272, 272, 272, 279, 280
  \__chk_if_exist_cs:N . . . . .
    . . . . . 24, 24, 279, 279, 280, 298

```

- _chk_if_exist_var:N 24,
24, 279, 279, 306, 306, 307, 307,
307, 307, 307, 307, 388, 388, 388,
388, 388, 389, 389, 389, 389, 389, 389
- _chk_if_free_cs:c 278, 279
- _chk_if_free_cs:N 24, 24,
278, 279, 279, 279, 280, 282, 328,
328, 328, 328, 348, 348, 369, 377,
381, 384, 384, 384, 428, 462, 471, 503
- _chk_if_free_msg:nn
..... 503, 503, 503, 504
- _chk_log:x
24, 24, 24, 24, 277, 277, 277, 278,
278, 278, 278, 279, 304, 504, 533, 533
- _chk_resume_log:
..... 24, 24, 24, 277, 277,
277, 277, 277, 278, 278, 278, 278, 481
- _chk_suspend_log: 24,
24, 24, 277, 277, 277, 277, 277, 278, 481
- choice commands:
.choice: 173, 538
- choices commands:
.choices:nn 173, 539
.choices:on 173, 539
.choices:Vn 173, 539
.choices:xn 173, 539
- circumflex commands:
\c_circumflex_str 117, 426, 427
- \cite 805
- \cleaders 242
- \clearmarks 253
- clist commands:
\clist_(g)clear:N 132
\clist_clear:c 445, 445
\clist_clear:N
131, 131, 445, 445, 445, 451, 542, 543
\clist_clear_new:c 445, 445
\clist_clear_new:N . 132, 132, 445, 445
\clist_concat:ccc 446
\clist_concat:NNN
.... 132, 132, 446, 446, 446, 448, 448
_clist_concat:NNNN 446, 446, 446, 446
\clist_const:cn 445
\clist_const:cx 445
\clist_const:Nn 131, 131, 445, 445, 445
\clist_const:Nx 445
\clist_count:c 457
\clist_count:N
136, 136, 139, 457, 457, 457, 458, 459
_clist_count:n 457, 457, 457
- \clist_count:n 136, 457, 457, 460
_clist_count:w ... 457, 457, 457, 457
\clist_gclear:c 445, 445
\clist_gclear:N ... 131, 445, 445, 446
\clist_gclear_new:c 445, 445
\clist_gclear_new:N ... 132, 445, 445
\clist_gconcat:ccc 446
\clist_gconcat:NNN
..... 132, 446, 446, 446, 448, 448
\clist_get:cN 449
\clist_get:cNTF 450
\clist_get:NN
..... 138, 138, 449, 449, 449, 450
\clist_get:NnF 450
\clist_get:NNT 450
\clist_get:NNTF ... 138, 138, 450, 450
_clist_get:wN ... 449, 449, 449, 450
\clist_gpop:cN 449
\clist_gpop:cNTF 450
\clist_gpop:NN
..... 138, 138, 449, 449, 449, 450
\clist_gpop:NnF 450
\clist_gpop:NNT 450
\clist_gpop:NNTF ... 138, 138, 450, 450
_clist_get:wN ... 449, 449, 449, 450
\clist_gpush:cn 449
\clist_gpush:co 450, 451
\clist_gpush:cV 450, 451
\clist_gpush:cx 450, 451
\clist_gpush:Nn 139, 450, 450
\clist_gpush:No 450, 450
\clist_gpush:Nv 450, 450
\clist_gpush:Nx 450, 451
\clist_gput_left:cn 448, 451
\clist_gput_left:co 448, 451
\clist_gput_left:cV 448, 451
\clist_gput_left:cx 448, 451
\clist_gput_left:Nn
..... 133, 448, 448, 448, 448, 450
\clist_gput_left:No 448, 450
\clist_gput_left:Nv 448, 450
\clist_gput_left:Nx 448, 451
\clist_gput_right:cn 448
\clist_gput_right:co 448
\clist_gput_right:cV 448
\clist_gput_right:cx 448
\clist_gput_right:Nn
..... 133, 448, 448, 448, 448
\clist_gput_right:No 448
\clist_gput_right:Nv 448
\clist_gput_right:Nx 448

\clist_gremove_all:cn [451](#)
\clist_gremove_all:Nn
..... [133](#), [451](#), [452](#), [452](#)
\clist_gremove_duplicates:c ... [451](#)
\clist_gremove_duplicates:N
..... [133](#), [451](#), [451](#), [451](#)
\clist_greverse:c [452](#)
\clist_greverse:N .. [134](#), [452](#), [452](#), [452](#)
\clist_gset:c [174](#), [539](#)
\clist_gset:cn [448](#)
\clist_gset:co [448](#)
\clist_gset:cV [448](#)
\clist_gset:cx [448](#)
\clist_gset:N [174](#), [539](#)
\clist_gset:Nn [132](#), [445](#), [448](#), [448](#), [448](#)
\clist_gset:No [448](#)
\clist_gset:Nv [448](#)
\clist_gset:Nx [448](#)
\clist_gset_eq:cc [445](#), [445](#)
\clist_gset_eq:cN [445](#), [445](#)
\clist_gset_eq:Nc [445](#), [445](#)
\clist_gset_eq:NN .. [132](#), [445](#), [445](#), [451](#)
\clist_gset_from_seq:cc [445](#)
\clist_gset_from_seq:cN [445](#)
\clist_gset_from_seq:Nc [445](#)
\clist_gset_from_seq:NN
..... [132](#), [445](#), [446](#), [446](#), [446](#)
\clist_if_empty:c [453](#)
\clist_if_empty:cTF [453](#)
\clist_if_empty:N [453](#)
\clist_if_empty:n [453](#)
\clist_if_empty:Nf
..... [446](#), [446](#), [452](#), [455](#), [456](#), [456](#)
\clist_if_empty:nf [460](#)
\clist_if_empty:Nf
..... [134](#), [134](#), [453](#), [460](#), [536](#)
\clist_if_empty:nf [134](#), [134](#), [453](#)
__clist_if_empty_n:w
..... [453](#), [453](#), [454](#), [454](#)
__clist_if_empty_n:wNw [453](#), [454](#), [454](#)
\clist_if_empty_p:c [453](#)
\clist_if_empty_p:N ... [134](#), [134](#), [453](#)
\clist_if_empty_p:n ... [134](#), [134](#), [453](#)
\clist_if_exist:c [447](#)
\clist_if_exist:cTF [447](#)
\clist_if_exist:N [447](#)
\clist_if_exist:Nf [556](#)
\clist_if_exist:Nf
..... [132](#), [132](#), [447](#), [458](#), [460](#), [555](#)
\clist_if_exist_p:c [447](#)
\clist_if_exist_p:N ... [132](#), [132](#), [447](#)
\clist_if_in:cnTF [454](#)
\clist_if_in:coTF [454](#)
\clist_if_in:cVTF [454](#)
\clist_if_in:Nn [454](#)
\clist_if_in:nn [454](#)
\clist_if_in:NnF [451](#), [454](#), [454](#)
\clist_if_in:nnf [454](#)
\clist_if_in:NnT [454](#), [454](#)
\clist_if_in:nnT [454](#)
\clist_if_in:NnTF
..... [134](#), [134](#), [454](#), [454](#), [454](#)
\clist_if_in:nnTF .. [134](#), [454](#), [454](#), [588](#)
\clist_if_in:NoTF [454](#)
\clist_if_in:NoTF [454](#)
\clist_if_in:NvTF [454](#)
\clist_if_in:nvTF [454](#)
__clist_if_in_return:nn
..... [454](#), [454](#), [454](#), [454](#)
\l__clist_internal_clist
..... [444](#), [444](#), [448](#), [448](#),
[448](#), [448](#), [454](#), [454](#), [456](#), [456](#), [456](#), [456](#)
\l__clist_internal_remove_clist .
..... [451](#), [451](#), [451](#), [451](#), [451](#), [451](#)
\clist_item:cn [459](#)
\clist_item:Nn
..... [139](#), [139](#), [459](#), [459](#), [459](#), [459](#)
\clist_item:nn [139](#), [459](#), [460](#)
__clist_item:nnNn .. [459](#), [459](#), [459](#), [460](#)
__clist_item_n:nw [459](#), [460](#), [460](#)
__clist_item_n_end:n . [459](#), [460](#), [460](#)
__clist_item_N_loop:nw
..... [459](#), [459](#), [459](#), [459](#)
__clist_item_n_loop:nw
..... [459](#), [460](#), [460](#), [460](#), [460](#)
__clist_item_n_strip:w [459](#), [460](#), [460](#)
\clist_log:c [771](#)
\clist_log:N .. [216](#), [216](#), [771](#), [771](#), [771](#)
\clist_log:n [216](#), [216](#), [771](#), [771](#)
\clist_map_break: [136](#), [136](#), [455](#), [455](#),
[455](#), [455](#), [456](#), [456](#), [457](#), [457](#), [457](#), [457](#)
\clist_map_break:n
..... [136](#), [136](#), [457](#), [457](#), [545](#)
\clist_map_function:cN [454](#)
\clist_map_function:NN
..... [46](#), [131](#), [135](#), [135](#),
[135](#), [429](#), [429](#), [454](#), [455](#), [455](#), [457](#), [460](#)
\clist_map_function:Nn [456](#)
\clist_map_function:nN
..... [135](#), [429](#), [429](#), [455](#), [455](#), [457](#), [461](#), [547](#)

```

\__clist_map_function:Nw .....
..... 454, 455, 455, 455, 455, 456
\__clist_map_function_n:Nn .....
..... 455, 455, 455, 455, 455
\clist_map_inline:cn ..... 455
\clist_map_inline:Nn .....
..... 135, 135, 135, 451,
455, 455, 456, 456, 456, 545, 556, 556
\clist_map_inline:nn .....
..... 135, 455, 456, 535, 817, 820, 820
\__clist_map_unbrace:Nw .....
..... 455, 455, 455, 455
\clist_map_variable:cNn ..... 456
\clist_map_variable:NNn .....
..... 135, 135, 456, 456, 456, 457
\clist_map_variable:nNn 135, 456, 456
\__clist_map_variable:Nnw .....
..... 456, 456, 456, 457
\clist_new:c ..... 445, 445
\clist_new:N .. 131, 131, 132, 444,
445, 445, 451, 461, 461, 461, 461, 529
\clist_pop:cN ..... 449
\clist_pop:cNTF ..... 450
\clist_pop:NN .....
..... 138, 138, 449, 449, 449, 450
\clist_pop:NNF ..... 450
\__clist_pop:NNN ... 449, 449, 449, 449
\clist_pop:NNT ..... 450
\clist_pop:NNTF ... 138, 138, 450, 450
\__clist_pop:wN ..... 449, 449, 449
\__clist_pop:wwNNN .....
..... 449, 449, 449, 449, 450
\__clist_pop_TF:NNN 450, 450, 450, 450
\clist_push:cn ..... 450, 450
\clist_push:co ..... 450, 450
\clist_push:cV ..... 450, 450
\clist_push:cx ..... 450, 450
\clist_push:Nn .... 139, 139, 450, 450
\clist_push:No ..... 450, 450
\clist_push:NV ..... 450, 450
\clist_push:Nx ..... 450, 450
\clist_put_left:cn ..... 448, 450
\clist_put_left:co ..... 448, 450
\clist_put_left:cV ..... 448, 450
\clist_put_left:cx ..... 448, 450
\clist_put_left:Nn .....
..... 133, 133, 448, 448, 448, 448, 450
\__clist_put_left:NNNn .....
..... 448, 448, 448, 448
\clist_put_left:No ..... 448, 450
\clist_put_left:NV ..... 448, 450
\clist_put_right:cn ..... 448
\clist_put_right:co ..... 448
\clist_put_right:cV ..... 448
\clist_put_right:cx ..... 448
\clist_put_right:Nn .....
..... 133, 133, 448, 448, 448, 448, 451
\__clist_put_right:NNNn .....
..... 448, 448, 448, 448
\clist_put_right:No ..... 448
\clist_put_right:NV ..... 448
\clist_put_right:Nx ..... 448, 547
\__clist_remove_all: 451, 452, 452, 452
\clist_remove_all:cn ..... 451
\clist_remove_all:Nn .....
..... 133, 133, 451, 452, 452
\__clist_remove_all:NNn .....
..... 451, 452, 452, 452
\__clist_remove_all:w .....
..... 451, 451, 451, 452, 452
\clist_remove_duplicates:c .... 451
\clist_remove_duplicates:N .....
..... 133, 133, 451, 451, 451
\__clist_remove_duplicates:NN ...
..... 451, 451, 451, 451
\clist_reverse:c ..... 452
\clist_reverse:N 134, 134, 452, 452, 452
\clist_reverse:n .....
..... 134, 134, 452, 452, 452, 452, 453, 453
\__clist_reverse:wwNnw .....
..... 453, 453, 453, 453, 453, 453, 453, 453
\__clist_reverse_end:ww .....
..... 453, 453, 453, 453
.clist_set:c ..... 174, 539
\clist_set:cn ..... 448
\clist_set:co ..... 448
\clist_set:cV ..... 448
\clist_set:cx ..... 448
.clist_set:N ..... 174, 539
\clist_set:Nn .....
..... 132, 132, 445, 448, 448, 448,
448, 448, 448, 448, 454, 456, 456, 536
\clist_set:No ..... 448
\clist_set:NV ..... 448
\clist_set:Nx ..... 448
\clist_set_eq:cc ..... 445, 445
\clist_set_eq:cN ..... 445, 445
\clist_set_eq:Nc ..... 445, 445
\clist_set_eq:NN 132, 132, 445, 445, 451

```

```

\clist_set_from_seq:cc ..... 445
\clist_set_from_seq:cN ..... 445
\clist_set_from_seq:Nc ..... 445
\clist_set_from_seq:NN .....
    ..... 132, 132, 445, 445, 446, 446
\__clist_set_from_seq:NNNN .....
    ..... 445, 445, 446, 446
\__clist_set_from_seq:w 445, 446, 446
\clist_show:c ..... 460
\clist_show:N .....
    ..... 139, 139, 216, 460, 460, 461, 771, 771
\clist_show:n .....
    ..... 139, 139, 216, 460, 460, 771
\__clist_tmp:w ..... 444,
    ..... 444, 451, 451, 451, 452, 452, 454, 454
\__clist_trim_spaces:n .....
    ..... 445, 447, 447, 448, 448
\__clist_trim_spaces:nn .....
    ..... 447, 447, 447, 447, 447, 447
\__clist_trim_spaces_generic:nn .
    ..... 447, 447, 447, 447
\__clist_trim_spaces_generic:nw .
    ..... 447, 447, 447, 447, 447, 455, 455, 455
\clist_use:cn ..... 458
\clist_use:cnnn ..... 458
\clist_use:Nn . 137, 137, 458, 458, 459
\clist_use:Nnnn .....
    .... 137, 137, 442, 458, 458, 458, 459
\__clist_use:nwn ..... 458, 458, 458
\__clist_use:nwwwnn .....
    ..... 458, 458, 458, 458, 458
\__clist_use:wn ..... 458, 458, 458, 458
\__clist_wrap_item:n .. 445, 446, 446
\closein ..... 242
\closeout ..... 242
\clubpenalties ..... 248
\clubpenalty ..... 242
cm ..... 208
code commands:
    .code:n ..... 174, 539
coffin commands:
    \__coffin_align:NnnNnnnnN .....
        ..... 491, 492, 492, 492, 493, 495
    \l__coffin_aligned_coffin .....
        ..... 484, 484, 491, 491,
        ..... 491, 491, 491, 491, 492, 492, 492,
        ..... 492, 492, 492, 492, 492, 492,
        ..... 492, 492, 492, 492, 492, 493,
        ..... 494, 495, 495, 500, 500, 500, 500, 500
    \l__coffin_aligned_internal_-
        coffin ..... 484, 484, 493, 493
    \coffin_attach:cnncnnnn ..... 492
    \coffin_attach:cnNnnnnn ..... 492
    \coffin_attach:Nnncnnnn ..... 492
    \coffin_attach:NnnNnnnn .....
        ..... 157, 157, 492, 492, 492, 500
    \coffin_attach_mark:NnnNnnnn ...
        ..... 492, 492, 497, 497, 498
    \l__coffin_bottom_corner_dim 771,
        ..... 771, 772, 773, 775, 775, 775, 776, 776
    \l__coffin_bounding_prop ... 771,
        ..... 771, 772, 773, 773, 773, 773, 775
    \l__coffin_bounding_shift_dim ...
        ..... 771, 771, 772, 775, 775, 775
    \__coffin_calculate_intersection:Nnn
        ..... 487, 487, 493, 493, 500
    \__coffin_calculate_intersection:nnnnnnnn
        ..... 487, 488, 488, 499
    \__coffin_calculate_intersection_-
        aux:nnnnnn .....
        .... 487, 488, 489, 489, 489, 490, 490
    \coffin_clear:c ..... 480
    \coffin_clear:N 155, 155, 480, 480, 480
    \c__coffin_corners_prop .....
        ..... 478, 478, 478, 478, 478, 478, 481, 485
    \l__coffin_cos_fp .....
        .... 771, 771, 772, 772, 774, 774, 774
    \__coffin_display_attach:Nnnnn ..
        ..... 498, 499, 499, 500, 500
    \l__coffin_display_coffin 495, 495,
        ..... 499, 499, 500, 500, 500, 500, 500, 500
    \l__coffin_display_coord_coffin .
        ..... 495, 495, 497, 498, 498, 499, 499, 500
    \l__coffin_display_font_tl .....
        ..... 497, 497, 497, 497, 497, 499
    \coffin_display_handles:cn .... 498
    \coffin_display_handles:Nn .....
        ..... 158, 158, 498, 498, 500
    \__coffin_display_handles_-
        aux:nnnn ..... 498, 500, 500, 500
    \__coffin_display_handles_-
        aux:nnnnnn ..... 498, 499, 499
    \l__coffin_display_handles_prop .
        ..... 495, 495,
        ..... 495, 495, 495, 495, 495, 496,
        ..... 496, 496, 496, 496, 496, 496, 496,
        ..... 496, 496, 496, 497, 497, 499, 499
    \l__coffin_display_offset_dim ...
        .... 496, 496, 496, 498, 498, 500, 500

```

\l__coffin_display_pole_coffin ..
 [495](#), [495](#), [497](#), [497](#), [498](#), [499](#)
 \l__coffin_display_poles_prop ...
 [496](#), [496](#), [498](#), [498](#), [499](#), [499](#), [499](#)
 \l__coffin_display_x_dim
 [496](#), [496](#), [499](#), [500](#)
 \l__coffin_display_y_dim
 [496](#), [496](#), [499](#), [500](#)
 \coffin_dp:c [484](#), [484](#)
 \coffin_dp:N
 [157](#), [157](#), [484](#), [484](#), [501](#), [776](#), [777](#)
 \l__coffin_error_bool [479](#),
 [479](#), [487](#), [488](#), [488](#), [489](#), [489](#), [499](#), [499](#)
 __coffin_find_bounding_shift: ..
 [772](#), [775](#), [775](#)
 __coffin_find_bounding_shift_-
 aux:nn [775](#), [775](#), [775](#)
 __coffin_find_corner_maxima:N ..
 [772](#), [775](#), [775](#)
 __coffin_find_corner_maxima_-
 aux:nn [775](#), [775](#), [775](#)
 __coffin_get_pole:NnN
 [485](#), [485](#), [487](#),
 [487](#), [494](#), [494](#), [494](#), [494](#), [498](#), [498](#), [498](#)
 __coffin_gset_eq_structure:NN ..
 [485](#), [485](#)
 \coffin_ht:c [484](#), [484](#)
 \coffin_ht:N
 [158](#), [158](#), [484](#), [484](#), [501](#), [776](#), [777](#)
 \coffin_if_exist:cTF [480](#)
 \coffin_if_exist:N [480](#)
 \coffin_if_exist:Nf [480](#)
 __coffin_if_exist:NT [480](#), [480](#), [480](#),
 [481](#), [482](#), [482](#), [483](#), [484](#), [486](#), [486](#), [501](#)
 \coffin_if_exist:NT [480](#)
 \coffin_if_exist:NTF
 [155](#), [155](#), [480](#), [480](#), [480](#)
 \coffin_if_exist_p:c [480](#)
 \coffin_if_exist_p:N [155](#), [155](#), [480](#), [480](#)
 \l__coffin_internal_box
 [478](#), [478](#), [482](#), [482](#), [482](#),
 [483](#), [483](#), [483](#), [772](#), [773](#), [773](#), [773](#), [773](#)
 \l__coffin_internal_dim . [478](#), [478](#),
 [491](#), [491](#), [491](#), [773](#), [773](#), [773](#), [777](#), [777](#)
 \l__coffin_internal_tl
 [478](#), [478](#), [479](#), [479](#), [479](#), [479](#),
 [479](#), [479](#), [479](#), [479](#), [479](#), [479](#), [479](#),
 [493](#), [493](#), [493](#), [497](#), [497](#), [497](#), [497](#),
 [498](#), [498](#), [499](#), [499](#), [499](#), [500](#), [500](#)
 \coffin_join:cnncnnnn [491](#)
 \coffin_join:cnncnnnn [491](#)
 \coffin_join:NnnNnnnn
 [157](#), [157](#), [491](#), [491](#), [492](#)
 \l__coffin_left_corner_dim . [771](#),
 [771](#), [772](#), [773](#), [775](#), [775](#), [775](#), [776](#), [776](#)
 \coffin_log_structure:c [779](#)
 \coffin_log_structure:N
 [216](#), [216](#), [779](#), [779](#), [779](#)
 \coffin_mark_handle:cnnc [497](#)
 \coffin_mark_handle:Nnnn
 [158](#), [158](#), [497](#), [497](#), [498](#)
 __coffin_mark_handle_aux:nnnnNnn
 [497](#), [498](#), [498](#), [498](#)
 \coffin_new:c [481](#)
 \coffin_new:N
 [155](#), [155](#), [481](#), [481](#), [481](#), [484](#),
 [484](#), [484](#), [484](#), [484](#), [484](#), [495](#), [495](#), [495](#)
 __coffin_offset_corner:Nnnnn
 [494](#), [494](#), [494](#)
 __coffin_offset_corners:Nnn
 [491](#), [491](#), [491](#), [491](#), [494](#), [494](#)
 __coffin_offset_pole:Nnnnnnn
 [493](#), [493](#), [493](#)
 __coffin_offset_poles:Nnn
 [491](#), [491](#), [491](#), [491](#), [492](#), [492](#), [493](#), [493](#)
 \l__coffin_offset_x_dim
 . [479](#), [479](#), [491](#), [491](#), [491](#), [491](#), [491](#),
 [491](#), [491](#), [491](#), [492](#), [493](#), [493](#), [500](#), [500](#)
 \l__coffin_offset_y_dim
 [479](#), [479](#), [491](#),
 [491](#), [491](#), [491](#), [492](#), [493](#), [493](#), [500](#), [500](#)
 \l__coffin_pole_a_tl [479](#), [479](#), [487](#),
 [488](#), [494](#), [494](#), [494](#), [494](#), [498](#), [498](#), [498](#)
 \l__coffin_pole_b_tl
 [479](#), [479](#), [487](#), [488](#),
 [494](#), [494](#), [494](#), [494](#), [498](#), [498](#), [498](#), [498](#)
 \c__coffin_poles_prop
 [479](#), [479](#), [479](#), [479](#), [479](#),
 [479](#), [479](#), [479](#), [479](#), [479](#), [481](#), [485](#)
 __coffin_reset_structure:N [480](#),
 [481](#), [482](#), [482](#), [483](#), [485](#), [485](#), [491](#), [492](#)
 \coffin_resize:cnnc [776](#)
 \coffin_resize:Nnn
 [216](#), [216](#), [776](#), [776](#), [776](#)
 __coffin_resize_common:Nnn
 [776](#), [777](#), [777](#), [777](#)
 \l__coffin_right_corner_dim
 [771](#), [771](#), [773](#), [775](#), [775](#), [775](#)
 \coffin_rotate:cn [772](#)

`\coffin_rotate:Nn` 216, 216, 772, 772, 773
`__coffin_rotate_bounding:nnn` ... 772, 773, 773
`__coffin_rotate_corner:Nnnn` ... 772, 773, 773
`__coffin_rotate_pole:Nnnnnn` ... 772, 774, 774
`__coffin_rotate_vector:nnNN` ... 773, 773, 774, 774, 774, 774
`\coffin_scale:cnn` 777
`\coffin_scale:Nnn` 216, 216, 777, 777, 777
`__coffin_scale_corner:Nnnn` 777, 778, 778
`__coffin_scale_pole:Nnnnnn` 777, 778, 778
`__coffin_scale_vector:nnNN` 778, 778, 778, 778
`\l__coffin_scale_x_fp` 776, 776, 776, 777, 777, 777, 777, 778
`\l__coffin_scale_y_fp` 776, 776, 776, 777, 777, 777, 778
`\l__coffin_scaled_total_height_dim` 776, 776, 777, 777
`\l__coffin_scaled_width_dim` 776, 776, 777, 777
`__coffin_set_bounding:N` 772, 773, 773
`\coffin_set_eq:cc` 484
`\coffin_set_eq:cN` 484
`\coffin_set_eq:Nc` 484
`\coffin_set_eq:NN` 155, 155, 484, 484, 492, 492, 493, 499
`__coffin_set_eq_structure:NN` ... 484, 485, 485
`\coffin_set_horizontal_pole:cnn` 485
`\coffin_set_horizontal_pole:Nnn` . 156, 156, 485, 486, 486
`__coffin_set_pole:Nnn` . 485, 486, 486
`__coffin_set_pole:Nnx` 482, 483, 485, 486, 486, 493, 494, 494, 495, 495, 774, 778
`\coffin_set_vertical_pole:cnn` .. 485
`\coffin_set_vertical_pole:Nnn` ... 156, 156, 485, 486, 486
`__coffin_shift_corner:Nnnn` 773, 775, 775
`__coffin_shift_pole:Nnnnnn` 773, 775, 776
`\coffin_show_structure:c` 501
`\coffin_show_structure:N` 158, 158, 216, 501, 501, 779, 779
`\l__coffin_sin_fp` 771, 771, 772, 772, 774, 774, 774
`\l__coffin_slope_x_fp` 479, 479, 489, 489, 490, 490
`\l__coffin_slope_y_fp` 479, 479, 489, 489, 490, 490
`\l__coffin_top_corner_dim` 771, 771, 773, 775, 775, 775
`\coffin_typeset:cnnnn` 495
`\coffin_typeset:Nnnnn` 157, 157, 495, 495, 495
`__coffin_update_B:nnnnnnnnN` ... 494, 494, 494
`__coffin_update_corners:N` 481, 482, 483, 483, 486, 486
`__coffin_update_poles:N` 481, 482, 483, 486, 487, 491, 492
`__coffin_update_T:nnnnnnnnN` ... 494, 494, 494
`__coffin_update_vertical_poles:NNN` 492, 492, 494, 494
`\coffin_wd:c` 484, 484
`\coffin_wd:N` 158, 158, 484, 484, 501, 776, 777
`\l__coffin_x_dim` 479, 479, 488, 488, 489, 489, 489, 489, 490, 490, 493, 493, 493, 499, 500, 773, 773, 773, 774, 774, 774, 778, 778, 778, 778
`\l__coffin_x_prime_dim` 479, 479, 493, 493, 500, 500, 774, 774
`__coffin_x_shift_corner:Nnnn` ... 777, 778, 778
`__coffin_x_shift_pole:Nnnnnn` ... 777, 778, 778
`\l__coffin_y_dim` ... 479, 479, 488, 488, 488, 489, 489, 489, 490, 493, 493, 493, 499, 500, 773, 773, 773, 774, 774, 774, 778, 778, 778, 778
`\l__coffin_y_prime_dim` 479, 479, 493, 493, 500, 500, 774, 774
colon commands:
`\c_colon_str` 117, 332, 337, 337, 426, 427
`\color` 497, 497, 498, 499
color commands:
`\color_ensure_current:` 159, 159, 481, 481, 482, 483, 502, 502, 502, 502, 502

<code>\color_group_begin:</code>	159,	<code>__cs_generate_from_signature:nnNNn</code>	285, 285
159, 159, 481, 482, 482, 483, 502, 502		<code>__cs_generate_internal_variant:n</code>	304, 304, 304
<code>\color_group_end:</code>	159,	<code>__cs_generate_internal_variant:wwnNwnn</code>	304, 304
159, 159, 481, 482, 482, 483, 502, 502		<code>__cs_generate_internal_variant:wwnw</code>	304
<code>\columnwidth</code>	482, 483	<code>__cs_generate_internal_variant_-</code>	
<code>\copy</code>	242	loop:n	304, 304, 305, 305
<code>cos</code>	206	<code>__cs_generate_variant:N</code>	298, 299, 299
<code>cosd</code>	206	<code>\cs_generate_variant:Nn</code>	12, 26, 27,
<code>cot</code>	206	27, 27, 28, 298, 298, 300, 300, 300,	
<code>cotd</code>	206	300, 305, 306, 306, 306, 306, 306,	
<code>\count</code>	242	306, 307, 307, 307, 307, 308, 314,	
<code>\countdef</code>	242	314, 314, 314, 320, 320, 320, 320,	
<code>\cr</code>	242	321, 321, 321, 321, 321, 321, 348,	
<code>\crampeddisplaystyle</code>	253	348, 349, 349, 349, 349, 349, 349,	
<code>\crampedscriptscriptstyle</code>	254	349, 349, 350, 350, 350, 350, 350,	
<code>\crampedscriptstyle</code>	254	350, 350, 350, 350, 350, 351, 367,	
<code>\crampedtextstyle</code>	254	369, 369, 370, 370, 370, 370, 370,	
<code>\crrcr</code>	242	370, 370, 370, 370, 370, 371, 371,	
cs commands:		371, 371, 375, 376, 377, 377, 378,	
<code>\cs:w</code>	18, 18, 18, 19, 262,	378, 378, 378, 378, 378, 378, 378,	
262, 263, 263, 263, 265, 275, 276,		378, 378, 379, 379, 379, 379, 380,	
284, 286, 289, 291, 292, 292, 292,		380, 380, 380, 381, 381, 381, 381,	
292, 292, 292, 293, 293, 294, 294,		381, 381, 382, 382, 382, 382, 382,	
294, 294, 294, 296, 296, 297, 305,		382, 382, 382, 382, 382, 383, 383,	
316, 316, 348, 351, 369, 375, 377,		384, 384, 384, 385, 385, 385, 385,	
380, 381, 471, 578, 581, 610, 613,		385, 385, 386, 386, 386, 386, 386,	
619, 620, 630, 632, 633, 705, 714, 730		386, 387, 387, 387, 387, 387, 387,	
<code>__cs_count_signature:c</code>	283, 283	387, 387, 387, 387, 387, 387, 387,	
<code>__cs_count_signature:N</code>	25, 25, 283, 283, 283	387, 387, 387, 390, 390, 390, 390,	
<code>__cs_count_signature:nnN</code>	283, 283, 283	393, 393, 393, 393, 396, 396, 396,	
<code>\cs_end:</code>	18, 18, 18, 262, 262, 263,	396, 396, 396, 396, 396, 396, 396,	
263, 263, 263, 265, 275, 275, 276,		396, 396, 397, 397, 397, 397, 397,	
276, 282, 284, 286, 289, 291, 292,		397, 397, 397, 398, 398, 398, 398,	
292, 292, 292, 292, 292, 293, 293,		399, 399, 399, 399, 399, 399, 401,	
294, 294, 294, 294, 294, 296, 296,		401, 401, 401, 401, 402, 402, 402,	
297, 305, 305, 315, 316, 316, 316,		403, 403, 403, 404, 404, 407, 407,	
316, 316, 316, 316, 316, 316, 316,		407, 408, 408, 409, 409, 409, 409,	
316, 348, 351, 369, 375, 377, 380,		413, 413, 414, 414, 414, 414, 416,	
381, 471, 578, 581, 587, 610, 613,		416, 416, 416, 416, 416, 416, 416,	
619, 620, 630, 632, 633, 705, 714, 733		417, 417, 417, 417, 417, 417, 417,	
<code>\cs_generate_from_arg_count:cNnn</code>	283, 284	417, 419, 421, 423, 423, 424, 425,	
<code>\cs_generate_from_arg_count:Ncnn</code>	283, 284	426, 426, 426, 427, 428, 428, 428,	
<code>\cs_generate_from_arg_count:NNnn</code>	16, 16, 283, 284, 284, 284, 285	429, 429, 429, 429, 429, 429, 429,	
<code>__cs_generate_from_signature:NNn</code>	284, 285	429, 431, 431, 431, 431, 431, 431,	
		431, 431, 432, 432, 432, 432, 432,	
		432, 433, 433, 434, 434, 435, 435,	

435, 435, 435, 435, 435, 435, 435,
 435, 436, 436, 436, 437, 438, 438,
 438, 438, 438, 438, 438, 438, 438,
 438, 438, 438, 438, 439, 439, 439,
 439, 439, 439, 439, 439, 440, 441,
 441, 441, 441, 442, 442, 444, 445,
 446, 446, 446, 446, 446, 446, 448,
 448, 448, 448, 448, 448, 448, 448,
 448, 448, 449, 449, 449, 450, 450,
 450, 450, 450, 450, 450, 450, 450,
 451, 451, 452, 452, 452, 452, 454,
 454, 454, 454, 454, 454, 454, 454,
 454, 455, 456, 457, 457, 458, 459,
 459, 461, 462, 462, 462, 462, 462,
 464, 464, 464, 464, 464, 464, 464,
 465, 465, 465, 465, 466, 466, 466,
 466, 466, 466, 466, 466, 466, 466,
 467, 467, 467, 467, 467, 467, 468,
 468, 468, 468, 468, 468, 468, 468,
 469, 469, 469, 469, 469, 469, 469,
 469, 470, 470, 471, 471, 471, 471,
 471, 471, 471, 472, 472, 472, 472,
 472, 472, 472, 472, 472, 472, 473,
 473, 473, 473, 473, 473, 473, 473,
 473, 473, 473, 473, 473, 473, 474,
 474, 474, 474, 475, 475, 475, 475,
 476, 476, 476, 476, 477, 477, 477,
 477, 477, 477, 477, 477, 478, 478,
 480, 480, 480, 480, 480, 481, 481,
 482, 483, 483, 484, 486, 486, 486,
 492, 492, 495, 498, 500, 501, 515,
 523, 531, 533, 533, 534, 535, 535,
 538, 542, 542, 542, 542, 543, 543,
 543, 543, 543, 544, 555, 555, 557,
 557, 558, 558, 558, 559, 559, 562,
 562, 562, 563, 563, 563, 564, 591,
 591, 642, 750, 751, 753, 754, 754,
 756, 757, 757, 757, 757, 757, 757,
 758, 758, 758, 758, 758, 758, 758,
 758, 759, 760, 765, 766, 766, 766,
 767, 767, 768, 770, 771, 771, 773,
 776, 777, 779, 781, 781, 782, 782,
 782, 782, 782, 783, 783, 784, 784,
 785, 787, 787, 788, 788, 789, 790, 811
 __cs_generate_variant:nnNN
 298, 299, 300
 __cs_generate_variant:Nnnw
 300, 300, 301, 301
 __cs_generate_variant:ww
 299, 299, 299
 __cs_generate_variant:wwNN
 300, 301, 301, 302, 302, 303, 303
 __cs_generate_variant:wwNw
 299, 299, 299
 __cs_generate_variant_loop:nNwN
 301, 301, 301, 301, 302, 302
 __cs_generate_variant_loop_-
 end:nwwwNNnn
 301, 301, 301, 301, 302, 302
 __cs_generate_variant_loop_-
 invalid:NNwNNnn 301, 301, 302, 303
 __cs_generate_variant_loop_-
 long:wNNnn 301, 301, 302, 302
 __cs_generate_variant_loop_-
 same:w 301, 301, 302, 302
 __cs_generate_variant_same:N ...
 301, 302, 303, 303
 __cs_get_function_name:N
 25, 25, 275, 275
 __cs_get_function_signature:N ..
 25, 25, 275, 275
 \cs_gset:cn 285
 \cs_gset:cpn
 281, 281, 401, 456, 504, 504, 611
 \cs_gset:cpx 281, 281
 \cs_gset:cx 285
 \cs_gset:Nn 15, 15, 284
 \cs_gset:Npn 11,
 13, 13, 265, 265, 280, 281, 440, 470
 \cs_gset:Npx 13, 265, 265, 280, 281, 440
 \cs_gset:Nx 284
 \cs_gset_eq:cc 281, 282, 306, 385
 \cs_gset_eq:cN 281,
 282, 282, 306, 385, 440, 470, 528, 528
 \cs_gset_eq:Nc
 281, 281, 306, 385, 441, 470
 \cs_gset_eq:NN
 ... 17, 17, 17, 281, 281, 281, 282,
 282, 282, 306, 306, 306, 307, 307,
 307, 384, 385, 388, 428, 462, 559,
 563, 811, 817, 817, 817, 817, 817,
 817, 817, 817, 818, 818, 818, 818,
 818, 818, 818, 818, 818, 818, 818,
 \cs_gset_nopar:cn 285
 \cs_gset_nopar:cpn 280, 280
 \cs_gset_nopar:cpx 280, 280
 \cs_gset_nopar:cx 285
 \cs_gset_nopar:Nn 15, 15, 284
 \cs_gset_nopar:Npn ... 13, 13, 265,
 265, 265, 265, 265, 280, 280, 357, 505

- \cs_gset_nopar:Npx
 . 13, 265, 265, 265, 265, 280,
 280, 357, 384, 384, 386, 386, 386,
 386, 387, 387, 387, 387, 387, 387, 387
- \cs_gset_nopar:Nx 284
- \cs_gset_protected:cn 285
- \cs_gset_protected:cpn 281, 281
- \cs_gset_protected:cpx 281, 281
- \cs_gset_protected:cx 285
- \cs_gset_protected:Nn ... 16, 16, 284
- \cs_gset_protected:Npn
 . 13, 13, 265, 265, 280, 281, 324, 503
- \cs_gset_protected:Npx
 13, 265, 265, 280, 281
- \cs_gset_protected:Nx 284
- \cs_gset_protected_nopar:cn ... 285
- \cs_gset_protected_nopar:cpn 281, 281
- \cs_gset_protected_nopar:cpx 281, 281
- \cs_gset_protected_nopar:cx ... 285
- \cs_gset_protected_nopar:Nn
 16, 16, 284
- \cs_gset_protected_nopar:Npn ...
 14, 14, 265, 265, 280, 281
- \cs_gset_protected_nopar:Npx ...
 14, 265, 265, 280, 281
- \cs_gset_protected_nopar:Nx ... 284
- \cs_if_eq:ccF 287
- \cs_if_eq:ccT 287
- \cs_if_eq:ccTF 286, 287
- \cs_if_eq:cNF 286
- \cs_if_eq:cNT 286
- \cs_if_eq:cNTF 286, 286, 510
- \cs_if_eq:NcF 286
- \cs_if_eq:NcT 286
- \cs_if_eq:NcTF 286, 286
- \cs_if_eq:NN 286
- \cs_if_eq:NNF 286, 286, 287
- \cs_if_eq:NNT 286, 286, 287
- \cs_if_eq:NNTF 23, 23, 286,
 286, 286, 287, 597, 597, 597, 597, 611
- \cs_if_eq_p:cc 286, 286
- \cs_if_eq_p:cN 286, 286
- \cs_if_eq_p:Nc 286, 286
- \cs_if_eq_p:NN 23, 23, 286, 286, 286, 286
- \cs_if_exist:c . 275, 308, 349, 370,
 378, 382, 385, 431, 447, 467, 472, 644
- \cs_if_exist:cF 533
- \cs_if_exist:cT 643
- \cs_if_exist:cTF
 275, 277, 277, 277, 277,
 480, 503, 511, 531, 547, 548, 548, 614
- \cs_if_exist:N 23, 275, 308, 349, 370,
 378, 382, 385, 431, 447, 467, 472, 644
- \cs_if_exist:Nf 279, 279
- \cs_if_exist:NT 348, 348,
 348, 415, 502, 551, 553, 554, 557,
 561, 611, 813, 813, 817, 817, 818, 818
- \cs_if_exist:NTF
 23, 23, 169, 275, 276, 276,
 276, 277, 287, 287, 348, 367, 480,
 502, 557, 560, 561, 813, 822, 824, 827
- \cs_if_exist_p:c 275
- \cs_if_exist_p:N
 23, 23, 24, 275, 818, 818
- \cs_if_exist_use:... 276
- \cs_if_exist_use:c 276, 277
- \cs_if_exist_use:cF
 277, 588, 613, 793, 803, 803
- \cs_if_exist_use:cT 277
- \cs_if_exist_use:cTF 276, 277
- \cs_if_exist_use:N .. 18, 18, 276, 276
- \cs_if_exist_use:Nf 276
- \cs_if_exist_use:NT 276
- \cs_if_exist_use:NTF 18, 18, 276, 276
- \cs_if_free:c 276
- \cs_if_free:cT 304
- \cs_if_free:cTF 276, 511, 511
- \cs_if_free:N 276
- \cs_if_free:Nf 279, 279
- \cs_if_free:NTF .. 23, 23, 37, 276, 304
- \cs_if_free_p:c 276
- \cs_if_free_p:N
 22, 23, 23, 24, 24, 37, 276
- \cs_log:c 759, 759, 760
- \cs_log:N .. 169, 212, 212, 759, 759, 760
- \cs_meaning:c 263, 263, 263, 264
- \cs_meaning:N 17, 17, 262, 262, 263, 287
- \cs_new:cn 285
- \cs_new:cpn 281, 281, 311,
 311, 311, 316, 316, 316, 316, 316,
 316, 316, 316, 316, 316, 316, 316,
 316, 316, 316, 316, 316, 316, 316,
 316, 316, 353, 353, 353, 353, 353,
 353, 353, 353, 373, 373, 373, 373,
 582, 609, 611, 627, 627, 636, 637,
 639, 639, 639, 639, 649, 654, 717, 801
- \cs_new:cpx 281, 281
- \cs_new:cx 285

`\cs_new:Nn` 14, 14, 38, 284
`\cs_new:Npn` 11, 12, 12,
 16, 37, 37, 39, 280, 280, 281, 283,
 283, 288, 288, 288, 289, 289, 289,
 289, 289, 289, 289, 289, 290, 290,
 290, 291, 291, 292, 292, 292, 292,
 292, 292, 292, 292, 292, 293, 293,
 293, 293, 293, 293, 293, 294, 294,
 294, 294, 295, 295, 295, 295, 295,
 296, 296, 296, 296, 296, 296, 296,
 296, 296, 296, 297, 297, 297, 297,
 297, 297, 297, 298, 298, 302, 302,
 302, 302, 303, 303, 305, 308, 310,
 310, 310, 310, 310, 310, 311, 313,
 313, 313, 313, 314, 314, 314, 314,
 314, 314, 314, 315, 315, 315, 316,
 316, 316, 319, 319, 319, 319, 320,
 320, 320, 321, 321, 322, 324, 326,
 326, 327, 327, 332, 334, 334, 334,
 335, 335, 336, 336, 336, 337, 338,
 338, 338, 338, 338, 339, 341, 344,
 344, 344, 345, 346, 346, 346, 346,
 347, 347, 347, 347, 347, 351, 351,
 352, 352, 352, 353, 354, 354, 354,
 354, 354, 354, 355, 355, 355, 355,
 355, 356, 356, 356, 356, 356, 357,
 358, 358, 358, 359, 360, 360, 360,
 360, 360, 360, 361, 361, 361, 362,
 363, 363, 363, 363, 363, 363, 363,
 363, 364, 364, 364, 364, 364, 364,
 365, 365, 365, 365, 365, 365, 366,
 366, 366, 371, 371, 371, 372, 372,
 372, 372, 373, 373, 373, 373, 373,
 373, 373, 375, 375, 375, 376, 376,
 376, 376, 379, 380, 380, 380, 380,
 383, 390, 397, 399, 399, 399, 399,
 400, 400, 400, 400, 400, 400, 400,
 401, 401, 401, 402, 403, 403, 403,
 403, 403, 403, 404, 404, 404, 404,
 405, 405, 405, 405, 406, 406, 406,
 406, 406, 406, 406, 407, 407, 407,
 407, 408, 408, 408, 408, 408, 410,
 410, 411, 412, 412, 413, 415, 415,
 416, 417, 417, 417, 417, 417, 417,
 417, 417, 418, 418, 418, 418, 418,
 418, 419, 419, 419, 419, 419, 420,
 420, 420, 420, 421, 421, 421, 421,
 421, 421, 422, 422, 422, 422, 422,
 423, 423, 423, 424, 424, 424, 424,
 425, 425, 425, 425, 426, 426, 426,
 426, 426, 426, 426, 426, 428, 430,
 430, 431, 432, 434, 436, 438, 439,
 439, 439, 440, 440, 441, 441, 442,
 442, 442, 442, 442, 442, 446, 446,
 447, 447, 447, 447, 449, 452, 452,
 453, 453, 453, 454, 454, 455, 455,
 455, 455, 455, 457, 457, 458, 458,
 458, 458, 458, 459, 459, 459, 460,
 460, 460, 460, 460, 461, 463, 465,
 465, 468, 468, 469, 469, 508, 508,
 508, 508, 508, 508, 510, 511, 515,
 521, 522, 522, 522, 522, 522, 525,
 525, 525, 525, 534, 535, 547, 547,
 547, 553, 567, 573, 574, 574, 574,
 574, 574, 574, 574, 574, 574, 575,
 575, 576, 576, 576, 576, 576, 577,
 577, 577, 577, 577, 578, 578, 578,
 578, 580, 580, 580, 580, 580, 581,
 581, 581, 582, 582, 582, 583, 583,
 583, 583, 583, 584, 584, 584, 584,
 584, 585, 585, 585, 585, 586, 587,
 591, 591, 591, 591, 591, 591, 591,
 591, 593, 593, 594, 594, 594, 594,
 594, 594, 595, 595, 596, 596, 596,
 597, 597, 598, 598, 598, 598, 598,
 598, 599, 599, 599, 608, 608, 608,
 609, 610, 610, 611, 612, 612, 612,
 612, 612, 613, 613, 613, 614, 614,
 615, 615, 616, 616, 616, 616, 617,
 617, 618, 618, 618, 618, 619, 620,
 620, 621, 621, 621, 622, 622, 623,
 624, 624, 624, 624, 625, 625, 627,
 628, 628, 630, 630, 630, 631, 631,
 632, 632, 632, 632, 632, 633, 633,
 633, 634, 634, 634, 635, 635, 635,
 636, 636, 637, 637, 638, 638, 638,
 639, 639, 640, 640, 640, 640, 641,
 641, 641, 642, 642, 643, 643, 643,
 644, 645, 645, 646, 646, 646, 647,
 647, 647, 647, 647, 647, 647, 648,
 648, 648, 649, 649, 649, 650, 650,
 650, 651, 651, 651, 651, 651, 652,
 652, 652, 652, 652, 654, 654, 655,
 655, 656, 656, 656, 657, 657, 657,
 657, 658, 658, 658, 658, 659, 659,
 659, 659, 660, 660, 660, 660, 660,
 661, 661, 662, 662, 662, 663, 664,
 665, 665, 665, 665, 666, 666, 667,
 671, 672, 672, 672, 673, 673, 674,
 674, 674, 675, 675, 675, 675, 676,

676, 677, 677, 678, 678, 679, 680, 281 , 282, 626, 817, 817, 817, 817
681, 681, 681, 681, 681, 681, 682,	<code>\cs_new_eq:Nc</code> 281 , 282
682, 682, 683, 683, 683, 685, 685,	<code>\cs_new_eq:NN</code> 16 , 16, 16, 278, 281 , 281,
685, 685, 686, 687, 687, 687, 687,	282, 282, 282, 282, 287, 288, 290,
687, 687, 688, 688, 688, 688, 689,	297, 298, 305, 305, 305, 306, 306,
689, 690, 690, 691, 691, 692, 692,	306, 306, 306, 306, 306, 306, 322,
692, 693, 693, 693, 693, 693, 694,	328, 328, 328, 328, 328, 328, 328,
694, 694, 694, 694, 696, 697, 697,	328, 328, 328, 339, 339, 339, 345,
697, 698, 698, 698, 699, 699, 699,	345, 345, 345, 345, 348, 348, 349,
699, 700, 700, 700, 700, 700, 701,	350, 354, 358, 369, 369, 369, 369,
701, 701, 701, 701, 701, 702, 702,	369, 369, 369, 369, 369, 374, 375,
702, 702, 704, 704, 704, 705, 705,	376, 380, 380, 380, 380, 383, 383,
705, 706, 707, 707, 707, 708, 708,	384, 384, 385, 385, 385, 385, 385,
708, 709, 709, 709, 709, 709, 709,	385, 385, 385, 396, 396, 401, 414,
710, 710, 710, 711, 711, 711, 712,	414, 418, 427, 427, 428, 429, 429,
712, 713, 713, 713, 713, 714, 714,	429, 429, 429, 429, 429, 429, 443,
714, 714, 714, 715, 715, 715, 716,	443, 443, 443, 443, 443, 443, 443,
716, 718, 719, 719, 720, 720, 720,	443, 443, 443, 443, 443, 443, 443,
721, 721, 721, 721, 721, 721, 722,	443, 443, 443, 443, 444, 445, 445,
722, 722, 723, 723, 724, 725, 725,	445, 445, 445, 445, 445, 445, 445,
726, 726, 726, 727, 727, 728, 728,	445, 445, 450, 450, 450, 450, 450,
729, 729, 729, 729, 734, 734, 734,	450, 450, 450, 450, 450, 450, 451,
734, 734, 734, 735, 735, 735, 735,	451, 451, 451, 451, 462, 462, 462,
736, 736, 736, 737, 738, 738, 740,	462, 462, 462, 462, 462, 470, 470,
741, 741, 742, 742, 743, 743, 743,	472, 472, 472, 472, 472, 473, 473,
743, 744, 744, 744, 745, 745, 745,	473, 476, 476, 476, 476, 478, 478,
746, 746, 747, 747, 748, 748, 749,	478, 484, 484, 484, 484, 484, 484,
749, 749, 749, 749, 749, 750, 751,	502, 556, 557, 560, 561, 561, 562,
751, 751, 751, 752, 752, 753, 753,	563, 565, 568, 578, 587, 587, 587,
753, 753, 754, 754, 755, 755, 755,	587, 595, 596, 596, 596, 596, 596,
755, 755, 756, 756, 756, 756, 782,	596, 756, 756, 757, 757, 757, 785,
782, 782, 782, 782, 782, 783, 783,	785, 785, 785, 786, 786, 797, 798,
783, 783, 784, 784, 785, 786, 786,	798, 818, 818, 818, 818, 818, 819,
787, 787, 787, 787, 787, 789, 789,	819, 819, 819, 819, 819, 819, 819,
789, 790, 790, 790, 790, 790, 791,	819, 819, 819, 819, 819, 819, 819,
791, 791, 791, 792, 792, 792, 793,	819, 819, 819, 819, 822, 822, 826, 827
793, 793, 794, 794, 794, 794, 794,	<code>\cs_new_nopar:cn</code> 285
795, 795, 795, 795, 795, 796, 796,	<code>\cs_new_nopar:cpn</code> 280 , 280,
796, 796, 796, 797, 797, 797, 797,	312, 312, 312, 312, 312, 312, 312,
798, 798, 798, 799, 799, 799, 800,	312, 629, 629, 629, 631, 631, 663, 667
800, 800, 800, 801, 801, 801, 802,	<code>\cs_new_nopar:cpx</code> .. 280 , 280, 304, 653
802, 802, 802, 802, 803, 803, 803,	<code>\cs_new_nopar:cx</code> 285
804, 804, 804, 804, 804, 812, 812,	<code>\cs_new_nopar:Nn</code> 14 , 14, 284
813, 813, 815, 820, 820, 820, 820, 823	<code>\cs_new_nopar:Npn</code> 12 , 12, 26, 278, 280 , 280,
<code>\cs_new:Npx</code> . 12 , 280 , 280, 281, 457,	280, 280, 283, 286, 286, 286, 286,
457, 521, 567, 608, 612, 754, 754, 757	286, 286, 286, 286, 286, 287, 287,
<code>\cs_new:Nx</code> 284	
<code>\cs_new...</code> 11	
<code>\cs_new_eq:cc</code> 271 , 281 , 282, 414	
<code>\cs_new_eq:cN</code>	

- 287, 287, 294, 294, 294, 294, 295,
 295, 295, 295, 295, 295, 295, 295,
 295, 295, 296, 296, 296, 296, 317,
 317, 339, 339, 339, 341, 341, 341,
 341, 342, 363, 363, 363, 363, 363,
 363, 363, 363, 363, 363, 363, 363,
 363, 363, 363, 401, 402, 402, 408,
 408, 413, 419, 421, 423, 423, 424,
 425, 435, 439, 440, 457, 457, 470,
 470, 505, 565, 591, 609, 630, 630,
 630, 630, 630, 630, 630, 630, 630,
 631, 631, 631, 642, 650, 688, 688,
 735, 740, 740, 750, 751, 753, 754,
 779, 780, 789, 789, 789, 789, 789, 789
 \cs_new_nopar:Npx
 . 12, 280, 280, 280, 298, 299, 299, 730
 \cs_new_nopar:Nx 284
 \cs_new_protected:cn 285
 \cs_new_protected:cpx
 . 281, 281, 508, 508, 516, 538, 538,
 538, 538, 538, 538, 538, 538, 539,
 539, 539, 539, 539, 539, 539, 539,
 539, 539, 539, 539, 539, 539, 539,
 539, 539, 540, 540, 540, 540, 540,
 540, 540, 540, 540, 540, 540, 540,
 540, 540, 541, 541, 541, 541, 541,
 541, 541, 541, 541, 541, 541, 541,
 541, 541, 541, 541, 541, 542, 542, 811
 \cs_new_protected:cpx
 281, 281, 414, 509, 509,
 509, 509, 509, 509, 509, 509, 516,
 516, 516, 516, 516, 516, 516, 516, 811
 \cs_new_protected:cx 285
 \cs_new_protected:Nn 14, 14, 284
 \cs_new_protected:Npn
 12, 12, 280, 280, 281, 281,
 282, 282, 282, 284, 285, 285, 287,
 287, 290, 296, 298, 298, 299, 299,
 300, 301, 303, 304, 305, 306, 306,
 306, 306, 306, 306, 308, 318, 322,
 323, 323, 324, 324, 324, 325, 325,
 325, 325, 325, 325, 325, 325, 325,
 325, 325, 325, 325, 325, 325, 325,
 325, 325, 325, 325, 326, 326, 326,
 326, 326, 326, 326, 326, 326, 326,
 326, 326, 326, 326, 326, 326, 326,
 327, 327, 327, 328, 339, 340, 340,
 340, 340, 340, 340, 348, 348, 349,
 349, 349, 349, 349, 349, 350, 350,
 350, 350, 350, 357, 357, 367, 369,
 369, 370, 370, 370, 370, 370, 370,
 370, 370, 371, 371, 371, 371, 377,
 377, 378, 378, 378, 378, 378, 378,
 378, 378, 379, 379, 379, 379, 381,
 381, 381, 381, 381, 381, 382, 382,
 382, 382, 382, 382, 382, 382, 384,
 384, 384, 385, 385, 385, 385, 385,
 385, 386, 386, 386, 386, 386, 386,
 386, 386, 386, 386, 386, 386, 387,
 387, 387, 387, 387, 387, 387, 387,
 387, 387, 390, 390, 391, 391, 391,
 392, 392, 393, 394, 394, 395, 396,
 396, 396, 396, 401, 402, 402, 402,
 404, 404, 407, 407, 413, 413, 428,
 428, 428, 429, 429, 429, 429, 429,
 429, 430, 431, 431, 431, 431, 432,
 432, 432, 432, 432, 433, 433, 433,
 434, 435, 436, 436, 436, 436, 437,
 437, 437, 440, 440, 440, 441, 441,
 444, 444, 445, 445, 446, 446, 446,
 448, 448, 448, 448, 449, 449, 449,
 449, 450, 451, 451, 451, 452, 452,
 452, 452, 452, 454, 456, 456, 456,
 456, 456, 460, 460, 462, 462, 462,
 462, 462, 463, 463, 463, 464, 464,
 464, 464, 466, 467, 470, 470, 471,
 471, 471, 471, 471, 471, 471, 472,
 472, 472, 472, 472, 472, 473, 473,
 473, 473, 473, 474, 474, 474, 475,
 475, 475, 475, 475, 475, 476, 476,
 476, 476, 476, 476, 476, 476, 477,
 477, 477, 477, 477, 477, 477, 477,
 477, 477, 478, 478, 480, 480, 481,
 481, 482, 482, 483, 484, 485, 485,
 485, 485, 486, 486, 486, 486, 487,
 487, 488, 490, 491, 492, 492, 493,
 493, 493, 494, 494, 494, 494, 494,
 495, 497, 498, 498, 499, 500, 500,
 501, 503, 504, 504, 504, 504, 504,
 504, 505, 506, 506, 507, 507, 508,
 510, 512, 512, 513, 513, 514, 514,
 514, 515, 515, 515, 515, 515, 523,
 523, 523, 524, 525, 525, 527, 527,
 527, 528, 528, 528, 528, 531, 531,
 531, 531, 531, 531, 532, 532, 532,
 533, 533, 533, 535, 535, 535, 536,
 536, 536, 536, 537, 537, 537, 542,
 542, 542, 543, 543, 543, 543, 543,
 544, 544, 546, 548, 548, 552, 553,
 553, 553, 554, 554, 554, 555, 555,

- 555, 555, 557, 557, 557, 558, 558,
 559, 559, 559, 560, 560, 562, 562,
 562, 562, 563, 563, 563, 564, 564,
 564, 567, 569, 570, 574, 587, 588,
 588, 589, 590, 590, 590, 642, 642,
 742, 757, 757, 757, 757, 758, 758,
 758, 758, 758, 759, 761, 761, 762,
 763, 763, 763, 763, 764, 764, 765,
 765, 765, 765, 766, 766, 766, 767,
 767, 768, 768, 770, 772, 773, 773,
 773, 774, 774, 775, 775, 775, 775,
 776, 776, 777, 777, 778, 778, 778,
 778, 778, 779, 779, 779, 779, 780,
 780, 784, 784, 787, 788, 816, 816,
 820, 820, 822, 823, 824, 824, 824, 824
 \cs_new_protected:Npx
 12, 280, 280, 281, 299, 304, 526
 \cs_new_protected:Nx 284
 \cs_new_protected_nopar:cn 285
 \cs_new_protected_nopar:cpn
 281, 281, 538, 541, 542, 550, 550
 \cs_new_protected_nopar:cpx
 281, 281, 284, 286, 304, 343
 \cs_new_protected_nopar:cx 285
 \cs_new_protected_nopar:Nn 14, 14, 284
 \cs_new_protected_nopar:Npn
 12, 12,
 280, 280, 280, 281, 281, 281, 281,
 281, 281, 282, 282, 282, 282, 282,
 284, 284, 287, 287, 294, 295, 295,
 295, 295, 295, 295, 295, 295, 295,
 295, 295, 295, 295, 295, 296, 308,
 318, 339, 339, 342, 350, 350, 350,
 350, 350, 351, 357, 367, 377, 380,
 383, 389, 389, 389, 393, 393, 393,
 393, 399, 399, 399, 402, 430, 430,
 434, 434, 436, 436, 437, 437, 440,
 446, 446, 448, 448, 448, 448, 449,
 449, 466, 466, 466, 467, 474, 483,
 483, 502, 502, 502, 512, 514, 522,
 523, 534, 534, 534, 534, 535, 535,
 537, 542, 542, 543, 544, 545, 545,
 546, 546, 547, 555, 558, 559, 562,
 563, 564, 564, 567, 569, 569, 569,
 570, 570, 571, 571, 588, 588, 588,
 589, 589, 589, 590, 590, 590, 590,
 590, 590, 758, 758, 758, 758, 759,
 760, 760, 760, 771, 771, 775, 779,
 780, 780, 780, 780, 780, 781, 781,
 781, 781, 781, 782, 782, 783, 784,
 784, 784, 784, 785, 785, 785, 786,
 787, 787, 788, 788, 811, 811, 815,
 816, 816, 816, 822, 822, 822, 822,
 825, 826, 827, 827, 827, 828, 828, 828
 \cs_new_protected_nopar:Npx
 12, 280, 280,
 281, 298, 299, 299, 299, 303, 304, 570
 \cs_new_protected_nopar:Nx 284
 __cs_parm_from_arg_count:nnF ...
 269, 282, 282, 284
 __cs_parm_from_arg_count_-
 test:nnF 282, 282, 283
 \cs_set:cn 285
 \cs_set:cpn ... 281, 281, 504, 504, 590
 \cs_set:cpx 281, 281
 \cs_set:cx 285
 \cs_set:Nn 15, 15, 284, 284, 284
 \cs_set:Npn
 ... 11, 12, 12, 37, 37, 39, 264, 264,
 265, 266, 266, 266, 266, 266, 266,
 266, 266, 266, 266, 266, 266, 266,
 266, 266, 266, 266, 266, 266, 266,
 267, 267, 267, 267, 267, 267, 267,
 267, 267, 272, 272, 272, 272, 274,
 274, 274, 274, 275, 275, 276, 276,
 276, 276, 277, 277, 277, 277, 280,
 280, 281, 281, 284, 284, 285, 342,
 343, 346, 346, 346, 371, 371, 374,
 374, 374, 374, 374, 374, 375, 375,
 392, 395, 399, 404, 415, 424, 425,
 425, 425, 452, 454, 463, 463, 588,
 589, 589, 589, 589, 780, 806, 806, 820
 \cs_set:Npx 12, 264, 265, 274, 281, 395
 \cs_set:Nx 284
 \cs_set_eq:cc . 271, 281, 281, 306, 385
 \cs_set_eq:cN
 281, 281, 306, 385, 537, 537, 587
 \cs_set_eq:Nc 281, 281, 306, 385
 \cs_set_eq:NN 17, 17, 17, 281, 281,
 281, 281, 281, 281, 281, 282, 290,
 299, 299, 304, 306, 306, 306, 306,
 307, 307, 329, 339, 340, 340, 340,
 343, 385, 388, 434, 434, 434, 437,
 437, 438, 567, 567, 567, 568, 811, 819
 \cs_set_nopar:cn 285
 \cs_set_nopar:cpn 280, 280, 280
 \cs_set_nopar:cpx 280, 280
 \cs_set_nopar:cx 285
 \cs_set_nopar:Nn 15, 15, 284

- \cs_set_nopar:Npn
 11, 13, 13, 56, 264, 264, 264, 264,
 265, 265, 265, 265, 265, 265, 267,
 267, 273, 278, 280, 280, 280, 635, 813
- \cs_set_nopar:Npx ... 13, 264, 264,
 265, 265, 265, 265, 280, 288, 290,
 296, 340, 340, 340, 340, 386, 386,
 386, 386, 386, 386, 386, 387, 387,
 387, 387, 552, 567, 567, 567, 567, 567
- \cs_set_nopar:Nx 284
- \cs_set_protected:cn 285
- \cs_set_protected:cpn
 281, 281, 533, 535
- \cs_set_protected:cpx 281, 281
- \cs_set_protected:cx 285
- \cs_set_protected:Nn 15, 15, 284
- \cs_set_protected:Npn
 11, 13, 13, 264, 265, 265,
 268, 268, 269, 269, 269, 270, 270,
 271, 271, 271, 271, 272, 278, 278,
 278, 279, 279, 279, 279, 280, 281,
 281, 282, 283, 306, 306, 307, 307,
 307, 307, 307, 307, 379, 388, 388,
 388, 388, 388, 389, 414, 414, 435,
 483, 508, 516, 516, 609, 627, 629,
 629, 636, 788, 806, 811, 814, 815, 820
- \cs_set_protected:Npx 13, 264, 265, 281
- \cs_set_protected:Nx 284
- \cs_set_protected_nopar:cn 285
- \cs_set_protected_nopar:cpn 281, 281
- \cs_set_protected_nopar:cpx 281, 281
- \cs_set_protected_nopar:cx 285
- \cs_set_protected_nopar:Nn 15, 15, 284
- \cs_set_protected_nopar:Npn
 13, 13, 240, 264, 265, 265,
 265, 265, 265, 265, 265, 265, 265,
 268, 268, 268, 268, 268, 268, 268,
 268, 271, 271, 277, 277, 277, 277,
 277, 277, 277, 278, 278, 278, 278,
 278, 278, 278, 279, 280, 281, 281,
 482, 502, 502, 507, 564, 564, 569, 569
- \cs_set_protected_nopar:Npx
 13, 240, 264, 265, 277, 281, 512
- \cs_set_protected_nopar:Nx 284
- \cs_show:c . 287, 287, 287, 759, 759, 760
- \cs_show:N 17,
 17, 23, 169, 212, 287, 287, 287, 759, 760
- __cs_split_function:NN
 25, 25, 268, 269, 271, 271,
 274, 274, 274, 275, 275, 283, 285, 298
- __cs_split_function_auxi:w
 274, 274, 274, 274
- __cs_split_function_auxii:w ...
 274, 274, 274
- __cs_tmp:w . 25, 280, 280, 280, 280,
 280, 280, 280, 280, 280, 280, 280,
 280, 280, 280, 280, 280, 280, 281,
 281, 281, 281, 281, 281, 281, 281,
 281, 281, 281, 281, 281, 281, 281,
 281, 281, 281, 284, 285, 285, 285,
 285, 285, 285, 285, 285, 285, 285,
 285, 285, 285, 285, 285, 285, 285,
 285, 286, 286, 286, 286, 286, 286,
 286, 286, 286, 286, 286, 286, 286,
 286, 286, 286, 286, 286, 286, 286,
 286, 286, 286, 286, 298, 299, 299,
 303, 304, 304, 379, 379, 388, 388, 388
- __cs_to_str:N 273, 273, 274, 274, 274
- \cs_to_str:N .. 4, 19, 19, 102, 109,
 273, 273, 273, 274, 274, 426, 426,
 426, 427, 427, 427, 427, 427, 427,
 427, 427, 427, 427, 565, 642, 794, 805
- __cs_to_str:w 273, 273, 273, 274, 274
- \cs_undefine:c 282, 282, 537
- \cs_undefine:N
 17, 17, 282, 282, 517, 517, 517
- csc 206
- cscd 206
- \csname 234, 234, 235, 235, 236,
 236, 237, 237, 238, 239, 239, 240, 242
- \currentgrouplevel 248
- \currentgrouptype 248
- \currentifbranch 248
- \currentiflevel 248
- \currentifttype 248
- \CYRA 807
- \cyra 807
- \CYRABHCH 807
- \cyrabhch 807
- \CYRABHCHDSC 807
- \cyrabhchdsc 807
- \CYRABHDZE 807
- \cyrabhdze 807
- \CYRABHHA 807
- \cyrabhha 807
- \CYRAE 807
- \cyrae 807
- \CYRB 807
- \cyrb 807

\CYRBYUS	807	\CYRHHCRS	807
\cyrbyus	807	\cyrhhcrs	807
\CYRC	807	\CYRHHK	807
\cyrC	807	\cyrhhk	807
\CYRCH	807	\CYRHRDSN	807
\cyrch	807	\cyrhrdsn	807
\CYRCHLDSC	807	\CYRI	808
\cyrchldsc	807	\cyri	808
\CYRCHRDSC	807	\CYRIE	808
\cyrchrdsc	807	\cyrie	808
\CYRCHVCRS	807	\CYRII	808
\cyrchvcrs	807	\cyrii	808
\CYRD	807	\CYRISHRT	808
\cyrd	807	\cyrishrt	808
\CYRDELTA	807	\CYRISHRTDSC	808
\cyrdelta	807	\cyrishrtdsc	808
\CYRDJE	807	\CYRIZH	808
\cyrdje	807	\cyrizh	808
\CYRDZE	807	\CYRJE	808
\cyrdze	807	\cyrje	808
\CYRDZHE	807	\CYRK	808
\cyrdzhe	807	\cyrk	808
\CYRE	807	\CYRKBEAK	808
\cyre	807	\cyrkbeak	808
\CYREPS	807	\CYRKDSC	808
\cyreps	807	\cyrkdsc	808
\CYREREV	807	\CYRKHCRS	808
\cyrerev	807	\cyrkhcrs	808
\CYRERY	807	\CYRKHK	808
\cyrery	807	\cyrkhk	808
\CYRF	807	\CYRKVCRS	808
\cyrf	807	\cyrkvcrs	808
\CYRFITA	807	\CYRL	808
\cyrfita	807	\cyr1	808
\CYRG	807	\CYRLDSC	808
\cyrg	807	\cyrldsc	808
\CYRGDSC	807	\CYRLHK	808
\cyrgdsc	807	\cyr1hk	808
\CYRGDSCHCRS	807	\CYRLJE	808
\cyrgdschcrs	807	\cyr1je	808
\CYRGHCRS	807	\CYRM	808
\cyrghcrs	807	\cyrm	808
\CYRGHK	807	\CYRMDSC	808
\cyrghk	807	\cyrmdsc	808
\CYRGUP	807	\CYRMHK	808
\cyrgup	807	\cyrmhk	808
\CYRH	807	\CYRN	808
\cyrh	807	\cyrn	808
\CYRHDSC	807	\CYRNDSC	808
\cyrhdsc	807	\cyrndsc	808

\CYRNG	808	\CYRTSHE	809
\cyrng	808	\cyrtshe	809
\CYRNHK	808	\CYRU	809
\cyrnhk	808	\cyru	809
\CYRNJE	808	\CYRUSHRT	809
\cyrnje	808	\cyrushrt	809
\CYRNLHK	808	\CYRV	809
\cyrnlhk	808	\cyrv	809
\CYRO	808	\CYRW	809
\cyro	808	\cyrw	809
\CYROTLD	808	\CYRY	809
\cyrotld	808	\cyry	809
\CYRP	808	\CYRYA	809
\cyrp	808	\cyrya	809
\CYRPHK	808	\CYRYAT	809
\cyrphk	808	\cyryat	809
\CYRQ	808	\CYRYHCRS	809
\cyrq	808	\cyryhcrs	809
\CYRR	808	\CYRYI	809
\cyrr	808	\cyryi	809
\CYRRDSC	808	\CYRYO	809
\cyrrdsc	808	\cyryo	809
\CYRRHK	808	\CYRYU	809
\cyrrhk	808	\cyryu	809
\CYRTICK	808	\CYRZ	809
\cyrtick	808	\cyrz	809
\CYRS	808	\CYRZDSC	809
\cyrs	808	\cyrzdsc	809
\CYRSACRS	808	\CYRZH	809
\cyrsacrs	808	\cyrzh	809
\CYRSCHWA	808	\CYRZHDSC	809
\cyrschwa	808	\cyrzhdsc	809
\CYRSDSC	808		
\cyrsdsc	808		
\CYRSEMISFTSN	808		
\cyrsemisftsn	808		
\CYRSFTSN	808		
\cyrsftsn	808		
\CYRSH	808		
\cyrsh	808		
\CYRSHCH	808		
\cyrshch	808		
\CYRSHHA	808		
\cyrshha	808		
\CYRT	809		
\cyr	809		
\CYRTDSC	809		
\cyrt	809		
\CYRTDSC	809		
\cyrt	809		
\CYRTETSE	809		
\cyrtetse	809		

D

\day	242
dd	208
\deadcycles	243
\def	235, 236, 237, 237, 237, 238, 238, 238, 238, 239, 239, 239, 239, 239, 240, 241, 243
default commands:	
.default:n	174, 539
.default:o	174, 539
.default:V	174, 539
.default:x	174, 539
\defaultthyphenchar	243
\defaultskewchar	243
deg	208
\delcode	243
\delimiter	243

- \delimiterfactor 243
- \delimitershortfall 243
- \detokenize 240, 248
- \DH 806
- \dh 806
- dim commands:
 - \dim_(g)zero:N 79
 - __dim_abs:N 371, 371, 371
 - \dim_abs:n 80, 80, 371, 371
 - \dim_add:cn 371
 - \dim_add:Nn . 80, 80, 371, 371, 371, 371
 - \dim_case:nn 83, 373, 373
 - \dim_case:nnF 373
 - \dim_case:nnT 373
 - __dim_case:nnTF
 - 373, 373, 373, 373, 373, 373
 - \dim_case:nnTF 83, 83, 373, 373
 - __dim_case:nw 373, 373, 373, 373
 - __dim_case_end:nw 373, 373, 374
 - \dim_compare:n 372
 - \dim_compare:n(TF) 78
 - \dim_compare:nF 374, 374
 - \dim_compare:nNn 372
 - \dim_compare:nNnF 375, 375
 - \dim_compare:nNnT
 - 374, 375, 491, 491, 770
 - \dim_compare:nNnTF
 - .. 81, 81, 83, 83, 84, 84, 372, 373,
 - 488, 488, 488, 488, 489, 489, 489,
 - 489, 491, 494, 494, 769, 769, 770, 770
 - \dim_compare:nT 374, 374
 - \dim_compare:nTF
 - 82, 82, 84, 84, 84, 84, 88, 372
 - __dim_compare:w 372, 372, 372
 - __dim_compare:wNN
 - 372, 372, 372, 372, 373
- dim_compare_
 - __dim_compare_>:w 372
 - __dim_compare_:w 372
 - __dim_compare_<:w 372
 - __dim_compare_end:w 372, 373
 - \dim_compare_p:n 82, 82, 372
 - \dim_compare_p:nNn 81, 81, 372
 - \dim_const:cn 369
 - \dim_const:Nn
 - 79, 79, 369, 369, 369, 377, 377
 - \dim_do_until:nn . 84, 84, 374, 374, 374
 - \dim_do_until:nNnn 83, 83, 374, 375, 375
 - \dim_do_while:nn . 84, 84, 374, 374, 374
 - \dim_do_while:nNnn 83, 83, 374, 375, 375
- \dim_eval:n . 81, 82, 84, 84, 85, 93,
 - 373, 373, 373, 373, 375, 375, 377,
 - 482, 483, 486, 486, 486, 486, 487,
 - 487, 487, 494, 494, 501, 501, 501,
 - 770, 770, 776, 776, 776, 776, 778, 778
- __dim_eval:w 93, 93,
 - 369, 369, 370, 371, 371, 371, 371,
 - 371, 371, 371, 371, 371, 372, 372,
 - 372, 372, 373, 373, 373, 375, 375,
 - 376, 376, 472, 472, 472, 472, 473,
 - 473, 473, 475, 476, 477, 477, 478,
 - 611, 612, 629, 769, 769, 770, 770, 772
- __dim_eval_end: 93, 93, 93, 93, 369,
 - 369, 370, 371, 371, 371, 371,
 - 372, 372, 375, 375, 376, 472, 472,
 - 472, 472, 473, 473, 473, 475, 476,
 - 477, 477, 478, 769, 769, 770, 770, 772
- \dim_gadd:cn 371
- \dim_gadd:Nn 80, 371, 371, 371
- .dim_gset:c 174, 539
- \dim_gset:cn 370
- .dim_gset:N 174, 539
- \dim_gset:Nn ... 80, 369, 370, 370, 370
- \dim_gset_eq:cc 370
- \dim_gset_eq:cN 370
- \dim_gset_eq:Nc 370
- \dim_gset_eq:NN 80, 370, 370, 370, 370
- \dim_gsub:cn 371
- \dim_gsub:Nn 80, 371, 371, 371
- \dim_gzero:c 370
- \dim_gzero:N ... 79, 370, 370, 370, 370
- \dim_gzero_new:c 370
- \dim_gzero_new:N ... 79, 370, 370, 370
- \dim_if_exist:c 370
- \dim_if_exist:cTF 370
- \dim_if_exist:N 370
- \dim_if_exist:NTF 79, 79, 370, 370, 370
- \dim_if_exist_p:c 370
- \dim_if_exist_p:N 79, 79, 370
- \dim_log:c 785, 785
- \dim_log:N 221, 221, 785, 785
- \dim_log:n 221, 221, 785, 785
- \dim_max:nn . 80, 80, 371, 371, 775, 775
- __dim_maxmin:wwN .. 371, 371, 371, 371
- \dim_min:nn
 - 80, 80, 371, 371, 775, 775, 775
- \dim_new:c 369
- \dim_new:N 79,
 - 79, 79, 369, 369, 369, 369, 370, 370,
 - 377, 377, 377, 377, 478, 479, 479,

- 479, 479, 479, 479, 496, 496, 496,
- 760, 760, 760, 760, 760, 760, 760,
- 760, 771, 771, 771, 771, 771, 776, 776
- _dim_ratio:n 372, 372, 372, 372
- \dim_ratio:nn
- 81, 81, 81, 372, 372, 376, 376
- .dim_set:c 174, 539
- \dim_set:cn 370
- .dim_set:N 174, 539
- \dim_set:Nn 80, 80, 370, 370, 370, 370,
- 482, 483, 488, 488, 489, 489, 489,
- 489, 490, 490, 491, 493, 493, 493,
- 493, 493, 493, 496, 499, 499, 500,
- 500, 500, 500, 761, 761, 761, 762,
- 763, 765, 765, 765, 765, 767, 767,
- 767, 767, 767, 767, 773, 774, 774,
- 775, 775, 775, 775, 775, 775, 775,
- 775, 775, 775, 777, 777, 777, 778, 778
- \dim_set_eq:cc 370
- \dim_set_eq:cN 370
- \dim_set_eq:Nc 370
- \dim_set_eq:NN 80, 80,
- 370, 370, 370, 370, 482, 482, 483, 483
- \dim_show:c 376
- \dim_show:N 86, 86, 376, 376, 376
- \dim_show:n . 86, 86, 377, 377, 785, 785
- _dim_strip_bp:n 384, 384
- _dim_strip_pt:n 384, 384
- \dim_sub:cn 371
- \dim_sub:Nn . 80, 80, 371, 371, 371, 371
- \dim_to_decimal:n
- 85, 85, 375, 375, 376, 376, 384
- _dim_to_decimal:w . . . 375, 375, 376
- \dim_to_decimal_in_bp:n
- 85, 85, 86, 376,
- 376, 384, 824, 824, 824, 825, 825, 825
- \dim_to_decimal_in_sp:n
- 85, 85, 86, 376, 376
- \dim_to_decimal_in_unit:nn
- 86, 86, 86, 376, 376
- \dim_to_fp:n 86,
- 86, 86, 376, 489, 489, 489, 489,
- 490, 490, 490, 490, 490, 490, 490,
- 490, 490, 611, 629, 755, 755, 755,
- 763, 763, 763, 763, 764, 764, 764,
- 764, 765, 765, 765, 766, 766, 766,
- 766, 766, 766, 766, 766, 774, 774,
- 774, 774, 776, 776, 776, 776, 778, 778
- \dim_until_do:nn . 84, 84, 374, 374, 374
- \dim_until_do:nNnn 84, 84, 374, 374, 375
- \dim_use:c 375, 375
- \dim_use:N
- 84, 85, 85, 85, 371, 371, 371, 371,
- 371, 371, 371, 372, 373, 375, 375,
- 375, 375, 375, 486, 486, 486, 486,
- 487, 487, 493, 493, 773, 773, 773,
- 773, 773, 773, 773, 773, 774, 774,
- 774, 774, 774, 774, 778, 778, 778, 778
- \dim_while_do:nn . 84, 84, 374, 374, 374
- \dim_while_do:nNnn 84, 84, 374, 374, 374
- \dim_zero:c 370
- \dim_zero:N 79, 79, 370, 370,
- 370, 370, 370, 488, 488, 761, 765, 767
- \dim_zero_new:c 370
- \dim_zero_new:N . 79, 79, 370, 370, 370
- \dimen 243
- \dimendef 243
- \dimexpr 248
- \directlua
- 234, 234, 235, 235, 236, 237, 237, 254
- \disablecjktoken 258
- \discretionary 243
- \displayindent 243
- \displaylimits 243
- \displaystyle 243
- \displaywidowpenalties 249
- \displaywidowpenalty 243
- \displaywidth 243
- \divide 243
- \DJ 806
- \dj 806
- \do 324
- dollar commands:
- \c_dollar_str 117, 426, 427
- \dospecials 323, 324, 324
- \doublehyphendemerits 243
- \dp 243
- driver commands:
- _driver_absolute_lengths:n
- 823, 823, 824
- _driver_box_rotate_begin:
- 233, 233, 762, 825, 825
- _driver_box_rotate_end:
- 233, 233, 762, 825, 826
- _driver_box_scale_begin:
- 233, 233, 767, 826, 826
- _driver_box_scale_end:
- 233, 233, 767, 826, 827
- _driver_box_use_clip:N
- 232, 232, 768, 824, 824

<code>__driver_color_ensure_current:</code> .	332, 332, 334, 334, 334, 334, 335,
..... 233,	335, 335, 335, 335, 335, 338, 338,
233, 502, 502, 502, 827, 827, 827, 828	338, 338, 341, 341, 342, 342, 342,
<code>__driver_color_reset:</code>	346, 346, 347, 347, 349, 353, 353,
.... 827, 827, 827, 827, 828, 828, 828	354, 355, 362, 362, 364, 371, 371,
<code>\l__driver_color_stack_int</code>	372, 372, 373, 379, 397, 397, 397,
..... 827, 827, 827, 828	397, 398, 398, 398, 400, 408, 409,
<code>\l__driver_current_color_tl</code>	409, 410, 410, 410, 410, 411, 411,
827, 827, 827, 827, 827, 827, 828, 828	412, 416, 416, 416, 417, 420, 422,
<code>__driver_literal:n</code> 822, 822, 823,	422, 422, 422, 422, 434, 435, 436,
824, 824, 824, 825, 826, 826, 828, 828	449, 449, 450, 450, 468, 473, 473,
<code>__driver_literal_direct:n</code>	473, 560, 560, 576, 577, 577, 577,
..... 822, 823, 823	577, 577, 577, 581, 584, 584, 584,
<code>__driver_matrix:n</code>	585, 585, 587, 594, 594, 594, 595,
..... 824, 824, 824, 824, 825, 826	595, 596, 597, 597, 598, 598, 599,
<code>__driver_state_restore:</code>	599, 609, 609, 609, 610, 610, 613,
.... 822, 822, 822, 822, 825, 826, 827	614, 614, 615, 615, 615, 616, 616,
<code>__driver_state_save:</code>	617, 618, 619, 619, 620, 620, 621,
.... 822, 822, 822, 822, 824, 825, 826	621, 621, 622, 622, 623, 623, 624,
<code>\dtou</code>	624, 624, 624, 625, 625, 626, 626,
<code>\dump</code>	626, 626, 627, 628, 631, 631, 633,
	633, 633, 634, 634, 635, 635, 636,
	637, 637, 637, 638, 638, 638, 639,
	640, 640, 640, 641, 641, 641, 641,
	644, 645, 645, 646, 646, 646, 646,
	646, 646, 648, 649, 649, 649, 649,
	650, 650, 650, 654, 654, 654, 654,
	655, 655, 655, 656, 657, 658, 659,
	660, 661, 661, 661, 661, 662, 663,
	663, 663, 663, 665, 672, 674, 674,
	675, 676, 680, 680, 680, 682, 693,
	694, 694, 702, 702, 704, 704, 705,
	705, 708, 710, 711, 711, 712, 712,
	712, 712, 712, 717, 717, 718, 719,
	719, 719, 720, 721, 721, 721, 721,
	722, 723, 723, 723, 723, 723, 723,
	724, 725, 725, 726, 726, 727, 728,
	729, 735, 737, 737, 737, 738, 741,
	741, 741, 741, 745, 745, 746, 746,
	748, 748, 751, 753, 753, 753, 755,
	755, 812, 812, 812, 812, 812, 813, 816
<code>em</code>	208
<code>\emergencystretch</code>	243
empty commands:	
<code>\c_empty_box</code>	149, 150, 471, 471, 474, 474, 495
<code>\c_empty_clist</code>	139, 444, 444, 449, 449, 450, 450
<code>\c_empty_coffin</code> 158, 484, 484, 484, 495	
<code>\c_empty_prop</code>	146,
E	
e commands:	
<code>\c_e_fp</code>	199, 202, 759, 759
<code>\ECIRCUMFLEX</code>	811
<code>\Ecircumflex</code>	810, 810
<code>\ecircumflex</code>	810, 810, 811
<code>\edef</code>	4, 238, 238, 239, 243
<code>\efcode</code>	252
eight commands:	
<code>\c_eight</code>	76, 325, 326, 365,
367, 367, 420, 420, 424, 585, 619,	
620, 712, 723, 723, 734, 734, 734, 736	
eleven commands:	
<code>\c_eleven</code>	76,
323, 325, 326, 367, 367, 659, 662, 663	
<code>\else</code>	234, 234, 235,
235, 236, 236, 236, 237, 237, 238, 243	
else commands:	
<code>\else:</code>	23, 38, 77, 77, 77,
77, 77, 93, 154, 154, 154, 190, 190,	
262, 262, 263, 267, 270, 275, 275,	
275, 275, 275, 276, 276, 276, 276,	
278, 282, 283, 283, 283, 286, 291,	
299, 299, 302, 302, 302, 303, 303,	
307, 309, 311, 311, 311, 317, 317,	
317, 317, 319, 320, 320, 329, 329,	
329, 329, 330, 330, 330, 330, 330,	
330, 331, 331, 331, 331, 332, 332,	

462, 462, 462, 462, 462, 467, 533	\etex_glueshrink:D 249, 785
\c_empty_seq 129, 428,	\etex_glueshrinkorder:D 249
428, 428, 428, 428, 428, 434, 435, 436	\etex_gluestretch:D 249, 785
\c_empty_tl 108, 360, 360,	\etex_gluestretchorder:D 249
360, 384, 385, 385, 386, 386, 397, 444	\etex_gluetomu:D 249
\enablecjktoken 258	\etex_ifcsname:D 249, 262
\end 238, 243, 261 249, 258, 259, 260, 260,
\endcsname . . . 234, 234, 235, 235, 236,	260, 260, 261, 261, 261, 262, 262, 264
236, 237, 237, 238, 239, 239, 240, 243	\etex_iffontchar:D 249
\endgroup 234, 234, 235,	\etex_interactionmode:D
236, 236, 237, 237, 238, 238, 239, 243 249, 474, 474, 474
\endinput 238, 243	\etex_interlinepenalties:D 249
\endL 249	\etex_lastlinefit:D 249
\endlinechar 240, 240, 240, 243	\etex_lastnodetype:D 249
\endR 249	\etex_marks:D 249
\ensuremath 805	\etex_middle:D 249
\eqno 243	\etex_muexpr:D 249, 382, 382, 382, 383
\errhelp 238, 238, 243	\etex_mutoglu:D 249
\errmessage 238, 238, 243	\etex_numexpr:D 249, 345
\ERROR 814	\etex_pagediscards:D 249
\errorcontextlines 243	\etex_parshapedimen:D 249
\errorstopmode 243	\etex_parshapeindent:D 249
\escapechar 243	\etex_parshapelength:D 249
etex commands:	\etex_predisplaydirection:D . . . 249
\etex... 9	\etex_protected:D
\etex_beginL:D 248 249, 264, 265, 265, 265,
\etex_beginR:D 248	265, 265, 265, 265, 265, 265, 265, 265
\etex_botmarks:D 248	\etex_readline:D 249, 560
\etex_clubpenalties:D 248	\etex_savinghyphcodes:D 249
\etex_currentgrouplevel:D 248	\etex_savingvdiscards:D 249
\etex_currentgrouptype:D 248	\etex_scantokens:D . 249, 390, 392, 392
\etex_currentifbranch:D 248	\etex_showgroups:D 249
\etex_currentiflevel:D 248	\etex_showifs:D 249
\etex_currentifttype:D 248	\etex_showtokens:D
\etex_detokenize:D 249, 261, 413, 524, 525
. 248, 263, 397, 398, 402, 415	\etex_splitbotmarks:D 249
\etex_dimexpr:D 248, 369	\etex_splitdiscards:D 249
\etex_displaywidowpenalties:D . . 249	\etex_splitfirstmarks:D 249
\etex_endL:D 249	\etex_TeXxTstate:D 249
\etex_endR:D 249	\etex_topmarks:D 249
\etex_eTeXrevision:D 249	\etex_tracingassigns:D 249
\etex_eTeXversion:D 249	\etex_tracinggroups:D 249
\etex_everyeof:D . . . 249, 390, 788, 788	\etex_tracingifs:D 249
\etex_firstmarks:D 249	\etex_tracingnesting:D 250
\etex_fontchardp:D 249	\etex_tracingscantokens:D 250
\etex_fontcharht:D 249	\etex_unexpanded:D
\etex_fontcharic:D 249 250, 261, 262, 297, 297, 297,
\etex_fontcharwd:D 249	297, 352, 407, 408, 408, 786, 789, 801
\etex_glueexpr:D 249,	\etex_unless:D 250, 260, 262
378, 379, 379, 379, 380, 380, 380, 755	

<code>\etex_widowpenalties:D</code>	250	275, 275, 276, 276, 276, 282, 282,
<code>\eTeXrevision</code>	249	282, 283, 283, 284, 286, 288, 288,
<code>\eTeXversion</code>	249	288, 289, 289, 289, 290, 290, 290,
<code>\euc</code>	257	290, 290, 290, 290, 291, 291, 291,
<code>\everycr</code>	243	291, 291, 292, 292, 292, 292, 292,
<code>\everydisplay</code>	243	292, 292, 292, 292, 292, 292, 292,
<code>\everyeof</code>	249	292, 292, 292, 292, 292, 292, 292,
<code>\everyhbox</code>	243	292, 293, 293, 293, 293, 293, 293,
<code>\everyjob</code>	236, 236, 237, 237, 243	293, 293, 293, 293, 293, 293, 293,
<code>\everymath</code>	243	293, 293, 293, 293, 293, 293, 293,
<code>\everypar</code>	243	293, 293, 293, 294, 294, 294, 294,
<code>\everyvbox</code>	243	294, 294, 294, 294, 294, 294, 294,
<code>ex</code>	208	294, 294, 294, 294, 295, 295, 295,
<code>\exhyphenpenalty</code>	243	295, 295, 295, 296, 296, 296, 296,
<code>exp</code>	204	296, 296, 296, 296, 296, 296, 296,
<code>exp</code> commands:		296, 296, 296, 296, 296, 296, 296,
<code>\exp:w</code>	34,	296, 296, 297, 297, 297, 297, 297,
34, 34, 35, 35, 35, 35, 35, 35, 262,		297, 297, 297, 298, 298, 298, 298,
267, 267, 267, 274, 290, 290, 290,		299, 299, 300, 301, 302, 305, 311,
290, 290, 290, 291, 292, 292, 293,		311, 311, 311, 311, 315, 319, 319,
293, 293, 293, 293, 293, 293, 294,		319, 320, 332, 332, 333, 333, 334,
295, 295, 296, 296, 296, 296, 296,		334, 335, 335, 336, 336, 336, 336,
296, 296, 297, 297, 297, 297, 297,		337, 338, 338, 338, 338, 338, 341,
298, 315, 315, 315, 342, 354, 354,		341, 341, 341, 341, 341, 341, 342,
354, 354, 372, 373, 373, 373, 373,		342, 342, 342, 342, 342, 342, 342,
400, 400, 400, 400, 405, 406, 407,		342, 344, 344, 345, 346, 346, 346,
412, 412, 417, 417, 417, 417, 417,		346, 346, 346, 347, 347, 347, 347,
417, 418, 418, 419, 420, 420, 421,		352, 352, 352, 353, 353, 356, 356,
521, 521, 578, 586, 593, 596, 596,		356, 361, 361, 361, 362, 362, 362,
596, 599, 599, 601, 601, 601, 601,		362, 363, 363, 364, 364, 364, 364,
603, 604, 604, 608, 609, 610, 610,		365, 366, 371, 371, 371, 371, 371,
611, 611, 611, 611, 612, 613, 613,		371, 372, 372, 373, 373, 375, 379,
613, 613, 614, 614, 614, 615, 615,		388, 390, 390, 392, 392, 392, 392,
616, 617, 617, 617, 619, 619, 619,		395, 395, 395, 395, 397, 397, 398,
620, 620, 621, 621, 621, 622, 623,		398, 398, 400, 400, 402, 406, 407,
623, 624, 624, 625, 625, 625, 627,		408, 408, 408, 408, 408, 408, 408,
627, 627, 627, 628, 628, 629, 629,		408, 409, 410, 410, 410, 410, 411,
630, 630, 632, 632, 632, 632, 632,		411, 411, 411, 411, 411, 411, 411,
632, 632, 632, 634, 634, 635, 635,		411, 412, 412, 412, 412, 412, 412,
636, 637, 638, 638, 638, 639, 640,		415, 418, 419, 419, 419, 419, 420,
640, 641, 641, 641, 641, 641, 642,		420, 420, 420, 420, 421, 421, 421,
644, 645, 645, 650, 650, 651, 651,		421, 421, 422, 422, 422, 422, 423,
651, 692, 712, 712, 713, 717, 722,		423, 424, 424, 424, 425, 425, 425,
729, 741, 741, 741, 750, 750, 751,		426, 426, 426, 431, 431, 431, 431,
751, 752, 753, 753, 754, 754, 757,		431, 431, 435, 436, 436, 437, 437,
786, 786, 789, 790, 790, 801, 802, 812		437, 437, 438, 439, 440, 440, 440,
<code>\exp_after:wN</code> 32, 32, 34, 35, 35, 262,		442, 442, 442, 449, 449, 450, 450,
262, 263, 263, 263, 263, 267, 267,		452, 452, 452, 458, 458, 458, 458,
267, 269, 270, 270, 270, 272, 272,		458, 463, 469, 469, 507, 521, 521,
272, 274, 274, 274, 274, 274, 275,		521, 521, 521, 521, 522, 524, 525,

- 713, 713, 714, 714, 714, 714, 714,
 714, 714, 715, 715, 715, 716, 717,
 717, 717, 717, 718, 719, 719, 719,
 719, 719, 719, 719, 719, 719, 719,
 719, 719, 719, 720, 720, 720, 720,
 720, 720, 720, 720, 720, 720, 720,
 720, 720, 721, 721, 721, 721, 721,
 721, 721, 721, 721, 721, 721, 721,
 721, 721, 722, 722, 722, 722, 722,
 722, 722, 722, 723, 723, 723, 723,
 723, 723, 727, 727, 727, 727, 727,
 727, 728, 728, 728, 728, 729, 729,
 729, 729, 729, 729, 729, 729, 729,
 729, 730, 734, 734, 734, 734, 734,
 735, 735, 735, 735, 735, 735, 735,
 735, 736, 736, 737, 737, 737, 737,
 737, 737, 738, 738, 739, 739, 739,
 741, 741, 741, 741, 742, 742, 742,
 742, 742, 743, 743, 744, 744, 745,
 745, 745, 745, 745, 745, 745, 747,
 747, 747, 749, 750, 750, 750, 750,
 750, 751, 751, 751, 751, 751, 751,
 752, 752, 752, 752, 752, 752, 752,
 752, 752, 752, 753, 753, 753, 753,
 753, 753, 753, 753, 753, 753, 753,
 754, 754, 754, 754, 755, 755, 755,
 755, 755, 755, 755, 755, 755, 757,
 757, 757, 782, 782, 782, 783, 783,
 784, 786, 787, 787, 787, 788, 788,
 789, 790, 791, 791, 793, 795, 795,
 796, 797, 798, 799, 799, 800, 800,
 801, 802, 802, 802, 803, 803, 805,
 805, 812, 812, 813, 813, 815, 815,
 815, 815, 815, 816, 816, 816, 816, 816
 \exp_arg:N 32
 __exp_arg_last_unbraced:nn
 295, 295, 295, 295, 295, 296
 __exp_arg_next:Nnn ... 289, 289, 289
 __exp_arg_next:nnn
 289, 289, 289, 290, 290, 290, 290
 \exp_args:cc
 263, 263, 270, 270, 271, 271, 292
 \exp_args:N<variant> 27
 \exp_args:Nc 29,
 29, 263, 263, 263, 263, 279, 280,
 280, 281, 282, 282, 283, 284, 285,
 286, 286, 286, 286, 286, 287, 292,
 394, 401, 414, 414, 547, 587, 597, 780
 \exp_args:Ncc 281, 282,
 282, 286, 287, 287, 287, 292, 292, 548
 \exp_args:Nccc 31, 292, 292
 \exp_args:Ncco 294, 294
 \exp_args:Nccx 295, 295
 \exp_args:Ncf 293, 293
 \exp_args:NcNc 294, 294
 \exp_args:NcNo 294, 294
 \exp_args:Ncnx 295, 295
 \exp_args:Nco 293, 293, 293
 \exp_args:Ncx 294, 295
 \exp_args:Nf 29, 29, 292,
 292, 354, 354, 354, 354, 358, 360,
 360, 360, 360, 361, 361, 364, 365,
 373, 373, 373, 373, 412, 413, 419,
 419, 421, 421, 421, 426, 439, 439,
 457, 459, 459, 460, 460, 522, 782,
 782, 787, 794, 795, 798, 798, 800, 803
 \exp_args:Nff 294, 294
 \exp_args:Nfo 294, 294, 459
 \exp_args:NNc 30, 30, 263, 281, 281,
 282, 284, 286, 286, 286, 286, 287,
 292, 292, 324, 357, 357, 522, 558, 562
 \exp_args:Nnc 294, 294
 \exp_args:NNf
 293, 293, 357, 558, 562, 727, 727
 \exp_args:Nnf 294, 294, 324
 \exp_args:Nnnc 31, 295, 295
 \exp_args:NNNo
 31, 31, 292, 292, 390, 788, 788
 \exp_args:NNno 295, 295
 \exp_args:Nnno 31, 295, 295
 \exp_args:NNNV 294, 294
 \exp_args:NNNx 31, 295, 295, 324
 \exp_args:NNnx 31, 31, 295, 295
 \exp_args:Nnnx 31, 295, 295
 \exp_args:NNo 26,
 26, 26, 30, 292, 292, 358, 463, 568
 \exp_args:Nno
 . 30, 294, 295, 324, 344, 372, 390,
 456, 588, 589, 589, 589, 589, 590, 788
 \exp_args:NNoo 31, 31, 295, 295
 \exp_args:NNox 295, 295
 \exp_args:Nnox 295, 295
 \exp_args:NNV 293, 293
 \exp_args:NNv 293, 293
 \exp_args:NnV 294, 295
 \exp_args:NNx 30, 30, 294, 295
 \exp_args:Nnx 30, 294, 295
 \exp_args:No 29, 29, 292,
 292, 358, 360, 360, 379, 390, 390,
 399, 399, 399, 399, 399, 399, 399,

- 401, 402, 402, 404, 404, 407, 407,
- 408, 408, 413, 419, 419, 421, 421,
- 423, 423, 424, 425, 430, 447, 452,
- 452, 452, 454, 454, 460, 460, 539,
- 539, 540, 541, 547, 564, 569, 780, 788
- \exp_args:Noc 30, 294, 295
- \exp_args:Nof 294, 295
- \exp_args:Noo 30, 294, 295
- \exp_args:Nooo 295, 295
- \exp_args:Noox 295, 295
- \exp_args:Nox 294, 295
- \exp_args:NV
 - .. 29, 29, 292, 293, 539, 539, 540, 541
- \exp_args:Nv 29, 29, 292, 292
- \exp_args:NVV 30, 293, 293
- \exp_args:Nx . 30, 30, 282, 294, 294,
- 506, 523, 524, 539, 539, 540, 541, 814
- \exp_args:Nxo 294, 295
- \exp_args:Nxx 294, 295
- \exp_end: 34, 34, 34, 34, 34,
- 34, 35, 262, 267, 270, 270, 270, 271,
- 271, 274, 290, 291, 291, 292, 297,
- 298, 316, 316, 316, 316, 316, 316,
- 316, 316, 316, 316, 316, 317, 401,
- 405, 406, 406, 412, 412, 412, 420,
- 420, 420, 521, 521, 632, 632, 787,
- 790, 812, 813, 813, 813, 813, 815, 815
- \exp_end_continue_f:nw 35, 35, 297, 298
- \exp_end_continue_f:w
 - 35, 35, 35, 35, 35, 35,
 - 290, 290, 290, 292, 293, 293, 295,
 - 296, 297, 297, 298, 342, 372, 578,
 - 586, 596, 599, 599, 603, 604, 604,
 - 608, 610, 611, 611, 611, 612, 613,
 - 614, 614, 627, 627, 629, 629, 632,
 - 632, 632, 632, 641, 641, 641, 641,
 - 644, 645, 645, 650, 651, 651, 651,
 - 692, 717, 722, 729, 750, 750, 751,
 - 751, 752, 753, 753, 754, 754, 757, 786
- __exp_eval_error_msg:w 291, 291, 292
- __exp_eval_register:c
 - 290, 291, 291, 292, 293, 296, 296, 297
- __exp_eval_register:N
 - . 290, 291, 291, 291, 293, 293, 293,
 - 293, 294, 295, 296, 296, 296, 296, 297
- \l__exp_internal_tl . 35, 265, 265,
- 265, 265, 265, 288, 290, 290, 296, 296
- __exp_last_two_unbraced:noN ...
 - 297, 297, 297
- \exp_last_two_unbraced:Noo
 - 32, 32, 297, 297, 488, 494, 494
- \exp_last_unbraced:Nco
 - 32, 296, 296, 456
- \exp_last_unbraced:NcV 296, 296
- \exp_last_unbraced:Nf
 - .. 32, 32, 296, 296, 360, 361, 392, 446
- \exp_last_unbraced:Nfo 296, 296
- \exp_last_unbraced:NNNo 296, 296
- \exp_last_unbraced:NnNo . 32, 296, 296
- \exp_last_unbraced:NNNV . 32, 296, 296
- \exp_last_unbraced:NNo 296,
- 296, 406, 455, 469, 493, 790, 792, 802
- \exp_last_unbraced:Nno
 - 32, 32, 296, 296, 783
- \exp_last_unbraced:NNV 296, 296
- \exp_last_unbraced:No
 - 296, 296, 460, 498, 498, 499, 500
- \exp_last_unbraced:Noo
 - 296, 296, 465, 468
- \exp_last_unbraced:NV 296, 296
- \exp_last_unbraced:Nv . 296, 296, 815
- \exp_last_unbraced:Nx 32, 32, 296, 296
- \exp_not:c
 - .. 33, 33, 297, 297, 302, 304, 324,
 - 343, 414, 509, 509, 509, 509, 509,
 - 509, 509, 509, 516, 516, 516, 516,
 - 516, 516, 516, 516, 521, 523, 527,
 - 528, 533, 533, 534, 534, 538, 653, 811
- \exp_not:f 33,
- 33, 297, 297, 431, 431, 757, 757, 757
- \exp_not:N 33, 33, 189,
- 209, 262, 262, 269, 271, 271, 274,
- 274, 274, 274, 274, 274, 274, 274,
- 274, 274, 274, 284, 284, 286, 286,
- 288, 291, 291, 291, 297, 299, 299,
- 299, 299, 299, 299, 299, 299, 299,
- 299, 299, 299, 299, 299, 299, 299,
- 299, 299, 299, 301, 301, 302, 302,
- 302, 304, 304, 304, 304, 304, 304,
- 304, 304, 304, 304, 304, 305, 324,
- 328, 329, 329, 329, 329, 329, 330,
- 330, 330, 330, 330, 331, 331, 331,
- 331, 331, 332, 332, 332, 332, 332,
- 332, 332, 332, 332, 332, 337, 337,
- 337, 337, 337, 337, 337, 337, 337,
- 337, 337, 337, 337, 338, 340, 340,
- 340, 341, 341, 341, 342, 342, 342,
- 342, 357, 364, 364, 376, 389, 390,
- 391, 391, 392, 395, 409, 409, 409,

- 409, 410, 410, 410, 410, 414, 426,
 430, 430, 441, 457, 457, 457, 457,
 466, 467, 474, 509, 516, 521, 526,
 526, 526, 526, 526, 526, 526, 526,
 526, 526, 526, 526, 526, 526, 526,
 533, 533, 534, 534, 535, 535, 538,
 548, 568, 571, 608, 608, 608, 608,
 608, 608, 608, 608, 608, 609, 609,
 609, 612, 612, 612, 612, 612, 612,
 612, 612, 612, 612, 614, 615, 615,
 616, 619, 620, 622, 623, 624, 624,
 624, 626, 626, 633, 633, 637, 638,
 638, 640, 754, 754, 757, 757, 757,
 757, 757, 757, 757, 784, 787,
 788, 788, 806, 806, 813, 814, 816, 816
\exp_not:n 33, 33, 105, 106,
 107, 122, 126, 126, 137, 137, 139,
 143, 189, 222, 224, 262, 262, 269,
 269, 271, 282, 288, 296, 302, 303,
 340, 340, 340, 340, 343, 343, 357,
 367, 384, 386, 386, 386, 386, 386,
 387, 387, 388, 388, 389, 392, 395,
 395, 395, 395, 395, 395, 395, 413,
 416, 416, 431, 431, 432, 434, 434,
 434, 434, 436, 438, 439, 441, 442,
 442, 445, 446, 446, 447, 449, 452,
 453, 457, 457, 458, 458, 459, 460,
 465, 465, 466, 466, 466, 467, 509,
 509, 512, 516, 516, 521, 523, 535,
 538, 548, 553, 560, 563, 564, 568,
 612, 653, 730, 806, 806, 820, 820, 820
\exp_not:o 33, 33,
 108, 277, 278, 278, 297, 297, 385,
 385, 385, 385, 386, 386, 386, 386,
 386, 386, 386, 386, 387, 387, 387,
 387, 387, 387, 387, 387, 387, 387,
 387, 387, 387, 387, 388, 389, 389,
 389, 389, 389, 390, 392, 395, 395,
 395, 404, 404, 404, 433, 446, 446,
 449, 452, 453, 453, 466, 467, 527,
 528, 528, 528, 542, 543, 547, 547, 814
\exp_not:V 33,
 33, 297, 297, 386, 387, 387, 387, 552
\exp_not:v 33, 33, 297, 297
\exp_stop_f:
 34, 34, 34, 35, 35, 35, 290,
 290, 290, 346, 346, 346, 353, 353,
 372, 407, 420, 421, 422, 422, 431,
 433, 434, 434, 522, 558, 562, 573,
 573, 584, 594, 595, 597, 609, 610,
 615, 616, 617, 617, 619, 620, 620,
 620, 621, 621, 622, 624, 625, 625,
 641, 646, 646, 646, 646, 646, 646,
 654, 654, 654, 656, 658, 659, 661,
 661, 676, 678, 683, 684, 684, 694,
 694, 699, 704, 704, 704, 711, 712,
 714, 717, 719, 719, 721, 722, 723,
 724, 725, 725, 726, 726, 727, 729,
 735, 743, 745, 745, 746, 748, 748,
 750, 752, 753, 753, 793, 795, 798,
 799, 800, 804, 806, 806, 810, 810,
 810, 810, 810, 810, 810, 810, 810,
 810, 810, 810, 810, 810, 810, 810,
 810, 810, 810, 810, 811, 811, 811,
 811, 811, 811, 811, 812, 812, 813, 815
\expandafter
 . 234, 234, 234, 234, 234, 235, 235,
 235, 235, 235, 236, 236, 236, 236,
 236, 236, 236, 236, 236, 237, 237,
 237, 237, 237, 237, 238, 239, 239, 243
\expanded 254
\expansionERROR 298
\ExplFileDate 7, 821, 821, 821, 821
\ExplFileDescription 7
\ExplFileName 7
\ExplFileVersion . . . 7, 821, 821, 821, 821
\ExplSyntaxOff . . . 4, 7, 7, 7, 7, 8, 239,
 239, 239, 239, 239, 240, 240, 240, 240
\ExplSyntaxOn 4, 7, 7, 7,
 7, 8, 239, 240, 240, 240, 240, 327, 391
- F**
- false** 208
false commands:
\c_false_bool 22,
 39, 269, 270, 271, 272, 273, 273,
 274, 274, 283, 283, 300, 300, 305,
 306, 306, 306, 307, 307, 310, 310,
 312, 312, 312, 314, 817, 818, 819, 819
\fam 243
\fi 234, 234, 235,
 236, 236, 236, 236, 237, 237, 237,
 237, 238, 238, 238, 238, 239, 239, 243
fi commands:
\fi: 23, 38, 77,
 77, 77, 93, 154, 154, 154, 190, 262,
 262, 263, 267, 269, 270, 270, 272,
 272, 272, 273, 274, 274, 275, 275,
 275, 275, 275, 276, 276, 276, 276,

278, 279, 279, 282, 283, 283, 283,
 286, 288, 291, 291, 292, 292, 299,
 300, 301, 301, 302, 302, 302, 302,
 303, 303, 303, 303, 303, 303, 303,
 303, 307, 307, 309, 311, 311, 311,
 317, 317, 317, 317, 317, 317, 317,
 317, 319, 319, 320, 320, 320, 329,
 329, 329, 329, 330, 330, 330, 330,
 330, 330, 331, 331, 331, 331, 332,
 332, 333, 333, 334, 334, 334, 334,
 335, 335, 335, 335, 336, 336, 338,
 338, 338, 338, 341, 341, 342, 342,
 342, 346, 346, 347, 347, 347, 348,
 349, 351, 351, 351, 352, 353, 353,
 354, 354, 355, 361, 362, 362, 364,
 364, 371, 371, 372, 372, 373, 373,
 379, 388, 395, 395, 395, 397, 397,
 397, 397, 398, 398, 398, 399, 399,
 400, 406, 408, 408, 408, 408, 409,
 410, 410, 410, 411, 411, 411, 411,
 412, 412, 412, 416, 416, 416, 417,
 418, 419, 419, 420, 420, 420, 422,
 422, 422, 422, 422, 422, 422, 423,
 424, 425, 425, 433, 433, 434, 435,
 436, 436, 437, 438, 449, 449, 450,
 450, 468, 469, 473, 473, 473, 504,
 560, 560, 576, 577, 577, 577, 577,
 577, 577, 577, 581, 583, 583, 583,
 583, 583, 583, 583, 583, 583, 583,
 583, 584, 584, 584, 584, 584, 584,
 585, 585, 585, 585, 587, 590, 593,
 593, 594, 594, 594, 594, 594, 594,
 594, 594, 594, 595, 595, 595, 595,
 595, 595, 595, 595, 595, 595, 595,
 596, 596, 596, 597, 597, 598, 598,
 598, 598, 599, 599, 599, 608, 608,
 608, 609, 609, 610, 610, 610, 613,
 613, 614, 614, 614, 614, 615, 615,
 615, 615, 615, 615, 615, 616, 616,
 616, 616, 617, 618, 618, 618, 618,
 618, 619, 619, 620, 620, 621, 621,
 621, 622, 622, 623, 623, 624, 624,
 624, 624, 624, 624, 625, 625, 625,
 626, 626, 626, 626, 627, 628, 631,
 631, 631, 633, 633, 633, 633, 634,
 634, 635, 635, 636, 637, 637, 637,
 638, 638, 638, 639, 640, 640, 640,
 640, 640, 641, 641, 641, 641, 645,
 645, 645, 645, 645, 646, 646, 646,
 646, 646, 646, 646, 646, 646, 646,

646, 648, 649, 649, 649, 649, 649,
 649, 649, 649, 649, 649, 649, 649,
 649, 649, 649, 650, 650, 650, 650,
 651, 651, 651, 651, 652, 652, 652,
 652, 654, 654, 654, 654, 655, 655,
 655, 656, 657, 657, 658, 659, 659,
 660, 661, 661, 661, 661, 661, 661,
 662, 663, 664, 664, 664, 664, 664,
 665, 672, 674, 674, 674, 675, 676,
 676, 676, 676, 678, 680, 680, 680,
 681, 682, 682, 682, 682, 682, 683,
 683, 683, 683, 683, 683, 683, 683,
 683, 683, 683, 684, 693, 693, 693,
 694, 694, 694, 694, 694, 694, 699,
 702, 702, 702, 704, 704, 704, 705,
 705, 708, 709, 710, 710, 710, 711,
 711, 712, 712, 712, 712, 712, 713,
 713, 713, 713, 714, 714, 714, 714,
 715, 715, 715, 716, 716, 717, 717,
 718, 718, 718, 718, 719, 719, 719,
 719, 719, 719, 720, 720, 721, 721,
 721, 721, 722, 722, 722, 723, 723,
 723, 723, 723, 723, 724, 725, 725,
 726, 726, 727, 727, 727, 728, 729,
 735, 737, 737, 737, 738, 738, 739,
 741, 741, 741, 741, 741, 741, 741,
 742, 742, 743, 743, 743, 744, 744,
 744, 745, 745, 746, 746, 747, 747,
 748, 748, 748, 750, 751, 751, 751,
 752, 753, 753, 753, 754, 754, 755,
 755, 783, 812, 812, 812, 812, 812,
 812, 812, 812, 812, 812, 812, 814,
 815, 815, 816, 816, 816, 816, 816, 816

fifteen commands:

\c_fifteen
 76, 325, 326, 367, 368, 630, 636

file commands:

\file_... 184
 __file_add_path:nN ... 553, 553, 553
 \file_add_path:nN
 184, 184, 191, 553, 553, 554, 557, 558
 __file_add_path_search:nN
 553, 553, 553
 \g_file_current_name_tl 184, 505,
551, 551, 551, 551, 551, 555, 555, 555
 \file_if_exist:n 554
 \file_if_exist:n(TF) 190
 __file_if_exist:nT
 190, 554, 554, 554, 788, 788
 \file_if_exist:nT 779, 779

- \file_if_exist:nTF 184, 184, 184, 184, 554, 554, 779, 779
- \file_if_exist_input:n . 217, 217, 779
- \file_if_exist_input:nF 779
- \file_if_exist_input:nT 779
- \file_if_exist_input:nTF 217, 217, 779, 779
- _file_input:n 554, 555
- \file_input:n 184, 184, 184, 185, 217, 217, 554, 554
- _file_input:n_file_input:V 554
- _file_input:V 554, 779, 779, 779, 779
- _file_input_aux:n 554, 555, 555, 555
- _file_input_aux:o 554, 555
- \g__file_internal_iior 190, 553, 553, 553, 553, 553, 561, 561
- \l__file_internal_name_tl 190, 551, 551, 551, 552, 552, 552, 552, 552, 553, 553, 554, 554, 554, 557, 558, 558, 558, 558, 558, 779, 779, 779, 779, 788, 788
- \l__file_internal_seq 552, 552, 553, 553, 555, 556, 556, 556
- \l__file_internal_tl 551, 551, 555, 555
- \file_list: 185, 185, 555, 555
- _file_name_sanitize:nn 190, 190, 552, 552, 553, 554, 555, 555, 557, 558, 562
- _file_name_sanitize_aux:n 552, 552, 553
- _file_path_include:n . 555, 555, 555
- \file_path_include:n 184, 185, 185, 217, 555, 555
- \file_path_remove:n 185, 185, 555, 555
- \g__file_record_seq 551, 551, 551, 554, 555, 555, 555, 556, 556
- \l__file_saved_search_path_seq .. 552, 552, 553, 554
- \l__file_search_path_seq 552, 552, 553, 553, 553, 553, 554, 555, 555, 555
- \g__file_stack_seq 551, 551, 554, 555, 555
- \finalhyphendemerits 243
- \firstmark 243
- \firstmarks 249
- five commands:
 - \c_five 76, 325, 326, 367, 367, 424, 594, 594, 594, 594, 636, 636, 674, 700, 712, 812
- \floatingpenalty 243
- floor 205
- \fmtname 239
- \font 243
- \fontchardp 249
- \fontcharht 249
- \fontcharic 249
- \fontcharwd 249
- \fontdimen 243
- \fontid 254
- \fontname 244
- \forcecjktoken 258
- \formatname 254
- four commands:
 - \c_four 76, 325, 326, 367, 367, 392, 424, 571, 571, 596, 636, 636, 692, 693, 701, 702, 705, 711, 729, 729, 729, 737, 741, 746, 748, 755
- fourteen commands:
 - \c_fourteen 76, 325, 326, 367, 368, 636
- fp commands:
 - \s__fp 572, 572, 572, 572, 573, 573, 574, 574, 574, 574, 574, 574, 575, 575, 575, 575, 575, 575, 575, 576, 576, 576, 577, 577, 577, 577, 577, 578, 578, 583, 583, 583, 583, 583, 583, 584, 584, 584, 584, 584, 584, 584, 590, 590, 597, 598, 598, 598, 606, 607, 610, 623, 623, 625, 626, 629, 644, 645, 645, 648, 648, 648, 648, 649, 649, 649, 649, 649, 649, 649, 649, 650, 650, 650, 650, 651, 653, 653, 653, 653, 654, 654, 654, 654, 654, 654, 654, 654, 655, 655, 655, 655, 655, 655, 655, 655, 656, 656, 658, 658, 663, 663, 664, 664, 664, 664, 664, 664, 667, 667, 667, 676, 676, 676, 683, 684, 704, 704, 704, 711, 712, 712, 717, 717, 718, 718, 718, 718, 718, 718, 718, 718, 719, 719, 719, 719, 719, 720, 722, 722, 722, 722, 722, 724, 724, 725, 725, 725, 725, 726, 726, 726, 726, 727, 727, 741, 741, 741, 742, 742, 742, 745, 746, 746, 746, 746, 747, 748, 748, 748, 748, 749, 749, 750, 751, 751, 751, 752, 752, 753, 754, 755, 755, 755
 - _fp_ 650, 650, 650
 - _fp_&o:ww 644, 650
 - \fp_(g)zero:N 193

fp*_o:ww	663	_fp_array_to_clist:n	598, 637, 637, 756, 756
fp+_o:ww	653, 653, 653, 653, 653, 653, 653, 683	_fp_array_to_clist_loop:Nw ...	756, 756, 757, 757
_fp_o:ww	653, 653, 653, 653	_fp_asec_o:w	748, 748
\s_fp...	572, 572, 573, 573, 573, 573, 573, 575, 575	_fp_asin_auxi_o:NnNww	747, 747, 747, 749
_fp..._o:ww	601	_fp_asin_auxi_o:nNww .	746, 746, 749
fp/_o:ww	663, 663, 666, 705	_fp_asin_isqrt:wn ...	747, 747, 747
fp&_o:ww	650	_fp_asin_normal_o:NfwNnnnw ...	746, 746, 746, 746
fp^_o:ww	717	_fp_asin_o:w	745, 745
\fp_abs:n	204, 208, 208, 756, 756, 756, 765, 767, 767, 777, 777	_fp_atan_auxi:ww .	743, 743, 743, 743
_fp_acos_o:w	745, 746, 746, 748	_fp_atan_auxii:w	743, 743, 744
_fp_acot_o:Nw ...	630, 630, 740, 740	_fp_atan_combine_aux:ww	744, 745, 745
_fp_acotii_o:Nww .	740, 740, 741, 741	_fp_atan_combine_o:NwwwwN ...	741, 742, 742, 742, 744, 745
_fp_acotii_o:ww	741	_fp_atan_dispatch_o:NNnNw	740, 740, 740, 740
_fp_acsc_normal_o:NfwNnw	748, 748, 748, 748, 749	_fp_atan_div:wnwnw	742, 743, 743, 743
_fp_acsc_o:w	748, 748	_fp_atan_inf_o:NNNw	741, 741, 741, 741, 741, 742, 746, 748
\fp_add:cn	758	_fp_atan_near:wwn ...	743, 743, 743
\fp_add:Nn .	194, 194, 756, 758, 758, 758	_fp_atan_near_aux:wwn	743, 743, 743
_fp_add:NNNn	758, 758, 758, 758, 758, 758	_fp_atan_normal_o:NNnNnw	741, 741, 742, 742
_fp_add_big_i:wNww	656	_fp_atan_o:Nw ...	630, 630, 740, 740
_fp_add_big_i_o:wNww	653, 656, 656, 656, 656, 712	_fp_atan_Taylor_break:w	744, 744, 744
_fp_add_big_ii:wNww	656	_fp_atan_Taylor_loop:www	743, 744, 744, 744, 744
_fp_add_big_ii_o:wNww	656, 656, 656	_fp_atan_test_o:NwwNwwN	742, 742, 742, 747, 747
_fp_add_inf_o:Nww ...	654, 655, 655	_fp_atanii_o:Nww	740, 740, 741, 741, 741
_fp_add_normal_o:Nww	654, 655, 655, 655	_fp_basics_pack_high:NNNNnw ...	652, 652, 657, 657, 662, 666, 666, 675, 675, 683, 702
_fp_add_npos_o:NnwNnw	655, 656, 656, 656	_fp_basics_pack_high_carry:w ...	652, 652, 652, 652
_fp_add_return_ii_o:Nww	654, 654, 654, 654	_fp_basics_pack_low:NNNNnw ...	652, 652, 657, 662, 665, 666, 666, 675, 675, 681, 681, 683, 702
_fp_add_significand_carry_o:wwwNN	657, 658, 658, 658	_fp_basics_pack_weird_high:NNNNNNnw	211, 652, 652, 658, 676
_fp_add_significand_no_carry_o:wwwNN	657, 657, 657, 657	_fp_basics_pack_weird_low:NNNNw	211, 652, 652, 658, 676
_fp_add_significand_o:NnnwnnnN	656, 656, 656, 657, 657, 657		
_fp_add_significand_pack:NNNNNN	657, 657, 657		
_fp_add_significand_test_o:N ...	657, 657, 657		
_fp_add_zeros_o:Nww .	654, 654, 654		
_fp_and_return:wNw ..	650, 650, 650		
_fp_array_count:n	585, 585, 596, 740		
_fp_array_count_loop:Nw	585, 585, 585, 585		

\c__fp_big_leading_shift_int ...
 580, 580, 679, 690, 690, 691
 \c__fp_big_middle_shift_int ...
 . 580, 580, 679, 679, 679, 679, 679,
 679, 679, 690, 691, 691, 691, 691, 691
 \c__fp_big_trailing_shift_int ...
 580, 580, 679, 692
 \c__fp_Bigg_leading_shift_int ...
 580, 580, 672, 672
 \c__fp_Bigg_middle_shift_int ...
 580, 580, 672, 672, 672, 672
 \c__fp_Bigg_trailing_shift_int ..
 580, 580, 672, 672
 __fp_case_return:nw 583,
 583, 584, 584, 584, 599, 711, 741,
 741, 741, 750, 752, 753, 753, 753, 755
 __fp_case_return_i_o:ww
 584, 584, 654, 654, 655, 664, 741
 __fp_case_return_ii_o:ww
 584, 584, 664, 719, 719, 741
 __fp_case_return_o:Nw
 583, 583, 583, 712, 712, 712,
 717, 717, 718, 718, 725, 726, 748, 748
 __fp_case_return_o:Nww 583,
 583, 664, 664, 664, 664, 719, 719, 719
 __fp_case_return_same_o:w
 583, 583, 584, 676, 676, 704,
 712, 718, 724, 724, 725, 725, 726,
 726, 726, 727, 746, 746, 746, 748, 748
 __fp_case_use:nw .. 583, 583, 655,
 664, 664, 664, 664, 667, 667, 676,
 704, 704, 718, 724, 724, 725, 725,
 725, 725, 726, 726, 726, 726, 727,
 727, 746, 746, 746, 746, 746, 748,
 748, 748, 748, 748, 750, 750, 752, 752
 __fp_chk:w
 . 572, 572, 572, 573, 573, 574, 574,
 574, 574, 574, 574, 574, 575, 575,
 575, 575, 575, 575, 575, 575, 576,
 576, 576, 577, 577, 577, 577, 577,
 578, 578, 584, 590, 590, 597, 598,
 598, 598, 623, 623, 629, 644, 645,
 645, 648, 648, 648, 648, 649, 649,
 649, 649, 649, 650, 650, 650, 651,
 653, 654, 654, 654, 654, 654, 654,
 654, 654, 655, 655, 655, 655, 655,
 655, 655, 656, 656, 658, 658, 663,
 663, 664, 664, 664, 664, 664, 664,
 667, 667, 667, 667, 676, 676, 676,
 683, 684, 704, 704, 704, 711, 712,
 712, 717, 717, 718, 718, 718, 718,
 718, 718, 718, 718, 719, 719, 719,
 719, 719, 720, 722, 722, 722, 722,
 722, 724, 724, 725, 725, 725, 725,
 726, 726, 726, 726, 726, 727, 727,
 741, 741, 741, 741, 742, 742, 742,
 745, 746, 746, 746, 746, 747, 748,
 748, 748, 748, 749, 749, 750, 751,
 751, 751, 752, 752, 753, 754, 755, 755
 \fp_compare:n 644
 \fp_compare:nF 647, 647
 \fp_compare:nNn 645
 \fp_compare:nNnF 647, 648
 \fp_compare:nNnT ... 647, 648, 777, 825
 \fp_compare:nNnTF
 196, 196, 197, 198, 198, 198, 489,
 645, 762, 762, 762, 768, 768, 825, 826
 \fp_compare:nT 647, 647
 \fp_compare:nTF
 197, 197, 198, 198, 198, 198, 204, 644
 __fp_compare:wNNNNw 639
 __fp_compare_aux:wn .. 645, 645, 645
 __fp_compare_back:ww
 641, 645, 645, 645, 645, 645, 649
 __fp_compare_nan:w
 645, 645, 645, 645, 646
 __fp_compare_npos:nwnw
 644, 645, 645, 646, 646, 646, 658, 694
 \fp_compare_p:n 197, 197, 644
 \fp_compare_p:nNn 196, 196, 645
 __fp_compare_return:w . 644, 644, 644
 __fp_compare_significand:nnnnnnn
 646, 646, 646
 \fp_const:cn 757
 \fp_const:Nn 193,
 193, 757, 757, 757, 759, 759, 759, 759
 __fp_cos_o:w 724, 725
 __fp_cot_o:w 726, 726, 726
 __fp_cot_zero_o:Nfw
 725, 725, 726, 726, 727, 727
 __fp_csc_o:w 725, 725
 __fp_decimate:nNnnnn .. 581, 581,
 584, 598, 656, 656, 659, 713, 713, 752
 __fp_decimate_:Nnnnn 581, 581
 __fp_decimate_auxi:Nnnnn 581
 __fp_decimate_auxii:Nnnnn 581
 __fp_decimate_auxiii:Nnnnn ... 581
 __fp_decimate_auxiv:Nnnnn 581
 __fp_decimate_auxix:Nnnnn 581
 __fp_decimate_auxv:Nnnnn 581

_fp_decimate_auxvi:Nnnnn [581](#)
 _fp_decimate_auxvii:Nnnnn ... [581](#)
 _fp_decimate_auxviii:Nnnnn .. [581](#)
 _fp_decimate_auxx:Nnnnn [581](#)
 _fp_decimate_auxxi:Nnnnn [581](#)
 _fp_decimate_auxxii:Nnnnn ... [581](#)
 _fp_decimate_auxxiii:Nnnnn .. [581](#)
 _fp_decimate_auxxiv:Nnnnn ... [581](#)
 _fp_decimate_auxxv:Nnnnn [581](#)
 _fp_decimate_auxxvi:Nnnnn ... [581](#)
 _fp_decimate_pack:nnnnnnnnnw .
 [581](#), [582](#), [582](#), [582](#)
 _fp_decimate_pack:nnnnnnnw [582](#), [582](#)
 _fp_decimate_tiny:Nnnnn .. [581](#), [581](#)
 _fp_div_npos_o:Nww
 [666](#), [667](#), [667](#), [667](#), [667](#)
 _fp_div_significand_calc:wwnnnnnn
[671](#), [671](#), [671](#), [671](#), [672](#), [673](#), [707](#), [707](#)
 _fp_div_significand_calc_
 i:wwnnnnnn [671](#), [672](#), [672](#)
 _fp_div_significand_calc_
 ii:wwnnnnnn [671](#), [672](#), [672](#)
 _fp_div_significand_i_o:wnnw ..
 [667](#), [667](#), [671](#), [671](#), [671](#)
 _fp_div_significand_ii:wnw ...
 [671](#), [671](#), [671](#), [673](#), [673](#)
 _fp_div_significand_iii:wwnnnn
 [671](#), [673](#), [673](#), [673](#)
 _fp_div_significand_iv:wwnnnnnn
 [673](#), [673](#), [673](#), [674](#)
 _fp_div_significand_large_
 o:wwNNNNwN [675](#), [675](#), [675](#), [675](#)
 _fp_div_significand_pack:NNN ..
 [673](#),
[674](#), [674](#), [674](#), [674](#), [675](#), [707](#), [707](#),
[707](#), [707](#), [707](#), [707](#), [707](#), [707](#), [707](#)
 _fp_div_significand_small_
 o:wwNNNNwN [675](#), [675](#), [675](#), [675](#)
 _fp_div_significand_test_o:w ..
 [671](#), [674](#), [675](#), [675](#), [675](#)
 _fp_div_significand_v:NN
 [674](#), [674](#), [674](#)
 _fp_div_significand_v:NNw ... [673](#)
 _fp_div_significand_vi:Nw
 [673](#), [674](#), [674](#), [674](#)
 \s_fp_division [575](#), [575](#)
 \l_fp_division_by_zero_flag_
 token [587](#), [587](#)
 _fp_division_by_zero_o:Nnw ...
 [588](#), [589](#), [591](#), [591](#), [704](#), [727](#), [727](#)
 _fp_division_by_zero_o:NNww ...
 [588](#), [589](#), [591](#), [591](#), [667](#), [667](#), [718](#)
 \fp_do_until:nn [198](#), [198](#), [647](#), [647](#), [647](#)
 \fp_do_until:nNnn
 [197](#), [197](#), [647](#), [647](#), [647](#)
 \fp_do_while:nn [198](#), [198](#), [647](#), [647](#), [647](#)
 \fp_do_while:nNnn
 [198](#), [198](#), [647](#), [647](#), [647](#)
 _fp_ep_compare:www .. [694](#), [694](#), [743](#)
 _fp_ep_compare_aux:www
 [694](#), [694](#), [694](#)
 _fp_ep_div:wwwn [696](#),
[696](#), [701](#), [738](#), [739](#), [743](#), [743](#), [743](#), [749](#)
 _fp_ep_div_eps_pack:NNNNnw ...
 [697](#), [698](#), [698](#), [698](#)
 _fp_ep_div_epsilon:wnNNNNn [697](#)
 _fp_ep_div_epsilon:wnNNNNnn
 [697](#), [697](#), [698](#)
 _fp_ep_div_epsilonii:wnNNNNnn ...
 [697](#), [698](#), [698](#)
 _fp_ep_div_esti:wwwn
 [696](#), [697](#), [697](#), [697](#)
 _fp_ep_div_estii:wwnnwn
 [697](#), [697](#), [697](#)
 _fp_ep_div_estiii:NNNNwwwn ...
 [697](#), [697](#), [697](#)
 _fp_ep_inv_to_float:wN [727](#)
 _fp_ep_inv_to_float:wwN
 [700](#), [701](#), [701](#), [725](#), [726](#), [736](#)
 _fp_ep_isqrt:wn [698](#), [699](#), [747](#)
 _fp_ep_isqrt_aux:wn [698](#)
 _fp_ep_isqrt_auxi:wn [699](#), [699](#)
 _fp_ep_isqrt_auxii:wwnnwn ...
 [698](#), [699](#), [699](#)
 _fp_ep_isqrt_epsilon:wN
 [699](#), [700](#), [700](#), [700](#)
 _fp_ep_isqrt_epsilonii:wwN
 [700](#), [700](#), [700](#), [700](#), [700](#)
 _fp_ep_isqrt_esti:wwnnwn
 [699](#), [699](#), [699](#), [699](#)
 _fp_ep_isqrt_estii:wwnnwn ...
 [699](#), [699](#), [700](#)
 _fp_ep_isqrt_estiii:NNNNwwwn .
 [699](#), [700](#), [700](#)
 _fp_ep_mul:wwwn
 [694](#), [694](#), [737](#), [738](#), [747](#), [747](#)
 _fp_ep_mul_raw:wwwN
 [694](#), [694](#), [694](#), [728](#), [736](#)
 _fp_ep_to_ep:wwN
 [693](#), [693](#), [694](#), [694](#), [696](#), [697](#), [699](#), [747](#)

```

__fp_ep_to_ep_end:ww . 693, 693, 693, 693
__fp_ep_to_ep_loop:N . . . . . 715, 715, 715, 715, 715, 715,
. . . . 693, 693, 693, 693, 693, 735, 736 715, 715, 715, 715, 715, 715, 715,
__fp_ep_to_ep_zero:ww . 693, 694, 694 715, 715, 715, 715, 715, 715, 716,
__fp_ep_to_fixed:wwn . . . . . 716, 716, 716, 716, 716, 716, 716,
. . . . . 692, 692, 728, 743, 743, 747 716, 716, 716, 716, 716, 716, 716
__fp_ep_to_fixed_auxi:www . . . . .
. . . . . 692, 692, 693
__fp_ep_to_fixed_auxii:nnnnnnwnn
. . . . . 692, 693, 693
__fp_ep_to_float:wN . . . . . 727
__fp_ep_to_float:wwN . . . . . 700,
700, 701, 701, 724, 724, 725, 736, 739
__fp_error:nffn . . . . .
. . . . 589, 589, 590, 590, 591, 597, 637
__fp_error:nnfn . . . . . 589, 589, 591
__fp_error:nnnn . . . 591, 591, 591, 597
\fp_eval:n . . . . . 194, 194, 197,
203, 203, 203, 204, 204, 204, 204,
204, 204, 204, 204, 204, 204, 204,
204, 205, 205, 205, 205, 205, 205,
206, 206, 206, 206, 206, 206, 206,
206, 206, 206, 206, 206, 206, 206,
206, 206, 207, 207, 207, 207, 207,
207, 207, 207, 207, 207, 207, 207,
208, 208, 209, 756, 756, 758, 826, 826
\s__fp_exact . . . . .
575, 575, 575, 575, 575, 575, 575, 648
__fp_exp_after_?_f:nw . . . . 610, 610
__fp_exp_after_array_f:w . . . .
578, 578, 578, 628, 650, 651, 651, 651
__fp_exp_after_f:nw . . . . .
. . . . . 577, 577, 610, 629, 633
__fp_exp_after_mark_f:nw . . . .
. . . . . 610, 610, 610
__fp_exp_after_normal:nnNw . . .
. . . . . 577, 577, 577, 578, 578
__fp_exp_after_normal:Nwwww . .
. . . . . 578, 578
__fp_exp_after_o:nw . . . . . 577, 577
__fp_exp_after_o:w . . . . . 577,
577, 577, 583, 584, 584, 598, 599,
599, 641, 649, 650, 654, 684, 722, 722
__fp_exp_after_special:nnNw . .
. . . . . 577, 577, 577, 578, 578
__fp_exp_after_stop_f:nw . . 578, 578
__fp_exp_large:w . . . . .
. . . . 714, 714, 714, 714, 714, 714,
714, 714, 714, 714, 715, 715,
715, 715, 715, 715, 715, 715,
__fp_exp_large:wN . . . 714, 716, 716
__fp_exp_large_after:wwn . . . .
. . . . . 714, 716, 716
__fp_exp_large_i:wN . . 714, 715, 715
__fp_exp_large_ii:wN . 714, 715, 715
__fp_exp_large_iii:wN . 714, 715, 715
__fp_exp_large_iv:wN . 714, 714, 714
__fp_exp_large_v:wN . . 714, 714, 721
__fp_exp_normal:w . . . . 712, 712, 712
__fp_exp_o:w . . . . . 711, 711
__fp_exp_overflow: . . . . . 712, 713
__fp_exp_pos:NNwnw . . . 712, 712, 712
__fp_exp_pos:Nwnnw . . . . . 712
__fp_exp_pos_large:NnnNwn . . . .
. . . . . 713, 714, 714
__fp_exp_Taylor:Nnnwn . . . . .
. . . . . 713, 713, 713, 716
__fp_exp_Taylor_break:Nww . . . .
. . . . . 713, 713, 714
__fp_exp_Taylor_ii:ww . . . . 713, 713
__fp_exp_Taylor_loop:www . . . .
. . . . . 713, 713, 713, 713
__fp_expand:n . . . . 585, 585, 756, 756
__fp_expand_loop:nwnN . . . . .
. . . . . 585, 586, 586, 586
__fp_exponent:w . . . . . 576, 576
__fp_fixed_add:nnNnnwn 688, 688, 688
__fp_fixed_add:Nnnnnwnn . . . .
. . . . . 688, 688, 688, 688
__fp_fixed_add:wwn 684, 684, 688,
688, 698, 709, 710, 710, 711, 743, 745
__fp_fixed_add_after:NNNNNwn . .
. . . . . 688, 688, 688
__fp_fixed_add_one:wN . . . . .
. . . . . 685, 685, 698, 713, 714, 747
__fp_fixed_add_pack:NNNNNwn . .
. . . . . 688, 688, 688, 688
__fp_fixed_continue:wn . . . 685,
685, 694, 694, 696, 714, 714, 715,
715, 715, 716, 721, 729, 737, 747, 747
__fp_fixed_div_int:wnN . . . . .
. . . . . 686, 687, 687, 687
__fp_fixed_div_int:wwN . . . . .
. . . . . 686, 687, 709, 713, 744

```

```

\__fp_fixed_div_int_after:Nw ...
    ..... 686, 686, 687, 687
\__fp_fixed_div_int_auxi:wnn ...
    .... 686, 687, 687, 687, 687, 687, 687
\__fp_fixed_div_int_auxii:wnn ...
    ..... 686, 687, 687, 687
\__fp_fixed_div_int_pack:Nw ....
    686, 687, 687, 687, 687, 687, 687, 687
\__fp_fixed_div_myriad:wn .....
    ..... 685, 685, 698
\__fp_fixed_inv_to_float:wN ....
    ..... 701, 701, 712, 720
\__fp_fixed_mul:nnnnnnnw 688, 689, 689
\__fp_fixed_mul:wnn .....
    ..... 684, 686, 688, 689,
    695, 697, 698, 698, 698, 700, 700,
    701, 709, 710, 711, 713, 714, 716,
    720, 734, 736, 737, 738, 744, 745, 745
\__fp_fixed_mul_add:nnnnwnnnn ...
    ..... 691, 691, 692
\__fp_fixed_mul_add:nnnnwnnN ...
    ..... 692, 692, 692
\__fp_fixed_mul_add:Nwnnnwnnn ...
    ..... 690, 691, 691, 691, 691
\__fp_fixed_mul_add:wwwn ... 690, 690
\__fp_fixed_mul_after:wnn 685, 685,
    685, 686, 689, 689, 690, 690, 691, 720
\__fp_fixed_mul_one_minus_-
    mul:wnn ..... 690
\__fp_fixed_mul_short:wnn .....
    .... 686, 686, 697, 698, 700, 700, 745
\__fp_fixed_mul_sub_back:wwwn ...
    ..... 690, 690, 700,
    737, 737, 737, 737, 737, 737, 737,
    737, 737, 737, 737, 737, 737, 737,
    737, 737, 737, 738, 738, 738, 738,
    738, 738, 739, 739, 739, 739, 744, 744
\__fp_fixed_one_minus_mul:wnn ...
    ..... 690, 691, 691
\__fp_fixed_sub:wnn .... 688, 688,
    700, 710, 711, 711, 729, 743, 745, 747
\__fp_fixed_to_float:Nw 701, 701, 711
\__fp_fixed_to_float:wN .....
    ..... 685, 701, 701,
    701, 701, 711, 711, 712, 719, 745, 745
\__fp_fixed_to_float_pack:ww 702, 702
\__fp_fixed_to_float_rad:wN ....
    ..... 701, 701, 745
\__fp_fixed_to_float_round_-
    up:wnnnnw ..... 702, 702
\__fp_fixed_to_float_zero:w 702, 702
\__fp_fixed_to_loop:N . 701, 701, 702
\__fp_fixed_to_loop_end:w .. 702, 702
\fp_flag_off:n .... 200, 200, 587, 587
\fp_flag_on:n ..... 200, 200,
    587, 587, 589, 589, 589, 589, 590, 590
\fp_format:nn ..... 209
\__fp_from_dim:wNnnnnnn 755, 755, 755
\__fp_from_dim:wnnnnwNn .... 755, 755
\__fp_from_dim:wnnnnwNw ..... 755
\__fp_from_dim:wNw .... 755, 755, 755
\__fp_from_dim_test:ww .....
    ..... 612, 629, 755, 755, 755, 755
\fp_function:Nw ..... 641, 641
\__fp_function_apply:nw .....
    .... 641, 641, 642, 642, 642, 642, 643
\__fp_function_args:Nwn .....
    ..... 642, 642, 642, 642
\__fp_function_store:wwNwnn ....
    ..... 642, 643, 643, 643, 643
\__fp_function_store_end:wnnn ...
    ..... 642, 643, 643, 643
\fp_gadd:cn ..... 758
\fp_gadd:Nn ..... 194, 758, 758, 758
.fp_gset:c ..... 174, 540
\fp_gset:cn ..... 757
.fp_gset:N ..... 174, 540
\fp_gset:Nn 194, 757, 757, 757, 758, 758
\fp_gset_eq:cc ..... 757
\fp_gset_eq:cN ..... 757
\fp_gset_eq:Nc ..... 757
\fp_gset_eq:NN 194, 757, 757, 757, 758
\fp_gsub:cn ..... 758
\fp_gsub:Nn ..... 194, 758, 758, 758
\fp_gzero:c ..... 758
\fp_gzero:N ... 193, 758, 758, 758, 758
\fp_gzero_new:c ..... 758
\fp_gzero_new:N ... 193, 758, 758, 758
\fp_if_exist:c ..... 644
\fp_if_exist:cTF ..... 644
\fp_if_exist:N ..... 644
\fp_if_exist:NTF .....
    ..... 196, 196, 644, 758, 758, 759
\fp_if_exist_p:c ..... 644
\fp_if_exist_p:N ..... 196, 196, 644
\fp_if_flag_on:n ..... 587
\fp_if_flag_on:nTF .... 200, 200, 587
\fp_if_flag_on_p:n .... 200, 200, 587
\fp_if_nan:nTF ..... 209
\__fp_inf_fp:N ..... 575, 575, 590

```

```

\s__fp_invalid ..... 575, 575
\__fp_invalid_operation:nnw ....
..... 587, 588,
588, 591, 591, 591, 750, 750, 752, 752
\l__fp_invalid_operation_flag_-
token ..... 587, 587
\__fp_invalid_operation_o:fw ...
..... 591, 724, 725, 725, 726,
726, 727, 746, 746, 747, 748, 748, 749
\__fp_invalid_operation_o:nw ...
..... 588, 591, 591, 591, 676, 704
\__fp_invalid_operation_o:Nww ...
..... 587,
589, 591, 591, 655, 655, 664, 664, 722
\__fp_invalid_operation_tl_o:ff .
..... 588, 589, 591, 591, 598
\c__fp_leading_shift_int .....
579, 580, 685, 686, 689, 720, 734, 735
\__fp_ln_c:NwNn ..... 709
\__fp_ln_c:NwNw ... 709, 710, 710, 710
\__fp_ln_div_after:Nw .....
..... 705, 707, 707, 708
\__fp_ln_div_i:w ..... 707, 707
\__fp_ln_div_ii:wn .....
..... 707, 707, 707, 707, 707
\__fp_ln_div_vi:wn ..... 707, 707
\__fp_ln_exponent:wn 704, 710, 710, 710
\__fp_ln_exponent_one:ww ... 711, 711
\__fp_ln_exponent_small:NNww ...
..... 711, 711, 711
\c__fp_ln_i_fixed_tl ..... 703, 703
\c__fp_ln_ii_fixed_tl ..... 703, 703
\c__fp_ln_iii_fixed_tl ..... 703, 703
\c__fp_ln_iv_fixed_tl ..... 703, 703
\c__fp_ln_ix_fixed_tl ..... 703, 703
\__fp_ln_npos_o:w .....
..... 703, 703, 704, 704, 704
\__fp_ln_o:w ..... 703, 703, 704, 719
\__fp_ln_significand:NNNNnnN ...
..... 704, 704, 704, 704, 720
\__fp_ln_square_t_after:w .. 708, 709
\__fp_ln_square_t_pack:NNNNNw ...
..... 708, 708, 708, 708, 709
\__fp_ln_t_large:NNw 708, 708, 708, 708
\__fp_ln_t_small:Nw ..... 708, 708
\__fp_ln_t_small:w ..... 708
\__fp_ln_Taylor:wwNw 709, 709, 709, 709
\__fp_ln_Taylor_break:w .... 709, 710
\__fp_ln_Taylor_loop:www 709, 709, 710
\__fp_ln_twice_t_after:w ... 708, 709
\__fp_ln_twice_t_pack:Nw .....
..... 708, 708, 709, 709, 709, 709
\c__fp_ln_vi_fixed_tl ..... 703, 703
\c__fp_ln_vii_fixed_tl ..... 703, 703
\c__fp_ln_viii_fixed_tl .... 703, 703
\c__fp_ln_x_fixed_tl 703, 703, 711, 711
\__fp_ln_x_ii:wnnnn ... 704, 705, 705
\__fp_ln_x_iii:NNNNNNw ..... 705, 705
\__fp_ln_x_iii_var:NNNNNw .. 705, 705
\__fp_ln_x_iv:wnnnnnnnn 705, 706, 706
\fp_log:c ..... 781
\fp_log:N ..... 218, 218, 781, 781, 781
\fp_log:n ..... 218, 218, 781, 781
\s__fp_mark 575, 575, 585, 586, 586,
586, 586, 606, 606, 610, 632, 632,
633, 634, 643, 643, 643, 643, 643, 643
\fp_max:nn ..... 209, 209, 756, 756
\c__fp_max_exponent_int .....
. 573, 573, 575, 575, 576, 576, 577,
577, 694, 702, 712, 713, 721, 750, 752
\__fp_max_fp:N ..... 576, 576
\c__fp_middle_shift_int .....
..... 579, 580, 686,
686, 686, 686, 689, 689, 689, 689,
720, 720, 720, 720, 734, 735, 736, 736
\fp_min:nn ..... 209, 756, 756
\__fp_min_fp:N ..... 576, 576
\__fp_minmax_auxi:ww 649, 649, 649, 649
\__fp_minmax_auxii:ww .....
..... 649, 649, 649, 649
\__fp_minmax_break_o:w . 648, 649, 649
\__fp_minmax_loop:Nww .....
..... 648, 648, 648, 648, 648, 649
\__fp_minmax_o:Nw .....
..... 630, 630, 644, 648, 648
\__fp_mul_cases_o:NnNnw .....
..... 663, 663, 667, 667
\__fp_mul_cases_o:nNnnw ..... 663
\__fp_mul_npos_o:Nw ..... 663,
663, 664, 664, 666, 755, 755, 755
\__fp_mul_significand_drop:NNNNNw
.... 664, 664, 665, 665, 665, 665, 665
\__fp_mul_significand_keep:NNNNNw
..... 664, 665, 665, 665
\__fp_mul_significand_large_-
f:NwNNNN ..... 665, 666, 666
\__fp_mul_significand_o:nnnnNnnnn
..... 664, 664, 664, 664, 665
\__fp_mul_significand_small_-
f:NNwwwN ..... 665, 666, 666

```

__fp_mul_significand_test_f:NNN
 665, 665, 665, 665
 __fp_neg_sign:N ... 576, 576, 653, 653
 \fp_new:N 193,
 193, 193, 479, 479, 757, 757, 757,
 758, 758, 759, 759, 759, 759, 760,
 760, 760, 764, 764, 771, 771, 776, 776
 __fp_new_function:Ncfnn ... 642, 642
 __fp_new_function:NNnnn 642, 642, 642
 \fp_new_function:Npn 642, 642
 __fp_not_o:w 627, 644, 649
 \c__fp_one_fixed_tl 685,
 685, 709, 714, 721, 721, 742, 744, 747
 \s__fp_overflow 575, 575, 575
 __fp_overflow:w
 576, 577, 588, 590, 590, 591, 591
 \l__fp_overflow_flag_token . 587, 587
 __fp_pack:NNNNNw .. 579, 580, 685,
 686, 686, 686, 686, 686, 689, 689,
 689, 689, 689, 720, 720, 720, 720, 720
 __fp_pack_big:NNNNNNw .. 580, 580,
 679, 679, 679, 679, 679, 679, 679,
 679, 690, 690, 691, 691, 691, 691, 692
 __fp_pack_Bigg:NNNNNNw
 580, 580, 672, 672, 672, 672, 672, 672
 __fp_pack_eight:wNNNNNNNN
 580, 580, 660, 662, 677, 693, 729, 729
 __fp_pack_twice_four:wNNNNNNNN .
 580, 580, 599, 599, 660, 660,
 693, 693, 693, 694, 694, 694, 702,
 702, 713, 713, 713, 729, 729, 734, 755
 __fp_parse:n .. 600, 612, 632, 632,
 632, 643, 643, 644, 645, 645, 750,
 751, 753, 754, 756, 756, 756, 756,
 756, 756, 757, 757, 757, 757, 758, 758
 \fp_parse:n 611
 __fp_parse_after:ww .. 632, 632, 632
 __fp_parse_apply_binary:NwNwN ..
 603,
 604, 604, 604, 632, 632, 636, 636, 637
 __fp_parse_apply_compare:NwNNNNNwN
 640, 640
 __fp_parse_apply_compare_-
 aux:NNwN 641, 641, 641
 __fp_parse_apply_juxtapose:NwNwN
 636, 636, 637, 637
 __fp_parse_apply_unary:NNNwN ...
 627, 627, 627, 630, 630
 __fp_parse_compare:NNNNNNN
 639, 639, 639, 639, 639, 639, 640, 641
 __fp_parse_compare_auxi:NNNNNNN
 639, 640, 640, 640, 640
 __fp_parse_compare_auxii:NNNNN .
 639, 640, 640, 640, 640, 640
 __fp_parse_compare_end:NNNNw ...
 639, 640, 640
 __fp_parse_continue 632
 __fp_parse_continue:NwN
 603, 604, 604, 604, 604,
 632, 632, 632, 632, 641, 651, 651, 651
 __fp_parse_continue_compare:NNwNN
 641, 641
 __fp_parse_digits_:N . 608, 609, 609
 __fp_parse_digits_i:N 608, 609
 __fp_parse_digits_ii:N 608, 609
 __fp_parse_digits_iii:N ... 608, 609
 __fp_parse_digits_iv:N 608, 609
 __fp_parse_digits_v:N 608, 609
 __fp_parse_digits_vi:N
 608, 609, 617, 619
 __fp_parse_digits_vii:N
 608, 616, 617, 619
 __fp_parse_excl_error: 639, 639, 640
 __fp_parse_expand:w
 . 607, 608, 608, 608, 608, 609, 610,
 611, 613, 614, 615, 615, 616, 616,
 617, 617, 618, 619, 619, 620, 621,
 621, 622, 623, 623, 624, 624, 624,
 625, 625, 627, 628, 628, 630, 630,
 632, 634, 634, 635, 636, 637, 638,
 638, 638, 639, 640, 640, 641, 642, 650
 __fp_parse_exponent:N
 611, 616, 621, 621, 624, 624, 624
 __fp_parse_exponent:Nw
 617, 618, 619, 620, 622, 623, 624, 624
 __fp_parse_exponent_aux:N
 624, 624, 624
 __fp_parse_exponent_body:N
 624, 625, 625
 __fp_parse_exponent_digits:N ...
 625, 625, 625, 625
 __fp_parse_exponent_keep:N ... 626
 __fp_parse_exponent_keep:NTF ...
 625, 625
 __fp_parse_exponent_sign:N
 624, 624, 624, 624
 __fp_parse_function:NNN
 630, 630, 630, 630,
 630, 630, 630, 630, 631, 631, 631, 631

```

\__fp_parse_infix:NN .....
. 610, 610, 612, 613, 614, 628, 628,
629, 629, 629, 633, 633, 634, 634, 643
fp_parse_infix_
  \__fp_parse_infix_>:N ..... 639
\__fp_parse_infix_ . 628, 633, 634,
634, 635, 635, 635, 636, 636, 636,
636, 637, 637, 638, 638, 638, 638, 638
\__fp_parse_infix_&:Nw ..... 637
\__fp_parse_infix_(:N ..... 636
\__fp_parse_infix_):N ..... 634
\__fp_parse_infix_*:N ..... 637
\__fp_parse_infix_+:N . 608, 635, 642
\__fp_parse_infix_-:N ..... 635
\__fp_parse_infix_/:N ..... 635
\__fp_parse_infix_::N .....
..... 638, 638, 639, 650
\__fp_parse_infix_:N ..... 639
\__fp_parse_infix_<:N ..... 639
\__fp_parse_infix_?:N ..... 638
\__fp_parse_infix_^:N ..... 635
\__fp_parse_infix_after_operand:NwN
..... 611, 612, 612, 627, 633, 633
\__fp_parse_infix_and:N 635, 636, 638
\__fp_parse_infix_check:NNN 633, 633
\__fp_parse_infix_comma:w .. 635, 635
\__fp_parse_infix_comma_gobble:w
..... 635, 635
\__fp_parse_infix_end:N .....
.... 632, 632, 632, 634, 634, 634, 634
\__fp_parse_infix_juxtapose:N ...
.... 633, 633, 636, 636, 636, 636, 637
\__fp_parse_infix_mark:NNN .....
..... 633, 634, 634
\__fp_parse_infix_mul:N 635, 636, 637
\__fp_parse_infix_or:N . 635, 636, 638
\__fp_parse_large:N 615, 615, 618, 618
\__fp_parse_large_leading:wwNN ..
..... 618, 619, 619
\__fp_parse_large_round:NN .....
..... 619, 620, 622, 622
\__fp_parse_large_round_aux:wNN .
..... 622, 622, 623
\__fp_parse_large_round_test:NN .
..... 622, 622, 622
\__fp_parse_large_trailing:wwNN .
..... 619, 619, 620
\__fp_parse_letters:N .....
..... 613, 613, 613, 613, 613, 614
\__fp_parse_lparen_after:NwN ...
..... 628, 628, 628
\__fp_parse_one ..... 632
\__fp_parse_one:Nw ..... 603,
603, 604, 605, 605, 605, 606, 606,
608, 609, 609, 614, 614, 626, 628, 632
\__fp_parse_one_digit:NN .....
..... 610, 612, 612, 627
\__fp_parse_one_fp:NN .....
..... 609, 609, 610, 610
\__fp_parse_one_other:NN 610, 613, 613
\__fp_parse_one_register:NN ....
..... 610, 611, 611
\__fp_parse_one_register_aux:Nw .
..... 611, 611, 612
\__fp_parse_one_register_-
auxii:wwwNw ..... 611, 612, 612
\__fp_parse_one_register_dim:ww .
..... 611, 612, 612, 612
\__fp_parse_one_register_int:www
..... 611, 612, 612
\__fp_parse_one_register_mu:www .
..... 611, 612, 612
\__fp_parse_operand ..... 632
\__fp_parse_operand:Nw .. 603, 603,
603, 603, 605, 605, 605, 606, 606,
607, 627, 627, 628, 628, 630, 630,
632, 632, 632, 632, 635, 636, 637,
638, 639, 640, 641, 641, 642, 642, 650
\__fp_parse_pack_carry:w .....
..... 618, 618, 618, 618
\__fp_parse_pack_leading:NNNNNww
..... 617, 618, 618, 618, 619
\__fp_parse_pack_trailing:NNNNNNww
..... 617, 618, 618, 619, 620, 620
\__fp_parse_prefix:NNN . 613, 614, 614
\__fp_parse_prefix_ ..... 628
\__fp_parse_prefix_(:Nw ..... 628
\__fp_parse_prefix_+:Nw ..... 626
\__fp_parse_prefix_-:Nw ..... 627
\__fp_parse_prefix_:Nw ..... 627
\__fp_parse_prefix_:Nw ..... 627
\__fp_parse_prefix_unknown:NNN ..
..... 614, 614, 614
\__fp_parse_return_semicolon:w ..
..... 608,
608, 609, 614, 621, 621, 624, 625, 625
\__fp_parse_round:Nw .....
..... 631, 631, 631, 631, 631

```

_fp_parse_round_after:wN	621, 621, 621, 621, 622, 623
_fp_parse_round_loop:N	620, 620,	620, 621, 621, 621, 622, 622, 622, 623
_fp_parse_round_up:N	620, 621, 621, 621
_fp_parse_small:N	616, 616, 616, 616	
_fp_parse_small_leading:wwNN	..	617, 617, 617, 619
_fp_parse_small_round:NN	617, 621, 621, 622
_fp_parse_small_trailing:wwNN	..	617, 617, 617, 620
_fp_parse_strim_end:w	615, 616, 616	
_fp_parse_strim_zeros:N	615, 615, 615, 616, 616, 627, 628
_fp_parse_trim_end:w	615, 615, 615	
_fp_parse_trim_zeros:N	612, 615, 615, 615
_fp_parse_unary_function:nNN	..	630, 630, 630, 630, 630, 630, 631, 631
_fp_parse_word:Nw	613, 613, 613, 613	
_fp_parse_word_abs:N	630, 630
_fp_parse_word_acos:N	631
_fp_parse_word_acosd:N	631
_fp_parse_word_acot:N	...	630, 630
_fp_parse_word_acotd:N	...	630, 630
_fp_parse_word_acsc:N	631
_fp_parse_word_acscd:N	631
_fp_parse_word_asec:N	631
_fp_parse_word_asecd:N	631
_fp_parse_word_asin:N	631
_fp_parse_word_asind:N	631
_fp_parse_word_atan:N	...	630, 630
_fp_parse_word_atand:N	...	630, 630
_fp_parse_word_bp:N	629
_fp_parse_word_cc:N	629
_fp_parse_word_ceil:N	...	631, 631
_fp_parse_word_cm:N	629
_fp_parse_word_cos:N	631
_fp_parse_word_cosd:N	631
_fp_parse_word_cot:N	631
_fp_parse_word_cotd:N	631
_fp_parse_word_csc:N	631
_fp_parse_word_cscd:N	631
_fp_parse_word_dd:N	629
_fp_parse_word_deg:N	628
_fp_parse_word_em:N	629
_fp_parse_word_ex:N	629
_fp_parse_word_exp:N	630, 630
_fp_parse_word_false:N	628
_fp_parse_word_floor:N	...	631, 631
_fp_parse_word_in:N	629
_fp_parse_word_inf:N	628
_fp_parse_word_ln:N	630, 630
_fp_parse_word_max:N	630, 630
_fp_parse_word_min:N	630, 630
_fp_parse_word_mm:N	629
_fp_parse_word_nan:N	628
_fp_parse_word_nc:N	629
_fp_parse_word_nd:N	629
_fp_parse_word_pc:N	629
_fp_parse_word_pi:N	628
_fp_parse_word_pt:N	629
_fp_parse_word_round:N	...	631, 631
_fp_parse_word_sec:N	631
_fp_parse_word_secd:N	631
_fp_parse_word_sin:N	631
_fp_parse_word_sind:N	631
_fp_parse_word_sp:N	629
_fp_parse_word_sqrt:N	...	630, 630
_fp_parse_word_tan:N	631
_fp_parse_word_tand:N	631
_fp_parse_word_true:N	628
_fp_parse_word_trunc:N	...	631, 631
_fp_parse_zero:	615, 615, 616, 616, 616
_fp_pow_B:wwN	720, 721
_fp_pow_C_neg:w	721, 721
_fp_pow_C_overflow:w	...	721, 721, 721
_fp_pow_C_pack:w	...	721, 721, 721
_fp_pow_C_pos:w	721, 721
_fp_pow_C_pos_loop:wN	721, 721, 721	
_fp_pow_exponent:Nwnnnnnw	...	720, 720, 720
_fp_pow_exponent:wnN	720, 720
_fp_pow_neg:www	...	717, 721, 722, 722
_fp_pow_neg_aux:wNN	721, 722, 722, 722
_fp_pow_neg_case:w	...	722, 722, 722
_fp_pow_neg_case_aux:nnnnn	...	722, 722, 723
_fp_pow_neg_case_aux:NNNNNNNw	722, 723, 723, 723
_fp_pow_normal:ww	717, 717, 717, 718, 719
_fp_pow_npos:Nww	...	719, 719, 719
_fp_pow_npos:ww	718
_fp_pow_npos_aux:NNnw	719, 720, 720, 720

```

\__fp_pow_zero_or_inf:ww .....
    ..... 717, 718, 718, 718
\__fp_reverse_args:Nww .....
    ..... 574, 574, 739, 743, 746, 747, 748, 748
\__fp_round:NNN .....
    ..... 593, 593, 593, 595, 595, 595, 657,
    ..... 658, 666, 666, 666, 675, 676, 683, 683
\__fp_round:Nwn 596, 598, 598, 598, 754
\__fp_round:Nww ... 596, 597, 598, 598
\__fp_round:Nwww ..... 596, 597, 597
\__fp_round_digit:Nw ..... 581,
    ..... 582, 582, 582, 582, 595, 595, 658,
    ..... 662, 664, 666, 666, 666, 676, 683, 683
\__fp_round_name_from_cs:N .....
    ..... 597, 597, 597, 598
\__fp_round_neg:NNN .....
    ..... 593, 596, 596, 661, 661, 662, 662, 662
\__fp_round_normal:NnnwNnn .....
    ..... 598, 598, 598
\__fp_round_normal:NNwNnn .....
    ..... 598, 598, 598
\__fp_round_normal:NwNnnw .....
    ..... 598, 598, 598
\__fp_round_normal_end:wwNnn ...
    ..... 598, 599, 599
\__fp_round_o:Nw .....
    ..... 596, 596, 631, 631, 631, 631
\__fp_round_pack:Nw ... 598, 598, 598
\__fp_round_return_one: .....
    ..... 593, 593, 593, 594, 594,
    ..... 594, 594, 594, 594, 594, 595, 596, 596
\__fp_round_s:NNNw .....
    ..... 593, 595, 595, 621, 622, 622
\__fp_round_special:NwwNnn .....
    ..... 598, 599, 599
\__fp_round_special_aux:Nw .....
    ..... 598, 599, 599
\__fp_round_to_nearest:NNN .....
    ..... 593, 594,
    ..... 595, 596, 596, 597, 597, 631, 631, 754
\__fp_round_to_nearest_neg:NNN ..
    ..... 596, 596, 596
\__fp_round_to_nearest_ninf:NNN .
    ..... 593, 594, 596, 597
\__fp_round_to_nearest_ninf_-
neg:NNN ..... 596, 596
\__fp_round_to_nearest_pinf:NNN .
    ..... 593, 594, 596, 597
\__fp_round_to_nearest_pinf_-
neg:NNN ..... 596, 596
\__fp_round_to_nearest_zero:NNN .
    ..... 593, 594, 597
\__fp_round_to_nearest_zero_-
neg:NNN ..... 596, 596
\__fp_round_to_ninf:NNN .....
    ..... 593, 593, 596, 597, 631, 631
\__fp_round_to_ninf_neg:NNN 596, 596
\__fp_round_to_pinf:NNN .....
    ..... 593, 594, 596, 597, 631, 631
\__fp_round_to_pinf_neg:NNN 596, 596
\__fp_round_to_zero:NNN .....
    ..... 593, 594, 597, 631, 631
\__fp_round_to_zero_neg:NNN 596, 596
\__fp_rrot:www ..... 574, 574, 744
\__fp_sanitiz:Nw ..... 576, 576,
    ..... 577, 599, 599, 656, 656, 659, 659,
    ..... 664, 664, 667, 667, 676, 676, 704,
    ..... 712, 719, 736, 737, 739, 744, 744, 745
\__fp_sanitiz:wN .....
    ..... 576, 577, 612, 612, 616, 627
\__fp_sanitiz_zero:w . 576, 577, 577
\__fp_sec_o:w ..... 725, 726
.fp_set:c ..... 174, 540
\fp_set:cn ..... 757
.fp_set:N ..... 174, 540
\fp_set:Nn .... 194, 194, 209, 489,
    ..... 489, 757, 757, 758, 758, 761,
    ..... 761, 761, 764, 764, 765, 766, 766,
    ..... 766, 766, 767, 767, 772, 772, 776,
    ..... 776, 777, 777, 825, 825, 826, 826, 826
\fp_set_eq:cc ..... 757
\fp_set_eq:cN ..... 757
\fp_set_eq:Nc ..... 757
\fp_set_eq:NN ..... 194,
    ..... 194, 757, 757, 757, 758, 765, 766, 766
\__fp_set_sign_o:w .... 627, 683, 683
\fp_show:c ..... 759
\fp_show:N .....
    ..... 201, 201, 759, 759, 759, 781, 781
\fp_show:n .... 201, 201, 759, 759, 781
\__fp_sin_o:w ..... 627, 724, 724, 746
\__fp_sin_series_aux_o:NNnwww ...
    ..... 736, 737, 737
\__fp_sin_series_o:NNwww .....
    ..... 724, 724, 725, 725, 726, 736, 736, 738
\__fp_small_int:wTF ... 584, 584, 598
\__fp_small_int_normal:NnwTF ...
    ..... 584, 584, 584, 584
\__fp_small_int_test:NnnwNnw 584, 585
\__fp_small_int_test:NnnwNTF 584, 584

```

```

\__fp_small_int_true:wTF .....
..... 584, 584, 584, 584, 584, 585
\__fp_sqrt_auxi_o:NNNNwnnN .....
..... 678, 678, 678
\__fp_sqrt_auxii_o:NnnnnnnnN ...
..... 678, 678, 678, 679, 680, 680, 681, 682
\__fp_sqrt_auxiii_o:wnnnnnnnn ...
..... 678, 680, 680, 681
\__fp_sqrt_auxiv_o:NNNNNw .....
..... 680, 680, 681
\__fp_sqrt_auxix_o:wnwnw 681, 681, 681
\__fp_sqrt_auxv_o:NNNNNw 680, 680, 681
\__fp_sqrt_auxvi_o:NNNNNw .....
..... 680, 680, 681
\__fp_sqrt_auxvii_o:NNNNNw .....
..... 680, 680, 681
\__fp_sqrt_auxviii_o:nnnnnnn ...
..... 681, 681, 681, 681, 681, 681
\__fp_sqrt_auxx_o:Nnnnnnnn .....
..... 681, 681, 682
\__fp_sqrt_auxxi_o:wnnnN 681, 682, 682
\__fp_sqrt_auxxii_o:nnnnnnnnw ...
..... 682, 682, 682
\__fp_sqrt_auxxiii_o:w . 682, 682, 683
\__fp_sqrt_auxxiv_o:wnnnnnnnN ...
..... 682, 682, 683, 683, 683
\__fp_sqrt_Newton_o:wnn .....
..... 676, 677, 677, 678, 678, 678
\__fp_sqrt_npos_auxi_o:wnnnN ...
..... 676, 676, 677
\__fp_sqrt_npos_auxii_o:wnnnnnnnN
..... 676, 677, 677
\__fp_sqrt_npos_o:w ... 676, 676, 676
\__fp_sqrt_o:w ..... 676, 676
\s__fp_stop 575, 575, 628, 632, 632,
643, 643, 643, 643, 650, 651, 651, 651
\fp_sub:cn ..... 758
\fp_sub:Nn .... 194, 194, 758, 758, 758
\__fp_sub_back_far_o:NnnwnnnnN ...
..... 659, 660, 660, 660
\__fp_sub_back_near_after:wNNNNw
..... 659, 659, 659, 662
\__fp_sub_back_near_o:nnnnnnnnN .
..... 659, 659, 659, 659
\__fp_sub_back_near_pack:NNNNNw
..... 659, 659, 659, 662
\__fp_sub_back_not_far_o:wwwNN .
..... 661, 661, 662
\__fp_sub_back_quite_far_ii:NN ...
..... 661, 661, 661
\__fp_sub_back_quite_far_o:wwNN .
..... 661, 661, 661
\__fp_sub_back_shift:wnnnn .....
..... 659, 660, 660, 660
\__fp_sub_back_shift_ii:ww .....
..... 660, 660, 660
\__fp_sub_back_shift_iii:NNNNNNNw
..... 660, 660, 660, 660
\__fp_sub_back_shift_iv:nnnnw ...
..... 660, 660, 660
\__fp_sub_back_very_far_ii_-
o:nnNwNN ..... 662, 662, 662
\__fp_sub_back_very_far_o:wwwNN
..... 661, 662, 662
\__fp_sub_eq_o:Nnwnw .. 658, 658, 658
\__fp_sub_npos_i_o:Nnwnw .....
..... 658, 658, 658, 659, 659
\__fp_sub_npos_ii_o:Nnwnw .....
..... 658, 658, 658
\__fp_sub_npos_o:NnwNnw .....
..... 655, 658, 658, 658
\__fp_tan_o:w ..... 726, 726
\__fp_tan_series_aux_o:Nnwww ...
..... 738, 738, 738
\__fp_tan_series_o:NNwww .....
..... 726, 726, 726, 738, 738
\__fp_ternary:NwnN . 638, 644, 650, 650
\__fp_ternary_auxi:NwnN .....
..... 644, 650, 650, 651
\__fp_ternary_auxii:NwnN .....
..... 639, 644, 650, 651, 651
\__fp_ternary_break_point:n ....
..... 650, 650, 651, 651
\__fp_ternary_loop:Nw .....
..... 650, 650, 651, 651
\__fp_ternary_loop_break:w .....
..... 650, 650, 651
\__fp_ternary_map_break: 650, 651, 651
\__fp_tmp:w .....
. 581, 582, 582, 582, 582, 582, 582,
582, 582, 582, 582, 582, 582, 582,
582, 582, 582, 582, 609, 609, 609,
609, 609, 609, 609, 609, 627, 627,
627, 629, 629, 629, 629, 629, 629,
629, 629, 629, 629, 629, 629, 629,
629, 629, 629, 629, 629, 629, 635,
636, 636, 636, 636, 636, 636, 636,
\fp_to_decimal:c ..... 751
\fp_to_decimal:N ..... 194,
194, 195, 586, 751, 751, 754, 756

```

```

\fp_to_decimal:n ..... 194, 194, 194, 195, 195,
751, 751, 754, 754, 756, 756, 756, 756
\_fp_to_decimal_dispatch:w ....
751, 751, 751, 751, 751, 753, 754, 754
\_fp_to_decimal_huge:wnnnn ..... 751, 752, 753
\_fp_to_decimal_large:Nnnw ..... 751, 752, 752
\_fp_to_decimal_normal:wnnnnn ..
..... 751, 752, 752, 754
\fp_to_dim:c ..... 754
\fp_to_dim:N .. 195, 195, 754, 754, 754
\fp_to_dim:n .....
..... 195, 195, 200, 490, 490, 754,
754, 763, 763, 765, 774, 774, 778, 778
\fp_to_int:c ..... 754
\fp_to_int:N .. 195, 195, 754, 754, 754
\fp_to_int:n ..... 195, 195, 754, 754
\_fp_to_int_dispatch:w .....
..... 754, 754, 754, 754
\fp_to_int_dispatch:w ..... 754
\fp_to_scientific:c ..... 749
\fp_to_scientific:N .....
..... 195, 195, 586, 749, 749, 750
\fp_to_scientific:n .....
..... 195, 195, 195, 749, 750
\_fp_to_scientific_dispatch:w ..
..... 749, 749, 750, 750, 750, 751, 753
\_fp_to_scientific_normal:wnnnnn
..... 750, 750, 751, 753, 753
\_fp_to_scientific_normal:wNw ..
..... 750, 751, 751, 751
\fp_to_tl:c ..... 753
\fp_to_tl:N 195, 195, 753, 753, 753, 759
\fp_to_tl:n 195, 195, 574, 589, 589,
589, 589, 590, 590, 590, 753, 753, 759
\_fp_to_tl_dispatch:w .....
..... 753, 753, 753, 753, 753, 757
\_fp_to_tl_normal:nnnnn 753, 753, 753
\c__fp_trailing_shift_int .....
579, 580, 685, 686, 689, 720, 734, 736
\fp_trap:nn ..... 200, 201,
201, 588, 588, 588, 591, 591, 591, 591
\_fp_trap_division_by_zero_-
set:N ..... 589, 589, 589, 589, 589
\_fp_trap_division_by_zero_set_-
error: ..... 589, 589
\_fp_trap_division_by_zero_set_-
flag: ..... 589, 589
\_fp_trap_division_by_zero_set_-
none: ..... 589, 589
\_fp_trap_invalid_operation_-
set:N ..... 588, 588, 588, 588, 588
\_fp_trap_invalid_operation_-
set_error: ..... 588, 588
\_fp_trap_invalid_operation_-
set_flag: ..... 588, 588
\_fp_trap_invalid_operation_-
set_none: ..... 588, 588
\_fp_trap_overflow_set:N .....
..... 590, 590, 590, 590, 590
\_fp_trap_overflow_set:NnNn ...
..... 590, 590, 590, 590
\_fp_trap_overflow_set_error: ..
..... 590, 590
\_fp_trap_overflow_set_flag: ...
..... 590, 590
\_fp_trap_overflow_set_none: ...
..... 590, 590
\_fp_trap_underflow_set:N .....
..... 590, 590, 590, 590, 590
\_fp_trap_underflow_set_error: .
..... 590, 590
\_fp_trap_underflow_set_flag: ..
..... 590, 590
\_fp_trap_underflow_set_none: ..
..... 590, 590
\_fp_trig:NNNNwn .....
724, 725, 725, 726, 726, 727, 727, 727
\_fp_trig_inverse_two_pi: .....
..... 730, 730, 733, 734
\_fp_trig_large:ww ... 728, 733, 734
\_fp_trig_large_auxi:wwwww ...
..... 733, 734, 734
\_fp_trig_large_auxii:ww .....
..... 733, 734, 734
\_fp_trig_large_auxiii:wNNNNNNNN
..... 733, 734, 734
\_fp_trig_large_auxiv:wN .....
..... 733, 734, 734
\_fp_trig_large_auxix:Nw .....
..... 735, 735, 735, 735
\_fp_trig_large_auxv:www .....
..... 734, 734, 734
\_fp_trig_large_auxvi:wnnnnnnnn
..... 734, 734, 735
\_fp_trig_large_auxvii:w .....
..... 734, 735, 735
\_fp_trig_large_auxviii:w .... 735

```

- _fp_trig_large_auxviii:ww 735, 735
 - _fp_trig_large_auxx:wNNNNN ... 735, 735, 736
 - _fp_trig_large_auxxi:w 735, 735, 736
 - _fp_trig_large_pack:NNNNw ... 734, 735, 735, 736
 - _fp_trig_small:ww ... 728, 728, 728, 728, 728, 735, 736
 - _fp_trigd_large:ww .. 728, 728, 729
 - _fp_trigd_large_auxi:nnnnwNNNN ... 728, 729, 729
 - _fp_trigd_large_auxii:wNw ... 728, 729, 729
 - _fp_trigd_large_auxiii:www ... 728, 729, 729
 - _fp_trigd_small:ww ... 728, 728, 728, 729, 729
 - _fp_trim_zeros:w ... 749, 749, 751, 752, 753
 - _fp_trim_zeros_dot:w . 749, 749, 749
 - _fp_trim_zeros_end:w . 749, 749, 749
 - _fp_trim_zeros_loop:w ... 749, 749, 749, 749
 - _fp_type_from_scan:N ... 578, 608, 608, 610, 610
 - _fp_type_from_scan:w . 608, 608, 608
 - \s_fp_underflow ... 575, 575, 575
 - _fp_underflow:w ... 576, 577, 588, 590, 590, 591, 591
 - \l_fp_underflow_flag_token 587, 587
 - \fp_until_do:nn 198, 198, 647, 647, 647
 - \fp_until_do:nNnn ... 198, 198, 647, 647, 648
 - \fp_use:c ... 756
 - \fp_use:N ... 195, 195, 756, 756, 756, 825, 826, 826, 826, 826, 826, 826, 826
 - _fp_use_i:ww ... 574, 574, 693, 694, 746, 747
 - _fp_use_i:www ... 574, 574
 - _fp_use_i_until_s:nw ... 574, 574, 576, 586, 729, 734, 734, 735, 735
 - _fp_use_ii_until_s:nnw 574, 574, 576
 - _fp_use_none_stop_f:n ... 573, 573, 701, 701, 701
 - _fp_use_none_until_s:w ... 574, 574, 678, 722, 747, 747
 - _fp_use_s:n ... 574, 574
 - _fp_use_s:nn ... 574, 574
 - \fp_while_do:nn 198, 198, 647, 647, 647
 - \fp_while_do:nNnn ... 198, 198, 647, 648, 648
 - \fp_zero:c ... 758
 - \fp_zero:N ... 193, 193, 758, 758, 758, 825
 - _fp_zero_fp:N ... 575, 575, 590, 599
 - \fp_zero_new:c ... 758
 - \fp_zero_new:N 193, 193, 758, 758, 758
 - \futurelet ... 244
- ## G
- \gdef ... 244
 - \GetIdInfo ... 7, 7, 7
 - \gleaders ... 254
 - \global ... 236, 237, 241, 244
 - \globaldefs ... 244
 - \glueexpr ... 249
 - \glueshrink ... 249
 - \glueshrinkorder ... 249
 - \gluestretch ... 249
 - \gluestretchorder ... 249
 - \gluetomu ... 249
 - group commands:
 - \group_align_safe_begin/end: .. 317
 - \group_align_safe_begin: ... 44, 44, 44, 309, 310, 317, 317, 340, 340, 395, 395, 399, 405, 789, 801
 - \group_align_safe_end: ... 44, 44, 44, 312, 312, 317, 317, 339, 340, 340, 340, 395, 396, 399, 406, 790
 - \group_begin: ... 10, 10, 10, 263, 263, 287, 304, 323, 323, 324, 324, 324, 327, 328, 328, 329, 333, 342, 390, 390, 391, 398, 414, 414, 418, 435, 475, 502, 502, 507, 507, 508, 516, 521, 526, 526, 552, 566, 566, 567, 611, 628, 633, 634, 635, 635, 637, 637, 638, 650, 750, 761, 764, 765, 766, 766, 766, 767, 788, 788, 806, 811, 811, 813, 814, 815
 - \c_group_begin_token ... 56, 106, 328, 328, 328, 329, 329, 409, 410, 410, 476, 477
 - \group_end: 10, 10, 10, 10, 263, 263, 287, 304, 323, 324, 324, 324, 325, 327, 328, 328, 330, 333, 344, 390, 390, 392, 398, 398, 414, 415, 418, 435, 435, 435, 475, 502, 502, 507, 507, 511, 517, 521, 526, 526, 553, 566, 566, 568, 611, 628, 634, 634,

635, 636, 637, 638, 639, 650, 750,
 761, 764, 766, 766, 767, 767,
 788, 788, 811, 811, 811, 814, 815, 816
 \c_group_end_token 56,
 328, 328, 328, 329, 329, 476, 476, 478
 \group_insert_after:N
 ... 10, 10, 10, 263, 263, 827, 827, 828
 groups commands:
 .groups:n 174, 540

H

\halign 244
 \hangafter 244
 \hangindent 244
 hash commands:
 \c_hash_str 117, 426, 427
 \hbadness 244
 \hbox 244
 hbox commands:
 \hbox:n 151,
 151, 233, 475, 475, 497, 498, 762, 768
 \hbox_gset:cn 475
 \hbox_gset:cw 476
 \hbox_gset:Nn 151, 475, 475, 475
 \hbox_gset:Nw 151, 476, 476, 476
 \hbox_gset_end: 151, 476, 476
 \hbox_gset_to_wd:cnn 475
 \hbox_gset_to_wd:Nnn 151, 475, 475, 475
 \hbox_overlap_left:n 151, 151, 476, 476
 \hbox_overlap_right:n
 151, 151, 476, 476, 767, 825
 \hbox_set:cn 475
 \hbox_set:cw 476
 \hbox_set:Nn
 151, 151, 151, 475, 475, 475,
 475, 481, 484, 491, 493, 500, 761,
 762, 762, 764, 765, 766, 766, 766,
 767, 767, 768, 769, 769, 769, 769,
 769, 770, 770, 770, 770, 770, 772, 773
 \hbox_set:Nw
 151, 151, 476, 476, 476, 476, 482
 \hbox_set_end: 151, 151, 476, 476, 482
 \hbox_set_to_wd:cnn 475
 \hbox_set_to_wd:Nnn
 151, 151, 475, 475, 475, 475
 \hbox_to_wd:nn 151, 151, 476, 476, 768
 \hbox_to_zero:n
 151, 151, 476, 476, 476, 476
 \hbox_unpack:c 476

\hbox_unpack:N
 152, 152, 476, 476, 476, 491, 495
 \hbox_unpack_clear:c 476
 \hbox_unpack_clear:N
 152, 152, 476, 476, 476
 hcoffin commands:
 \hcoffin_set:cn 481
 \hcoffin_set:cw 482
 \hcoffin_set:Nn 155,
 155, 481, 481, 497, 497, 498, 499
 \hcoffin_set:Nw 156, 156, 482, 482, 483
 \hcoffin_set_end:
 156, 156, 482, 482, 483
 \hfil 244
 \hfill 244
 \hfilneg 244
 \hfuzz 244
 \hoffset 244
 \holdinginserts 244
 \hrule 244
 \hsize 244
 \hskip 244
 \hss 244
 \ht 244
 \hyphenation 244
 \hyphenchar 244
 \hyphenpenalty 244

I

\i 807, 807
 \if 244
 if commands:
 \if:w 24, 51, 51, 51, 262, 262,
 273, 273, 273, 273, 274, 301, 302,
 302, 302, 303, 303, 338, 364, 364,
 615, 615, 616, 619, 620, 620, 622,
 623, 624, 624, 624, 637, 638, 638, 719
 \if_bool:N 44, 305, 305, 306, 503
 \if_box_empty:N 154, 154, 473, 473, 473
 \if_case:w 77, 77, 282, 345,
 345, 361, 361, 362, 420, 420, 420,
 422, 422, 576, 583, 584, 596, 597,
 640, 641, 654, 658, 661, 661, 663,
 667, 684, 694, 704, 704, 711, 712,
 714, 714, 714, 714, 715, 715, 715,
 716, 717, 719, 722, 722, 724, 725,
 725, 726, 726, 727, 740, 741, 743,
 745, 746, 748, 748, 750, 752, 753, 815
 \if_catcode:w
 . 24, 262, 262, 329, 329, 329, 329,

329, 330, 330, 330, 330, 330, 331,
 331, 332, 341, 341, 341, 400, 400,
 409, 410, 410, 411, 411, 411, 609,
 614, 624, 626, 633, 637, 640, 816, 816
 \if_charcode:w 24, 51, 262, 262, 331,
 341, 341, 409, 409, 410, 410, 425, 425
 \if_cs_exist:N
 24, 262, 262, 275, 276, 332, 338
 \if_cs_exist:w
 . 24, 262, 262, 263, 275, 276, 282, 587
 \if_dim:w 93, 93, 369, 369, 371, 372, 373
 \if_eof:w 190, 190, 560, 560, 560
 \if_false:
 .. 23, 39, 262, 262, 317, 317, 352,
 352, 372, 395, 395, 395, 399, 399,
 399, 407, 408, 408, 408, 411, 411,
 412, 412, 433, 433, 437, 437, 438, 814
 \if_hbox:N 154, 154, 473, 473, 473
 \if_int_compare:w
 23, 77, 77, 263, 263,
 317, 317, 332, 338, 345, 346, 348,
 351, 351, 351, 353, 353, 353, 353,
 353, 353, 353, 353, 353, 353, 379,
 416, 416, 416, 416, 420, 421, 421,
 422, 422, 422, 560, 577, 577, 581,
 584, 585, 593, 594, 594, 594, 594,
 594, 595, 595, 596, 596, 598, 598,
 609, 610, 613, 613, 614, 614, 615,
 616, 617, 617, 619, 620, 620, 621,
 621, 622, 624, 625, 625, 626, 626,
 627, 628, 633, 633, 633, 634, 635,
 635, 636, 636, 638, 638, 640, 645,
 646, 646, 646, 646, 646, 646, 646,
 646, 649, 651, 654, 654, 656, 659,
 661, 661, 661, 661, 663, 663, 674,
 678, 680, 680, 680, 681, 682, 682,
 682, 682, 682, 683, 694, 694, 699,
 702, 704, 705, 709, 711, 712, 712,
 712, 713, 719, 719, 719, 719, 720,
 721, 721, 722, 723, 723, 723, 723,
 723, 728, 729, 741, 742, 743, 744,
 747, 747, 751, 753, 753, 753, 812,
 812, 812, 812, 812, 812, 812, 812, 812
 \if_int_odd:w 77, 77, 345,
 345, 348, 354, 355, 594, 595, 595,
 641, 662, 676, 723, 735, 737, 737,
 738, 738, 739, 745, 812, 812, 813, 816
 \if_meaning:w 23, 262, 262,
 269, 270, 270, 271, 272, 272, 275,
 275, 276, 276, 283, 283, 286, 288,
 291, 291, 299, 300, 301, 307, 311,
 311, 311, 319, 319, 320, 320, 320,
 331, 332, 334, 334, 334, 334, 335,
 335, 335, 335, 335, 335, 338, 341,
 342, 346, 347, 347, 347, 352, 371,
 372, 388, 396, 397, 397, 397, 398,
 398, 398, 406, 408, 410, 410, 418,
 423, 424, 434, 435, 435, 436, 449,
 449, 450, 450, 468, 469, 469, 576,
 577, 577, 577, 577, 584, 585, 585,
 590, 593, 594, 594, 594, 594, 595,
 595, 595, 595, 595, 595, 597, 598,
 598, 599, 599, 599, 609, 609, 614,
 618, 618, 626, 631, 631, 631, 633,
 641, 641, 644, 645, 645, 645, 645,
 645, 645, 648, 649, 649, 649, 649,
 650, 650, 652, 652, 654, 655, 655,
 655, 657, 657, 659, 660, 663, 663,
 664, 665, 672, 674, 674, 675, 676,
 676, 676, 693, 693, 702, 702, 704,
 708, 710, 712, 712, 717, 717, 718,
 718, 718, 719, 721, 721, 737, 738,
 741, 741, 741, 741, 742, 742, 745,
 748, 750, 751, 753, 755, 755, 783, 816
 \if_mode_horizontal: 24, 262, 262, 317
 \if_mode_inner: ... 24, 262, 262, 317
 \if_mode_math: 24, 262, 262, 317
 \if_mode_vertical: . 24, 262, 262, 317
 \if_predicate:w 37, 39, 44, 305, 305, 309
 \if_true: ... 23, 39, 262, 262, 397, 397
 \if_vbox:N 154, 154, 473, 473, 473
 \ifcase 244
 \ifcat 244
 \ifcsname 249
 \ifdbbox 257
 \ifddir 257
 \ifdefined 249
 \ifdim 244
 \ifeof 244
 \iffalse 244
 \iffontchar 249
 \ifhbox 244
 \ifhmode 244
 \ifincsname 252
 \ifinner 244
 \ifmdir 257
 \ifmmode 244
 \ifnum ... 235, 235, 236, 237, 237, 238, 244
 \ifodd 244
 \ifpdfprimitive 251

- \iftbox 258
- \iftdir 258
- \iftrue 244
- \ifvbox 244
- \ifvmode 244
- \ifvoid 244
- \ifx 234, 234, 235, 235,
236, 236, 237, 237, 238, 239, 239, 244
- \ifybox 258
- \ifydir 258
- \ignorespaces 244
- \immediate 244
- in 208
- \indent 244
- inf 208
- inf commands:
 - \c_inf_fp 199, 208, 575, 575,
629, 664, 667, 712, 718, 718, 719, 727
- \inhibitglue 258
- \inhibitxspcode 258
- \initcatcodetable 236, 254
- initial commands:
 - .initial:n 175, 540
 - .initial:o 175, 540
 - .initial:V 175, 540
 - .initial:x 175, 540
- \input 244
- \inputlineno 244
- \insert 244
- \insertpenalties 244
- int commands:
 - \int_(g)zero:N 67
 - __int_abs:N 346, 346, 346
 - \int_abs:n 65, 65, 346, 346
 - \int_add:cn 350
 - \int_add:Nn 67,
67, 350, 350, 350, 350, 569, 570, 571
 - \int_case:nn 70, 354, 354, 358, 358, 361
 - \int_case:nnF . 354, 442, 458, 799, 800
 - \int_case:nnT 354
 - __int_case:nnTF
..... 354, 354, 354, 354, 354, 354
 - \int_case:nnTF ... 25, 70, 70, 354, 354
 - __int_case:nw 354, 354, 354, 354
 - __int_case_end:nw 354, 354, 354
 - \int_compare:n 352
 - \int_compare:n(TF) 78
 - \int_compare:nF 355, 355
 - \int_compare:nNn 353
 - __int_compare:nnN 351, 353,
353, 353, 353, 353, 353, 353, 353
 - \int_compare:nNnF .. 323, 356, 356, 357
 - \int_compare:nNnT 323, 323,
324, 355, 356, 390, 392, 412, 439, 804
 - \int_compare:nNnTF
..... 68, 68, 68, 70, 71, 71, 71,
314, 323, 348, 348, 353, 354, 356,
356, 358, 360, 360, 360, 361, 364,
365, 365, 366, 376, 391, 392, 392,
413, 419, 419, 420, 439, 459, 459,
459, 459, 460, 552, 564, 569, 642,
749, 752, 752, 796, 797, 797, 798, 801
 - __int_compare:NNw
..... 351, 352, 352, 352, 353
 - \int_compare:nT ... 355, 355, 559, 563
 - \int_compare:nTF
.. 69, 69, 71, 71, 71, 71, 197, 351, 372
 - __int_compare:Nw
.... 351, 351, 351, 352, 352, 353, 353
 - __int_compare:w ... 351, 352, 352, 352
 - int_compare_
 - __int_compare_>:NNw 351
 - __int_compare_<:NNw 351
 - \int_compare_p:n 69, 69, 351
 - \int_compare_p:nNn
... 23, 68, 68, 353, 798, 799, 799,
799, 800, 804, 804, 805, 805, 818, 818
 - \int_const:cn
.. 348, 365, 365, 365, 365, 365, 365,
365, 365, 366, 366, 366, 366, 366, 366
 - \int_const:Nn 66,
66, 348, 348, 348, 367, 367, 367,
367, 367, 367, 367, 367, 367, 367,
367, 368, 368, 368, 368, 368, 368,
368, 368, 368, 368, 368, 575, 580,
580, 580, 580, 580, 580, 580, 580,
580, 813, 813, 817, 817, 817, 817, 817
 - __int_constdef:Nw
..... 348, 348, 348, 348, 348, 349
 - \int_decr:c 350
 - \int_decr:N . 67, 67, 350, 350, 350, 350
 - \int_div_round:nn ... 66, 66, 347, 347
 - \int_div_truncate:nn 66,
66, 66, 347, 347, 358, 360, 361, 794, 817
 - __int_div_truncate:NwNw
..... 347, 347, 347, 347
 - \int_do_until:nn . 71, 71, 355, 355, 355
 - \int_do_until:nNnn 70, 70, 355, 356, 356
 - \int_do_while:nn . 71, 71, 355, 355, 355

```

\int_do_while:nNnn 71, 71, 355, 356, 356
\int_eval:n ..... 16,
    28, 28, 65, 65, 65, 65, 65, 66, 67, 68,
    69, 70, 77, 78, 170, 282, 283, 284,
    346, 346, 346, 354, 354, 354, 354,
    357, 358, 360, 360, 363, 363, 364,
    364, 365, 365, 365, 366, 367, 376,
    403, 403, 412, 413, 421, 423, 424,
    439, 439, 441, 457, 457, 459, 459,
    460, 474, 474, 524, 559, 562, 576,
    600, 642, 645, 668, 668, 670, 787, 787
\__int_eval:w ... 78, 78, 282, 315,
    324, 324, 326, 326, 326, 326, 326,
    326, 327, 327, 327, 327, 327, 327,
    345, 345, 346, 346, 346, 346, 346,
    346, 346, 346, 347, 347, 347, 347,
    347, 347, 347, 347, 348, 350, 350,
    350, 352, 352, 353, 353, 353, 354,
    355, 356, 356, 356, 361, 362, 362,
    362, 419, 419, 420, 420, 421, 421,
    421, 422, 426, 576, 581, 581, 582,
    585, 593, 595, 595, 595, 595, 595,
    595, 595, 596, 598, 598, 599, 608,
    612, 613, 614, 617, 617, 619, 620,
    620, 620, 621, 622, 622, 622, 622,
    622, 623, 623, 624, 628, 633, 640,
    645, 656, 656, 656, 657, 657, 657,
    657, 658, 658, 658, 659, 659, 659,
    659, 662, 662, 662, 662, 662, 663,
    664, 664, 665, 665, 665, 665, 665,
    665, 665, 665, 665, 666, 666, 666,
    666, 667, 667, 668, 671, 671, 672,
    672, 672, 672, 672, 672, 672, 672,
    673, 673, 673, 673, 674, 674, 674,
    674, 675, 675, 676, 676, 676, 678,
    678, 679, 679, 679, 679, 679, 679,
    679, 679, 679, 680, 680, 680, 680,
    681, 681, 681, 682, 683, 683, 683,
    685, 685, 685, 686, 686, 686, 686,
    686, 686, 687, 687, 687, 687, 688,
    688, 688, 689, 689, 689, 689, 689,
    689, 690, 690, 690, 691, 691, 691,
    691, 691, 691, 692, 692, 693, 693,
    695, 697, 697, 697, 698, 698, 698,
    699, 699, 700, 700, 701, 701, 701,
    702, 702, 704, 705, 705, 705, 705,
    707, 707, 707, 707, 707, 708, 708,
    708, 708, 708, 708, 708, 708, 708,
    708, 708, 708, 708, 709, 709,
    709, 710, 710, 712, 712, 713, 714,
    720, 720, 720, 720, 720, 720, 720,
    721, 721, 722, 722, 728, 729, 729,
    734, 734, 734, 735, 735, 735, 736,
    736, 737, 737, 737, 738, 739, 739,
    740, 742, 742, 742, 743, 743, 744,
    745, 745, 747, 751, 755, 793, 812, 812
\__int_eval_end: .....
    78, 78, 78, 78, 282, 315, 324, 324,
    326, 326, 326, 326, 326, 326, 327,
    327, 327, 327, 327, 327, 345, 345,
    346, 346, 346, 347, 347, 347, 348,
    350, 350, 350, 353, 354, 355, 361,
    362, 362, 362, 426, 576, 585, 596,
    598, 598, 640, 645, 652, 658, 662,
    664, 674, 687, 693, 721, 722, 729,
    729, 737, 737, 738, 739, 740, 743, 793
\__int_from_alph:N . 364, 364, 364, 364
\int_from_alph:n ... 74, 74, 364, 364
\__int_from_alph:nN .....
    ..... 364, 364, 364, 364, 364
\__int_from_base:N . 365, 365, 365, 365
\int_from_base:nn .....
    ..... 75, 75, 365, 365, 365, 365, 365
\__int_from_base:nnN .....
    ..... 365, 365, 365, 365, 365
\int_from_bin:n . 74, 74, 365, 365, 369
\int_from_binary:n ..... 369, 369
\int_from_hex:n . 75, 75, 365, 365, 369
\int_from_hexadecimal:n .... 369, 369
\int_from_oct:n . 75, 75, 365, 365, 369
\int_from_octal:n ..... 369, 369
\int_from_roman:n ... 75, 75, 366, 366
\__int_from_roman:NN .....
    ..... 366, 366, 366, 366, 366
\c__int_from_roman_C_int ..... 365
\c__int_from_roman_c_int ..... 365
\c__int_from_roman_D_int ..... 365
\c__int_from_roman_d_int ..... 365
\__int_from_roman_error:w .....
    ..... 366, 366, 366, 366
\c__int_from_roman_I_int ..... 365
\c__int_from_roman_i_int ..... 365
\c__int_from_roman_L_int ..... 365
\c__int_from_roman_l_int ..... 365
\c__int_from_roman_M_int ..... 365
\c__int_from_roman_m_int ..... 365
\c__int_from_roman_V_int ..... 365
\c__int_from_roman_v_int ..... 365
\c__int_from_roman_X_int ..... 365
\c__int_from_roman_x_int ..... 365

```

- \int_gadd:cn [350](#)
- \int_gadd:Nn [67](#), [350](#), [350](#), [350](#)
- \int_gdecr:c [350](#)
- \int_gdecr:N [67](#), [350](#), [350](#),
[350](#), [357](#), [402](#), [441](#), [456](#), [470](#), [528](#), [780](#)
- \int_gincr:c [350](#)
- \int_gincr:N ... [67](#), [350](#), [350](#), [350](#),
[357](#), [357](#), [401](#), [440](#), [456](#), [470](#), [528](#), [780](#)
- .int_gset:c [175](#), [540](#)
- \int_gset:cn [350](#)
- .int_gset:N [175](#), [540](#)
- \int_gset:Nn [67](#), [348](#), [348](#), [350](#), [350](#), [350](#)
- \int_gset_eq:cc [349](#)
- \int_gset_eq:cN [349](#)
- \int_gset_eq:Nc [349](#)
- \int_gset_eq:NN
..... [67](#), [349](#), [349](#), [349](#), [349](#), [508](#)
- \int_gsub:cn [350](#)
- \int_gsub:Nn [68](#), [350](#), [350](#), [350](#)
- \int_gzero:c [349](#)
- \int_gzero:N ... [66](#), [349](#), [349](#), [349](#), [349](#)
- \int_gzero_new:c [349](#)
- \int_gzero_new:N ... [67](#), [349](#), [349](#), [349](#)
- \int_if_even:n [354](#)
- \int_if_even:nTF [70](#), [354](#)
- \int_if_even_p:n [70](#), [354](#)
- \int_if_exist:c [349](#)
- \int_if_exist:cF [366](#), [366](#)
- \int_if_exist:cTF [349](#)
- \int_if_exist:N [349](#)
- \int_if_exist:NTF [67](#), [67](#), [349](#), [349](#), [349](#)
- \int_if_exist_p:c [349](#)
- \int_if_exist_p:N [67](#), [67](#), [349](#)
- \int_if_odd:n [354](#)
- \int_if_odd:nTF ... [70](#), [70](#), [354](#), [699](#)
- \int_if_odd_p:n [70](#), [70](#), [354](#)
- \int_incr:c [350](#)
- \int_incr:N
.. [67](#), [67](#), [350](#), [350](#), [350](#), [350](#), [535](#), [570](#)
- \int_log:c [781](#)
- \int_log:N ... [218](#), [218](#), [781](#), [781](#), [781](#)
- \int_log:n [219](#), [219](#), [781](#), [781](#)
- \int_max:nn
..... [66](#), [66](#), [346](#), [346](#), [692](#), [729](#), [756](#)
- __int_maxmin:wwN .. [346](#), [346](#), [346](#), [346](#)
- \int_min:nn [66](#), [66](#), [346](#), [346](#)
- \int_mod:nn [66](#),
[66](#), [347](#), [347](#), [358](#), [360](#), [361](#), [552](#), [817](#)
- __int_mod:ww [347](#), [347](#), [347](#)
- \int_new:c [348](#)
- \int_new:N [66](#),
[66](#), [67](#), [318](#), [348](#), [348](#), [348](#), [348](#),
[348](#), [349](#), [349](#), [368](#), [368](#), [368](#), [368](#),
[526](#), [529](#), [565](#), [565](#), [565](#), [565](#), [565](#), [827](#)
- __int_pass_signs:wn
..... [364](#), [364](#), [364](#), [364](#), [364](#), [365](#)
- __int_pass_signs_end:wn [364](#), [364](#), [364](#)
- .int_set:c [175](#), [540](#)
- \int_set:cn [350](#)
- .int_set:N [175](#), [540](#)
- \int_set:Nn [67](#), [67](#),
[350](#), [350](#), [350](#), [350](#), [475](#), [475](#), [535](#),
[560](#), [564](#), [564](#), [565](#), [567](#), [569](#), [569](#), [570](#)
- \int_set_eq:cc [349](#)
- \int_set_eq:cN [349](#)
- \int_set_eq:Nc [349](#)
- \int_set_eq:NN .. [67](#), [67](#), [349](#), [349](#),
[349](#), [349](#), [475](#), [475](#), [560](#), [566](#), [567](#), [568](#)
- \int_show:c [367](#)
- \int_show:N
.. [75](#), [75](#), [367](#), [367](#), [367](#), [781](#), [781](#), [781](#)
- \int_show:n
..... [75](#), [75](#), [367](#), [367](#), [524](#), [781](#), [781](#)
- __int_step:NnnnN
..... [356](#), [356](#), [357](#), [357](#), [357](#)
- __int_step:NNnnnn .. [357](#), [357](#), [357](#), [357](#)
- __int_step:wwwN [356](#), [356](#), [356](#)
- \int_step_function:nnnN [72](#),
[72](#), [356](#), [356](#), [357](#), [357](#), [814](#), [814](#), [814](#)
- \int_step_inline:nnnn
..... [72](#), [72](#), [357](#), [357](#), [557](#), [561](#)
- \int_step_variable:nnnN
..... [72](#), [72](#), [357](#), [357](#)
- \int_sub:cn [350](#)
- \int_sub:Nn
..... [68](#), [68](#), [350](#), [350](#), [350](#), [350](#), [571](#)
- \int_to_Alph:n ... [73](#), [73](#), [74](#), [358](#), [359](#)
- \int_to_alph:n
..... [73](#), [73](#), [73](#), [73](#), [74](#), [358](#), [358](#)
- \int_to_arabic:n ... [72](#), [72](#), [358](#), [358](#)
- \int_to_Base:n [74](#)
- \int_to_base:n [74](#)
- __int_to_Base:nn [360](#), [360](#), [360](#)
- \int_to_Base:nn .. [74](#), [75](#), [360](#), [360](#), [363](#)
- __int_to_base:nn [360](#), [360](#), [360](#)
- \int_to_base:nn
... [74](#), [74](#), [75](#), [360](#), [360](#), [363](#), [363](#), [363](#)
- __int_to_Base:nnN
..... [360](#), [360](#), [360](#), [361](#), [361](#)

```

\__int_to_base:nnN .....
..... 360, 360, 360, 360, 360
\__int_to_Base:nnnN ... 360, 361, 361
\__int_to_base:nnnN ... 360, 360, 360
\int_to_bin:n .....
..... 73, 73, 74, 74, 362, 363, 369
\int_to_binary:n ..... 369, 369
\int_to_Hex:n 74, 74, 75, 362, 363, 369
\int_to_hex:n . 74, 74, 74, 75, 362, 363
\int_to_hexadecimal:n ..... 369, 369
\__int_to_Letter:n . 360, 361, 361, 362
\__int_to_letter:n . 360, 360, 360, 361
\int_to_oct:n 74, 74, 75, 362, 363, 369
\int_to_octal:n ..... 369, 369
\int_to_Roman:n .. 74, 74, 75, 363, 363
\__int_to_roman:N .....
..... 363, 363, 363, 363, 363
\int_to_roman:n .....
..... 74, 74, 74, 75, 363, 363, 794
\__int_to_roman:w ..... 77, 77,
263, 263, 345, 352, 352, 363, 363,
363, 581, 619, 620, 705, 714, 814, 815
\__int_to_Roman_aux:N . 363, 363, 363
\__int_to_Roman_c:w ..... 363, 363
\__int_to_roman_c:w ..... 363, 363
\__int_to_Roman_d:w ..... 363, 363
\__int_to_roman_d:w ..... 363, 363
\__int_to_Roman_i:w ..... 363, 363
\__int_to_roman_i:w ..... 363, 363
\__int_to_Roman_l:w ..... 363, 363
\__int_to_roman_l:w ..... 363, 363
\__int_to_Roman_m:w ..... 363, 363
\__int_to_roman_m:w ..... 363, 363
\__int_to_Roman_Q:w ..... 363, 364
\__int_to_roman_Q:w ..... 363, 363
\__int_to_Roman_v:w ..... 363, 363
\__int_to_roman_v:w ..... 363, 363
\__int_to_Roman_x:w ..... 363, 363
\__int_to_roman_x:w ..... 363, 363
\int_to_symbols:nnn .....
... 73, 73, 73, 358, 358, 358, 358, 359
\__int_to_symbols:nnnn . 358, 358, 358
\int_until_do:nn . 71, 71, 355, 355, 355
\int_until_do:nNnn 71, 71, 355, 356, 356
\int_use:c ..... 350, 351
\int_use:N .....
65, 68, 68, 68, 346, 346, 346, 346,
346, 347, 347, 347, 350, 350, 351,
352, 356, 356, 356, 357, 357, 401,
401, 419, 419, 420, 420, 421, 421,
421, 422, 426, 440, 441, 456, 456,
470, 470, 505, 518, 527, 528, 528,
528, 535, 560, 564, 576, 585, 595,
595, 598, 598, 599, 600, 612, 617,
617, 619, 620, 620, 620, 621, 622,
622, 622, 622, 623, 628, 656, 657,
657, 657, 657, 658, 658, 658, 659,
659, 659, 659, 662, 662, 662, 662,
664, 665, 665, 665, 665, 665, 665,
665, 666, 666, 666, 666, 667, 667,
671, 671, 672, 672, 672, 672, 672,
672, 672, 672, 673, 673, 673, 674,
674, 674, 675, 675, 676, 676, 676,
678, 678, 679, 679, 679, 679, 679,
679, 679, 679, 679, 680, 680, 680,
680, 681, 681, 681, 682, 683, 683,
683, 685, 685, 685, 686, 686, 686,
686, 686, 686, 687, 687, 687, 687,
688, 688, 688, 689, 689, 689, 689,
689, 689, 690, 690, 690, 691, 691,
691, 691, 691, 691, 692, 692, 693,
693, 695, 697, 697, 697, 698, 698,
698, 699, 699, 700, 700, 701, 702,
702, 704, 705, 705, 705, 707, 707,
707, 707, 707, 708, 708, 708, 708,
708, 708, 708, 708, 708, 708, 708,
708, 708, 708, 709, 709, 709, 710,
710, 712, 713, 720, 720, 720, 720,
720, 720, 721, 722, 728, 729, 729,
734, 734, 734, 735, 735, 735, 736,
736, 737, 739, 742, 742, 744, 745,
745, 750, 751, 755, 780, 793, 812, 812
\__int_value:w ..... 78, 78, 78,
274, 311, 311, 311, 315, 345, 345,
346, 346, 346, 346, 346, 347, 347,
347, 347, 352, 352, 353, 362, 362,
372, 372, 419, 421, 480, 481, 481,
481, 481, 485, 485, 485, 485, 485,
485, 485, 485, 485, 485, 485, 486,
486, 486, 486, 486, 487, 487, 487,
487, 487, 491, 492, 492, 493, 494,
494, 498, 501, 576, 578, 578, 578,
578, 578, 582, 582, 582, 584, 584,
585, 595, 598, 598, 598, 601, 601,
601, 601, 601, 601, 608, 611, 611,
612, 612, 616, 616, 616, 617, 617,
618, 619, 619, 619, 620, 624, 624,
626, 626, 629, 642, 645, 646, 654,
654, 654, 654, 654, 656, 656, 658,
660, 660, 660, 662, 662, 662, 666,

```

- 666, 671, 671, 671, 671, 671, 676,
- 682, 682, 683, 684, 701, 701, 701,
- 704, 704, 705, 707, 707, 707, 707,
- 707, 711, 711, 711, 712, 714, 719,
- 721, 721, 721, 721, 721, 734, 737,
- 738, 738, 751, 753, 755, 755, 755,
- 772, 772, 773, 773, 774, 775, 776,
- 776, 777, 777, 777, 777, 778, 778, 778
- \int_while_do:nn . 71, 71, 355, 355, 355
- \int_while_do:nNn 71, 71, 355, 355, 356
- \int_zero:c 349
- \int_zero:N 66,
- 66, 349, 349, 349, 349, 535, 568, 570
- \int_zero_new:c 349
- \int_zero_new:N . 67, 67, 349, 349, 349
- \interactionmode 249
- \interlinepenalties 249
- \interlinepenalty 244
- ior commands:
- \ior_... 184
- \ior_close:c 559
- \ior_close:N 185,
- 185, 186, 186, 553, 558, 559, 559, 559
- \ior_get:NN
- ... 186, 186, 187, 190, 560, 560, 780
- \ior_get_str:NN
- ... 187, 187, 187, 560, 560, 780
- \ior_if_eof:N 560, 780
- \ior_if_eof:Nf 553, 780, 780
- \ior_if_eof:Ntf ... 187, 187, 553, 560
- \ior_if_eof_p:N 187, 187, 560
- \l_ior_internal_tl 780, 780, 780, 780
- \ior_list_streams:
- ... 186, 186, 559, 559, 780, 780
- _ior_list_streams:Nn
- ... 559, 559, 559, 563
- \ior_log_streams: . . 218, 218, 780, 780
- \ior_map_... 217, 217, 218, 218
- \ior_map_break:
- ... 217, 217, 779, 779, 779, 780, 780
- \ior_map_break:n ... 218, 218, 779, 780
- \ior_map_inline:Nn . 217, 217, 780, 780
- _ior_map_inline:NnN
- ... 780, 780, 780, 780
- _ior_map_inline:NnNn . 780, 780, 780
- _ior_map_inline_loop:NnN
- ... 780, 780, 780, 780
- \ior_new:c 557
- _ior_new:N 558, 558, 558, 559
- \ior_new:N . 185, 185, 557, 557, 557, 561
- \ior_open:cn 557
- \ior_open:cnTF 558
- _ior_open:Nn
- ... 191, 191, 553, 553, 558, 558, 559
- \ior_open:Nn 185, 185, 557, 557, 557, 558
- \ior_open:Nnf 558
- \ior_open:Nnt 558
- \ior_open:Nntf ... 185, 185, 558, 558
- _ior_open:No 558, 558, 558
- _ior_open_aux:Nn 557, 557, 557
- _ior_open_aux:Nntf . . 558, 558, 558
- _ior_open_stream:Nn
- ... 558, 559, 559, 559
- \ior_str_map_inline:Nn
- ... 217, 217, 780, 780
- \l_ior_stream_tl
- ... 557, 557, 558, 559, 559
- \g_ior_streams_prop
- ... 557, 557, 557, 559, 559, 559
- \g_ior_streams_seq
- ... 556, 556, 556, 558, 559, 559, 561
- iow commands:
- \iow_... 184
- \iow_char:N ... 188, 188, 565, 565, 717
- \iow_close:c 563
- \iow_close:N
- ... 186, 186, 186, 562, 563, 563, 563
- \l_iow_current_indentation_int
- ... 565, 565, 570, 570, 571, 571, 571
- \l_iow_current_indentation_tl
- ... 565, 565, 568, 569, 570, 571, 571
- \l_iow_current_line_int ... 565,
- 565, 568, 569, 569, 570, 570, 570, 570
- \l_iow_current_line_tl . 565, 565,
- 568, 569, 570, 570, 570, 570, 570, 571
- \l_iow_current_word_int
- ... 565, 565, 569, 569, 570
- \l_iow_current_word_tl
- 565, 565, 569, 569, 569, 569, 570, 570
- _iow_indent:n 567, 567, 567
- \iow_indent:n
- 189, 189, 189, 518, 549, 567, 567,
- 567, 567, 568, 568, 568, 571, 591, 592
- _iow_indent_error:n
- ... 567, 567, 567, 568
- \l_iow_line_count_int 189,
- 189, 189, 565, 565, 565, 568, 568, 569
- \l_iow_line_start_bool
- ... 566, 566, 568, 569, 569, 570

- \iow_list_streams: 186, 186, 563, 563, 780, 780
 - __iow_list_streams:Nn . 563, 563, 563
 - \iow_log:n 187, 187, 507, 508, 508, 511, 523, 556, 556, 564, 564
 - \iow_log:x 24, 277, 277, 277, 277, 524, 525, 564, 564
 - \iow_log_streams: .. 218, 218, 780, 780
 - \iow_new:c 562
 - __iow_new:N 562, 562, 562
 - \iow_new:N 185, 185, 562, 562, 562
 - \iow_newline: 188, 188, 188, 188, 191, 506, 506, 507, 507, 525, 564, 565, 565, 568, 568
 - \l_iow_newline_tl . 566, 566, 568, 568, 568, 568, 569, 570, 570
 - \iow_now:cn 564
 - \iow_now:cx 564
 - \iow_now:Nn ... 187, 187, 187, 187, 187, 188, 188, 564, 564, 564, 564, 564
 - \iow_now:Nx 564, 564, 564
 - \iow_open:cn 562
 - __iow_open:Nn 562, 562, 562, 562
 - \iow_open:Nn .. 186, 186, 562, 562, 562
 - __iow_open_stream:Nn 562, 562, 562, 562
 - \iow_shipout:cn 563
 - \iow_shipout:cx 563
 - \iow_shipout:Nn 188, 188, 188, 188, 563, 563, 563, 564
 - \iow_shipout:Nx 563
 - \iow_shipout_x:cn 563
 - \iow_shipout_x:cx 563
 - \iow_shipout_x:Nn 188, 188, 188, 188, 563, 563, 563, 564
 - \iow_shipout_x:Nx 563
 - \l_iow_stream_tl 561, 561, 562, 562, 562
 - \g_iow_streams_prop 561, 561, 562, 562, 563, 563
 - \g__iow_streams_seq 561, 561, 561, 562, 563, 563
 - \l_iow_target_count_int 565, 565, 568, 568, 568, 569, 569
 - \iow_term:n 187, 187, 508, 508, 508, 523, 564, 564
 - \iow_term:x ... 277, 277, 507, 564, 564
 - __iow_with:Nnn 191, 191, 506, 507, 507, 525, 525, 564, 564, 564, 564
 - __iow_with_aux:nNnn .. 564, 564, 564
 - \iow_wrap:nnnN 166, 166, 166, 188, 188, 189, 189, 189, 189, 189, 287, 367, 506, 506, 508, 508, 511, 522, 523, 524, 524, 525, 567, 567, 568, 568, 571
 - __iow_wrap_end: 571
 - __iow_wrap_end:w 570
 - \c__iow_wrap_end_marker_tl . 566, 568
 - __iow_wrap_indent: 570
 - __iow_wrap_indent:w 570
 - \c__iow_wrap_indent_marker_tl ... 566, 567
 - __iow_wrap_loop:w 568, 569, 569, 569, 570
 - \c__iow_wrap_marker_tl 566, 566, 566, 566, 566, 569, 570
 - __iow_wrap_newline: 570
 - __iow_wrap_newline:w 570
 - \c__iow_wrap_newline_marker_tl .. 566, 567, 568
 - __iow_wrap_set:Nx 567, 567, 568
 - __iow_wrap_set_target: 568, 568, 568, 569, 569, 569, 570, 570
 - __iow_wrap_special:w 569, 570, 570, 570
 - \l_iow_wrap_tl 566, 566, 567, 567, 568, 568, 570, 570, 571
 - __iow_wrap_unindent: 571
 - __iow_wrap_unindent:w 570
 - \c__iow_wrap_unindent_marker_tl . 566, 567
 - __iow_wrap_word: 569, 569, 569
 - __iow_wrap_word_fits: . 569, 569, 569
 - __iow_wrap_word_newline: 569, 569, 570
- J**
- \j 807
 - \jcharwidowpenalty 258
 - \jfam 258
 - \jfont 258
 - \jis 258
 - job commands:
 - \c_job_name_tl 819
 - \jobname 244
- K**
- \kanjiskip 258
 - \kansuji 258
 - \kansujichar 258

[illegible]

- 255, 255, 255, 255, 255, 255, 255,
- 255, 256, 256, 256, 256, 256, 256,
- 256, 256, 256, 256, 256, 256, 256,
- 256, 256, 256, 256, 256, 256, 256,
- 256, 256, 256, 256, 256, 256, 256,
- 256, 256, 256, 256, 256, 256, 256,
- 256, 256, 256, 256, 256, 256, 256,
- 256, 256, 256, 256, 256, 256, 256,
- 256, 256, 257, 257, 257, 257, 257,
- 257, 257, 257, 257, 257, 257, 257,
- 257, 257, 257, 257, 257, 257, 257,
- 257, 257, 257, 257, 257, 257, 257,
- 257, 257, 257, 257, 257, 257, 257,
- 257, 257, 257, 257, 257, 257, 257,
- 257, 258, 258, 258, 258, 258, 258,
- 258, 258, 258, 258, 258, 258, 258,
- 258, 258, 258, 258, 258, 258, 258,
- 258, 258, 258, 258, 258, 258, 258,
- 258, 258, 258, 258, 258, 258, 258,
- _kernel_register_log:c 760, 785, 785, 786
- _kernel_register_log:N ... 212,
- 212, 760, 760, 760, 781, 785, 785, 786
- _kernel_register_show:c .. 287, 287
- _kernel_register_show:N 25, 25, 212, 287, 287,
- 287, 367, 376, 380, 383, 413, 760, 760
- keys commands:
- _keys_bool_set:cn ... 533, 538, 538
- _keys_bool_set:Nn 533, 533, 533, 538, 538
- _keys_bool_set_inverse:cn 533, 538, 538
- _keys_bool_set_inverse:Nn 533, 533, 534, 538, 538
- _keys_check_groups: 545, 545
- _keys_choice_find:n 534, 547, 547, 547
- \l_keys_choice_int 173, 175, 177, 177, 177,
- 177, 179, 529, 529, 535, 535, 535, 535
- _keys_choice_make: 533, 533, 534, 534, 535, 538
- _keys_choice_make:N 534, 534, 534, 534
- _keys_choice_make_aux:N 534, 534, 534, 534
- \l_keys_choice_tl 173,
- 175, 177, 177, 177, 179, 529, 529, 535
- _keys_choices_make:nn 535, 535, 539, 539, 539, 539
- _keys_choices_make:Nnn 535, 535, 535, 535
- _keys_cmd_set:nn 533, 534, 534, 534, 535, 535, 535, 539
- _keys_cmd_set:nx 533, 533, 533, 534, 535, 535, 538
- _keys_cmd_set:Vn 535, 537
- _keys_cmd_set:Vo 535, 537
- \c_keys_code_root_tl 529, 529, 533,
- 533, 535, 537, 547, 547, 548, 548, 548
- _keys_default_set:n 533, 534, 535, 535, 539, 539, 539, 539
- \keys_define:nn 172, 172, 172, 531, 531, 549
- _keys_define:nnn 531, 531, 531
- _keys_define:onn 531, 531
- _keys_define_elt:n .. 531, 531, 531
- _keys_define_elt:nn . 531, 531, 531
- _keys_define_elt_aux:nn 531, 531, 531, 531
- _keys_define_key:n .. 531, 532, 532
- _keys_define_key:w .. 532, 532, 532
- _keys_ensure_exist:n . 532, 533, 533
- _keys_ensure_exist:V 532, 535, 536, 536, 536, 537
- _keys_execute: 545, 546, 546
- _keys_execute:nn 546, 546, 547, 547, 547, 547
- _keys_execute_unknown: 546, 546, 546
- \l_keys_filtered_bool 530, 530, 543, 543, 545, 545, 546, 546
- _keys_find_key_module:w 543, 544, 544, 544
- \l_keys_groups_clist 529, 529, 536, 536, 536, 545, 545
- _keys_groups_set:n .. 536, 536, 540
- \keys_if_choice_exist:nnn 548
- \keys_if_choice_exist:nnnTF 182, 182, 548
- \keys_if_choice_exist_p:nnn 182, 182, 548
- \keys_if_exist:nn 547
- \keys_if_exist:nn(TF) 548
- \keys_if_exist:nnTF 181, 181, 547, 548
- \keys_if_exist_p:nn ... 181, 181, 547
- _keys_if_value:n 546
- _keys_if_value_p:n .. 544, 544, 546

```

\c__keys_info_root_tl .....
    .... 529, 529, 533, 533, 534, 534,
    534, 536, 536, 536, 536, 537, 537,
    537, 537, 545, 545, 546, 546, 546, 548
\__keys_initialise:n .....
    .... 536, 536, 540, 540, 540, 540
\__keys_initialise:wn . 536, 536, 536
\l_keys_key_tl .....
    179, 179, 529, 529, 533, 534, 544, 547
\keys_log:nn ..... 219, 219, 781, 781
\__keys_meta_make:n ... 536, 536, 540
\__keys_meta_make:nn .. 536, 537, 541
\l__keys_module_tl .....
    .... 529, 529, 531, 531,
    531, 532, 537, 542, 542, 542, 543,
    544, 544, 544, 544, 544, 544, 547, 547
\__keys_multichoice_find:n .....
    .... 534, 547, 547
\__keys_multichoice_make: .....
    .... 534, 534, 535, 541
\__keys_multichoices_make:nn ...
    .... 535, 535, 541, 541, 541, 541
\l__keys_no_value_bool .....
    .... 530, 530, 531,
    531, 532, 543, 544, 544, 544, 546, 547
\l__keys_only_known_bool .....
    .... 530, 530, 542, 543, 547
\__keys_parent:n ..... 534, 534, 535
\__keys_parent:o ... 534, 534, 534, 534
\__keys_parent:wn .. 534, 534, 535, 535
\l_keys_path_tl ..... 179, 179,
    530, 530, 531, 532, 532, 532, 532,
    532, 532, 533, 533, 533, 533, 534,
    534, 534, 534, 534, 534, 534, 534,
    534, 534, 535, 535, 536, 536, 536,
    536, 536, 536, 536, 536, 537,
    537, 537, 537, 537, 537, 537, 537,
    538, 539, 544, 544, 544, 545, 545,
    545, 546, 546, 546, 546, 547, 547, 547
\__keys_property_find:n 531, 531, 531
\__keys_property_find:w .....
    .... 531, 532, 532, 532
\l__keys_property_tl 530, 530, 531,
    531, 531, 532, 532, 532, 532, 532, 532
\c__keys_props_root_tl .....
    .... 529, 529, 531, 532,
    532, 538, 538, 538, 538, 538, 538,
    538, 538, 538, 539, 539, 539, 539,
    539, 539, 539, 539, 539, 539, 539,
    539, 539, 539, 539, 539, 539, 540,
    540, 540, 540, 540, 540, 540, 540,
    540, 540, 540, 540, 540, 541,
    541, 541, 541, 541, 541, 541, 541,
    541, 541, 541, 541, 541, 541,
    541, 541, 541, 542, 542, 542, 550, 550
\l__keys_selective_bool .....
    .... 530, 530, 543, 543, 543, 543, 544
\l__keys_selective_seq .....
    .... 530, 530, 543, 543, 545
\keys_set:nn ..... 171,
    175, 179, 179, 179, 179, 180, 536,
    537, 537, 542, 542, 542, 542, 543, 543
\__keys_set:nnn ..... 542, 542, 542
\keys_set:no ..... 542
\keys_set:nV ..... 542
\keys_set:nv ..... 542
\__keys_set:onn ..... 542, 542
\__keys_set_elt:n ..... 542, 543, 543
\__keys_set_elt:nn .... 542, 543, 543
\__keys_set_elt_aux: .....
    .... 543, 544, 544, 545, 545, 546, 546
\__keys_set_elt_aux:nnn 543, 544, 544
\__keys_set_elt_aux:onn 543, 543, 544
\__keys_set_elt_selective: .....
    .... 543, 544, 545
\keys_set_filter:nnn .....
    .... 181, 181, 543, 543, 543, 543
\keys_set_filter:nnnN .....
    .... 181, 181, 181, 543, 543, 543
\__keys_set_filter:nnnnN 543, 543, 543
\keys_set_filter:nnV ..... 543
\keys_set_filter:nnv\keys_-
    set_filter:nno ..... 543
\keys_set_filter:nnVN ..... 543
\keys_set_filter:nnvN\keys_-
    set_filter:nnoN ..... 543
\__keys_set_filter:onnnN ... 543, 543
\keys_set_groups:nnn .....
    .... 181, 181, 543, 543, 543
\keys_set_groups:nnV ..... 543
\keys_set_groups:nnv\keys_-
    set_groups:nno ..... 543
\keys_set_known:nn .....
    .... 180, 180, 542, 542, 542, 543
\keys_set_known:nnN .....
    180, 180, 180, 180, 542, 542, 542, 543
\__keys_set_known:nnnN . 542, 542, 542
\keys_set_known:no ..... 542
\keys_set_known:noN ..... 542
\keys_set_known:nV ..... 542

```

\keys_set_known:nv	542		
\keys_set_known:nVN	542		
\keys_set_known:nvN	542		
_keys_set_known:onnN	542, 542		
_keys_show:NN	548, 548, 548		
\keys_show:nn	182, 182, 548, 548, 781, 781		
_keys_store_unused:	545, 545, 546, 546, 546, 547, 547		
\l_keys_tmp_bool	530, 530, 545, 545, 545		
_keys_undefine:	537, 537, 542		
\l_keys_unused_clist	530, 530, 542, 542, 542, 542, 543, 543, 543, 547		
_keys_value_or_default:n	544, 546, 546		
_keys_value_requirement:nn	537, 537, 542, 542, 550, 550		
\l_keys_value_tl	179, 179, 530, 530, 534, 545, 546, 546, 546, 547, 547		
_keys_variable_set:cnnN	537, 539, 539, 539, 539, 540, 540, 540, 540, 541, 541, 541, 541, 541, 541		
_keys_variable_set:NnnN	537, 537, 538, 539, 539, 539, 539, 540, 540, 540, 540, 541, 541, 541, 541, 541, 541		
keyval commands:			
\l_keyval_key_tl	526, 526, 527, 527, 527, 528, 528		
\g_keyval_level_int	526, 526, 527, 528, 528, 528, 528, 528		
_keyval_parse:n	526, 526, 528		
\keyval_parse:NNn	183, 183, 183, 526, 528, 528, 531, 542		
_keyval_parse_elt:w	526, 527, 527, 527		
\l_keyval_parse_tl	526, 526, 526, 526, 527, 528		
\l_keyval_sanitise_tl	526, 526, 526, 526, 526, 526		
_keyval_split:Nn	527, 528, 528, 528, 528, 528		
_keyval_split:Nw	528, 528, 528		
_keyval_split_key:w	527, 527, 527		
_keyval_split_key_value:w	527, 527, 527		
_keyval_split_value:w	527, 528, 528		
\l_keyval_value_tl	526, 526, 528, 528		
\kuten	258, 258		
		L	
\L	806		
\l	806		
\label	805		
\language	245		
\lastbox	245		
\lastkern	245		
\lastlinefit	249		
\lastnodetype	249		
\lastpenalty	245		
\lastskip	245		
\latelua	254		
LaTeX3 error commands:			
\LaTeX3_error:	521, 521		
\lccode	245		
\leaders	245		
\left	245		
left commands:			
\c_left_brace_str	117, 426, 427		
\leftghost	254		
\lefthyphenmin	245		
\leftmarginkern	252		
\leftskip	245		
\leqno	245		
\let	1, 235, 241, 241, 241, 245		
\letterspacefont	252		
\limits	245		
\LineBreak	237, 237, 237, 237, 237, 237, 237, 237, 237, 238, 238, 238, 238, 238, 238, 238		
\linepenalty	245		
\lineskip	245		
\lineskiplimit	245		
\linewidth	482, 483		
\ln	720, 720, 720, 720		
ln	205		
\localbrokenpenalty	254		
\localinterlinepenalty	254		
\localleftbox	254		
\localrightbox	254		
\loccount	557, 561		
log commands:			
\c_log_iow	190, 561, 561, 561, 564, 564		
\long	241, 245		
\LongText	237, 238, 238		
\looseness	245		
\lower	245		
\lowercase	245		
\lpcode	252		

lua commands:

`\lua_escape:n` . 231, 231, 820, 820, 820
`\lua_escape_x:n`
 231, 231, 231, 820, 820, 820, 820
`\lua_now:n` . 230, 230, 230, 820, 820, 820
`\lua_now_x:n` 230,
 230, 230, 236, 237, 820, 820, 820, 820
`\lua_shipout:n`
 230, 230, 230, 820, 820, 820
`\lua_shipout_x` 820
`\lua_shipout_x:n` 230, 230, 820, 820, 820
`\luaescapestring` 254
`\luafunction` 254
`\luastartup` 254

luatex commands:

`\luatex_...` 9
`\luatex_alignmark:D` 253, 259
`\luatex_aligntab:D` 253, 259
`\luatex_attribute:D` 253, 259
`\luatex_attributedef:D` 253, 259
`\luatex_bodydir:D` 254, 259, 261
`\luatex_boxdir:D` 254, 260
`\luatex_catcodetable:D` 253, 259
`\luatex_chardp:D` 260
`\luatex_charht:D` 260
`\luatex_charit:D` 260
`\luatex_charwd:D` 260
`\luatex_clearmarks:D` 253, 259
`\luatex_crampeddisplaystyle:D` .
 253, 259
`\luatex_crampedscriptscriptstyle:D`
 254, 259
`\luatex_crampedscriptstyle:D` 254, 259
`\luatex_crampedtextstyle:D` . 254, 259
`\luatex_directlua:D`
 254, 415, 813, 813, 813, 820
`\luatex_expanded:D` 254, 261, 415
`\luatex_fontid:D` 254, 259
`\luatex_formatname:D` 254, 259
`\luatex_gleaders:D` 254, 259
`\luatex_if_engine:` 819
`\luatex_initcatcodetable:D` . 254, 259
`\luatex_latelua:D` 254, 259, 820
`\luatex_leftghost:D` 254, 260
`\luatex_localbrokenpenalty:D` 254, 260
`\luatex_localinterlinepenalty:D` .
 254, 260
`\luatex_localleftbox:D` 254, 260
`\luatex_localrightbox:D` 254, 260

`\luatex_luaescapestring:D`
 254, 259, 415, 415, 820
`\luatex_luafunction:D` 254, 259
`\luatex_luastartup:D` 254, 259
`\luatex luatexdatestamp:D` 254
`\luatex luatexrevision:D` 254
`\luatex luatexversion:D`
 254, 260, 261, 264, 348, 415, 817
`\luatex_mathdir:D` 254, 260
`\luatex_mathstyle:D` 254, 259
`\luatex_nokerns:D` 254, 259
`\luatex_noligs:D` 254, 259
`\luatex_outputbox:D` 254, 259
`\luatex_pagebottomoffset:D` . 254, 260
`\luatex_pagedir:D` 254, 260, 261
`\luatex_pageheight:D` 260
`\luatex_pageleftoffset:D` ... 254, 259
`\luatex_pagerightoffset:D` .. 254, 260
`\luatex_pagetopoffset:D` 254, 259
`\luatex_pagewidth:D` 260
`\luatex_pardir:D` 254, 260
`\luatex_postexhyphenchar:D` . 254, 259
`\luatex_posthyphenchar:D` ... 254, 259
`\luatex_preexhyphenchar:D` .. 254, 259
`\luatex_prehyphenchar:D` 254, 259
`\luatex_rightghost:D` 254, 260
`\luatex_savecatcodetable:D` . 254, 259
`\luatex_scantextokens:D` 254, 259
`\luatex_suppressifcurnameerror:D` .
 254, 259
`\luatex_suppresslongerror:D` 254, 259
`\luatex_suppressmathparerror:D` ..
 254, 259
`\luatex_suppressoutererror:D` 254, 259
`\luatex_textdir:D` 254, 260
`\luatexalignmark` 259
`\luatexaligntab` 259
`\luatexattribute` 259
`\luatexattributedef` 259
`\luatexbodydir` 259
`\luatexboxdir` 260
`\luatexcatcodetable` 259, 259
`\luatexchardp` 260
`\luatexcharht` 260
`\luatexcharit` 260
`\luatexcharwd` 260
`\luatexclearmarks` 259
`\luatexcrampeddisplaystyle` 259
`\luatexcrampedscriptscriptstyle` ... 259
`\luatexcrampedscriptstyle` 259

<code>\luatexcrampedtextstyle</code>	259	mark commands:	
<code>\luatexdatestamp</code>	254	<code>\q_mark</code>	25,
<code>\luatexfontid</code>	259		25, 47, 108, 108, 274, 274, 274, 274,
<code>\luatexformatname</code>	259		274, 274, 274, 274, 299, 299, 299,
<code>\luatexgleaders</code>	259		299, 299, 299, 300, 301, 302, 302,
<code>\luatexinitcatcodetable</code>	259		302, 302, 302, 303, 303, 304, 304,
<code>\luatexlatalua</code>	259		304, 310, 310, 310, 310, 310, 310,
<code>\luatexleftghost</code>	260		310, 310, 310, 310, 310, 310, 318,
<code>\luatexlocalbrokenpenalty</code>	260		318, 352, 352, 354, 354, 373, 373,
<code>\luatexlocalinterlinepenalty</code>	260		393, 393, 393, 393, 394, 394, 400,
<code>\luatexlocalleftbox</code>	260		400, 401, 401, 401, 403, 403, 403,
<code>\luatexlocalrightbox</code>	260		403, 404, 404, 404, 404, 404, 404,
<code>\luatexluaescapestring</code>	259		404, 404, 404, 404, 405, 405, 405,
<code>\luatexluafunction</code>	259		405, 417, 417, 418, 418, 418, 418,
<code>\luatexluastartup</code>	259		418, 419, 425, 425, 425, 425, 442,
<code>\luatexmathdir</code>	260		442, 442, 442, 447, 447, 447, 447,
<code>\luatexmathstyle</code>	259		447, 449, 449, 449, 449, 449, 450,
<code>\luatexnokerns</code>	259		451, 451, 451, 452, 452, 452, 453,
<code>\luatexnoligs</code>	259		453, 453, 453, 453, 453, 453, 453,
<code>\luatexoutputbox</code>	259		453, 453, 454, 454, 454, 455, 455,
<code>\luatexpagebottomoffset</code>	260		458, 458, 458, 458, 458, 458, 458,
<code>\luatexpagedir</code>	260		460, 463, 463, 463, 463, 513, 513,
<code>\luatexpageheight</code>	260		513, 513, 608, 608, 608, 816, 816, 816
<code>\luatexpageleftoffset</code>	259	<code>\marks</code>	249
<code>\luatexpagerightoffset</code>	260	math commands:	
<code>\luatexpagetopoffset</code>	259	<code>\c_math_subscript_token</code>	
<code>\luatexpagewidth</code>	260		56, 328, 328, 330, 330
<code>\luatexpardir</code>	260	<code>\c_math_superscript_token</code>	
<code>\luatexpostexhyphenchar</code>	259		56, 328, 328, 330, 330
<code>\luatexpostthyphenchar</code>	259	<code>\c_math_toggle_token</code>	
<code>\luatexpreehyphenchar</code>	259		56, 328, 328, 329, 329
<code>\luatexprehyphenchar</code>	259	<code>\mathaccent</code>	245
<code>\luatexrevision</code>	254	<code>\mathbin</code>	245
<code>\luatexrightghost</code>	260	<code>\mathchar</code>	245, 333
<code>\luatexsavecatcodetable</code>	259	<code>\mathchardef</code>	245
<code>\luatexscantextokens</code>	259	<code>\mathchoice</code>	245
<code>\luatexsuppressfontnotfounderror</code> ..	259	<code>\mathclose</code>	245
<code>\luatexsuppressifcsnameerror</code>	259	<code>\mathcode</code>	245
<code>\luatexsuppresslongerror</code>	259	<code>\mathdir</code>	254
<code>\luatexsuppressmathparerror</code>	259	<code>\mathinner</code>	245
<code>\luatexsuppressoutererror</code>	259	<code>\mathop</code>	245
<code>\luatextextdir</code>	260	<code>\mathopen</code>	245
<code>\luatexUchar</code>	259	<code>\mathord</code>	245
<code>\luatexversion</code>	235, 236, 238, 254	<code>\mathpunct</code>	245
		<code>\mathrel</code>	245
		<code>\mathstyle</code>	254
		<code>\mathsurround</code>	245
		max	205
		max commands:	
<code>\mag</code>	245	<code>\c__max_constdef_int</code>	348, 348, 348, 349
<code>\mark</code>	245		

M

- \c_max_dim 86, 89, 377, 377, 380, 775, 775, 775, 775, 775
- \c_max_int 76, 368, 368, 474, 474, 474, 474
- \c_max_muskip 92, 383, 383
- \c_max_register_int 76, 264, 264, 264, 345, 518
- \c_max_skip 89, 380, 380
- \maxdeadcycles 245
- \maxdepth 245
- \meaning 245
- \medmuskip 245
- \message 245
- \MessageBreak 238
- meta commands:
 - .meta:n 175, 540
 - .meta:nn 175, 541
- \middle 249
- min 205
- minus commands:
 - \c_minus_inf_fp 199, 208, 575, 575, 664, 667, 704, 727
 - \c_minus_one 76, 264, 264, 264, 264, 264, 277, 283, 348, 350, 367, 390, 392, 475, 507, 508, 525, 559, 560, 561, 563, 566, 567, 632, 632, 640, 648, 654, 687, 722, 722, 723, 723, 744
 - \c_minus_zero_fp 199, 575, 575, 664, 748
- \mkern 245
- mm 208
- mode commands:
 - \mode_if_horizontal: 317
 - \mode_if_horizontal:TF ... 44, 44, 317
 - \mode_if_horizontal_p: ... 44, 44, 317
 - \mode_if_inner: 317
 - \mode_if_inner:TF 44, 44, 317
 - \mode_if_inner_p: 44, 44, 317
 - \mode_if_math: 317
 - \mode_if_math:TF 44, 44, 317
 - \mode_if_math_p: 44, 317
 - \mode_if_vertical: 317
 - \mode_if_vertical:TF ... 44, 44, 317
 - \mode_if_vertical_p: ... 44, 44, 317
- \month 245
- \moveleft 245
- \moveright 245
- msg commands:
 - _msg_class_chk_exist:nT 511, 511, 512, 514, 514, 514
 - \l_msg_class_loop_seq 512, 512, 514, 514, 515, 515, 515, 515
 - _msg_class_new:nn 508, 508, 509, 510, 510, 510, 511, 511, 511, 516
 - \l_msg_class_tl 511, 511, 512, 512, 512, 513, 513, 513, 513, 514, 515, 515, 515, 515
 - \c_msg_coding_error_text_tl ... 170, 501, 501, 504, 504, 517, 517, 518, 518, 518, 518, 518, 519, 519, 519, 519, 520, 549, 549, 550, 550
 - \c_msg_continue_text_tl 504, 504, 506
 - \msg_critical:nn 163, 509
 - \msg_critical:nnn 163, 509
 - \msg_critical:nnnn 163, 509
 - \msg_critical:nnnnn 163, 509
 - \msg_critical:nnxxx 509
 - \msg_critical:nnxxx 509
 - \msg_critical:nnxxx 509
 - \msg_critical_text:n 161, 161, 508, 508, 510
 - \c_msg_critical_text_tl 504, 505, 510
 - \l_msg_current_class_tl 511, 511, 512, 513, 513, 513, 513, 514, 514, 515
 - _msg_error:cnnnnn ... 510, 510, 510
 - \msg_error:nn 163, 510
 - \msg_error:nnn 163, 510
 - \msg_error:nnnn 163, 510
 - \msg_error:nnnnn 163, 510
 - \msg_error:nnnnnn .. 163, 163, 219, 510
 - \msg_error:nnx 510
 - \msg_error:nnxxx 510
 - \msg_error:nnxxxx 510
 - _msg_error_code:nnnnnn 517
 - \msg_error_text:n 161, 161, 508, 508, 510
 - _msg_expandable_error:n 169, 169, 521, 521, 522, 522
 - \msg_expandable_error:nn 219, 782, 782
 - \msg_expandable_error:nnf 782
 - \msg_expandable_error:nnff 782
 - \msg_expandable_error:nnfff ... 782
 - \msg_expandable_error:nnn 219, 782, 782, 782
 - \msg_expandable_error:nnnn 219, 782, 782, 782

```

\msg_expandable_error:nnnnn ....
..... 219, 782, 782, 782
\msg_expandable_error:nnnnnn 219,
219, 782, 782, 782, 782, 782, 782
\_msg_expandable_error:w .....
..... 521, 521, 521, 521
\_msg_expandable_error_module:nn
..... 782, 782, 782
\msg_fatal:nn ..... 162, 509
\msg_fatal:nnn ..... 162, 509
\msg_fatal:nnnn ..... 162, 509
\msg_fatal:nnnnn ..... 162, 509
\msg_fatal:nnnnnn ..... 162, 162, 509
\msg_fatal:nnx ..... 509
\msg_fatal:nnxx ..... 509
\msg_fatal:nnxxx ..... 509
\msg_fatal:nnxxxx ..... 509
\_msg_fatal_code:nnnnnn ..... 516
\msg_fatal_text:n .....
..... 161, 161, 508, 508, 509
\c__msg_fatal_text_tl . 504, 505, 509
\msg_gset:nnn ..... 161, 504, 504
\msg_gset:nnnn 161, 504, 504, 504, 504
\c__msg_help_text_tl . 504, 505, 506
\l__msg_hierarchy_seq .....
..... 512, 512, 512, 512, 513, 513, 513
\msg_if_exist:nn ..... 503
\msg_if_exist:nnT ..... 503, 503
\msg_if_exist:nnTF . 161, 161, 503, 512
\msg_if_exist_p:nn .... 161, 161, 503
\msg_info:nn ..... 163, 511
\msg_info:nnn ..... 163, 511
\msg_info:nnnn ..... 163, 511
\msg_info:nnnnnn ..... 163, 511
\msg_info:nnnnnnn ... 163, 163, 164, 511
\msg_info:nnx ..... 511
\msg_info:nnxx ..... 511
\msg_info:nnxxx ..... 511
\msg_info:nnxxxx ..... 511, 517
\msg_info_text:n 162, 162, 508, 508, 511
\l__msg_internal_tl .....
..... 503, 503, 524, 525, 525, 525, 525
\msg_interrupt:nnn .....
..... 166, 166, 505, 505, 509, 510, 510
\_msg_interrupt_more_text:n ...
..... 506, 506, 506, 506
\_msg_interrupt_text:n 506, 506, 507
\_msg_interrupt_wrap:nn .....
..... 506, 506, 506, 506

\_msg_kernel_class_new:nN .....
..... 516, 516, 516, 517, 517, 517, 517
\_msg_kernel_class_new_aux:nN ..
..... 516, 516, 516
\_msg_kernel_error:nn .....
..... 167, 278, 278, 488, 516, 517, 527
\_msg_kernel_error:nnn 167, 516, 820
\_msg_kernel_error:nnnn ... 167, 516
\_msg_kernel_error:nnnnn .. 167, 516
\_msg_kernel_error:nnnnnn .....
..... 167, 167, 516
\_msg_kernel_error:nnx .....
..... 269, 270, 271, 272, 278, 278,
279, 280, 285, 300, 322, 394, 475,
480, 511, 516, 517, 524, 532, 533,
534, 537, 544, 553, 554, 558, 574, 588
\_msg_kernel_error:nnxx 269, 270,
272, 278, 278, 278, 278, 279, 279,
284, 303, 485, 503, 504, 512, 516,
517, 531, 532, 534, 534, 545, 547, 588
\_msg_kernel_error:nnxxx ..... 516
\_msg_kernel_error:nnxxxx . 303, 516
\_msg_kernel_expandable_-
error:nn .....
..... 168, 316, 428, 461, 522, 522,
567, 610, 635, 812, 812, 812, 812, 812
\_msg_kernel_expandable_-
error:nnn ..... 168,
292, 351, 356, 403, 442, 458, 522,
522, 611, 611, 611, 613, 614, 614,
626, 626, 626, 626, 628, 633, 634, 820
\_msg_kernel_expandable_-
error:nnnn .....
..... 168, 522, 522, 638, 639, 651
\_msg_kernel_expandable_-
error:nnnnn .....
..... 168, 522, 522, 591, 642, 741
\_msg_kernel_expandable_-
error:nnnnnn ..... 168,
168, 522, 522, 522, 522, 522, 522
\_msg_kernel_fatal:nn .....
..... 167, 516, 559, 562
\_msg_kernel_fatal:nnn .... 167, 516
\_msg_kernel_fatal:nnnn ... 167, 516
\_msg_kernel_fatal:nnnnn .. 167, 516
\_msg_kernel_fatal:nnnnnn .....
..... 167, 167, 516
\_msg_kernel_fatal:nnx ..... 516
\_msg_kernel_fatal:nnxx ..... 516
\_msg_kernel_fatal:nnxxx ..... 516

```

- _msg_kernel_fatal:nnxxxx [516](#)
- _msg_kernel_info:nn [168](#), [517](#)
- _msg_kernel_info:nnn [168](#), [517](#)
- _msg_kernel_info:nnnn [168](#), [517](#)
- _msg_kernel_info:nnnnn [168](#), [517](#)
- _msg_kernel_info:nnnnnn [168](#), [168](#), [517](#)
- _msg_kernel_info:nnx [517](#)
- _msg_kernel_info:nnxx [517](#)
- _msg_kernel_info:nnxxx [517](#)
- _msg_kernel_info:nnxxxx [517](#)
- _msg_kernel_new:nnn [167](#),
[501](#), [515](#), [515](#), [520](#), [520](#), [520](#), [520](#),
[520](#), [520](#), [520](#), [520](#), [520](#), [520](#), [550](#),
[571](#), [592](#), [592](#), [592](#), [592](#), [592](#), [592](#),
[643](#), [643](#), [643](#), [643](#), [643](#), [643](#), [643](#),
[643](#), [643](#), [644](#), [815](#), [815](#), [815](#), [815](#), [815](#)
- _msg_kernel_new:nnnn [167](#),
[167](#), [501](#), [501](#), [501](#), [515](#), [515](#), [517](#),
[517](#), [517](#), [517](#), [518](#), [518](#), [518](#), [518](#),
[518](#), [518](#), [519](#), [519](#), [519](#), [519](#), [519](#),
[519](#), [520](#), [529](#), [549](#), [549](#), [549](#), [549](#),
[549](#), [549](#), [549](#), [549](#), [550](#), [550](#), [550](#),
[571](#), [571](#), [571](#), [571](#), [586](#), [591](#), [592](#), [820](#)
- _msg_kernel_set:nnn [167](#), [515](#), [515](#)
- _msg_kernel_set:nnnn
. [167](#), [167](#), [515](#), [515](#)
- _msg_kernel_warning:nn [168](#), [517](#)
- _msg_kernel_warning:nnn [168](#), [517](#)
- _msg_kernel_warning:nnnn [168](#), [517](#)
- _msg_kernel_warning:nnnnn [168](#), [517](#)
- _msg_kernel_warning:nnnnnn
. [168](#), [168](#), [517](#)
- _msg_kernel_warning:nnx [517](#)
- _msg_kernel_warning:nnxx [517](#)
- _msg_kernel_warning:nnxxx [517](#)
- _msg_kernel_warning:nnxxxx [515](#), [517](#)
- \msg_line_context:
. [161](#), [161](#), [278](#), [278](#),
[279](#), [304](#), [504](#), [505](#), [505](#), [505](#), [519](#), [533](#)
- \msg_line_number:
. [161](#), [161](#), [505](#), [505](#), [505](#), [529](#)
- \msg_log:n [166](#), [166](#), [507](#), [507](#), [511](#)
- \msg_log:nn [164](#), [511](#)
- \msg_log:nnn [164](#), [511](#)
- \msg_log:nnnn [164](#), [511](#)
- \msg_log:nnnnn [164](#), [511](#)
- \msg_log:nnnnnn [164](#), [164](#), [511](#)
- \msg_log:nnx [511](#)
- \msg_log:nnxx [511](#)
- \msg_log:nnxxx [511](#)
- \msg_log:nnxxxx [511](#)
- _msg_log_next:
[169](#), [169](#), [169](#), [169](#), [522](#), [522](#), [759](#),
[760](#), [760](#), [760](#), [771](#), [771](#), [779](#), [780](#),
[780](#), [781](#), [781](#), [781](#), [781](#), [781](#), [782](#),
[782](#), [783](#), [785](#), [785](#), [785](#), [786](#), [811](#), [811](#)
- \g_msg_log_next_bool [522](#),
[522](#), [522](#), [523](#), [523](#), [524](#), [525](#), [525](#), [525](#)
- _msg_log_wrap:n [523](#)
- \c_msg_more_text_prefix_tl
. [503](#), [503](#), [504](#), [504](#), [510](#)
- \msg_new:nnn [160](#), [504](#), [504](#), [515](#)
- \msg_new:nnnn
[160](#), [160](#), [503](#), [503](#), [504](#), [504](#), [515](#)
- \c_msg_no_info_text_tl [504](#), [505](#), [506](#)
- _msg_no_more_text:nnnn [510](#), [510](#), [510](#)
- \msg_none:nn [164](#), [511](#)
- \msg_none:nnn [164](#), [511](#)
- \msg_none:nnnn [164](#), [511](#)
- \msg_none:nnnnn [164](#), [511](#)
- \msg_none:nnnnnn [164](#), [164](#), [511](#)
- \msg_none:nnx [511](#)
- \msg_none:nnxx [511](#)
- \msg_none:nnxxx [511](#)
- \msg_none:nnxxxx [511](#)
- \c_msg_on_line_text_tl [504](#), [505](#), [505](#)
- _msg_redirect:nnn [514](#), [514](#), [514](#), [514](#)
- \msg_redirect_class:nn
. [165](#), [165](#), [514](#), [514](#)
- _msg_redirect_loop_chk:nnn
. [514](#), [514](#), [515](#), [515](#)
- _msg_redirect_loop_chk:onn [515](#)
- _msg_redirect_loop_list:n
. [514](#), [515](#), [515](#)
- \msg_redirect_module:nnn
. [165](#), [165](#), [514](#), [514](#)
- \msg_redirect_name:nnn
. [165](#), [165](#), [514](#), [514](#)
- \l_msg_redirect_prop
. [511](#), [511](#), [512](#), [514](#), [514](#)
- \c_msg_return_text_tl
. [504](#), [505](#), [505](#), [517](#), [517](#), [517](#)
- \msg_see_documentation_text:n
. [162](#), [162](#), [508](#), [508](#), [509](#), [510](#), [510](#)
- \msg_set:nnn [161](#), [504](#), [504](#), [515](#)
- \msg_set:nnnn
. [161](#), [161](#), [504](#), [504](#), [504](#), [515](#)
- _msg_show_item:n [170](#), [170](#),
[170](#), [444](#), [460](#), [461](#), [523](#), [524](#), [525](#), [525](#)

- _msg_show_item:nn 170, 170, 170, 470, 470, 525, 525
- _msg_show_item_unbraced:nn 170, 170, 501, 525, 525, 548, 548, 559, 560
- _msg_show_pre:nnnnnn 169, 169, 522, 523, 523, 523
- _msg_show_pre:nnnnnV 522
- _msg_show_pre:nnxxxx 460, 501, 522, 523, 524, 548, 548, 560
- _msg_show_pre_aux:n 522, 522, 523, 523
- _msg_show_variable:NNNnn 169, 169, 169, 170, 287, 287, 308, 367, 413, 444, 444, 460, 460, 470, 470, 523, 523, 523, 759, 759
- _msg_show_wrap:n 169, 169, 170, 170, 170, 287, 325, 326, 326, 327, 327, 413, 413, 460, 501, 523, 523, 524, 524, 524, 524, 524, 524, 524, 524, 525, 548, 548, 548, 560
- _msg_show_wrap:Nn 169, 170, 170, 308, 367, 377, 380, 383, 524, 524, 524, 759
- _msg_show_wrap_aux:n . 524, 525, 525
- _msg_show_wrap_aux:w . 524, 525, 525
- \msg_term:n ... 166, 166, 507, 508, 510
- \c_msg_text_prefix_tl 503, 503, 503, 504, 504, 509, 510, 510, 510, 511, 511, 522, 523, 782
- \c_msg_trouble_text_tl 504, 505
- _msg_use:nnnnnnnn 509, 512, 512
- _msg_use_code: 512, 512, 512, 512, 512, 512, 513, 513
- _msg_use_hierarchy:nwWN 512, 513, 513, 513
- _msg_use_redirect_module:n ... 512, 513, 513, 513, 513
- _msg_use_redirect_name:n 512, 512, 512
- \msg_warning:nn 163, 510
- \msg_warning:nnn 163, 510
- \msg_warning:nnnn 163, 510
- \msg_warning:nnnnn 163, 510
- \msg_warning:nnnnnn 163, 510
- \msg_warning:nnx 510
- \msg_warning:nnxx 510
- \msg_warning:nnxxx 510
- \msg_warning:nnxxxx ... 163, 510, 517
- \msg_warning_text:n 162, 162, 508, 508, 510
- \mskip 245
- \muexpr 249
- multichoice commands:
 - .multichoice: 175, 541
- multichoices commands:
 - .multichoices:nn 175, 541
 - .multichoices:on 175, 541
 - .multichoices:Vn 175, 541
 - .multichoices:xn 175, 541
- \multiply 245
- \muskip 245
- muskip commands:
 - \muskip_(g)zero:N 91
 - \muskip_add:cn 382
 - \muskip_add:Nn 91, 91, 382, 382, 382, 382
 - \muskip_const:cn 381
 - \muskip_const:Nn 90, 90, 381, 381, 381, 383, 383
 - \muskip_eval:n 92, 92, 92, 383, 383, 383
 - \muskip_gadd:cn 382
 - \muskip_gadd:Nn ... 91, 382, 382, 382
 - \muskip_gset:cn 382
 - \muskip_gset:Nn 91, 381, 382, 382, 382
 - \muskip_gset_eq:cc 382
 - \muskip_gset_eq:cN 382
 - \muskip_gset_eq:Nc 382
 - \muskip_gset_eq:NN 91, 382, 382, 382, 382
 - \muskip_gsub:cn 382
 - \muskip_gsub:Nn ... 91, 382, 382, 382
 - \muskip_gzero:c 381
 - \muskip_gzero:N 90, 381, 381, 381, 381
 - \muskip_gzero_new:c 381
 - \muskip_gzero_new:N 91, 381, 381, 381
 - \muskip_if_exist:c 382
 - \muskip_if_exist:cTF 382
 - \muskip_if_exist:N 382
 - \muskip_if_exist:NTF 91, 91, 381, 381, 382
 - \muskip_if_exist_p:c 382
 - \muskip_if_exist_p:N ... 91, 91, 382
 - \muskip_log:c 786, 786
 - \muskip_log:N 222, 222, 786, 786
 - \muskip_log:n 222, 222, 786, 786
 - \muskip_new:c 381
 - \muskip_new:N 90, 90, 91, 381, 381, 381, 381, 381, 381, 381, 383, 383, 383, 383
 - \muskip_set:cn 382
 - \muskip_set:Nn 91, 91, 382, 382, 382, 382
 - \muskip_set_eq:cc 382

- `\muskip_set_eq:cN` 382
 - `\muskip_set_eq:Nc` 382
 - `\muskip_set_eq:NN`
 - 91, 91, 382, 382, 382, 382
 - `\muskip_show:c` 383
 - `\muskip_show:N` .. 92, 92, 383, 383, 383
 - `\muskip_show:n` 92, 92, 383, 383, 786, 786
 - `\muskip_sub:cn` 382
 - `\muskip_sub:Nn` 91, 91, 382, 382, 382, 382
 - `\muskip_use:c` 383
 - `\muskip_use:N`
 - ... 92, 92, 92, 92, 383, 383, 383, 383
 - `\muskip_zero:c` 381
 - `\muskip_zero:N`
 - 90, 381, 381, 381, 381, 381
 - `\muskip_zero_new:c` 381
 - `\muskip_zero_new:N` 91, 91, 381, 381, 381
 - `\muskipdef` 245
 - `\mutoglu` 249
- N**
- `nan` 208
 - `nan` commands:
 - `\c_nan_fp` 208, 575, 575, 589, 589, 591, 591, 597, 597, 610, 611, 611, 613, 613, 614, 629, 642, 717, 741
 - `nc` 208
 - `nd` 208
 - `\newbox` 348
 - `\newcatcodetable` 236
 - `\newcount` 348
 - `\newdimen` 348
 - `\newlinechar` 238, 246
 - `\next` 63, 63, 64, 237, 238, 238, 238, 239, 239, 239
 - `\NG` 806
 - `\ng` 806
 - `nil` commands:
 - `\q_nil` 21, 21, 47, 47, 47, 47, 266, 266, 266, 310, 310, 310, 310, 310, 310, 310, 310, 310, 310, 310, 318, 318, 318, 320, 321, 321, 321, 321, 321, 321, 321, 394, 394, 396, 396, 397, 397, 397, 397, 398, 398, 404, 404, 404, 405, 405, 405, 408, 408, 526, 527, 527, 527, 527, 527, 527, 527, 527, 528, 528, 528, 528, 528, 528, 528, 528, 528, 794, 794, 794, 794, 794
 - `nine` commands:
 - `\c_nine` 76, 325, 326, 367, 367, 423, 424, 602, 607, 609, 610, 615, 616, 617, 617, 619, 620, 620, 621, 621, 622, 625, 625, 636, 636, 636, 636, 663, 729, 729, 729, 729, 729, 812
 - `no` commands:
 - `\q_no_value`
 - 46, 47, 47, 47, 47, 121, 121, 121, 121, 121, 127, 127, 127, 138, 142, 142, 142, 184, 313, 318, 318, 318, 320, 320, 321, 321, 435, 436, 436, 436, 437, 437, 449, 449, 449, 464, 464, 464, 464, 464, 553, 554
 - `\noalign` 246
 - `\noautospacing` 258
 - `\noautoxspacing` 258
 - `\noboundary` 246
 - `\noexpand` 238, 238, 238, 238, 246
 - `\noindent` 246
 - `\nokerns` 254
 - `\noligs` 254
 - `\nolimits` 246
 - `\nonscript` 246
 - `\nonstopmode` 246
 - `\normalend` 261, 261, 557, 561
 - `\normaleveryjob` 261
 - `\normalexpanded` 261
 - `\normalhoffset` 261
 - `\normalinput` 261
 - `\normalitaliccorrection` 261, 261
 - `\normallanguage` 261
 - `\normalleft` 261, 261
 - `\normalmathop` 261
 - `\normalmiddle` 261
 - `\normalmonth` 261
 - `\normalouter` 261
 - `\normalover` 261
 - `\normalright` 261
 - `\normalshowtokens` 261
 - `\normalunexpanded` 261
 - `\normalvcenter` 261
 - `\normalvoffset` 261
 - `\nulldelimiterspace` 246
 - `\nullfont` 246
 - `\number` 237, 237, 246
 - `\numexpr` 249
- O**
- `\O` 806

<code>\o</code>	806	422, 422, 422, 422, 422, 422, 577,
<code>\OCIRCUMFLEX</code>	811	577, 577, 584, 584, 596, 597, 640,
<code>\Ocircumflex</code>	810, 810	640, 640, 641, 641, 654, 654, 654,
<code>\ocircumflex</code>	810, 810, 811	658, 661, 664, 664, 664, 664, 664,
<code>\OE</code>	806	664, 664, 664, 664, 667, 667, 684,
<code>\oe</code>	806	684, 694, 704, 704, 705, 705, 705,
<code>\OHORN</code>	811	705, 705, 711, 712, 712, 712, 714,
<code>\Ohorn</code>	810, 810	714, 714, 714, 714, 714, 714, 714,
<code>\ohorn</code>	810, 810, 811	714, 714, 714, 714, 714, 715, 715,
<code>\omit</code>	246	715, 715, 715, 715, 715, 715, 715,
one commands:		715, 715, 715, 715, 715, 715, 715,
<code>\c_one</code>	76, 324, 325, 325,	715, 715, 715, 715, 715, 715, 715,
347, 350, 366, 367, 367, 403, 412,		715, 715, 715, 715, 715, 715, 715,
413, 421, 422, 424, 439, 439, 441,		715, 715, 716, 716, 716, 716, 716,
457, 457, 459, 474, 515, 515, 553,		716, 716, 716, 716, 716, 716, 716,
557, 557, 561, 561, 569, 577, 585,		716, 716, 716, 716, 716, 717, 719,
593, 593, 593, 593, 593, 593, 595,		719, 719, 722, 722, 724, 724, 725,
595, 595, 596, 596, 596, 598, 599,		725, 725, 725, 726, 726, 726, 726,
616, 618, 618, 620, 621, 621, 621,		727, 727, 741, 741, 741, 743, 746,
622, 628, 635, 635, 635, 639, 639,		746, 746, 746, 748, 748, 748, 748,
639, 639, 639, 639, 639, 648, 652,		748, 750, 750, 750, 752, 752, 752,
652, 652, 657, 658, 659, 661, 661,		753, 753, 813, 814, 814, 814, 814,
661, 662, 662, 663, 664, 666, 667,		814, 814, 814, 814, 814, 814, 814,
674, 674, 676, 677, 680, 680, 680,		
682, 682, 683, 687, 693, 697, 697,		<code>\outer</code>
699, 699, 702, 702, 704, 705, 709,		6, 6, 246, 348, 635
712, 712, 713, 721, 722, 722, 723,		<code>\output</code>
723, 723, 726, 728, 729, 739, 740,		246
741, 743, 747, 749, 751, 751, 812, 816		<code>\outputbox</code>
<code>\c_one_degree_fp</code> 199, 208, 629, 759, 759		254
<code>\c_one_fp</code> .. 199, 629, 640, 641, 649,		<code>\outputpenalty</code>
712, 717, 719, 725, 726, 741, 759, 759		246
<code>\c_one_hundred</code>	76, 368, 368	<code>\over</code>
<code>\c_one_thousand</code>	76, 368, 368	246
<code>\openin</code>	246	<code>\overfullrule</code>
<code>\openout</code>	246	246
<code>\or</code>	246	<code>\overline</code>
or commands:		246
<code>\or:</code>	77,	
77, 77, 262, 262, 283, 283, 283, 283,		
283, 283, 283, 283, 283, 345, 361,		
361, 361, 361, 361, 361, 361, 361,		
361, 361, 361, 361, 361, 361, 361,		
361, 362, 362, 362, 362, 362, 362,		
362, 362, 362, 362, 362, 362, 362,		
362, 362, 362, 362, 362, 362, 362,		
362, 362, 362, 362, 362, 362, 362,		
362, 362, 362, 362, 362, 362, 362,		
420, 420, 420, 420, 420, 420, 420,		
420, 420, 420, 420, 420, 422, 422,		
		P
		<code>\PackageError</code>
		238, 238
		<code>\pagebottomoffset</code>
		254
		<code>\pagedepth</code>
		246
		<code>\pagedir</code>
		254
		<code>\pagediscards</code>
		249
		<code>\pagefilllstretch</code>
		246
		<code>\pagefillstretch</code>
		246
		<code>\pagefilstretch</code>
		246
		<code>\pagegoal</code>
		246
		<code>\pageleftoffset</code>
		254
		<code>\pagerightoffset</code>
		254
		<code>\pageshrink</code>
		246
		<code>\pagestretch</code>
		246
		<code>\pagetopoffset</code>
		254
		<code>\pagetotal</code>
		246
		<code>\par</code>
		11, 11,
		12, 12, 13, 13, 13, 14, 14, 14, 15, 15,

15, 16, 186, 186, 246, 281, 281, 476, 476, 476, 477, 477, 477, 477, 478		
parameter commands:		
\c_parameter_token		
..... 56, 328, 328, 329, 329, 329, 329		
\paddir	254	
\parfillskip	246	
\parindent	246	
\parshape	246	
\parshapedimen	249	
\parshapeindent	249	
\parshapelength	249	
\parskip	246	
\patterns	246	
\pausing	246	
pc	208	
\pdf...	250	
\pdfadjustspacing	251	
\pdfannot	250	
\pdfcatalog	250	
\pdfcolorstack	250	
\pdfcolorstackinit	250	
\pdfcompresslevel	250	
\pdfcopyfont	251	
\pdfcreationdate	250	
\pdfdecimaldigits	250	
\pdfdest	250	
\pdfdestmargin	250	
\pdfdraftmode	251	
\pdfeachlinedepth	251	
\pdfeachlineheight	251	
\pdfendlink	250	
\pdfendthread	250	
\pdffirstlineheight	251	
\pdffontattr	250	
\pdffontexpand	251	
\pdffontname	250	
\pdffontobjnum	250	
\pdffontsize	251	
\pdfgamma	250	
\pdfgentounicode	250	
\pdfglyphtounicode	250	
\pdfhorigin	250	
\pdfignoreddimen	251	
\pdfimageapplygamma	250	
\pdfimagegamma	250	
\pdfimagehicolor	250	
\pdfimageresolution	250	
\pdfincludechars	250	
\pdfinclusioncopyfonts	250	
\pdfinclusionerrorlevel	250	
\pdfinfo	250	
\pdfinsertht	251	
\pdflastannot	250	
\pdflastlinedepth	251	
\pdflastlink	250	
\pdflastobj	250	
\pdflastxform	250	
\pdflastximage	250	
\pdflastximagecolordepth	250	
\pdflastximagepages	250	
\pdflastxpos	251	
\pdflastypos	251	
\pdflinkmargin	250	
\pdfliteral	250	
\pdfmapfile	251	
\pdfmapline	251	
\pdfminorversion	250	
\pdfnames	250	
\pdfnoligatures	251	
\pdfnormaldeviate	251	
\pdfobj	250	
\pdfobjcompresslevel	251	
\pdfoutline	251	
\pdfoutput	251	
\pdfpageattr	251	
\pdfpagebox	251	
\pdfpageheight	251	
\pdfpageref	251	
\pdfpageresources	251	
\pdfpagesattr	251	
\pdfpagewidth	251	
\pdfpkmode	252	
\pdfpkresolution	252	
\pdfprimitive	252	
\pdfprotrudechars	252	
\pdfpxdimen	252	
\pdfrandomseed	252	
\pdfrefobj	251	
\pdfrefxform	251	
\pdfrefximage	251	
\pdfrestore	251	
\pdfretval	251	
\pdfsave	251	
\pdfsavepos	252	
\pdfsetmatrix	251	
\pdfsetrandomseed	252	
\pdfshellescape	252	
\pdfstartlink	251	
\pdfstartthread	251	

<code>\pdfstrcmp</code>	235, 237, 252	<code>\pdftex_pdfimagegamma:D</code>	250
pdftex commands:		<code>\pdftex_pdfimagehicolor:D</code>	250
<code>\pdftex_...</code>	9	<code>\pdftex_pdfimageresolution:D</code> ..	250
<code>\pdftex_adjustspacing:D</code>	251	<code>\pdftex_pdfincludechars:D</code>	250
<code>\pdftex_copyfont:D</code>	251	<code>\pdftex_pdfinclusioncopyfonts:D</code>	250
<code>\pdftex_draftmode:D</code>	251	<code>\pdftex_pdfinclusionerrorlevel:D</code>	250
<code>\pdftex_eachlinedepth:D</code>	251	<code>\pdftex_pdfinfo:D</code>	250
<code>\pdftex_eachlineheight:D</code>	251	<code>\pdftex_pdflastannot:D</code>	250
<code>\pdftex_efcode:D</code>	252	<code>\pdftex_pdflastlink:D</code>	250
<code>\pdftex_firstlineheight:D</code>	251	<code>\pdftex_pdflastobj:D</code>	250
<code>\pdftex_fontexpand:D</code>	251	<code>\pdftex_pdflastxform:D</code>	250
<code>\pdftex_fontsize:D</code>	251	<code>\pdftex_pdflastximage:D</code>	250
<code>\pdftex_if_engine:F</code>	819, 819	<code>\pdftex_pdflastximagecolordepth:D</code>	
<code>\pdftex_if_engine:T</code>	819, 819	250
<code>\pdftex_if_engine:TF</code>	819, 819	<code>\pdftex_pdflastximagepages:D</code> ..	250
<code>\pdftex_if_engine:p</code>	819, 819	<code>\pdftex_pdflinkmargin:D</code>	250
<code>\pdftex_ifincsnam:D</code>	252	<code>\pdftex_pdfliteral:D</code>	
<code>\pdftex_ifprimitive:D</code>	251, 260	250, 822, 822, 823, 823
<code>\pdftex_ignoredimen:D</code>	251	<code>\pdftex_pdfminorversion:D</code> ..	250, 821
<code>\pdftex_insertht:D</code>	251	<code>\pdftex_pdfnames:D</code>	250
<code>\pdftex_lastlinedepth:D</code>	251	<code>\pdftex_pdfobj:D</code>	250
<code>\pdftex_lastxpos:D</code>	251	<code>\pdftex_pdfobjcompresslevel:D</code> ...	
<code>\pdftex_lastypos:D</code>	251	251, 821
<code>\pdftex_leftmarginkern:D</code>	252	<code>\pdftex_pdfoutline:D</code>	251
<code>\pdftex_letterspacefont:D</code>	252	<code>\pdftex_pdfoutput:D</code>	
<code>\pdftex_lpcode:D</code>	252	251, 818, 818, 821, 821
<code>\pdftex_mapfile:D</code>	251, 260	<code>\pdftex_pdfpageattr:D</code>	251
<code>\pdftex_mapline:D</code>	251, 260	<code>\pdftex_pdfpagebox:D</code>	251
<code>\pdftex_noligatures:D</code>	251	<code>\pdftex_pdfpageheight:D</code>	251
<code>\pdftex_normaldeviate:D</code>	251	<code>\pdftex_pdfpageref:D</code>	251
<code>\pdftex_pdfannot:D</code>	250	<code>\pdftex_pdfpageresources:D</code>	251
<code>\pdftex_pdfcatalog:D</code>	250	<code>\pdftex_pdfpagesattr:D</code>	251
<code>\pdftex_pdfcolorstack:D</code>		<code>\pdftex_pdfpagewidth:D</code>	251
.....	250, 827, 827, 828	<code>\pdftex_pdfpkresolution:D</code>	821
<code>\pdftex_pdfcolorstackinit:D</code> ...	250	<code>\pdftex_pdfrefobj:D</code>	251
<code>\pdftex_pdfcompresslevel:D</code> ..	250, 821	<code>\pdftex_pdfrefxform:D</code>	251
<code>\pdftex_pdfcreationdate:D</code>	250	<code>\pdftex_pdfrefximage:D</code>	251
<code>\pdftex_pdfdecimaldigits:D</code> ..	250, 821	<code>\pdftex_pdfrestore:D</code>	251, 822
<code>\pdftex_pdfdest:D</code>	250	<code>\pdftex_pdfretval:D</code>	251
<code>\pdftex_pdfdestmargin:D</code>	250	<code>\pdftex_pdfsave:D</code>	251, 822, 822
<code>\pdftex_pdfendlink:D</code>	250	<code>\pdftex_pdfsetmatrix:D</code> ..	251, 824, 824
<code>\pdftex_pdfendthread:D</code>	250	<code>\pdftex_pdfstartlink:D</code>	251
<code>\pdftex_pdffontattr:D</code>	250	<code>\pdftex_pdfstartthread:D</code>	251
<code>\pdftex_pdffontname:D</code>	250	<code>\pdftex_pdftexbanner:D</code> ..	252, 261, 261
<code>\pdftex_pdffontobjnum:D</code>	250	<code>\pdftex_pdftexrevision:D</code>	252
<code>\pdftex_pdfgamma:D</code>	250	<code>\pdftex_pdftexversion:D</code>	
<code>\pdftex_pdfgentounicode:D</code>	250	252, 260, 261, 817
<code>\pdftex_pdfglyphtounicode:D</code> ...	250	<code>\pdftex_pdfthread:D</code>	251
<code>\pdftex_pdfhorigin:D</code>	250, 821	<code>\pdftex_pdfthreadmargin:D</code>	251
<code>\pdftex_pdfimageapplygamma:D</code> ..	250	<code>\pdftex_pdftrailer:D</code>	251

- \pdfTEX_pdfuniqueresname:D 251
- \pdfTEX_pdfvorigin:D 251, 821
- \pdfTEX_pdfxform:D 251
- \pdfTEX_pdfxformattr:D 251
- \pdfTEX_pdfxformname:D 251
- \pdfTEX_pdfxformresources:D 251
- \pdfTEX_pdfximage:D 251
- \pdfTEX_pdfximagebbox:D 251
- \pdfTEX_pkmode:D 252
- \pdfTEX_pkresolution:D 252
- \pdfTEX_primitive:D 252, 260
- \pdfTEX_protrudechars:D 252
- \pdfTEX_pxdimen:D 252
- \pdfTEX_quitvmode:D 252
- \pdfTEX_randomseed:D 252
- \pdfTEX_rightmarginkern:D 252
- \pdfTEX_rpcode:D 252
- \pdfTEX_savepos:D 252
- \pdfTEX_setrandomseed:D 252
- \pdfTEX_shellescape:D 252, 260
- \pdfTEX_strcmp:D 252, 415
- \pdfTEX_synctex:D 252
- \pdfTEX_tagcode:D 252
- \pdfTEX_tracingfonts:D 252
- \pdfTEX_uniformdeviate:D 252
- \pdfTEXbanner 252
- \pdfTEXrevision 252
- \pdfTEXversion 252
- \pdfTHREAD 251
- \pdfTHREADmargin 251
- \pdfTRACINGfonts 252
- \pdfTRAILER 251
- \pdfUNIFORMdeviate 252
- \pdfUNIQUEsname 251
- \pdfVORIGIN 251
- \pdfXFORM 251
- \pdfXFORMattr 251
- \pdfXFORMname 251
- \pdfXFORMresources 251
- \pdfXIMAGE 251
- \pdfXIMAGEbbox 251
- peek commands:
 - \peek_after:Nw 44, 60, 60, 60, 60, 339, 339, 340, 340, 342
 - \peek_catcode:NTF 60, 60, 343
 - \peek_catcode_ignore_spaces:NTF 61, 61, 343
 - \peek_catcode_remove:NTF 61, 61, 343
 - \peek_catcode_remove_ignore_spaces:NTF 61, 61, 343
 - \peek_charcode:NTF 61, 61, 343
 - \peek_charcode_ignore_spaces:NTF 61, 61, 343
 - \peek_charcode_remove:NTF 62, 62, 343
 - \peek_charcode_remove_ignore_spaces:NTF 62, 62, 343
 - __peek_def:nmm 342, 342, 343, 343, 343, 343, 343, 343, 343, 343, 344, 344, 344, 344
 - __peek_def:nmmmm 342, 342, 342, 342, 343
 - __peek_execute_branches: 342, 342, 343
 - __peek_execute_branches_catcode: 341, 341, 343, 343, 343, 343
 - __peek_execute_branches_catcode_aux: 341, 341, 341, 341
 - __peek_execute_branches_catcode_auxii:N 341, 341, 341
 - __peek_execute_branches_catcode_auxiii: 341, 341, 342
 - __peek_execute_branches_charcode: 341, 341, 343, 343, 343, 343
 - __peek_execute_branches_meaning: 341, 341, 344, 344, 344, 344
 - __peek_execute_branches_N_type: 815, 815, 816, 816, 816
 - __peek_false:w 339, 339, 340, 340, 341, 342, 342, 815, 815, 816, 816
 - \peek_gafter:Nw 60, 60, 60, 339, 339
 - __peek_get_prefix_arg_replacement:wN 344, 344, 344, 344, 345
 - __peek_ignore_spaces_execute_branches: 342, 342, 342, 343, 343, 343, 344, 344
 - \peek_meaning:NTF 62, 62, 344
 - \peek_meaning_ignore_spaces:NTF 62, 62, 344
 - \peek_meaning_remove:NTF 62, 62, 344
 - \peek_meaning_remove_ignore_spaces:NTF 63, 63, 344
 - \peek_N_type:F 816
 - \peek_N_type:T 816
 - \peek_N_type:TF 227, 227, 815, 816
 - __peek_N_type:w 815, 816, 816
 - __peek_N_type_aux:nnw 815, 816, 816
 - \l__peek_search_tl 339, 339, 339, 340, 340, 341, 342, 342
 - \l__peek_search_token 339, 339, 339, 340, 340, 341

- __peek_tmp:w . 339, 339, 339, 815, 816
- \g_peek_token . . . 60, 60, 339, 339, 339
- \l_peek_token 60, 60, 339, 339, 339, 341, 341, 341, 342, 342, 342, 815, 815, 816, 816, 816, 816
- __peek_token_generic:NNTF . . 340, 816
- __peek_token_generic:NNT . . 340, 816
- __peek_token_generic:NNTF 340, 340, 340, 340, 815, 816
- __peek_token_remove_generic:NNTF 340
- __peek_token_remove_generic:NNT 340
- __peek_token_remove_generic:NNTF 340, 340, 340, 340
- __peek_true:w . 339, 339, 340, 340, 341, 342, 342, 815, 815, 816, 816, 816
- __peek_true_aux:w . 339, 339, 339, 340
- __peek_true_remove:w . 339, 339, 340
- \penalty 246
- percent commands:
 - \c_percent_str 117, 426, 427
- pi 208
- pi commands:
 - \c_pi_fp . . 199, 208, 623, 629, 759, 759
- \postbreakpenalty 258
- \postdisplaypenalty 246
- \postexhyphenchar 254
- \poshyphenchar 254
- \prebreakpenalty 258
- \predisplaydirection 249
- \predisdisplaypenalty 246
- \predisplaysize 246
- \preexhyphenchar 254
- \prehyphenchar 254
- \pretolerance 246
- \prevdepth 246
- \prevgraf 246
- prg commands:
 - __prg_break: 45, 288, 288, 318, 413, 439, 468, 585, 756, 784, 784
 - __prg_break:n 45, 45, 45, 288, 288, 318, 413, 435, 439, 465
 - __prg_break_point: 45, 45, 288, 288, 288, 288, 318, 412, 435, 439, 465, 468, 585, 756, 784, 784
 - __prg_break_point:Nn 45, 45, 45, 45, 101, 102, 125, 125, 136, 136, 217, 218, 287, 287, 287, 287, 288, 288, 318, 357, 357, 401, 402, 402, 439, 440, 441, 441, 455, 455, 456, 456, 469, 470, 780, 783
 - \prg_break_point:Nn 49
 - __prg_case_end:nw 25, 25, 354, 374, 400, 401, 401, 418
 - __prg_compare_error: 78, 78, 351, 351, 351, 351, 352, 352, 352, 372, 372, 372
 - __prg_compare_error:Nw 78, 78, 351, 351, 351, 352, 352, 353
 - \prg_do_nothing: 10, 10, 45, 271, 271, 287, 287, 288, 302, 302, 389, 390, 390, 392, 392, 395, 395, 395, 395, 395, 430, 430, 430, 430, 438, 438, 449, 459, 460, 460, 460, 586, 588, 589, 590, 590, 590, 640, 755, 755, 788
 - __prg_generate_conditional:nnNnnnnn 268, 269, 269, 269
 - __prg_generate_conditional:nnnnnnw 269, 269, 269, 270
 - __prg_generate_conditional-count:nnNnn 268, 268, 268, 268, 268, 268
 - __prg_generate_conditional-count:nnNnnnn 268, 269, 269
 - __prg_generate_conditional-parm:nnNpnn 268, 268, 268, 268, 268, 268
 - __prg_generate_F_form:wnnnnnnn 270, 271
 - __prg_generate_p_form:wnnnnnnn 270, 270
 - __prg_generate_T_form:wnnnnnnn 270, 270
 - __prg_generate_TF_form:wnnnnnnn 270, 271
 - __prg_map_1:w 45
 - __prg_map_2:w 45
 - __prg_map_break:Nn 45, 45, 287, 288, 288, 288, 318, 402, 402, 402, 439, 440, 457, 457, 457, 470, 470, 470, 779, 780
 - \g__prg_map_int 45, 45, 318, 318, 357, 357, 357, 357, 357, 357, 401, 401, 401, 401, 402, 440, 440, 441, 441, 456, 456, 456, 456, 470, 470, 470, 470, 780, 780, 780
 - \prg_new_conditional:Nnn . 37, 37, 268, 268, 305, 320, 320, 321, 321, 560
 - \prg_new_conditional:Npnn 37, 37, 38, 268, 268, 286, 305, 307, 309, 317, 317, 317,

317, 329, 329, 329, 329, 329, 329,
 330, 330, 330, 330, 330, 331, 331,
 331, 331, 332, 332, 332, 333, 333,
 334, 334, 335, 335, 335, 336, 336,
 336, 337, 342, 352, 353, 354, 354,
 372, 372, 379, 379, 396, 397, 397,
 398, 398, 400, 409, 409, 410, 411,
 411, 412, 416, 416, 416, 434, 453,
 467, 468, 473, 473, 473, 480, 503,
 546, 547, 548, 587, 626, 644, 645, 786
 \prg_new_eq_conditional:NNn
 39, 39, 271, 271, 305,
 308, 308, 349, 349, 370, 370, 378,
 378, 382, 382, 385, 385, 415, 415,
 415, 415, 431, 431, 443, 443, 443,
 443, 443, 443, 447, 447, 453, 453,
 467, 467, 472, 472, 644, 644, 819, 819
 \prg_new_protected_conditional:Nnn
 37, 37, 268, 268, 305
 \prg_new_protected_conditional:Npnn
 37, 37,
 268, 268, 305, 398, 399, 435, 438,
 438, 438, 438, 438, 438, 450, 450,
 450, 454, 454, 465, 465, 469, 554, 558
 __prg_replicate:N . 315, 316, 316, 316
 \prg_replicate:nn 43, 43,
 315, 315, 315, 315, 315, 520, 570,
 571, 692, 721, 723, 723, 729, 734,
 734, 734, 734, 734, 735, 752, 752, 752
 __prg_replicate_ 315, 316
 __prg_replicate_0:n 315
 __prg_replicate_1:n 315
 __prg_replicate_2:n 315
 __prg_replicate_3:n 315
 __prg_replicate_4:n 315
 __prg_replicate_5:n 315
 __prg_replicate_6:n 315
 __prg_replicate_7:n 315
 __prg_replicate_8:n 315
 __prg_replicate_9:n 315
 __prg_replicate_first:N 315, 315, 316
 __prg_replicate_first-:n 315
 __prg_replicate_first_0:n 315
 __prg_replicate_first_1:n 315
 __prg_replicate_first_2:n 315
 __prg_replicate_first_3:n 315
 __prg_replicate_first_4:n 315
 __prg_replicate_first_5:n 315
 __prg_replicate_first_6:n 315
 __prg_replicate_first_7:n 315
 __prg_replicate_first_8:n 315
 __prg_replicate_first_9:n 315
 \prg_return_false: 38,
 39, 39, 39, 117, 267, 267, 275, 275,
 275, 276, 276, 276, 286, 305, 307,
 309, 317, 317, 317, 317, 320, 320,
 329, 329, 329, 329, 330, 330, 330,
 330, 330, 330, 331, 331, 331, 331,
 332, 332, 332, 333, 334, 334, 334,
 334, 335, 335, 335, 335, 335, 335,
 337, 338, 338, 338, 351, 351, 352,
 353, 353, 354, 354, 355, 372, 373,
 373, 373, 379, 379, 397, 397, 398,
 398, 398, 399, 400, 409, 409, 410,
 410, 410, 411, 411, 412, 412, 416,
 416, 416, 417, 434, 435, 435, 436,
 450, 450, 453, 454, 454, 465, 466,
 467, 468, 469, 473, 473, 473, 480,
 480, 503, 546, 546, 548, 548, 554,
 558, 560, 587, 626, 626, 644, 645, 786
 \prg_return_true/false: 416
 \prg_return_true: 38, 39, 39, 39, 117,
 267, 267, 275, 275, 276, 276, 276,
 276, 286, 305, 307, 309, 317, 317,
 317, 317, 320, 320, 329, 329, 329,
 329, 330, 330, 330, 330, 330, 330,
 331, 331, 331, 331, 332, 332, 332,
 334, 334, 338, 338, 353, 353, 354,
 355, 372, 373, 379, 379, 397, 397,
 398, 398, 398, 399, 400, 409, 409,
 410, 410, 410, 411, 411, 412, 412,
 416, 416, 416, 417, 434, 435, 435,
 436, 450, 450, 454, 454, 465, 465,
 465, 467, 468, 469, 473, 473, 473,
 480, 503, 546, 548, 548, 554, 558,
 560, 560, 560, 587, 626, 626, 645, 645
 \prg_set_conditional:Nnn
 37, 268, 268, 305
 \prg_set_conditional:Npnn . . . 37,
 38, 39, 268, 268, 275, 275, 276, 276, 305
 \prg_set_eq_conditional:NNn
 39, 271, 271, 305
 __prg_set_eq_conditional:NNNn . .
 271, 271, 271, 271
 __prg_set_eq_conditional:nnNnnNNw
 271, 271, 271
 __prg_set_eq_conditional_F-
 form:nnn 271
 __prg_set_eq_conditional_F-
 form:wNnnnn 272

- __prg_set_eq_conditional_-
 loop:nnnnNw [271](#), [272](#), [272](#), [272](#)
- __prg_set_eq_conditional_p_-
 form:nnn [271](#)
- __prg_set_eq_conditional_p_-
 form:wNnnnn [272](#)
- __prg_set_eq_conditional_T_-
 form:nnn [271](#)
- __prg_set_eq_conditional_T_-
 form:wNnnnn [272](#)
- __prg_set_eq_conditional_TF_-
 form:nnn [271](#)
- __prg_set_eq_conditional_TF_-
 form:wNnnnn [272](#)
- \prg_set_protected_conditional:Nnn
 [37](#), [268](#), [268](#), [305](#)
- \prg_set_protected_conditional:Npnn
 [37](#), [268](#), [268](#), [305](#)
- \primitive [260](#)
- prop commands:
- \s__prop [146](#), [146](#), [461](#), [461](#), [461](#), [461](#),
 [461](#), [461](#), [461](#), [462](#), [463](#), [463](#), [463](#),
 [463](#), [465](#), [465](#), [466](#), [467](#), [468](#), [468](#),
 [469](#), [469](#), [469](#), [470](#), [470](#), [783](#), [783](#), [783](#)
- \prop_(g)clear:N [141](#)
- \prop_clear:c [462](#), [491](#)
- \prop_clear:N
 [141](#), [141](#), [462](#), [462](#), [462](#), [462](#)
- \prop_clear_new:c . . [462](#), [481](#), [481](#), [532](#)
- \prop_clear_new:N
 [141](#), [141](#), [462](#), [462](#), [462](#)
- \prop_gclear:c [462](#)
- \prop_gclear:N [141](#), [462](#), [462](#), [462](#), [462](#)
- \prop_gclear_new:c [462](#)
- \prop_gclear_new:N . [141](#), [462](#), [462](#), [462](#)
- \prop_get:cn [470](#), [470](#)
- \prop_get:cnN [464](#)
- \prop_get:cnNF [485](#), [546](#)
- \prop_get:cnNT [515](#)
- \prop_get:cnNTF . . . [469](#), [513](#), [534](#), [545](#)
- \prop_get:coN [464](#)
- \prop_get:coNTF [469](#)
- \prop_get:cVN [464](#)
- \prop_get:cVNTF [469](#)
- \prop_get:Nn [45](#), [470](#), [470](#)
- \prop_get:NnN
 [46](#), [47](#), [142](#), [142](#), [143](#), [464](#),
 [464](#), [464](#), [464](#), [469](#), [497](#), [497](#), [499](#), [499](#)
- \prop_get:NnNF [469](#), [469](#)
- \prop_get:NnNT [469](#), [469](#)
- \prop_get:NnNTF
 [142](#), [143](#), [144](#), [144](#), [469](#), [469](#), [469](#), [512](#)
- \prop_get:NoN [464](#)
- \prop_get:NoNTF [469](#)
- \prop_get:NVN [464](#)
- \prop_get:NVNTF [469](#)
- \prop_gpop:cnN [464](#)
- \prop_gpop:cnNTF [465](#)
- \prop_gpop:coN [464](#)
- \prop_gpop:NnN
 [142](#), [142](#), [464](#), [464](#), [465](#), [465](#), [465](#)
- \prop_gpop:NnNF [466](#)
- \prop_gpop:NnNT [466](#)
- \prop_gpop:NnNTF [142](#), [144](#), [144](#), [465](#), [466](#)
- \prop_gpop:NoN [464](#)
- \prop_gput:cnN [466](#)
- \prop_gput:cno [466](#)
- \prop_gput:cnV [466](#)
- \prop_gput:cnx [466](#)
- \prop_gput:con [466](#)
- \prop_gput:coo [466](#)
- \prop_gput:cVn [466](#)
- \prop_gput:cVV [466](#)
- \prop_gput:Nnn
 [142](#), [466](#), [466](#), [466](#), [466](#), [557](#), [562](#)
- \prop_gput:Nno [466](#)
- \prop_gput:NnV [466](#)
- \prop_gput:Nnx [466](#)
- \prop_gput:Non [466](#)
- \prop_gput:Noo [466](#)
- \prop_gput:NVn [466](#), [559](#), [562](#)
- \prop_gput:NVV [466](#)
- \prop_gput_if_new:cnN [466](#)
- \prop_gput_if_new:Nnn
 [142](#), [466](#), [467](#), [467](#)
- \prop_gremove:cn [463](#)
- \prop_gremove:cV [463](#)
- \prop_gremove:Nn [143](#), [463](#), [464](#), [464](#), [464](#)
- \prop_gremove:NV [463](#), [559](#), [563](#)
- \prop_gset_eq:cc . . . [462](#), [462](#), [485](#), [485](#)
- \prop_gset_eq:cN . . . [462](#), [462](#), [481](#), [481](#)
- \prop_gset_eq:Nc [462](#), [462](#)
- \prop_gset_eq:NN . . . [141](#), [462](#), [462](#), [462](#)
- \prop_if_empty:cTF [467](#)
- \prop_if_empty:N [467](#)
- \prop_if_empty:NF [467](#), [560](#)
- \prop_if_empty:NT [467](#)
- \prop_if_empty:NTF
 [143](#), [143](#), [467](#), [467](#), [470](#)
- \prop_if_empty_p:c [467](#)

`\prop_if_empty_p:N` . 143, 143, 467, 467
`\prop_if_exist:c` 467
`\prop_if_exist:cF` 533
`\prop_if_exist:cTF` . 467, 534, 545, 546
`\prop_if_exist:N` 467
`\prop_if_exist:NTF`
. 143, 143, 462, 462, 467, 470
`\prop_if_exist_p:c` 467
`\prop_if_exist_p:N` 143, 143, 467
`\prop_if_in:cnTF` 467, 546
`\prop_if_in:coTF` 467
`\prop_if_in:cVTF` 467
`__prop_if_in:N` . . . 467, 468, 468, 468
`\prop_if_in:Nn` 468
`\prop_if_in:NnF` 468, 468
`\prop_if_in:NnT` 468, 468
`\prop_if_in:NnTF` 143, 143, 467, 468, 468
`\prop_if_in:NoTF` 467
`\prop_if_in:NVTF` 467
`__prop_if_in:nwn`
. 467, 468, 468, 468, 468
`\prop_if_in_p:cn` 467
`\prop_if_in_p:co` 467
`\prop_if_in_p:cV` 467
`\prop_if_in_p:Nn` . . . 143, 467, 468, 468
`\prop_if_in_p:No` 467
`\prop_if_in_p:NV` 467
`\l_prop_internal_tl` . . 146, 461,
461, 466, 466, 466, 466, 467, 467
`\prop_item:cn` 465, 470
`\prop_item:Nn`
. . . . 143, 143, 220, 465, 465, 470
`__prop_item_Nn:nwn` 465
`__prop_item_Nn:nwn` 465, 465, 465, 465
`\prop_log:c` 783
`\prop_log:N` . . . 220, 220, 783, 783, 783
`\prop_map...` 145, 145, 145, 145
`\prop_map_break:` 145, 145, 469, 469,
470, 470, 470, 470, 470, 783, 783, 783
`\prop_map_break:n` . . 145, 145, 470, 470
`\prop_map_function:cc` 469
`\prop_map_function:cN` 469, 501
`\prop_map_function:Nc` 469
`\prop_map_function:NN`
. 144, 144, 220, 468,
469, 469, 469, 469, 470, 548, 560, 783
`__prop_map_function:Nwn`
. 469, 469, 469, 469

`\prop_map_inline:cn`
. 469, 493, 494, 772,
772, 773, 773, 775, 777, 777, 777, 777
`\prop_map_inline:Nn` 145,
145, 469, 470, 470, 499, 499, 772, 775
`\prop_map_tokens:cn` 783
`\prop_map_tokens:Nn`
. 220, 220, 783, 783, 783
`__prop_map_tokens:nwn`
. 783, 783, 783, 783, 783
`\prop_new:c` 462, 508, 533
`\prop_new:N` 141, 141, 141, 462, 462,
462, 462, 462, 463, 463, 463, 463,
478, 479, 495, 496, 511, 557, 561, 771
`__prop_pair:wn`
. . . . 146, 146, 146, 461, 461, 461,
461, 461, 463, 463, 463, 463, 465,
465, 466, 467, 468, 468, 468, 469,
469, 469, 470, 470, 470, 470, 783, 783
`\prop_pop:cnN` 464
`\prop_pop:cnNTF` 465
`\prop_pop:coN` 464
`\prop_pop:NnN`
. . . . 142, 142, 464, 464, 464, 465, 465
`\prop_pop:NnNF` 466
`\prop_pop:NnNT` 466
`\prop_pop:NnNTF` 142, 144, 144, 465, 466
`\prop_pop:NoN` 464
`\prop_put:cnn`
. . . . 466, 486, 514, 515, 534, 536, 537
`\prop_put:cno` 466
`\prop_put:cnV` 466, 536
`\prop_put:cnx` 466, 486,
486, 486, 486, 487, 487, 487, 487,
487, 494, 774, 776, 776, 778, 778, 778
`\prop_put:con` 466
`\prop_put:coo` 466
`\prop_put:cVn` 466
`\prop_put:cVV` 466
`\prop_put:Nnn` 142, 142, 146,
300, 461, 466, 466, 466, 466, 478,
478, 478, 478, 495, 495, 495, 495,
495, 495, 496, 496, 496, 496, 496,
496, 496, 496, 496, 496, 496, 514
`__prop_put:NNnn` . . . 466, 466, 466, 466
`\prop_put:Nno` 466, 479,
479, 479, 479, 479, 479, 479, 479, 479
`\prop_put:NnV` 466
`\prop_put:Nnx`
. 466, 773, 773, 773, 773, 773

\prop_put:Nnon	466
\prop_put:Noo	466
\prop_put:NVn	466
\prop_put:NVV	466
\prop_put_if_new:cnn	466
\prop_put_if_new:Nnn	142, 142, 466, 466, 467
__prop_put_if_new:NNnn	466, 466, 467, 467
\prop_remove:cn	463, 514, 536, 536, 537, 537
\prop_remove:cV	463
\prop_remove:Nn 463, 463, 464, 464, 498, 498, 499, 514	143, 143,
\prop_remove:NV	463
\prop_set_eq:cc	462, 462, 485, 485, 492
\prop_set_eq:cN	462, 462, 485, 485, 533, 533
\prop_set_eq:Nc	462, 462, 498
\prop_set_eq>NN	141, 141, 462, 462, 462
\prop_show:c	470
\prop_show:N	145, 145, 470, 470, 470, 783, 783
__prop_split:NnTF 463, 464, 464, 464, 464, 465, 465, 466, 466, 466, 466, 466, 467, 468, 469	146, 146, 463, 463,
__prop_split_aux:NnTF	463, 463, 463
__prop_split_aux:w	463, 463, 463, 463, 463, 463
\prosgegrammeni	810, 810
\protect	567, 611
\protected ...	239, 239, 239, 240, 249, 336
\ProvidesExplClass	7
\ProvidesExplFile	7, 821
\ProvidesExplPackage	7, 7
pt	208
ptex commands:	
\ptex_autospacing:D	257
\ptex_autoxspacing:D	257
\ptex_dtou:D	257
\ptex_euc:D	257
\ptex_ifdbox:D	257
\ptex_ifddir:D	257
\ptex_ifmdir:D	257
\ptex_iftbbox:D	258
\ptex_iftdir:D	258
\ptex_ifybox:D	258
\ptex_ifydir:D	258
\ptex_inhibitglue:D	258
\ptex_inhibitxspcode:D	258
\ptex_jcharwidowpenalty:D	258
\ptex_jfam:D	258
\ptex_jfont:D	258
\ptex_jis:D	258, 348, 818
\ptex_kanjiskip:D	258, 818
\ptex_kansuji:D	258
\ptex_kansujichar:D	258
\ptex_kcatcode:D	258
\ptex_kuten:D	258
\ptex_noautospadding:D	258
\ptex_noautoxspacing:D	258
\ptex_postbreakpenalty:D	258
\ptex_prebreakpenalty:D	258
\ptex_showmode:D	258
\ptex_sjis:D	258
\ptex_tate:D	258
\ptex_tbaselineshift:D	258
\ptex_tfont:D	258
\ptex_xkanjiskip:D	258
\ptex_xspcode:D	258
\ptex_ybaselineshift:D	258
\ptex_yoko:D	258
Q	
quark commands:	
\quark_if_nil:N	320
\quark_if_nil:n ...	321, 321, 321, 321
\quark_if_nil:nF	321
\quark_if_nil:nT	321
\quark_if_nil:NTF	47, 47, 320
\quark_if_nil:nTF	47, 47, 319, 321, 321, 394
\quark_if_nil:oTF	321, 528
\quark_if_nil:vTF	321
__quark_if_nil:w	321, 321, 321, 321, 321
\quark_if_nil_p:N	47, 47, 320
\quark_if_nil_p:n ...	47, 47, 321, 321
\quark_if_nil_p:o	321
\quark_if_nil_p:V	321
\quark_if_no_value:cTF	320
\quark_if_no_value:N	320
\quark_if_no_value:n	321
\quark_if_no_value:Nf	321
\quark_if_no_value:NT	320
\quark_if_no_value:NTF	47, 47, 313, 320, 321, 497, 497, 499, 499, 554, 558, 558
\quark_if_no_value:nTF	47, 47, 321

- _quark_if_no_value:w . 321, 321, 321
- \quark_if_no_value_p:c 320
- \quark_if_no_value_p:N 47, 47, 320, 320
- \quark_if_no_value_p:n . . . 47, 47, 321
- _quark_if_recursion_tail:w 319, 319, 319, 319, 320, 320
- _quark_if_recursion_tail_-break:NN 49, 320, 320, 402
- _quark_if_recursion_tail_-break:nN 49, 49, 320, 320, 401, 413, 455, 455
- \quark_if_recursion_tail_stop:N . . . 48, 48, 319, 319, 366, 426, 457, 806
- \quark_if_recursion_tail_stop:n . . . 48, 48, 319, 319, 320, 412, 447, 457
- \quark_if_recursion_tail_stop:o 319, 527
- \quark_if_recursion_tail_stop... 320
- \quark_if_recursion_tail_stop_-do:Nn 48, 48, 319, 319, 364, 365, 366, 791, 791, 795, 802, 802
- \quark_if_recursion_tail_stop_-do:nn 48, 48, 319, 319, 320, 803
- \quark_if_recursion_tail_stop_-do:on 319
- \quark_new:N 47, 47, 318, 318, 318, 318, 318, 319, 319, 321, 321
- \quitvmode 252
- R**
- \radical 246
- \raise 246
- \read 246
- \readline 249
- recursion commands:
 - \q_recursion_stop 21, 21, 48, 48, 48, 48, 48, 49, 266, 266, 266, 269, 271, 271, 298, 319, 319, 319, 364, 365, 366, 366, 388, 396, 426, 426, 426, 447, 456, 457, 526, 789, 789, 790, 790, 790, 790, 791, 791, 791, 791, 791, 792, 792, 792, 792, 792, 792, 795, 796, 796, 796, 796, 796, 797, 797, 797, 797, 797, 798, 799, 799, 799, 799, 800, 800, 800, 800, 800, 801, 801, 801, 802, 802, 802, 802, 802, 803, 803, 803, 803, 803, 803, 804, 804, 804, 804, 805, 805, 806, 806
 - \q_recursion_tail 48, 48, 48, 48, 48, 48, 48, 48, 49, 49, 49, 269, 270, 271, 271, 272, 319, 319, 319, 319, 319, 319, 319, 320, 320, 320, 320, 364, 365, 366, 366, 388, 388, 401, 402, 402, 412, 426, 447, 455, 455, 455, 456, 456, 457, 468, 468, 468, 469, 469, 469, 469, 526, 783, 783, 783, 789, 791, 792, 795, 796, 801, 802, 803, 806, 806, 807, 808, 809, 809
- \ref 805
- \relax 234, 234, 234, 234, 234, 234, 234, 235, 235, 236, 236, 237, 237, 238, 239, 239, 239, 239, 239, 239, 239, 239, 240, 240, 240, 240, 240, 240, 240, 240, 240, 240, 240, 240, 246
- \relpenalty 246
- \RequirePackage 239
- reverse commands:
 - \reverse_if:N 23, 23, 262, 262, 351, 351, 353, 353, 353, 353, 373, 373, 373, 373, 425, 425, 609, 609, 719, 738, 739
- \right 246
- right commands:
 - \c_right_brace_str 117, 426, 427
- \rightghost 254
- \righthyphenmin 247
- \rightmarginkern 252
- \rightskip 247
- \romannumeral 247
- round 205
- \rPCODE 252
- \rule 497, 498
- S**
- \savecatcodetable 254
- \savingshyphcodes 249
- \savingsdiscards 249
- scan commands:
 - \scan_align_safe_stop: 318, 318
 - \g__scan_marks_tl . . 321, 321, 322, 322
 - __scan_new:N 50, 50, 322, 322, 322, 322, 461, 574, 575, 575, 575, 575, 575, 575, 575
 - \scan_stop: . . 10, 10, 50, 50, 63, 63, 63, 64, 130, 240, 241, 263, 263, 270, 270, 275, 275, 275, 275, 275, 276, 276, 279, 291, 291, 291, 298, 301, 302, 302, 322, 322, 329, 332, 332, 332, 341, 341, 344, 344, 345, 345, 352, 357, 357, 378, 379, 379, 380,

- 380, 380, 382, 382, 382, 383, 388,
 389, 390, 391, 403, 425, 425, 425,
 437, 461, 470, 474, 475, 497, 498,
 557, 557, 559, 559, 561, 561, 562,
 562, 587, 606, 609, 609, 609, 610,
 610, 611, 614, 614, 614, 624, 626,
 626, 633, 633, 640, 785, 785, 788,
 788, 788, 815, 816, 816, 816, 821,
 821, 821, 821, 821, 821, 821, 821, 828
 \scantextokens 254
 \scantokens 249
 \scriptfont 247
 \scriptscriptfont 247
 \scriptscriptstyle 247
 \scriptspace 247
 \scriptstyle 247
 \scrollmode 247
 sec 206
 sec d 206
 seq commands:
 __seq 130, 322, 322,
 427, 427, 428, 429, 429, 429, 429,
 430, 431, 431, 431, 431, 431, 431,
 437, 437, 439, 442, 442, 783, 783, 784
 \seq_(g)clear:N 119
 \seq_clear:c 428
 \seq_clear:N 119, 119,
 128, 428, 428, 428, 429, 432, 512, 514
 \seq_clear_new:c 429
 \seq_clear_new:N 119, 119, 429, 429, 429
 \seq_concat:ccc 431
 \seq_concat:NNN
 120, 120, 128, 129, 431, 431, 431, 553
 \seq_count:c 441
 \seq_count:N 122,
 125, 125, 128, 439, 441, 441, 441, 442
 __seq_count:n 441, 441, 441
 \seq_elt:w 427, 427
 \seq_elt_end: 427, 427
 \seq_gclear:c 428
 \seq_gclear:N 119, 428, 428, 428, 429
 \seq_gclear_new:c 429
 \seq_gclear_new:N 119, 429, 429, 429
 \seq_gconcat:ccc 431
 \seq_gconcat:NNN 120, 431, 431, 431
 \seq_get:cN 443, 443, 443
 \seq_get:cNTF 443
 \seq_get:NN 127, 127, 443, 443, 443
 \seq_get:NNTF 127, 127, 443
 \seq_get_left:cN 436, 443, 443
 \seq_get_left:cNTF 438
 \seq_get_left:NN 121,
 121, 436, 436, 436, 438, 438, 443, 443
 \seq_get_left:NNTF 438
 \seq_get_left:NNT 438
 \seq_get_left:NNTF 122, 122, 438, 438
 __seq_get_left:wnw 436, 436, 436
 \seq_get_right:cN 437
 \seq_get_right:cNTF 438
 \seq_get_right:NN
 121, 121, 437, 437, 437, 438, 438
 \seq_get_right:NNTF 438
 \seq_get_right:NNT 438
 \seq_get_right:NNTF 122, 122, 438, 438
 __seq_get_right_loop:nn
 437, 437, 437, 437, 437
 \seq_gpop:cN 443, 443, 443
 \seq_gpop:cNTF 443
 \seq_gpop:NN 127, 127, 443, 443, 443, 555
 \seq_gpop:NNTF 127, 127, 443, 558, 562
 \seq_gpop_left:cN 436, 443, 443
 \seq_gpop_left:cNTF 438
 \seq_gpop_left:NN
 121, 121, 436, 436, 436, 438, 443, 443
 \seq_gpop_left:NNTF 438
 \seq_gpop_left:NNT 438
 \seq_gpop_left:NNTF 122, 122, 438, 439
 \seq_gpop_right:cN 437
 \seq_gpop_right:cNTF 438
 \seq_gpop_right:NN
 121, 121, 437, 437, 438, 438
 \seq_gpop_right:NNTF 439
 \seq_gpop_right:NNT 439
 \seq_gpop_right:NNTF 123, 123, 438, 439
 \seq_gpush:cn 443, 443
 \seq_gpush:co 443, 443
 \seq_gpush:cV 443, 443
 \seq_gpush:cv 443, 443
 \seq_gpush:cx 443, 443
 \seq_gpush:Nn 127, 443, 443
 \seq_gpush:No 26, 443, 443, 555
 \seq_gpush:Nv 443, 443, 559, 563
 \seq_gpush:Nv 443, 443
 \seq_gpush:Nx 443, 443
 \seq_gput_left:cn 431, 443
 \seq_gput_left:co 431, 443
 \seq_gput_left:cV 431, 443
 \seq_gput_left:cv 431, 443
 \seq_gput_left:cx 431, 443

[illegible]

\seq_map_function:cN [440](#)
 \seq_map_function:NN
 [4](#), [124](#), [124](#), [124](#), [440](#),
 [440](#), [440](#), [441](#), [444](#), [446](#), [515](#), [523](#), [524](#)
 _seq_map_function:NNn
 [440](#), [440](#), [440](#), [440](#)
 \seq_map_inline:cn [441](#)
 \seq_map_inline:Nn . [124](#), [124](#), [124](#),
 [128](#), [128](#), [129](#), [129](#), [129](#), [432](#), [441](#),
 [441](#), [441](#), [513](#), [545](#), [552](#), [553](#), [556](#), [784](#)
 \seq_map_variable:ccn [441](#)
 \seq_map_variable:cNn [441](#)
 \seq_map_variable:Ncn [441](#)
 \seq_map_variable:NNn
 [124](#), [124](#), [441](#), [441](#), [441](#), [441](#)
 \seq_mapthread_function:ccN ... [783](#)
 \seq_mapthread_function:cNN ... [783](#)
 \seq_mapthread_function:NcN ... [783](#)
 \seq_mapthread_function:NNN
 [220](#), [220](#), [783](#), [783](#), [784](#), [784](#)
 _seq_mapthread_function:Nnnwnn
 [783](#), [784](#), [784](#), [784](#)
 _seq_mapthread_function:wNN ...
 [783](#), [783](#), [783](#)
 _seq_mapthread_function:wNw ...
 [783](#), [784](#), [784](#)
 \seq_new:c [428](#)
 \seq_new:N [4](#), [119](#), [119](#), [119](#),
 [327](#), [327](#), [428](#), [428](#), [428](#), [429](#), [429](#),
 [432](#), [444](#), [444](#), [444](#), [444](#), [512](#), [512](#),
 [530](#), [551](#), [551](#), [552](#), [552](#), [552](#), [556](#), [561](#)
 \seq_pop:cN [443](#), [443](#), [443](#)
 \seq_pop:cNTF [443](#)
 \seq_pop:NN ... [127](#), [127](#), [443](#), [443](#), [443](#)
 _seq_pop:NNNN
 [435](#), [435](#), [436](#), [436](#), [437](#), [437](#)
 \seq_pop:NNTF [127](#), [127](#), [443](#)
 _seq_pop_item_def: .. [130](#), [130](#),
 [130](#), [433](#), [440](#), [440](#), [441](#), [441](#), [784](#), [785](#)
 \seq_pop_left:cN [436](#), [443](#), [443](#)
 \seq_pop_left:cNTF [438](#)
 \seq_pop_left:NN
 [121](#), [121](#), [436](#), [436](#), [436](#), [438](#), [443](#), [443](#)
 \seq_pop_left:NNF [438](#)
 _seq_pop_left:NNN
 [436](#), [436](#), [436](#), [436](#), [438](#), [438](#)
 \seq_pop_left:NNT [438](#)
 \seq_pop_left:NNTF . [122](#), [122](#), [438](#), [438](#)
 _seq_pop_left:wnwnnn . [436](#), [436](#), [436](#)
 \seq_pop_right:cN [437](#)
 \seq_pop_right:cNTF [438](#)
 \seq_pop_right:NN
 [121](#), [121](#), [437](#), [437](#), [438](#), [438](#)
 \seq_pop_right:NNF [439](#)
 _seq_pop_right:NNN
 [433](#), [437](#), [437](#), [437](#), [437](#), [438](#), [438](#)
 \seq_pop_right:NNT [439](#)
 \seq_pop_right:NNTF [123](#), [123](#), [438](#), [439](#)
 _seq_pop_right_loop:nn
 [437](#), [438](#), [438](#), [438](#)
 _seq_pop_TF:NNNN [435](#),
 [436](#), [438](#), [438](#), [438](#), [438](#), [438](#), [438](#), [438](#)
 \seq_push:cn [443](#), [443](#)
 \seq_push:co [443](#), [443](#)
 \seq_push:cV [443](#), [443](#), [443](#)
 \seq_push:cv [443](#)
 \seq_push:cx [443](#), [443](#)
 \seq_push:Nn [127](#), [127](#), [443](#), [443](#)
 \seq_push:No [443](#), [443](#)
 \seq_push:NV [443](#), [443](#)
 \seq_push:Nv [443](#), [443](#)
 \seq_push:Nx [443](#), [443](#)
 _seq_push_item_def:
 [440](#), [440](#), [440](#), [440](#)
 _seq_push_item_def:n . [130](#), [130](#),
 [130](#), [130](#), [433](#), [440](#), [440](#), [441](#), [784](#), [784](#)
 _seq_push_item_def:x . [440](#), [440](#), [441](#)
 \seq_put_left:cn [431](#), [443](#)
 \seq_put_left:co [431](#), [443](#)
 \seq_put_left:cV [431](#), [443](#)
 \seq_put_left:cv [431](#), [443](#)
 \seq_put_left:cx [431](#), [443](#)
 \seq_put_left:Nn
 [120](#), [120](#), [431](#), [431](#), [431](#), [431](#), [443](#), [513](#)
 \seq_put_left:No [431](#), [443](#)
 \seq_put_left:NV [431](#), [443](#)
 \seq_put_left:Nv [431](#), [443](#)
 \seq_put_left:Nx [431](#), [443](#)
 _seq_put_left_aux:w
 [431](#), [431](#), [431](#), [431](#), [431](#)
 \seq_put_right:cn [432](#)
 \seq_put_right:co [432](#)
 \seq_put_right:cV [432](#)
 \seq_put_right:cv [432](#)
 \seq_put_right:cx [432](#)
 \seq_put_right:Nn
 [120](#), [120](#), [128](#), [128](#), [129](#), [323](#),
 [324](#), [432](#), [432](#), [432](#), [432](#), [432](#), [515](#), [555](#)
 \seq_put_right:No [432](#), [556](#)
 \seq_put_right:NV [432](#)

- \seq_put_right:Nv [432](#)
- \seq_put_right:Nx [432](#)
- \seq_remove_all:cn [433](#)
- \seq_remove_all:Nn
 - [120](#), [123](#), [123](#), [128](#), [128](#),
 - [129](#), [129](#), [129](#), [323](#), [433](#), [433](#), [433](#), [555](#)
- __seq_remove_all_aux:NNn
 - [433](#), [433](#), [433](#), [433](#)
- \seq_remove_duplicates:c [432](#)
- \seq_remove_duplicates:N
 - [123](#), [123](#), [128](#), [128](#), [432](#), [432](#), [432](#), [556](#)
- __seq_remove_duplicates:NN
 - [432](#), [432](#), [432](#), [432](#)
- \l__seq_remove_seq
 - [432](#), [432](#), [432](#), [432](#), [432](#), [432](#)
- \seq_reverse:c [433](#)
- \seq_reverse:N
 - [123](#), [123](#), [433](#), [433](#), [434](#), [434](#)
- __seq_reverse:NN .. [433](#), [434](#), [434](#), [434](#)
- __seq_reverse_item:nw [434](#), [434](#)
- __seq_reverse_item:nwn [433](#), [434](#), [434](#)
- \seq_set_eq:cc [429](#), [429](#)
- \seq_set_eq:cN [429](#), [429](#)
- \seq_set_eq:Nc [429](#), [429](#)
- \seq_set_eq:NN
 - [119](#), [119](#), [128](#), [129](#), [129](#),
 - [129](#), [428](#), [429](#), [432](#), [553](#), [554](#), [555](#)
- \seq_set_filter:NNn
 - [221](#), [221](#), [784](#), [784](#), [784](#)
- __seq_set_filter:NNNn
 - [784](#), [784](#), [784](#), [784](#)
- \seq_set_from_clist:cc [429](#)
- \seq_set_from_clist:cN [429](#)
- \seq_set_from_clist:cn [429](#)
- \seq_set_from_clist:Nc [429](#)
- \seq_set_from_clist:NN
 - [119](#), [119](#), [429](#), [429](#), [429](#), [429](#)
- \seq_set_from_clist:Nn
 - [119](#), [429](#), [429](#), [429](#), [543](#), [543](#)
- \seq_set_map:NNn ... [221](#), [221](#), [784](#), [784](#)
- __seq_set_map:NNNn [784](#), [784](#), [784](#), [784](#)
- \seq_set_split:Nnn
 - [120](#), [120](#), [120](#), [327](#), [327](#), [430](#), [430](#), [431](#)
- __seq_set_split:NNnn
 - [430](#), [430](#), [430](#), [430](#)
- \seq_set_split:NnV [430](#), [553](#)
- __seq_set_split_auxi:w
 - [430](#), [430](#), [430](#), [430](#), [430](#), [430](#)
- __seq_set_split_auxii:w
 - [430](#), [430](#), [430](#), [430](#)
- __seq_set_split_end:
 - [430](#), [430](#), [430](#), [430](#), [430](#), [430](#), [430](#)
- \seq_show:c [444](#)
- \seq_show:N
 - [130](#), [130](#), [444](#), [444](#), [444](#), [785](#), [785](#)
- __seq_tmp:w [428](#), [428](#), [434](#), [434](#), [437](#), [438](#)
- \seq_use:cn [442](#)
- \seq_use:cnnn [442](#)
- \seq_use:Nn ... [126](#), [126](#), [442](#), [442](#), [442](#)
- \seq_use:Nnnn
 - [126](#), [126](#), [442](#), [442](#), [442](#), [442](#)
- __seq_use:NNnNnn .. [442](#), [442](#), [442](#), [442](#)
- __seq_use:nwnn [442](#), [442](#), [442](#)
- __seq_use:nwwwnwn [442](#), [442](#), [442](#), [442](#)
- __seq_use_setup:w ... [442](#), [442](#), [442](#), [442](#)
- __seq_wrap_item:n
 - [429](#), [429](#), [429](#), [429](#),
 - [430](#), [430](#), [430](#), [432](#), [432](#), [433](#), [784](#), [784](#)
- \setbox [247](#)
- \setlanguage [247](#)
- seven commands:
 - \c_seven
 - . [76](#), [325](#), [326](#), [367](#), [367](#), [422](#), [422](#),
 - [424](#), [620](#), [640](#), [640](#), [660](#), [663](#), [742](#), [742](#)
- \sfcode [247](#)
- \sffamily [497](#)
- \shellescape [260](#)
- \shipout [247](#)
- \ShortText [237](#), [238](#), [238](#)
- \show [247](#)
- \showbox [247](#)
- \showboxbreadth [247](#)
- \showboxdepth [247](#)
- \showgroups [249](#)
- \showifs [249](#)
- \showlists [247](#)
- \showmode [258](#)
- \showthe [247](#)
- \showtokens [249](#)
- sin [206](#)
- sind [206](#)
- six commands:
 - \c_six
 - . [76](#), [323](#), [324](#), [325](#), [326](#), [367](#), [367](#), [424](#)
- sixteen commands:
 - \c_sixteen [76](#), [264](#), [264](#), [264](#),
 - [277](#), [365](#), [367](#), [556](#), [559](#), [560](#), [561](#),
 - [563](#), [581](#), [584](#), [598](#), [628](#), [630](#), [641](#),
 - [642](#), [713](#), [723](#), [723](#), [752](#), [752](#), [752](#), [753](#)
- \sjis [258](#)

- \skewchar 247
- \skip 247
- skip commands:
 - \skip_(g)zero:N 87
 - \skip_add:cn 379
 - \skip_add:Nn 88, 88, 379, 379, 379, 379
 - \skip_const:cn 377
 - \skip_const:Nn
 - 87, 87, 377, 377, 377, 380, 380
 - \skip_eval:n 88,
 - 88, 89, 89, 89, 379, 379, 380, 380, 380
 - \skip_gadd:cn 379
 - \skip_gadd:Nn 88, 379, 379, 379
 - .skip_gset:c 175, 541
 - \skip_gset:cn 378
 - .skip_gset:N 175, 541
 - \skip_gset:Nn .. 88, 377, 378, 378, 378
 - \skip_gset_eq:cc 378
 - \skip_gset_eq:cN 378
 - \skip_gset_eq:Nc 378
 - \skip_gset_eq:NN 88, 378, 378, 378, 378
 - \skip_gsub:cn 379
 - \skip_gsub:Nn 88, 379, 379, 379
 - \skip_gzero:c 378
 - \skip_gzero:N .. 87, 378, 378, 378, 378
 - \skip_gzero_new:c 378
 - \skip_gzero_new:N .. 87, 378, 378, 378
 - \skip_horizontal:c 380
 - \skip_horizontal:N
 - 90, 90, 90, 380, 380, 380, 380
 - \skip_horizontal:n 90, 90, 380, 380, 825
 - \skip_if_eq:nn 379
 - \skip_if_eq:nnTF 88, 379
 - \skip_if_eq_p:nn 88, 88, 379
 - \skip_if_exist:c 378
 - \skip_if_exist:cTF 378
 - \skip_if_exist:N 378
 - \skip_if_exist:NTF 87, 87, 378, 378, 378
 - \skip_if_exist_p:c 378
 - \skip_if_exist_p:N 87, 87, 378
 - \skip_if_finite:n 379
 - \skip_if_finite:nTF .. 88, 88, 379, 785
 - _skip_if_finite:wwNw . 379, 379, 379
 - \skip_if_finite_p:n 88, 88, 379
 - \skip_log:c 785, 785
 - \skip_log:N 222, 222, 785, 785
 - \skip_log:n 222, 222, 785, 785
 - \skip_new:c 377
 - \skip_new:N .. 87, 87, 87, 377, 377,
 - 377, 377, 378, 378, 381, 381, 381, 381
 - .skip_set:c 175, 541
 - \skip_set:cn 378
 - .skip_set:N 175, 541
 - \skip_set:Nn 88, 88, 378, 378, 378, 378
 - \skip_set_eq:cc 378
 - \skip_set_eq:cN 378
 - \skip_set_eq:Nc 378
 - \skip_set_eq:NN
 - 88, 88, 378, 378, 378, 378
 - \skip_show:c 380
 - \skip_show:N 89, 89, 380, 380, 380
 - \skip_show:n 89, 89, 380, 380, 785, 785
 - \skip_split_finite_else_action:nnNN
 - 221, 221, 785, 785
 - \skip_sub:cn 379
 - \skip_sub:Nn 88, 88, 379, 379, 379, 379
 - \skip_use:c 380, 380
 - \skip_use:N 89,
 - 89, 89, 89, 379, 380, 380, 380, 380
 - \skip_vertical:c 380
 - \skip_vertical:N
 - 90, 90, 90, 380, 380, 380, 380
 - \skip_vertical:n 90, 90, 380, 380
 - \skip_zero:c 378
 - \skip_zero:N
 - ... 87, 87, 90, 378, 378, 378, 378, 378
 - \skip_zero_new:c 378
 - \skip_zero_new:N . 87, 87, 378, 378, 378
 - \skipdef 247
 - sp 208
 - spac commands:
 - \spac_directions_normal_body_dir 261
 - \spac_directions_normal_page_dir 261
 - space commands:
 - \c_space_tl 108,
 - 386, 386, 406, 426, 426, 457, 457,
 - 505, 555, 568, 568, 569, 571, 571,
 - 571, 571, 757, 790, 792, 802, 824,
 - 824, 825, 826, 826, 826, 826, 826, 826
 - \c_space_token
 - . 56, 107, 108, 227, 328, 328, 328,
 - 330, 330, 342, 409, 410, 411, 788, 816
 - \spacefactor 247
 - \spaceskip 247
 - \span 247
 - \special 247
 - \splitbotmark 247
 - \splitbotmarks 249
 - \splitdiscards 249
 - \splitfirstmark 247

- `\splitfirstmarks` 249
- `\splitmaxdepth` 247
- `\splittopskip` 247
- `sqrt` 208
- sr commands:
 - `\sr_if_empty_p:N` 111
- `\SS` 806
- `\ss` 806
- stop commands:
 - `\q_stop` .. 21, 21, 25, 25, 32, 46, 47, 47, 47, 105, 266, 266, 266, 270, 270, 270, 271, 271, 272, 272, 272, 272, 272, 274, 274, 274, 274, 274, 274, 299, 299, 300, 301, 302, 302, 302, 303, 303, 303, 303, 304, 304, 310, 310, 310, 313, 313, 318, 318, 318, 332, 332, 333, 333, 334, 334, 334, 334, 334, 335, 335, 335, 335, 336, 336, 336, 336, 336, 337, 337, 337, 337, 338, 344, 344, 345, 345, 351, 352, 352, 352, 352, 353, 354, 364, 364, 364, 365, 372, 373, 373, 379, 379, 394, 394, 400, 400, 400, 401, 403, 403, 403, 404, 404, 404, 405, 408, 408, 408, 409, 410, 410, 417, 418, 418, 418, 418, 419, 419, 419, 421, 422, 422, 423, 424, 425, 425, 425, 425, 425, 436, 436, 436, 436, 439, 439, 442, 442, 442, 442, 449, 449, 449, 449, 449, 449, 449, 450, 450, 452, 453, 453, 453, 453, 453, 453, 453, 454, 454, 458, 458, 458, 458, 458, 459, 460, 463, 463, 463, 513, 521, 525, 525, 527, 527, 527, 527, 527, 528, 528, 528, 532, 532, 532, 532, 534, 535, 535, 536, 536, 544, 544, 544, 568, 608, 608, 612, 612, 782, 783, 783, 784, 784, 784, 784, 784, 794, 794, 815, 816, 816, 816
 - `\s_stop` 50, 50, 50, 50, 322, 322, 322, 713, 714, 749, 749
- str commands:
 - `\str_(g)clear:N` 110
 - `\str_...:N` 109
 - `\str_...:n` 109
 - `\str_..._ignore_spaces:n` 109
 - `\str_...:N` 109
 - `\str_case:nn` .. 112, 417, 417, 417, 521
 - `\str_case:nn(TF)` 354, 373
 - `\str_case:nnF` 417, 417
 - `\str_case:nnT` 417, 417
 - `_str_case:nnTF` 417, 417, 417, 417, 417, 417
 - `\str_case:nnTF` 112, 112, 417, 417, 417
 - `\str_case:nV` 417
 - `\str_case:nv` 417
 - `\str_case:nVF` 798, 804
 - `\str_case:nvF` . 426, 793, 793, 793, 795
 - `\str_case:nVTF` 417
 - `\str_case:nvTF` 417
 - `_str_case:nw` 417, 417, 417, 417
 - `\str_case:on` 417
 - `\str_case:onTF` 417
 - `_str_case_end:nw` . 417, 417, 418, 418
 - `\str_case_x:nn` 112, 417, 417
 - `\str_case_x:nnF` 112, 418
 - `\str_case_x:nnT` 417
 - `_str_case_x:nnTF` 417, 417, 418, 418, 418, 418
 - `\str_case_x:nnTF` 112, 417, 418
 - `_str_case_x:nw` ... 417, 418, 418, 418
 - `_str_change_case:nn` 425, 425, 426, 426, 426
 - `_str_change_case_aux:nn` 425, 426, 426
 - `_str_change_case_char:nN` 425, 426, 426
 - `_str_change_case_char:NNNNNNNN` 425, 426, 426
 - `_str_change_case_loop:nw` 425, 426, 426, 426, 426
 - `_str_change_case_space:n` 425, 426, 426
 - `\str_clear:c` 414
 - `\str_clear:N` 110, 110, 414
 - `\str_clear_new:c` 414
 - `\str_clear_new:N` 110, 110, 414
 - `_str_collect_delimit_by_q-stop:w` 421, 422, 422
 - `_str_collect_end:nnnnnnnw` ... 422, 422, 422, 422
 - `_str_collect_end:wn` . 422, 422, 422
 - `_str_collect_loop:wn` 422, 422, 422, 422
 - `_str_collect_loop:wnNNNNNN` ... 422, 422, 422
 - `\str_const:cn` 414
 - `\str_const:cx` 414
 - `\str_const:Nn` 109, 109, 414, 817, 818, 818, 818, 818, 819, 819

```

\str_const:Nx .....
. 414, 426, 426, 427, 427, 427, 427,
427, 427, 427, 427, 427, 427, 817, 817
\str_count:c ..... 423
\str_count:N .. 113, 113, 423, 423, 423
\_str_count:n .....
.... 118, 118, 419, 421, 423, 423, 424
\str_count:n .....
.... 113, 113, 113, 118, 423, 423, 423
\_str_count_aux:n .....
..... 423, 423, 424, 424, 424
\str_count_ignore_spaces:n .....
..... 113, 113, 423, 423, 424, 569
\_str_count_loop:NNNNNNNN .....
..... 423, 423, 424, 424, 424, 424
\str_count_spaces:c ..... 423
\str_count_spaces:N 113, 423, 423, 423
\str_count_spaces:n .....
.... 113, 113, 423, 423, 423, 423, 423
\_str_count_spaces_loop:w .....
..... 423, 423, 423, 423
\_str_escape_x:n .. 415, 415, 415, 415
\str_fold_case:n ... 115, 115, 116,
116, 116, 116, 116, 223, 425, 425, 426
\str_fold_case:V ..... 425
\str_gclear:c ..... 414
\str_gclear:N ..... 110, 414
\str_gclear_new:c ..... 414
\str_gclear_new:N ..... 414
\str_gput_left:cn ..... 414
\str_gput_left:cx ..... 414
\str_gput_left:Nn ..... 110, 414
\str_gput_left:Nx ..... 414
\str_gput_right:cn ..... 414
\str_gput_right:cx ..... 414
\str_gput_right:Nn ..... 110, 414
\str_gput_right:Nx ..... 414
\str_gset:cn ..... 414
\str_gset:cx ..... 414
\str_gset:Nn ..... 110, 414
\str_gset:Nx ..... 414
\str_gset_eq:cc ..... 414
\str_gset_eq:cN ..... 414
\str_gset_eq:Nc ..... 414
\str_gset_eq:NN ... 110, 414, 414, 414
\str_head:c ..... 424
\str_head:N 113, 113, 424, 424, 424, 424
\str_head:n 113, 113, 113, 391, 392,
409, 409, 410, 424, 424, 424, 424, 424
\_str_head:w .....
..... 424, 424, 424, 424, 424, 425
\str_head_ignore_spaces:n .....
..... 113, 113, 424, 425
\str_if_empty:c ..... 415
\str_if_empty:cTF ..... 415
\str_if_empty:N ..... 415
\str_if_empty:NTF ..... 111, 111, 415
\str_if_empty_p:c ..... 415
\str_if_empty_p:N ..... 111, 415
\str_if_eq:ccTF ..... 416
\str_if_eq:cNTF ..... 416
\str_if_eq:NcTF ..... 416
\str_if_eq:NN ..... 416, 416
\str_if_eq:nn ..... 141, 146, 416
\str_if_eq:NNF ..... 417
\str_if_eq:nnF ..... 416, 416, 537
\str_if_eq:NNT ..... 417
\str_if_eq:nnT .....
220, 416, 416, 433, 433, 513, 545, 806
\str_if_eq:NNTF ... 111, 111, 416, 417
\str_if_eq:nnTF ..... 111, 111,
112, 112, 143, 416, 416, 416, 416,
417, 508, 537, 550, 611, 794, 794, 795
\str_if_eq:noTF ..... 416
\str_if_eq:nVTF ..... 416
\str_if_eq:onTF ..... 416
\str_if_eq:VnTF ..... 416
\str_if_eq:VVTF ..... 416
\str_if_eq_p:cc ..... 416
\str_if_eq_p:cN ..... 416
\str_if_eq_p:Nc ..... 416
\str_if_eq_p:NN ... 111, 111, 416, 417
\str_if_eq_p:nn 111, 111, 416, 416, 416
\str_if_eq_p:no ..... 416
\str_if_eq_p:nV ..... 416
\str_if_eq_p:on ..... 416
\str_if_eq_p:Vn ..... 416
\str_if_eq_p:VV ..... 416
\_str_if_eq_x:nn .....
117, 117, 332, 379, 415, 415, 415,
416, 416, 416, 416, 626, 626, 633, 719
\str_if_eq_x:nn ..... 416, 468, 468
\str_if_eq_x:nn(TF) ..... 117
\str_if_eq_x:nnF ..... 515, 531
\str_if_eq_x:nnTF ..... 111, 111,
416, 418, 465, 468, 570, 794, 794, 794
\str_if_eq_x_p:nn ..... 111, 111, 416

```

- _str_if_eq_x_return:nn
 [117](#), [117](#), [333](#), [333](#), [334](#), [334](#),
 [335](#), [335](#), [336](#), [336](#), [336](#), [336](#), [416](#), [416](#)
- \str_if_exist:c [415](#)
- \str_if_exist:cTF [415](#)
- \str_if_exist:N [415](#)
- \str_if_exist:NTF [111](#), [111](#), [415](#)
- \str_if_exist_p:c [415](#)
- \str_if_exist_p:N [111](#), [111](#), [415](#)
- \str_item:cn [419](#)
- \str_item:Nn .. [114](#), [114](#), [419](#), [419](#), [419](#)
- _str_item:nn
 [419](#), [419](#), [419](#), [419](#), [419](#), [419](#)
- \str_item:nn
 [114](#), [114](#), [114](#), [419](#), [419](#), [419](#), [419](#), [423](#)
- _str_item:w [419](#), [419](#), [419](#), [419](#)
- \str_item_ignore_spaces:nn
 [114](#), [114](#), [419](#), [419](#), [419](#)
- \str_lower_case:f [425](#)
- \str_lower_case:n
 [115](#), [115](#), [223](#), [425](#), [426](#), [426](#)
- \str_new:c [414](#)
- \str_new:N
 [109](#), [109](#), [110](#), [414](#), [427](#), [427](#), [427](#), [427](#)
- \str_put_left:cn [414](#)
- \str_put_left:cx [414](#)
- \str_put_left:Nn [110](#), [110](#), [414](#)
- \str_put_left:Nx [414](#)
- \str_put_right:cn [414](#)
- \str_put_right:cx [414](#)
- \str_put_right:Nn [110](#), [110](#), [414](#)
- \str_put_right:Nx [414](#)
- \str_range:Nnn [114](#), [420](#), [421](#), [421](#)
- _str_range:nnn
 [118](#), [118](#), [420](#), [421](#), [421](#), [421](#)
- \str_range:nnn
 [114](#), [114](#), [118](#), [420](#), [421](#), [421](#), [423](#)
- _str_range:nnw [420](#), [421](#), [421](#)
- _str_range:w [420](#), [421](#), [421](#)
- \str_range_ignore_spaces:nnn ...
 [114](#), [420](#), [421](#)
- _str_range_normalize:nn
 [421](#), [421](#), [421](#), [421](#)
- \str_set:cn [414](#)
- \str_set:cx [414](#)
- \str_set:Nn [110](#), [110](#), [414](#)
- \str_set:Nx [414](#)
- \str_set_eq:cc [414](#)
- \str_set_eq:cN [414](#)
- \str_set_eq:Nc [414](#)
- \str_set_eq:NN [110](#), [110](#), [414](#), [414](#), [414](#)
- \str_show:c [427](#)
- \str_show:N ... [116](#), [116](#), [427](#), [427](#), [427](#)
- \str_show:n [116](#), [427](#), [427](#)
- _str_skip_end:NNNNNNNN
 [420](#), [420](#), [420](#), [420](#), [420](#)
- _str_skip_end:w [420](#), [420](#), [420](#)
- _str_skip_exp_end:w
 [419](#), [420](#), [420](#), [420](#), [420](#), [420](#), [421](#), [422](#)
- _str_skip_loop:wNNNNNNNN
 [420](#), [420](#), [420](#)
- \str_tail:c [425](#)
- \str_tail:N ... [113](#), [113](#), [425](#), [425](#), [425](#)
- \str_tail:n [113](#), [113](#), [113](#), [425](#), [425](#), [425](#)
- _str_tail_auxi:w [425](#), [425](#), [425](#)
- _str_tail_auxii:w [425](#), [425](#), [425](#), [425](#)
- \str_tail_ignore_spaces:n
 [113](#), [113](#), [425](#), [425](#)
- _str_tmp:n [414](#), [414](#), [414](#), [414](#), [414](#), [414](#)
- _str_to_other:n [118](#),
 [118](#), [118](#), [118](#), [418](#), [418](#), [419](#), [421](#), [423](#)
- _str_to_other_end:w
 [418](#), [418](#), [418](#), [419](#)
- _str_to_other_loop:w
 [418](#), [418](#), [418](#), [418](#), [419](#)
- \str_upper_case:f [425](#)
- \str_upper_case:n
 [115](#), [115](#), [223](#), [425](#), [426](#), [426](#)
- \str_use:c [414](#)
- \str_use:N [112](#), [112](#), [414](#)
- \strcmp [235](#)
- \string [237](#), [247](#)
- \suppressfontnotfounderror [252](#)
- \suppressifcsnameerror [254](#)
- \suppresslongerror [254](#)
- \suppressmathparerror [254](#)
- \suppressoutererror [254](#)
- \synctex [252](#)
- sys commands:
- \c_sys_day_int [228](#), [817](#), [817](#)
- \c_sys_engine_str
 [228](#), [817](#), [817](#), [818](#), [818](#), [818](#), [818](#)
- \c_sys_hour_int [228](#), [817](#), [817](#)
- \sys_if_engine luatex: [819](#)
- \sys_if_engine luatex:F [817](#), [820](#)
- \sys_if_engine luatex:T [230](#), [817](#)
- \sys_if_engine luatex:TF [228](#), [817](#), [817](#)
- \sys_if_engine luatex_p:
 [228](#), [817](#), [817](#), [819](#)
- \sys_if_engine pdftex:F [817](#)

- `\sys_if_engine_pdftex:T` 817
 - `\sys_if_engine_pdftex:TF`
 - 228, 228, 817, 817
 - `\sys_if_engine_pdftex_p:` 228, 817, 817
 - `\sys_if_engine_ptex:F` 818
 - `\sys_if_engine_ptex:T` 818
 - `\sys_if_engine_ptex:TF` . 228, 817, 818
 - `\sys_if_engine_ptex_p:` . 228, 817, 818
 - `\sys_if_engine_uptex:F` 818
 - `\sys_if_engine_uptex:T` 818
 - `\sys_if_engine_uptex:TF` 228, 817, 818
 - `\sys_if_engine_uptex_p:` 228, 817, 818
 - `\sys_if_engine_xetex:` 819
 - `\sys_if_engine_xetex:F` 818
 - `\sys_if_engine_xetex:T` 818
 - `\sys_if_engine_xetex:TF` 228, 817, 818
 - `\sys_if_engine_xetex_p:`
 - 228, 817, 818, 819
 - `\sys_if_output_dvi:F` 818, 819
 - `\sys_if_output_dvi:T` 818, 819
 - `\sys_if_output_dvi:TF`
 - 229, 229, 818, 818, 819
 - `\sys_if_output_dvi_p:`
 - 229, 818, 818, 819
 - `\sys_if_output_pdf:F` 819, 819
 - `\sys_if_output_pdf:T` 818, 819
 - `\sys_if_output_pdf:TF`
 - 229, 818, 819, 819
 - `\sys_if_output_pdf_p:`
 - 229, 818, 819, 819
 - `\c_sys_jobname_str`
 - 184, 228, 816, 817, 817, 819
 - `\c_sys_minute_int` 228, 817, 817
 - `\c_sys_month_int` 228, 817, 817
 - `\c_sys_output_str` . . 229, 818, 819, 819
 - `\c_sys_year_int` 228, 817, 817
- T**
- `\tabskip` 247
 - `\tagcode` 252
 - `tan` 206
 - `tand` 206
 - `\tate` 258
 - `\tbaselineshift` 258
 - `\tempa` 235, 235, 236, 236, 237, 237
 - ten commands:
 - `\c_ten` 76, 325, 326, 361,
 - 362, 367, 367, 392, 607, 636, 636,
 - 636, 636, 636, 637, 663, 705, 741, 812
 - `\c_ten_thousand`
 - . 76, 368, 368, 685, 686, 686, 689, 692
 - term commands:
 - `\c_term_...` 185
 - `\c_term_ior` 190, 556, 556, 557, 559, 563
 - `\c_term_iow`
 - 190, 561, 561, 561, 562, 564, 564
 - TeX and L^AT_EX 2_ε commands:
 - `\...mark` 331, 337
 - `\@` 236, 426, 611
 - `\@@end` 258, 258, 258
 - `\@@hyph` 259
 - `\@@input` 259
 - `\@@italiccorr` 259
 - `\@@underline` 259
 - `\@addtofilelist` 555
 - `\@currname` 551, 551, 551
 - `\@filelist`
 - 551, 554, 555, 555, 556, 556, 556, 556
 - `\@firstoftwo` 266
 - `\@makeother` 324, 324
 - `\@sanitize` 52,
 - 53, 53, 322, 323, 323, 324, 324, 324, 324
 - `\@secondoftwo` 266
 - `\@tempa` 239, 239
 - `\@unexpandable@protect` . 610, 611, 611
 - `\botmark` 337
 - `\box` 148
 - `\chardef` 333, 348
 - `\copy` 148
 - `\count` 334
 - `\countdef` 334
 - `\cr` 317
 - `\csname` 18
 - `\currentgrouplevel` 367, 524, 781
 - `\currentgrouptype` 367, 524, 781
 - `\detokenize` 400
 - `\dimen` 334, 334
 - `\dimendef` 334
 - `\dimexpr` 93
 - `\directlua` 230
 - `\dospecials` 52, 53, 53, 322
 - `\dp` 148
 - `\edef` 2, 384
 - `\endcsname` 18
 - `\endinput` 163
 - `\endlinechar`
 - . . 97, 98, 337, 390, 390, 391, 391, 391
 - `\endtemplate` 44, 317
 - `\errhelp` 505, 506

- \errmessage ... 505, 506, 506, 506, 507
- \errorcontextlines . 191, 475, 506, 525
- \escapechar ... 102, 102, 102, 273, 566
- \everyeof 389, 391
- \expandafter 32, 34
- \expanded 253
- \firstmark 299, 337
- \frozen@everydisplay 258
- \frozen@everymath 259
- \futurelet 317, 339, 341
- \global 241
- \halign 44, 317
- \hskip 90
- \ht 149
- \hyphen 337, 337
- \ifcase 77
- \ifdim 93
- \ifeof 190
- \iffalse 39
- \ifhbox 154
- \ifnum 77
- \ifodd 77, 815
- \iftrue 39
- \ifvbox 154
- \ifvoid 154
- \ifx 23, 239
- \input@path 553, 553, 554
- \italiccorr 337, 337
- \jobname 228
- \l@expl@check@declarations@bool .
..... 279, 306, 387, 519
- \l@expl@log@functions@bool
..... 277, 279, 503
- \let 241
- \LGR@accDialytika 810, 810, 810
- \LGR@accdropped
..... 810, 810, 810, 810, 810, 810
- \LGR@hiatus ... 810, 810, 810, 810, 810
- \lower 769
- \lowercase 98, 322, 323
- \luaescapestring 231
- \m@ne 264
- \makeatletter 7
- \MakeUppercase 224
- \mathchardef 333, 348
- \meaning 17, 56, 334, 341, 815
- \newif 39
- \newlinechar . 97, 98, 191, 278, 390,
390, 390, 390, 391, 506, 525, 564, 564
- \newread 558, 558, 558
- \newwrite 562
- \noexpand 33
- \nullfont 337, 337, 337
- \number 78, 660
- \numexpr 78, 323
- \or 77
- \outer 239, 323, 558, 558, 562, 815, 815
- \pageheight 253
- \par 502
- \pdfcolorstack 827
- \pdfliteral 822
- \pdfmapfile 260
- \pdfmapline 260
- \pdfpageheight 253
- \pdfsave 822
- \pdfstrcmp
235, 235, 235, 237, 237, 238, 239, 252
- \protect 609, 610, 610, 610, 610
- \protected@edef 568, 568
- \ProvidesClass 7
- \ProvidesFile 7
- \ProvidesPackage 7
- \read 186
- \readline 187
- \relax 239,
269, 275, 287, 572, 574, 574, 595, 625
- \RequirePackage 7, 239
- \reserveinserts 239, 239
- \robustify 223
- \romannumeral 77
- \scantokens 389, 390, 391
- \set@color 502, 502, 502
- \show 17, 107, 287
- \showbox 474
- \showthe 287, 367, 377, 380, 383
- \showtokens 108, 169, 523, 524, 524, 525
- \space 337, 337
- \splitbotmark 337
- \splitfirstmark 337
- \strcmp 235, 252
- \string 56
- \synctex 252
- \tex_lowercase:D 814
- \the 68, 85, 89, 92, 291, 291
- \topmark 337
- \tracingonline 475
- \Ucharcat 236, 812, 813, 813
- \ucharcat@table
..... 236, 236, 236, 236, 237, 237
- \Ucharchar 236

- `\unexpanded` 33, 103, 104,
104, 107, 122, 126, 126, 134, 137,
137, 139, 143, 222, 352, 384, 407, 408
- `\unhbox` 152
- `\unhcopy` 152
- `\unless` 23
- `\unvbox` 153
- `\unvcopy` 153
- `\uppercase` 98
- `\valign` 317
- `\vbox` 152
- `\vskip` 90
- `\vsplit` 153
- `\vtop` 152, 482
- `\wd` 149
- `\write` 188, 564
- tex commands:
 - `\tex_...` 9
 - `\tex_above:D` 242
 - `\tex_abovedisplayshortskip:D` .. 242
 - `\tex_abovedisplayskip:D` 242
 - `\tex_abovewithdelims:D` 242
 - `\tex_accent:D` 242
 - `\tex_adjdemerits:D` 242
 - `\tex_advance:D` 242, 350, 350,
350, 350, 371, 371, 379, 379, 382, 382
 - `\tex_afterassignment:D` 242, 339
 - `\tex_aftergroup:D` 242, 263
 - `\tex_atop:D` 242
 - `\tex_atopwithdelims:D` 242
 - `\tex_badness:D` 242
 - `\tex_baselineskip:D` 242
 - `\tex_batchmode:D` 242
 - `\tex_begingroup:D` 242, 263
 - `\tex_belowdisplayshortskip:D` .. 242
 - `\tex_belowdisplayskip:D` 242
 - `\tex_binoppenalty:D` 242
 - `\tex_botmark:D` 242
 - `\tex_box:D` 242, 472, 472
 - `\tex_boxmaxdepth:D` 242
 - `\tex_brokenpenalty:D` 242
 - `\tex_catcode:D`
.... 242, 298, 298, 322, 323, 323,
323, 323, 323, 324, 324, 324, 324, 391
 - `\tex_char:D` 242
 - `\tex_chardef:D` 242,
262, 264, 264, 264, 264, 273, 273,
306, 306, 307, 307, 337, 348, 559, 562
 - `\tex_cleaders:D` 242
 - `\tex_closein:D` 242, 559
 - `\tex_closeout:D` 242, 563
 - `\tex_clubpenalty:D` 242
 - `\tex_copy:D` 242, 471, 472
 - `\tex_count:D` 242, 334, 557, 557, 561, 561
 - `\tex_countdef:D` 242, 264, 334
 - `\tex_cr:D` 242
 - `\tex_crcr:D` 242
 - `\tex_csname:D` 242, 260, 260, 262
 - `\tex_day:D` 242, 817
 - `\tex_deadcycles:D` 243
 - `\tex_def:D` . 243, 263, 263, 263, 263, 264
 - `\tex_defaultthyphenchar:D` 243
 - `\tex_defaultskewchar:D` 243
 - `\tex_delcode:D` 243
 - `\tex_delimiter:D` 243
 - `\tex_delimiterfactor:D` 243
 - `\tex_delimitershortfall:D` 243
 - `\tex_dimen:D` 243, 334
 - `\tex_dimendef:D` 243, 334
 - `\tex_discretionary:D` 243
 - `\tex_displayindent:D` 243
 - `\tex_displaylimits:D` 243
 - `\tex_displaystyle:D` 243
 - `\tex_displaywidowpenalty:D` 243
 - `\tex_displaywidth:D` 243
 - `\tex_divide:D` 243
 - `\tex_doublehyphenemerits:D` ... 243
 - `\tex_dp:D` 243, 472
 - `\tex_dump:D` 243
 - `\tex_edef:D` 243, 264
 - `\tex_else:D` 243, 262, 264
 - `\tex_emergencystretch:D` 243
 - `\tex_end:D` 243, 258, 261, 278, 509
 - `\tex_endcsname:D` ... 243, 260, 260, 262
 - `\tex_endgroup:D` 243, 258, 263
 - `\tex_endinput:D` 243, 510
 - `\tex_endlinechar:D` . 240, 240, 241,
243, 390, 390, 390, 392, 560, 560, 560
 - `\tex_eqno:D` 243
 - `\tex_errhelp:D` 243, 506
 - `\tex_errmessage:D` 243, 278, 507
 - `\tex_errorcontextlines:D`
..... 243, 475, 507, 508, 525
 - `\tex_errorstopmode:D` 243
 - `\tex_escapechar:D` .. 243, 566, 567, 567
 - `\tex_everycr:D` 243
 - `\tex_everydisplay:D` 243, 258
 - `\tex_everyhbox:D` 243
 - `\tex_everyjob:D`
243, 261, 551, 551, 551, 551, 816, 816

<code>\tex_everymath:D</code>	243, 259	<code>\tex_ifdim:D</code>	244, 369
<code>\tex_everypar:D</code>	243	<code>\tex_ifeof:D</code>	244, 560
<code>\tex_everyvbox:D</code>	243	<code>\tex_iffalse:D</code>	244, 262
<code>\tex_exhyphenpenalty:D</code>	243	<code>\tex_ifhbox:D</code>	244, 473
<code>\tex_expandafter:D</code> .	243, 260, 260, 262	<code>\tex_ifhmode:D</code>	244, 262
<code>\tex_fam:D</code>	243	<code>\tex_ifinner:D</code>	244, 262
<code>\tex_fi:D</code>		<code>\tex_ifmmode:D</code>	244, 262
.	243, 259, 260, 260, 260, 260, 260,	<code>\tex_ifnum:D</code>	244, 260, 263
261, 261, 261, 261, 261, 262, 264,	389	<code>\tex_ifodd:D</code>	
<code>\tex_finalhyphendemerits:D</code>	243	244, 277, 279, 279, 305, 305, 345, 387	
<code>\tex_firstmark:D</code>	243	<code>\tex_iftrue:D</code>	244, 262
<code>\tex_floatingpenalty:D</code>	243	<code>\tex_ifvbox:D</code>	244, 473
<code>\tex_font:D</code>	243	<code>\tex_ifvmode:D</code>	244, 262
<code>\tex_fontdimen:D</code>	243	<code>\tex_ifvoid:D</code>	244, 473
<code>\tex_fontname:D</code>	244	<code>\tex_ifx:D</code>	244, 262
<code>\tex_futurelet:D</code>	244, 339, 339	<code>\tex_ignorespaces:D</code>	244
<code>\tex_gdef:D</code>	244, 265	<code>\tex_immediate:D</code>	
<code>\tex_global:D</code> 241, 241, 241, 244, 281,		244, 277, 277, 562, 563, 564	
282, 292, 306, 307, 328, 328, 328,		<code>\tex_indent:D</code>	244
339, 348, 349, 349, 350, 350, 350,		<code>\tex_input:D</code> 244, 259, 261, 555, 788, 788	
350, 350, 370, 370, 370, 371, 371,		<code>\tex_inputlineno:D</code>	244, 278, 505
378, 378, 378, 379, 379, 381, 382,		<code>\tex_insert:D</code>	244
382, 382, 382, 471, 472, 473, 475,		<code>\tex_insertpenalties:D</code>	244
475, 476, 477, 477, 477, 477, 559, 562		<code>\tex_interlinepenalty:D</code>	244
<code>\tex_globaldefs:D</code>	244	<code>\tex_italiccorrection:D</code> 242, 259, 261	
<code>\tex_halign:D</code>	244	<code>\tex_jobname:D</code>	244, 551, 817, 817
<code>\tex_hangafter:D</code>	244	<code>\tex_kern:D</code>	
<code>\tex_hangindent:D</code>	244	245, 491, 491, 493, 493, 500, 500,	
<code>\tex_hbadness:D</code>	244	762, 768, 768, 769, 769, 770, 770, 772	
<code>\tex_hbox:D</code>		<code>\tex_language:D</code>	245, 261
....	244, 475, 475, 475, 476, 476, 476	<code>\tex_lastbox:D</code>	245, 473
<code>\tex_hfil:D</code>	244	<code>\tex_lastkern:D</code>	245
<code>\tex_hfill:D</code>	244	<code>\tex_lastpenalty:D</code>	245
<code>\tex_hfilneg:D</code>	244	<code>\tex_lastskip:D</code>	245
<code>\tex_hfuzz:D</code>	244	<code>\tex_lccode:D</code> .	245, 326, 326, 418, 418
<code>\tex_hoffset:D</code>	244, 261	<code>\tex_leaders:D</code>	245
<code>\tex_holdinginserts:D</code>	244	<code>\tex_left:D</code>	245, 261
<code>\tex_hrule:D</code>	244	<code>\tex_lefthyphenmin:D</code>	245
<code>\tex_hsize:D</code>		<code>\tex_leftskip:D</code>	245
....	244, 482, 482, 482, 483, 483, 483	<code>\tex_leqno:D</code>	245
<code>\tex_hskip:D</code>	244, 380	<code>\tex_let:D</code>	
<code>\tex_hss:D</code>	244, 476, 476, 768, 768	241, 241, 241, 245, 258, 258,
<code>\tex_ht:D</code>	244, 472	259, 259, 259, 259, 259, 259, 259,	
<code>\tex_hyphen:D</code>	242, 259	259, 259, 259, 259, 259, 259, 259,	
<code>\tex_hyphenation:D</code>	244	259, 259, 259, 259, 259, 259, 259,	
<code>\tex_hyphenchar:D</code>	244	259, 259, 259, 259, 259, 259, 259,	
<code>\tex_hyphenpenalty:D</code>	244	259, 259, 259, 259, 259, 259, 259,	
<code>\tex_if:D</code>	51, 244, 262, 262	259, 259, 259, 259, 259, 259, 259,	
<code>\tex_ifcase:D</code>	244, 345	260, 260, 260, 260, 260, 260, 260,	
<code>\tex_ifcat:D</code>	244, 262	260, 260, 260, 260, 260, 260, 260,	

260, 260, 260, 260, 260, 260, 260,	
260, 260, 260, 260, 260, 260, 260,	
260, 260, 261, 261, 261, 261, 261,	
261, 261, 261, 261, 261, 261, 261,	
261, 261, 261, 261, 261, 261, 261,	
261, 262, 262, 262, 262, 262, 262,	
262, 262, 262, 262, 262, 262, 262,	
262, 262, 262, 262, 262, 262, 262,	
262, 262, 262, 262, 263, 263, 263,	
263, 263, 263, 263, 263, 264, 264,	
264, 264, 265, 265, 281, 328, 328, 328	
<code>\tex_limits:D</code>	245
<code>\tex_linepenalty:D</code>	245
<code>\tex_lineskip:D</code>	245
<code>\tex_lineskiplimit:D</code>	245
<code>\tex_long:D</code>	245, 263, 263, 263,
264, 265, 265, 265, 265, 265, 265, 265	
<code>\tex_looseness:D</code>	245
<code>\tex_lower:D</code>	245, 473
<code>\tex_lowercase:D</code>	
....	245, 323, 324, 333, 336, 391,
392, 392, 418, 507, 521, 566, 811, 814	
<code>\tex_mag:D</code>	245
<code>\tex_mark:D</code>	245
<code>\tex_mathaccent:D</code>	245
<code>\tex_mathbin:D</code>	245
<code>\tex_mathchar:D</code>	245
<code>\tex_mathchardef:D</code>	
.....	245, 264, 264, 349, 349
<code>\tex_mathchoice:D</code>	245
<code>\tex_mathclose:D</code>	245
<code>\tex_mathcode:D</code>	245, 326, 326
<code>\tex_mathinner:D</code>	245
<code>\tex_mathop:D</code>	245, 261
<code>\tex_mathopen:D</code>	245
<code>\tex_mathord:D</code>	245
<code>\tex_mathpunct:D</code>	245
<code>\tex_mathrel:D</code>	245
<code>\tex_mathsurround:D</code>	245
<code>\tex_maxdeadcycles:D</code>	245
<code>\tex_maxdepth:D</code>	245
<code>\tex_meaning:D</code>	245, 262, 262
<code>\tex_medmuskip:D</code>	245
<code>\tex_message:D</code>	245
<code>\tex_middle:D</code>	261
<code>\tex_mkern:D</code>	245
<code>\tex_month:D</code>	245, 261, 817
<code>\tex_moveleft:D</code>	245, 472
<code>\tex_moveright:D</code>	245, 473
<code>\tex_mskip:D</code>	245
<code>\tex_multiply:D</code>	245
<code>\tex_muskip:D</code>	245, 335
<code>\tex_muskipdef:D</code>	245, 335
<code>\tex_newlinechar:D</code>	
246, 278, 390, 391, 391, 507, 525, 564	
<code>\tex_noalign:D</code>	246
<code>\tex_noboundary:D</code>	246
<code>\tex_noexpand:D</code>	246, 262
<code>\tex_noindent:D</code>	246
<code>\tex_nolimits:D</code>	246
<code>\tex_nonscript:D</code>	246
<code>\tex_nonstopmode:D</code>	246
<code>\tex_nulldelimiterspace:D</code>	246
<code>\tex_nullfont:D</code>	246, 338
<code>\tex_number:D</code>	246, 345
<code>\tex_omit:D</code>	246
<code>\tex_openin:D</code>	246, 559
<code>\tex_openout:D</code>	246, 562
<code>\tex_or:D</code>	246, 262
<code>\tex_outer:D</code>	246, 261
<code>\tex_output:D</code>	246
<code>\tex_outputpenalty:D</code>	246
<code>\tex_over:D</code>	246, 261
<code>\tex_overfullrule:D</code>	246
<code>\tex_overline:D</code>	246
<code>\tex_overwithdelims:D</code>	246
<code>\tex_pagedepth:D</code>	246
<code>\tex_pagefilllstretch:D</code>	246
<code>\tex_pagefillstretch:D</code>	246
<code>\tex_pagefilstretch:D</code>	246
<code>\tex_pagegoal:D</code>	246
<code>\tex_pageshrink:D</code>	246
<code>\tex_pagestretch:D</code>	246
<code>\tex_pagetotal:D</code>	246
<code>\tex_par:D</code>	246, 502
<code>\tex_parfillskip:D</code>	246
<code>\tex_parindent:D</code>	246
<code>\tex_parshape:D</code>	246
<code>\tex_parskip:D</code>	246
<code>\tex_patterns:D</code>	246
<code>\tex_pausing:D</code>	246
<code>\tex_penalty:D</code>	246
<code>\tex_postdisplaypenalty:D</code>	246
<code>\tex_predisdisplaypenalty:D</code>	246
<code>\tex_predisplaysize:D</code>	246
<code>\tex_pretolerance:D</code>	246
<code>\tex_prevdepth:D</code>	246
<code>\tex_prevgraf:D</code>	246
<code>\tex_radical:D</code>	246

<code>\tex_raise:D</code>	246, 473	<code>\tex_thinmuskip:D</code>	247
<code>\tex_read:D</code>	246, 560	<code>\tex_time:D</code>	247, 817, 817
<code>\tex_relax:D</code> 246, 263, 278, 345, 369, 575		<code>\tex_toks:D</code>	247, 335
<code>\tex_relpemalty:D</code>	246	<code>\tex_toksdef:D</code>	247, 335
<code>\tex_right:D</code>	246, 261	<code>\tex_tolerance:D</code>	247
<code>\tex_righthyphenmin:D</code>	247	<code>\tex_topmark:D</code>	247
<code>\tex_rightskip:D</code>	247	<code>\tex_topskip:D</code>	247
<code>\tex_romannumeral:D</code> 247, 262, 263,		<code>\tex_tracingcommands:D</code>	247
273, 273, 273, 273, 274, 274, 297, 297		<code>\tex_tracinglostchars:D</code>	247
<code>\tex_romannumeral:D</code>	298	<code>\tex_tracingmacros:D</code>	247
<code>\tex_scriptfont:D</code>	247	<code>\tex_tracingonline:D</code>	247, 475
<code>\tex_scriptscriptfont:D</code>	247	<code>\tex_tracingoutput:D</code>	247
<code>\tex_scriptscriptstyle:D</code>	247	<code>\tex_tracingpages:D</code>	247
<code>\tex_scriptspace:D</code>	247	<code>\tex_tracingparagraphs:D</code>	247
<code>\tex_scriptstyle:D</code>	247	<code>\tex_tracingrestores:D</code>	248
<code>\tex_scrollmode:D</code>	247	<code>\tex_tracingstats:D</code>	248
<code>\tex_setbox:D</code> .. 247, 471, 472, 473,		<code>\tex_uccode:D</code>	248, 327, 327
475, 475, 476, 477, 477, 477, 477, 478		<code>\tex_uchyph:D</code>	248
<code>\tex_setlanguage:D</code>	247	<code>\tex_undefined:D</code> 241, 241, 260, 260,	
<code>\tex_sfcode:D</code>	247, 327, 327	261, 261, 261, 282, 282, 337, 337,	
<code>\tex_shipout:D</code>	247	337, 537, 537, 587, 587, 587, 587, 587	
<code>\tex_show:D</code>	247	<code>\tex_underline:D</code>	248, 259
<code>\tex_showbox:D</code>	247, 475	<code>\tex_unhbox:D</code>	248, 476
<code>\tex_showboxbreadth:D</code>	247, 475	<code>\tex_unhcopy:D</code>	248, 476
<code>\tex_showboxdepth:D</code>	247, 475	<code>\tex_unkern:D</code>	248
<code>\tex_showlists:D</code>	247	<code>\tex_unpenalty:D</code>	248
<code>\tex_showthe:D</code>	247	<code>\tex_unskip:D</code>	248
<code>\tex_skewchar:D</code>	247	<code>\tex_unvbox:D</code>	248, 478
<code>\tex_skip:D</code>	247, 335	<code>\tex_unvcopy:D</code>	248, 478
<code>\tex_skipdef:D</code>	247, 335	<code>\tex_uppercase:D</code>	248, 393
<code>\tex_space:D</code>	242	<code>\tex_vadjust:D</code>	248
<code>\tex_spacefactor:D</code>	247	<code>\tex_valign:D</code>	248
<code>\tex_spaceskip:D</code>	247	<code>\tex_vbadness:D</code>	248
<code>\tex_span:D</code>	247	<code>\tex_vbox:D</code>	
<code>\tex_special:D</code> 247, 822, 822,	 248, 476, 477, 477, 477, 477, 477	
822, 822, 823, 823, 823, 823, 827, 827		<code>\tex_vcenter:D</code>	248, 261
<code>\tex_splitbotmark:D</code>	247	<code>\tex_vfil:D</code>	248
<code>\tex_splitfirstmark:D</code>	247	<code>\tex_vfill:D</code>	248
<code>\tex_splitmaxdepth:D</code>	247	<code>\tex_vfilneg:D</code>	248
<code>\tex_splittopskip:D</code>	247	<code>\tex_vfuzz:D</code>	248
<code>\tex_string:D</code>	247, 263	<code>\tex_voffset:D</code>	248, 261
<code>\tex_tabskip:D</code>	247	<code>\tex_vrule:D</code>	248, 497, 498
<code>\tex_textfont:D</code>	247	<code>\tex_vsize:D</code>	248
<code>\tex_textstyle:D</code>	247	<code>\tex_vskip:D</code>	248, 380
<code>\tex_the:D</code>		<code>\tex_vsplit:D</code>	248, 478
.. 240, 247, 278, 287, 291, 291, 292,		<code>\tex_vss:D</code>	248
324, 326, 326, 327, 327, 350, 351,		<code>\tex_vtop:D</code>	248, 476, 477
367, 375, 375, 380, 380, 383, 474,		<code>\tex_wd:D</code>	248, 472
551, 551, 606, 611, 611, 612, 626, 816		<code>\tex_widowpenalty:D</code>	248
<code>\tex_thickmuskip:D</code>	247	<code>\tex_write:D</code> 248, 277, 277, 563, 563, 564	

<code>\tex_xdef:D</code>	248, 265	<code>\textsampi</code>	810
<code>\tex_xleaders:D</code>	248	<code>\textSigma</code>	810, 810, 810
<code>\tex_xspaceskip:D</code>	248	<code>\textsigma</code>	810, 810
<code>\tex_year:D</code>	248, 817	<code>\textStigma</code>	810, 810
tex... commands:		<code>\textstigma</code>	810
<code>\tex...:D</code>	262	<code>\textstyle</code>	247
<code>\textAlpha</code>	809	<code>\textTau</code>	810
<code>\textalpha</code>	809	<code>\texttau</code>	810
<code>\textautosigma</code>	810	<code>\textTheta</code>	810
<code>\textBeta</code>	809	<code>\texttheta</code>	810
<code>\textbeta</code>	809	<code>\textUpsilon</code>	810
<code>\textChi</code>	809	<code>\textupsilon</code>	810
<code>\textchi</code>	809	<code>\textvarsigma</code>	810
<code>\textDelta</code>	809	<code>\textvarstigma</code>	810
<code>\textdelta</code>	809	<code>\textXi</code>	810
<code>\textDigamma</code>	809	<code>\textxi</code>	810
<code>\textdigamma</code>	809	<code>\textZeta</code>	810
<code>\textdir</code>	254	<code>\textzeta</code>	810
<code>\textEpsilon</code>	809	<code>\TeXeTstate</code>	249
<code>\textepsilon</code>	809	<code>\tfont</code>	258
<code>\textEta</code>	809	<code>\TH</code>	807
<code>\texteta</code>	809	<code>\th</code>	807
<code>\textfont</code>	247	<code>\the</code>	236, 237, 239, 239, 239, 239, 239, 239, 239, 240, 240, 240, 247
<code>\textGamma</code>	809	<code>\thickmuskip</code>	247
<code>\textgamma</code>	809	<code>\thinmuskip</code>	247
<code>\textIota</code>	809	thirteen commands:	
<code>\textiota</code>	809	<code>\c_thirteen</code>	76, 323, 323, 325, 326, 367, 368, 734, 735, 812, 812
<code>\textKappa</code>	809	thirty commands:	
<code>\textkappa</code>	809	<code>\c_thirty_two</code> 76, 368, 368, 613, 614, 633	
<code>\textLambda</code>	809	three commands:	
<code>\textlambda</code>	809	<code>\c_three</code>	76, 325, 326, 367, 367, 392, 424, 577, 602, 613, 614, 633, 638, 638, 638, 654, 663, 727, 743
<code>\textMu</code>	809	tilde commands:	
<code>\textmu</code>	809	<code>\c_tilde_str</code>	117, 426, 427
<code>\textNu</code>	809	<code>\time</code>	247
<code>\textnu</code>	809	<code>\tiny</code>	497
<code>\textOmega</code>	809	tl commands:	
<code>\textomega</code>	809	<code>\tl_(g)clear:N</code>	95
<code>\textOmicron</code>	809	<code>\tl_...:N</code>	109
<code>\textomicron</code>	809	<code>\c__tl_accents_lt_tl</code>	798
<code>\textPhi</code>	809	<code>\tl_act</code>	405
<code>\textphi</code>	809	<code>__tl_act:NNNnn</code>	405, 405, 405, 405, 407, 407, 407, 786, 787, 787
<code>\textPi</code>	809	<code>__tl_act_count_group:nn</code> 787, 787, 787	
<code>\textpi</code>	809	<code>__tl_act_count_normal:nN</code>	
<code>\textPsi</code>	809	<code>__tl_act_count_space:n</code> 787, 787, 787	
<code>\textpsi</code>	809		
<code>\textQoppa</code>	809		
<code>\textqoppa</code>	809		
<code>\textRho</code>	810		
<code>\textrho</code>	810		
<code>\textSampi</code>	810		

- _tl_act_end:w 405
- _tl_act_end:wn 406, 406, 787
- _tl_act_group:nwnNNN . 405, 406, 406
- _tl_act_group_recurse:Nnn 787, 787, 787
- _tl_act_loop:w 405, 405, 406, 406, 406, 406
- \q_tl_act_mark 321, 321, 405, 405, 405, 405, 406, 406
- _tl_act_normal:NwnNNN 405, 406, 406
- _tl_act_output:n 405, 406, 406
- _tl_act_result:n 405, 405, 406, 406, 406, 406, 406
- _tl_act_reverse 406
- _tl_act_reverse_output:n 405, 406, 407, 407, 407, 787
- _tl_act_space:wwnNNN 405, 406, 406, 406
- \q_tl_act_stop 321, 321, 405, 405, 405, 406, 406, 406, 406, 406, 406, 406, 406
- \tl_case:cn 400
- \tl_case:cnTF 400
- \tl_case:Nn 100, 400, 400, 401
- \tl_case:nn(TF) 417
- \tl_case:NnF 400, 401
- \tl_case:NnT 400, 401
- _tl_case:NnTF 400, 400, 400, 400, 400
- \tl_case:NnTF . 100, 100, 400, 400, 401
- _tl_case:nnTF 400
- _tl_case:Nw 400, 400, 400, 400
- \l_tl_case_change_exclude_tl ... 224, 224, 224, 795, 805, 805, 805
- \l_tl_case_change_math_tl 223, 223, 791, 802, 805, 805, 805
- _tl_case_end:nw 400, 400, 401
- _tl_change_case:nnn 789, 789, 789, 789, 789, 789
- \c_tl_change_case_acc_lower_tl 810
- \c_tl_change_case_acc_upper_tl . 806, 806, 810
- _tl_change_case_aux:nnn 789, 789, 789, 790
- _tl_change_case_char:Nn 789, 793, 793, 804
- _tl_change_case_char:Nnn 789, 792, 793
- _tl_change_case_char:NNNNNNNNn 789, 793, 793
- _tl_change_case_cs:N 789, 792, 792, 795, 803, 803
- _tl_change_case_cs:NN 789, 795, 795, 795
- _tl_change_case_cs:NNn 789, 795, 795
- _tl_change_case_cs:Nnnn 789, 792, 794, 803
- _tl_change_case_cs:nNnnn 789, 794, 794
- _tl_change_case_cs_cyr:NnNNNNw 789, 794, 794
- _tl_change_case_cs_expand:NN . 789, 796, 796
- _tl_change_case_cs_expand:Nnw . 789, 795, 796
- _tl_change_case_cs_four:NNNNw . 789, 794, 794
- _tl_change_case_cs_three:NNNw . 789, 794, 794, 794
- _tl_change_case_cs_type:nnn ... 789, 795, 795
- _tl_change_case_cs_type:Nnnnn . 789, 794, 794, 794, 794, 794, 795
- \c_tl_change_case_cyrillic-lower_i_tl 806
- \c_tl_change_case_cyrillic-lower_ii_tl 806
- \c_tl_change_case_cyrillic-lower_iii_tl 806
- \c_tl_change_case_cyrillic-lower_iv_tl 806
- \c_tl_change_case_cyrillic-upper_i_tl 806
- \c_tl_change_case_cyrillic-upper_ii_tl 806
- \c_tl_change_case_cyrillic-upper_iii_tl 806
- \c_tl_change_case_cyrillic-upper_iv_tl 806
- _tl_change_case_end:wn 789, 790, 791, 792, 802
- \c_tl_change_case_greek_lower_tl 806
- \c_tl_change_case_greek_upper_tl 806
- _tl_change_case_group:nwnn ... 789, 789, 790
- _tl_change_case_if_expandable:NTF 789, 796, 796, 797, 797, 799, 800, 804

- \c_tl_change_case_latin_lower_tl [806](#)
- \c_tl_change_case_latin_upper_tl [806](#)
- _tl_change_case_loop:wn [797](#)
- _tl_change_case_loop:wnn [789, 789, 789, 790, 790, 790, 791, 792, 792, 802, 803, 803](#)
- _tl_change_case_lower_az:Nnw .. [797, 798](#)
- _tl_change_case_lower_lt:nNnw .. [798, 798, 798](#)
- _tl_change_case_lower_lt:NNw .. [798, 799, 799](#)
- _tl_change_case_lower_lt:Nnw .. [798, 798](#)
- _tl_change_case_lower_lt:nnw .. [798, 798, 799](#)
- _tl_change_case_lower_lt:Nw ... [798, 799, 799, 799](#)
- _tl_change_case_lower_sigma:Nnw [796, 796](#)
- _tl_change_case_lower_sigma:Nw [796, 796, 797](#)
- _tl_change_case_lower_sigma:w .. [796, 796, 796, 797](#)
- _tl_change_case_lower_tr:Nnw .. [797, 797, 798](#)
- _tl_change_case_lower_tr_auxi:Nw [797, 797, 797, 798](#)
- _tl_change_case_lower_tr_auxii:Nw [797, 797, 797](#)
- _tl_change_case_map:NN [806, 806, 806](#)
- _tl_change_case_map:NNN [806, 806, 806](#)
- _tl_change_case_math:NNNnnn ... [789, 791, 791, 791, 802](#)
- _tl_change_case_math:NwNNnn ... [789, 791, 791](#)
- _tl_change_case_math_group:nwNNnn [789, 791, 792](#)
- _tl_change_case_math_loop:wNNnn [789, 791, 791, 792, 792, 792](#)
- _tl_change_case_math_space:wNNnn [789, 791, 792](#)
- \c_tl_change_case_misc_lower_tl [806, 810](#)
- \c_tl_change_case_misc_upper_tl [806, 810](#)
- _tl_change_case_mixed_nl:NNw .. [804, 804, 804](#)
- _tl_change_case_mixed_nl:Nnw .. [804, 804](#)
- _tl_change_case_mixed_nl:Nw ... [804, 804, 804, 805](#)
- _tl_change_case_N_type:Nnnn ... [789, 791, 792](#)
- _tl_change_case_N_type:NNNnnn .. [789, 791, 791, 791](#)
- _tl_change_case_N_type:Nwnn ... [789, 789, 790](#)
- _tl_change_case_output:fwn ... [789, 793, 796](#)
- _tl_change_case_output:nwn [789, 790, 790, 790, 791, 792, 792, 792, 795, 795, 795, 797, 798, 799, 799, 800, 802, 804, 804, 804, 805](#)
- _tl_change_case_output:own ... [789, 790, 802](#)
- _tl_change_case_output:Vwn ... [789, 797, 798, 798, 800, 801](#)
- _tl_change_case_result:n [789, 790, 790, 790, 801](#)
- _tl_change_case_setup:nnnn ... [806, 806, 807, 807, 808, 809, 809](#)
- _tl_change_case_space:wnn [789, 790, 790](#)
- _tl_change_case_upper_az:Nnw .. [797, 798](#)
- _tl_change_case_upper_de-alt:Nnw [801](#)
- _tl_change_case_upper_lt:NNw .. [798, 800, 800](#)
- _tl_change_case_upper_lt:Nnw .. [798, 800](#)
- _tl_change_case_upper_lt:nnw .. [798, 800, 800](#)
- _tl_change_case_upper_lt:Nw ... [798, 800, 800, 800](#)
- _tl_change_case_upper_sigma:Nnw [796, 797](#)
- _tl_change_case_upper_tr:Nnw .. [797, 798, 798](#)
- \tl_clear:c [385, 445](#)
- \tl_clear:N [95, 95, 385, 385, 385, 385, 445, 525, 526, 544, 546, 568, 568, 570](#)
- \tl_clear_new:c [385, 445](#)
- \tl_clear_new:N [95, 95, 385, 385, 385, 445](#)

- \tl_concat:ccc [385](#)
- \tl_concat:NNN [95](#), [95](#), [385](#), [385](#), [385](#), [388](#)
- \tl_const:cn [384](#), [814](#)
- \tl_const:cx [384](#), [566](#), [806](#), [806](#)
- \tl_const:Nn [95](#), [95](#), [318](#),
[328](#), [384](#), [384](#), [384](#), [386](#), [386](#), [428](#),
[462](#), [503](#), [503](#), [504](#), [504](#), [505](#), [505](#),
[505](#), [505](#), [505](#), [505](#), [505](#), [529](#), [529](#),
[529](#), [566](#), [566](#), [575](#), [575](#), [575](#), [575](#),
[575](#), [685](#), [703](#), [703](#), [703](#), [703](#), [703](#),
[703](#), [703](#), [703](#), [703](#), [810](#), [810](#), [810](#), [810](#)
- \tl_const:Nx
..... [384](#), [384](#), [384](#), [389](#), [445](#), [566](#), [757](#)
- \tl_count:c [403](#)
- \tl_count:N
..... [100](#), [103](#), [103](#), [103](#), [403](#), [403](#), [403](#), [569](#)
- __tl_count:n . [403](#), [403](#), [403](#), [403](#), [403](#)
- \tl_count:n ... [100](#), [103](#), [103](#), [103](#),
[269](#), [269](#), [283](#), [283](#), [285](#), [347](#), [403](#),
[403](#), [403](#), [412](#), [423](#), [423](#), [585](#), [642](#), [642](#)
- \tl_count:o [403](#)
- \tl_count:V [403](#)
- \tl_count_tokens:n
..... [222](#), [222](#), [787](#), [787](#), [787](#)
- __tl_from_file_do:w .. [787](#), [788](#), [788](#)
- \tl_gclear:c [385](#), [445](#)
- \tl_gclear:N [95](#), [385](#), [385](#), [385](#), [445](#)
- \tl_gclear_new:c [385](#), [445](#)
- \tl_gclear_new:N [95](#), [385](#), [385](#), [385](#), [445](#)
- \tl_gconcat:ccc [385](#)
- \tl_gconcat:NNN [95](#), [385](#), [385](#), [385](#), [389](#)
- \tl_gput_left:cn [386](#)
- \tl_gput_left:co [386](#)
- \tl_gput_left:cV [386](#)
- \tl_gput_left:cx [386](#)
- \tl_gput_left:Nn [96](#), [386](#), [386](#), [387](#), [388](#)
- \tl_gput_left:No ... [386](#), [387](#), [387](#), [388](#)
- \tl_gput_left:NV ... [386](#), [386](#), [387](#), [388](#)
- \tl_gput_left:Nx ... [386](#), [387](#), [387](#), [388](#)
- \tl_gput_right:cn [387](#)
- \tl_gput_right:co [387](#)
- \tl_gput_right:cV [387](#)
- \tl_gput_right:cx [387](#)
- \tl_gput_right:Nn
..... [96](#), [322](#), [387](#), [387](#), [387](#), [388](#), [432](#)
- \tl_gput_right:No .. [387](#), [387](#), [387](#), [388](#)
- \tl_gput_right:NV .. [387](#), [387](#), [387](#), [388](#)
- \tl_gput_right:Nx .. [387](#), [387](#), [387](#), [388](#)
- \tl_gremove_all:cn [396](#)
- \tl_gremove_all:Nn . [97](#), [396](#), [396](#), [396](#)
- \tl_gremove_once:cn [396](#)
- \tl_gremove_once:Nn [96](#), [396](#), [396](#), [396](#)
- \tl_greplace_all:cn [393](#)
- \tl_greplace_all:Nnn
..... [96](#), [393](#), [393](#), [393](#), [396](#)
- \tl_greplace_once:cn [393](#)
- \tl_greplace_once:Nnn
..... [96](#), [393](#), [393](#), [393](#), [396](#)
- \tl_greverse:c [407](#)
- \tl_greverse:N [103](#), [407](#), [407](#), [407](#)
- .tl_gset:c [175](#), [541](#)
- \tl_gset:cf [386](#)
- \tl_gset:cn [386](#)
- \tl_gset:co [386](#)
- \tl_gset:cV [386](#)
- \tl_gset:cv [386](#)
- \tl_gset:cx [386](#)
- .tl_gset:N [175](#), [541](#)
- \tl_gset:Nf [386](#), [431](#)
- \tl_gset:Nn [96](#),
[120](#), [386](#), [386](#), [386](#), [386](#), [388](#), [389](#),
[436](#), [438](#), [464](#), [464](#), [465](#), [555](#), [787](#), [788](#)
- \tl_gset:No [386](#), [386](#), [388](#)
- \tl_gset:NV [386](#)
- \tl_gset:Nv [386](#)
- \tl_gset:Nx [385](#), [386](#), [386](#),
[386](#), [388](#), [389](#), [393](#), [393](#), [393](#), [404](#),
[407](#), [429](#), [429](#), [430](#), [431](#), [433](#), [434](#),
[437](#), [438](#), [446](#), [446](#), [448](#), [449](#), [450](#),
[452](#), [452](#), [466](#), [467](#), [551](#), [757](#), [784](#), [784](#)
- \tl_gset_eq:cc [385](#), [385](#), [429](#), [445](#), [462](#)
- \tl_gset_eq:cN [385](#), [385](#), [429](#), [445](#), [462](#)
- \tl_gset_eq:Nc [385](#), [385](#), [429](#), [445](#), [462](#)
- \tl_gset_eq:NN . [95](#), [385](#), [385](#), [385](#),
[388](#), [414](#), [429](#), [445](#), [462](#), [551](#), [555](#), [757](#)
- \tl_gset_from_file:cn [787](#)
- \tl_gset_from_file:Nnn
..... [225](#), [787](#), [787](#), [787](#)
- \tl_gset_from_file_x:cn [788](#)
- \tl_gset_from_file_x:Nnn
..... [226](#), [788](#), [788](#), [788](#)
- \tl_gset_rescan:cn [389](#)
- \tl_gset_rescan:cno [389](#)
- \tl_gset_rescan:cnx [389](#)
- \tl_gset_rescan:Nnn
..... [97](#), [389](#), [389](#), [390](#), [390](#)
- \tl_gset_rescan:Nno [389](#)
- \tl_gset_rescan:Nnx [389](#)
- .tl_gset_x:c [176](#), [541](#)
- .tl_gset_x:N [176](#), [541](#)

- \tl_gtrim_spaces:c [404](#)
- \tl_gtrim_spaces:N . [104](#), [404](#), [404](#), [404](#)
- \tl_head:f [407](#)
- \tl_head:N [105](#), [407](#), [408](#)
- \tl_head:n [105](#), [105](#),
[105](#), [105](#), [407](#), [407](#), [408](#), [408](#), [408](#), [408](#)
- \tl_head:V [407](#)
- \tl_head:v [407](#)
- \tl_head:w [105](#), [105](#),
[407](#), [408](#), [409](#), [409](#), [409](#), [410](#), [410](#), [410](#)
- __tl_head_auxi:nw . [407](#), [408](#), [408](#), [408](#)
- __tl_head_auxii:n [407](#), [408](#), [408](#)
- \tl_if_blank:n [396](#)
- \tl_if_blank:nF
..... [105](#), [396](#), [396](#), [414](#), [414](#), [457](#)
- \tl_if_blank:nT [396](#), [396](#)
- \tl_if_blank:nTF [99](#), [99](#),
[105](#), [106](#), [396](#), [396](#), [396](#), [408](#), [460](#),
[527](#), [535](#), [544](#), [795](#), [798](#), [799](#), [800](#), [804](#)
- \tl_if_blank:oF [527](#)
- \tl_if_blank:oTF [396](#), [527](#)
- \tl_if_blank:VTF [396](#)
- \tl_if_blank_p:n . [99](#), [99](#), [396](#), [396](#), [396](#)
- __tl_if_blank_p:NNw [396](#)
- \tl_if_blank_p:o [396](#)
- \tl_if_blank_p:V [396](#)
- \tl_if_empty:c [415](#), [453](#)
- \tl_if_empty:cTF [397](#)
- \tl_if_empty:N [397](#), [415](#), [453](#)
- \tl_if_empty:n [397](#)
- \tl_if_empty:n(TF) [397](#), [399](#)
- \tl_if_empty:NF [397](#), [544](#)
- \tl_if_empty:nF [270](#), [272](#),
[343](#), [397](#), [455](#), [517](#), [517](#), [520](#), [523](#), [756](#)
- \tl_if_empty:NT [397](#)
- \tl_if_empty:nT [397](#)
- \tl_if_empty:NTF . [99](#), [99](#), [169](#), [397](#), [397](#)
.. [99](#), [99](#), [390](#), [394](#), [397](#), [397](#), [399](#),
[430](#), [447](#), [506](#), [514](#), [514](#), [520](#), [520](#),
[520](#), [521](#), [521](#), [532](#), [536](#), [597](#), [786](#), [816](#)
- \tl_if_empty:o [398](#)
- \tl_if_empty:oTF [319](#), [319](#), [320](#), [337](#),
[397](#), [399](#), [411](#), [412](#), [446](#), [453](#), [454](#), [454](#)
- \tl_if_empty:VTF [397](#)
- \tl_if_empty_p:c [397](#)
- \tl_if_empty_p:N [99](#), [99](#), [397](#), [397](#)
- \tl_if_empty_p:n [99](#), [99](#), [397](#), [397](#)
- \tl_if_empty_p:o [397](#)
- \tl_if_empty_p:V [397](#)
- __tl_if_empty_return:o . [321](#), [321](#),
[396](#), [396](#), [397](#), [397](#), [398](#), [786](#), [786](#)
- \tl_if_eq:ccTF [398](#)
- \tl_if_eq:cNTF [398](#)
- \tl_if_eq:NcTF [398](#)
- \tl_if_eq:NN [398](#), [416](#)
- \tl_if_eq:nn [398](#)
- \tl_if_eq:nn(TF) ... [123](#), [123](#), [133](#), [133](#)
- \tl_if_eq:NNF [398](#)
- \tl_if_eq:NNT . [398](#), [433](#), [433](#), [498](#), [498](#)
- \tl_if_eq:nnT [433](#)
- \tl_if_eq:NNTF [46](#), [99](#), [99](#),
[100](#), [398](#), [398](#), [400](#), [467](#), [513](#), [515](#), [569](#)
- \tl_if_eq:nnTF [99](#), [99](#), [398](#)
- \tl_if_eq_p:cc [398](#)
- \tl_if_eq_p:cN [398](#)
- \tl_if_eq_p:Nc [398](#)
- \tl_if_eq_p:NN [99](#), [99](#), [398](#), [398](#)
- \tl_if_exist:c [385](#), [415](#)
- \tl_if_exist:cTF [385](#)
- \tl_if_exist:N [385](#), [415](#)
- \tl_if_exist:NT ... [323](#), [324](#), [324](#), [324](#)
- \tl_if_exist:NTF
..... [95](#), [95](#), [385](#), [385](#), [385](#), [403](#), [413](#)
- \tl_if_exist_p:c [385](#)
- \tl_if_exist_p:N [95](#), [95](#), [385](#)
- \tl_if_head_eq_catcode:nN .. [409](#), [409](#)
- \tl_if_head_eq_catcode:nNTF
..... [106](#), [106](#), [409](#), [789](#)
- \tl_if_head_eq_catcode:oNTF [789](#), [794](#)
- \tl_if_head_eq_catcode_p:nN
..... [106](#), [106](#), [409](#)
- \tl_if_head_eq_charcode:fNTF .. [409](#)
- \tl_if_head_eq_charcode:nN . [409](#), [409](#)
- \tl_if_head_eq_charcode:nNF ... [409](#)
- \tl_if_head_eq_charcode:nNT ... [409](#)
- \tl_if_head_eq_charcode:nNTF ...
..... [106](#), [106](#), [409](#), [409](#)
- \tl_if_head_eq_charcode_p:fN .. [409](#)
- \tl_if_head_eq_charcode_p:nN ...
..... [106](#), [106](#), [409](#), [409](#)
- \tl_if_head_eq_meaning:nN .. [410](#), [410](#)
- \tl_if_head_eq_meaning:nNTF
..... [106](#), [106](#), [409](#)
- __tl_if_head_eq_meaning_-
normal:nN [410](#), [410](#)
- \tl_if_head_eq_meaning_p:nN
..... [106](#), [106](#), [409](#)
- __tl_if_head_eq_meaning_-
special:nN [410](#), [410](#)

`\tl_if_head_is_group:n` 411
`\tl_if_head_is_group:nTF` ... 106,
106, 406, 410, 410, 411, 789, 791, 802
`\tl_if_head_is_group_p:n` 106, 106, 411
`\tl_if_head_is_N_type:n` 409, 411
`\tl_if_head_is_N_type:nT` 799, 800, 804
`\tl_if_head_is_N_type:nTF`
..... 107, 107, 406, 409, 409,
410, 411, 786, 789, 791, 796, 797, 801
`__tl_if_head_is_N_type:w`
..... 411, 411, 411, 411
`\tl_if_head_is_N_type_p:n`
..... 107, 107, 411
`\tl_if_head_is_space:n` 412
`\tl_if_head_is_space:nTF`
..... 107, 107, 412, 426
`__tl_if_head_is_space:w` 412, 412, 412
`\tl_if_head_is_space_p:n` 107, 107, 412
`\tl_if_in:cnTF` 398
`\tl_if_in:Nn` 454
`\tl_if_in:nn` 399
`\tl_if_in:nn(TF)` 398, 398
`\tl_if_in:NnF` 399, 399
`\tl_if_in:nnF` 399, 399
`\tl_if_in:NnT` 399, 399, 552
`\tl_if_in:nnT` 399, 399
`\tl_if_in:NnTF`
..... 99, 99, 322, 394, 398, 399, 399
`\tl_if_in:nnTF` 99, 99, 391,
394, 399, 399, 399, 493, 532, 532, 555
`\tl_if_in:noTF` 399, 816
`\tl_if_in:onTF` 394, 399
`\tl_if_in:VnTF` 399
`\tl_if_single:n` 399, 400
`\tl_if_single:Nf` 399
`\tl_if_single:nF` 399
`__tl_if_single:nnw` ... 399, 400, 400
`\tl_if_single:NT` 399
`\tl_if_single:nT` 399
`\tl_if_single:NTF` .. 100, 100, 399, 399
`__tl_if_single:nTF` 399
`\tl_if_single:nTF`
..... 100, 100, 399, 399, 525
`\tl_if_single_p:N` .. 100, 100, 399, 399
`__tl_if_single_p:n` 399
`\tl_if_single_p:n` .. 100, 100, 399, 399
`\tl_if_single_token:n` 786
`\tl_if_single_token:nTF` 222, 222, 786
`\tl_if_single_token_p:n` 222, 222, 786
`\l__tl_internal_a_tl`
..... 389, 390, 390, 392,
398, 398, 398, 398, 788, 788, 788, 788
`\l__tl_internal_b_tl` 398, 398, 398, 398
`\tl_item:cn` 412
`\tl_item:Nn` 107, 412, 413, 413
`__tl_item:nn` 412, 412, 413, 413
`\tl_item:nn` ... 107, 107, 412, 412, 413
`\tl_log:c` 811
`\tl_log:N` 226, 226, 811, 811, 811
`\tl_log:n` 226, 226, 811, 811
`\tl_lower_case:n` 223, 789, 789
`\tl_lower_case:n(n)` 115
`\tl_lower_case:nn` 223, 789, 789
`\tl_map...` ... 101, 101, 102, 102, 387
`\tl_map_break:` ... 101, 101, 401,
401, 402, 402, 402, 402, 402, 402, 402
`\tl_map_break:n` 101, 102, 102, 402, 402
`\tl_map_function:cn` 401
`\tl_map_function:NN` 100,
100, 100, 101, 401, 401, 403, 552
`__tl_map_function:Nn`
..... 401, 401, 401, 401, 401, 401
`\tl_map_function:nN` 100,
100, 100, 101, 401, 401, 403, 430
`\tl_map_inline:cn` 401
`\tl_map_inline:Nn`
..... 101, 101, 101, 401, 402, 402
`\tl_map_inline:nn` ... 49, 101, 101,
101, 333, 401, 401, 402, 566, 629, 631
`\tl_map_variable:cNn` 402
`\tl_map_variable:NNn`
..... 101, 101, 402, 402, 402
`__tl_map_variable:Nnn`
..... 402, 402, 402, 402
`\tl_map_variable:nNn`
..... 101, 101, 402, 402, 402, 402
`\tl_mixed_case:n` 223, 789, 789
`\tl_mixed_case:n(n)` 115, 224
`__tl_mixed_case:nn` 789, 789, 801, 801
`\tl_mixed_case:nn` .. 223, 789, 789, 789
`__tl_mixed_case_aux:nn`
..... 801, 801, 801, 802
`__tl_mixed_case_char:Nn` 801, 803, 803
`__tl_mixed_case_char:nN` 801, 803, 804
`__tl_mixed_case_group:nwn`
..... 801, 802, 802
`\l_tl_mixed_case_ignore_tl`
..... 225, 803, 805, 805, 805

`__tl_mixed_case_loop:wn`
 801, 801, 801, 802, 802, 803, 803, 804
`__tl_mixed_case_N_type:Nnn`
 801, 802, 802
`__tl_mixed_case_N_type:NNNnn` ...
 801, 802, 802, 802
`__tl_mixed_case_N_type:Nwn`
 801, 801, 802
`__tl_mixed_case_skip:N` 801, 803, 803
`__tl_mixed_case_skip:NN`
 801, 803, 803, 804
`__tl_mixed_case_skip_tidy:Nwn` ..
 801, 804, 804
`__tl_mixed_case_space:wn`
 801, 802, 802
`\l_tl_mixed_change_ignore_tl` .. 225
`\tl_new:c` 384, 445
`\tl_new:N`
 56, 95, 95, 95, 321, 339, 384, 384,
 384, 385, 385, 386, 398, 398, 413,
 413, 413, 413, 428, 428, 444, 445,
 461, 478, 479, 479, 497, 503, 511,
 511, 519, 526, 526, 526, 526, 529,
 529, 529, 530, 530, 530, 530, 551,
 551, 551, 557, 561, 565, 565, 565,
 566, 566, 780, 805, 805, 805, 813, 827
`\tl_put_left:cn` 386
`\tl_put_left:co` 386
`\tl_put_left:cV` 386
`\tl_put_left:cx` 386
`\tl_put_left:Nn`
 96, 96, 386, 386, 387, 388
`\tl_put_left:No` ... 386, 386, 387, 388
`\tl_put_left:Nv` ... 386, 386, 387, 388
`\tl_put_left:Nx` ... 386, 386, 387, 388
`\tl_put_right:cn` 387
`\tl_put_right:co` 387
`\tl_put_right:cV` 387
`\tl_put_right:cx` 387
`\tl_put_right:Nn`
 96, 96, 323, 387, 387,
 387, 388, 432, 814, 814, 814, 814,
 814, 814, 814, 814, 814, 814, 814, 814
`\tl_put_right:No` 387, 387, 387, 388, 814
`\tl_put_right:Nv` ... 387, 387, 387, 388
`\tl_put_right:Nx`
 387, 387, 387, 388, 527,
 528, 544, 569, 570, 570, 570, 571, 571
`\tl_remove_all:cn` 396
`\tl_remove_all:Nn` 96,
 97, 97, 97, 323, 324, 396, 396, 396, 552
`\tl_remove_once:cn` 396
`\tl_remove_once:Nn` 96, 96, 396, 396, 396
`__tl_replace:NnNNNnn` 393,
 393, 393, 393, 393, 393, 394, 394, 394
`\tl_replace_all:cn` 393
`\tl_replace_all:Nnn` 96, 96, 393, 393,
 393, 396, 430, 430, 451, 526, 526, 569
`__tl_replace_auxi:NnnNNNnn`
 393, 394, 394, 394, 394, 394
`__tl_replace_auxii:nNNNnn`
 393, 393, 394, 394, 394, 395, 395
`__tl_replace_next:w` 393, 393, 393,
 393, 395, 395, 395, 395, 395, 396
`\tl_replace_once:cn` 393
`\tl_replace_once:Nnn`
 96, 96, 393, 393, 393, 396, 814
`__tl_replace_wrap:w`
 393, 393, 393, 393,
 395, 395, 395, 395, 395, 395, 396
`\tl_rescan:nn` 97, 98, 98, 98, 98, 389, 389
`__tl_rescan:w`
 389, 390, 390, 391, 391, 392, 392, 392
`\c__tl_rescan_marker_tl`
 389, 389, 390, 390, 391, 392, 788, 788
`\tl_reverse:c` 407
`\tl_reverse:N`
 103, 103, 103, 407, 407, 407
`\tl_reverse:n` 103, 103,
 103, 104, 407, 407, 407, 407, 407, 786
`\tl_reverse:o` 407
`\tl_reverse:V` 407
`__tl_reverse_group:nn` . 786, 786, 786
`__tl_reverse_group_preserve:nn` .
 407, 407, 407
`\tl_reverse_items:n`
 103, 103, 104, 104, 403, 403
`__tl_reverse_items:nwNwn`
 403, 403, 403, 403, 403
`__tl_reverse_items:wn`
 403, 403, 404, 404
`__tl_reverse_normal:nN`
 407, 407, 407, 786
`__tl_reverse_space:n`
 407, 407, 407, 786
`\tl_reverse_tokens:n`
 222, 222, 222, 786, 786, 787
`.tl_set:c` 175, 541
`\tl_set:cf` 386

<code>\tl_set:cn</code>	386	<code>__tl_set_rescan:n</code>	
<code>\tl_set:co</code>	386	389 , 389 , 390 , 390 , 391
<code>\tl_set:cV</code>	386	<code>\tl_set_rescan:Nnn</code>	
<code>\tl_set:cv</code>	386	...	97 , 97 , 97 , 98 , 389 , 389 , 390 , 390
<code>\tl_set:cx</code>	386	<code>__tl_set_rescan:NNnn</code>	
<code>.tl_set:N</code>	175 , 541	389 , 389 , 389 , 390 , 390
<code>\tl_set:Nf</code>	386 , 431 , 525	<code>\tl_set_rescan:Nno</code>	389
<code>\tl_set:Nn</code>	96 , 96 , 97 , 98 , 120 , 298 , 340 , 340 , 357 , 386 , 386 , 386 , 386 , 388 , 389 , 398 , 398 , 402 , 430 , 430 , 433 , 433 , 435 , 435 , 435 , 436 , 436 , 436 , 437 , 438 , 441 , 449 , 449 , 449 , 449 , 456 , 463 , 464 , 464 , 464 , 464 , 464 , 464 , 464 , 465 , 465 , 465 , 466 , 467 , 469 , 479 , 479 , 485 , 493 , 493 , 497 , 497 , 512 , 512 , 514 , 519 , 526 , 531 , 532 , 532 , 535 , 542 , 542 , 543 , 544 , 544 , 546 , 553 , 553 , 568 , 569 , 787 , 788 , 805 , 805 , 813 , 827 , 827	<code>__tl_set_rescan:NnTF</code> ..	390 , 391 , 391
<code>\tl_set:No</code>	386 , 386 , 386 , 388 , 788	<code>\tl_set_rescan:Nnx</code>	389
<code>\tl_set:NV</code>	386	<code>__tl_set_rescan_multi:n</code>	
<code>\tl_set:Nv</code>	386	389 , 389 , 390 , 391 , 391
<code>\tl_set:Nx</code>	176 , 385 , 386 , 386 , 386 , 388 , 389 , 390 , 392 , 393 , 393 , 393 , 404 , 407 , 429 , 429 , 430 , 430 , 431 , 433 , 434 , 436 , 437 , 438 , 438 , 445 , 446 , 448 , 449 , 450 , 452 , 452 , 466 , 466 , 528 , 528 , 531 , 532 , 532 , 532 , 542 , 542 , 543 , 544 , 552 , 552 , 552 , 554 , 559 , 562 , 567 , 568 , 568 , 570 , 571 , 757 , 784 , 784 , 788 , 805 , 814	<code>__tl_set_rescan_multiple:n</code> ...	390
<code>\tl_set_eq:cc</code> ..	385 , 385 , 429 , 445 , 462	<code>__tl_set_rescan_single:nn</code>	390 , 390 , 391 , 391 , 392
<code>\tl_set_eq:cN</code> ..	385 , 385 , 429 , 445 , 462	<code>__tl_set_rescan_single_aux:nn</code> ..	
<code>\tl_set_eq:Nc</code> ..	385 , 385 , 429 , 445 , 462	390 , 392 , 392 , 392
<code>\tl_set_eq:NN</code> ..	95 , 95 , 385 , 385 , 385 , 388 , 414 , 429 , 445 , 462 , 513 , 513 , 757	<code>.tl_set_x:c</code>	176 , 541
<code>\tl_set_from_file:cnn</code>	787	<code>.tl_set_x:N</code>	176 , 541
<code>\tl_set_from_file:Nnn</code>		<code>\tl_show:c</code>	413
.....	225 , 225 , 787 , 787 , 787	<code>\tl_show:N</code>	107 , 107 , 226 , 413 , 413 , 413 , 427 , 811 , 811
<code>__tl_set_from_file:NNnn</code>		<code>\tl_show:n</code>	
.....	787 , 787 , 787 , 787	108 , 108 , 226 , 413 , 413 , 427 , 811 , 811
<code>\tl_set_from_file_x:cnn</code>	788	<code>\tl_tail:f</code>	407
<code>\tl_set_from_file_x:Nnn</code>		<code>\tl_tail:N</code>	106 , 407 , 408
.....	226 , 226 , 788 , 788 , 788	<code>\tl_tail:n</code>	
<code>__tl_set_from_file_x:NNnn</code>	106 , 106 , 106 , 407 , 408 , 408 , 408
.....	788 , 788 , 788 , 788	<code>\tl_tail:V</code>	407
<code>\tl_set_rescan:cnn</code>	389	<code>\tl_tail:v</code>	407
<code>\tl_set_rescan:cno</code>	389	<code>__tl_tmp:w</code> ..	399 , 399 , 399 , 404 , 404 , 405
<code>\tl_set_rescan:cnx</code>	389	<code>\tl_to_lowercase:n</code>	
		...	54 , 94 , 98 , 98 , 392 , 392 , 750 , 750
		<code>\tl_to_str:c</code>	402
		<code>\tl_to_str:N</code>	103 , 103 , 109 , 189 , 402 , 402 , 402 , 413 , 416 , 416 , 552 , 567 , 568 , 568 , 568 , 568 , 568
		<code>\tl_to_str:n</code>	94 , 97 , 98 , 102 , 102 , 102 , 103 , 109 , 109 , 115 , 115 , 116 , 116 , 142 , 142 , 170 , 172 , 179 , 179 , 189 , 263 , 263 , 264 , 269 , 269 , 271 , 298 , 299 , 299 , 299 , 299 , 332 , 332 , 337 , 344 , 364 , 365 , 366 , 372 , 376 , 379 , 390 , 394 , 397 , 399 , 399 , 400 , 400 , 402 , 408 , 413 , 413 , 414 , 418 , 418 , 419 , 419 , 421 , 421 , 423 , 423 , 424 , 424 , 425 , 425 , 425 , 425 , 426 , 426 , 463 , 465 , 465 , 466 , 466 , 467 , 468 , 468 , 468 , 497 , 499 , 509 , 509 , 509 , 509 , 516 , 516 , 516 , 516 , 523 , 523 , 523 , 523

- 524, 525, 525, 525, 525, 525, 531,
- 532, 535, 542, 544, 547, 548, 548,
- 548, 548, 548, 548, 555, 556, 556,
- 566, 608, 608, 612, 612, 612, 612,
- 637, 753, 753, 754, 754, 782, 816, 816
- \tl_to_uppercase:n 54, 94, 98, 98, 392, 393
- \tl_trim_spaces:c 404
- \tl_trim_spaces:N 104, 104, 404, 404, 404
- \tl_trim_spaces:n 104,
- 104, 108, 404, 404, 404, 404, 430, 528
- _tl_trim_spaces:nn 108, 108, 404, 404, 447, 460
- _tl_trim_spaces_auxi:w 404, 404, 404, 404, 405, 405
- _tl_trim_spaces_auxii:w 404, 404, 404, 405
- _tl_trim_spaces_auxiii:w 404, 404, 404, 405, 405, 405
- _tl_trim_spaces_auxiv:w 404, 404, 404, 405
- \tl_trim_spacs:n 404
- \tl_upper_case:n ... 223, 223, 789, 789
- \tl_upper_case:n(n) 115
- \tl_upper_case:nn .. 223, 223, 789, 789
- \tl_use:c 403
- \tl_use:N 65,
- 84, 89, 92, 103, 103, 403, 403, 403
- tmpa commands:
- \g_tmpa_bool 41, 308, 308
- \l_tmpa_bool 40, 308, 308
- \g_tmpa_box 150, 474, 474
- \l_tmpa_box 150, 474, 474
- \g_tmpa_clist 140, 461, 461
- \l_tmpa_clist 139, 461, 461
- \l_tmpa_coffin 158, 484, 484
- \g_tmpa_dim 87, 377, 377
- \l_tmpa_dim 87, 377, 377
- \g_tmpa_fp 199, 759, 759
- \l_tmpa_fp 199, 759, 759
- \g_tmpa_int 76, 368, 368
- \l_tmpa_int 2, 76, 368, 368
- \g_tmpa_muskip 93, 383, 383
- \l_tmpa_muskip 93, 383, 383
- \g_tmpa_prop 146, 463, 463
- \l_tmpa_prop 146, 463, 463
- \g_tmpa_seq 129, 444, 444
- \l_tmpa_seq 129, 444, 444
- \g_tmpa_skip 90, 381, 381
- \l_tmpa_skip 90, 381, 381
- \g_tmpb_bool 41, 308, 308
- \l_tmpb_bool 40, 308, 308
- \g_tmpb_box 150, 474, 474
- \l_tmpb_box 150, 474, 474
- \g_tmpb_clist 140, 461, 461
- \l_tmpb_clist 139, 461, 461
- \l_tmpb_coffin 158, 484, 484
- \g_tmpb_dim 87, 377, 377
- \l_tmpb_dim 87, 377, 377
- \g_tmpb_fp 199, 759, 759
- \l_tmpb_fp 199, 759, 759
- \g_tmpb_int 76, 368, 368
- \l_tmpb_int 2, 76, 368, 368
- \g_tmpb_muskip 93, 383, 383
- \l_tmpb_muskip 93, 383, 383
- \g_tmpb_prop 146, 463, 463
- \l_tmpb_prop 146, 463, 463
- \g_tmpb_seq 129, 444, 444
- \l_tmpb_seq 129, 444, 444
- \g_tmpb_skip 90, 381, 381
- \l_tmpb_skip 90, 381, 381
- \g_tmpb_str 117, 427, 427
- \l_tmpb_str 117, 427, 427
- \g_tmpb_tl 108, 413, 413
- \l_tmpb_tl 108, 413, 413
- token commands:
- \c_token_A_int 337, 338
- \token_get_arg_spec:N 63, 63, 344, 344
- \token_get_prefix_spec:N 64, 64, 344, 344
- \token_get_replacement_spec:N ... 63, 63, 344, 345, 548
- \token_if_active:N 331
- \token_if_active:NTF 58, 58, 331
- \token_if_active_p:N 58, 58, 331
- \token_if_alignment:N 329
- \token_if_alignment:NTF 57, 57, 57, 329
- \token_if_alignment_p:N .. 57, 57, 329
- \token_if_chardef:N 333
- \token_if_chardef:NTF 59, 59, 333, 334
- _token_if_chardef:w 333, 333, 333, 334
- \token_if_chardef_p:N 59, 59, 333
- \token_if_cs:N 332

`\token_if_cs:N` NTF . 58, 58, 332, 792, 803
`\token_if_cs_p:N` 58, 58, 332, 798, 799, 800, 805
`\token_if_dim_register:N` 334
`\token_if_dim_register:N` NTF 59, 59, 333
`_token_if_dim_register:w` 333, 334, 334
`\token_if_dim_register_p:N` 59, 59, 333
`\token_if_eq_catcode:NN` 331
`\token_if_eq_catcode:NNTF` 58, 58, 60, 61, 61, 331
`\token_if_eq_catcode_p:NN` 58, 58, 331
`\token_if_eq_charcode:NN` 331
`\token_if_eq_charcode:NNT` 553
`\token_if_eq_charcode:NNTF` 58, 58, 61, 61, 62, 62, 331
`\token_if_eq_charcode_p:NN` 58, 58, 331
`\token_if_eq_meaning:NN` 331
`\token_if_eq_meaning:NNTF` 590
`\token_if_eq_meaning:NNTF` 58, 58, 62, 62, 62, 63, 331, 342, 628, 727, 791, 792, 792, 802
`\token_if_eq_meaning_p:NN` 58, 58, 331, 796
`\token_if_expandable:N` 332
`\token_if_expandable:N` NTF 58, 58, 332, 796
`\token_if_expandable_p:N` . 58, 58, 332
`\token_if_group_begin:N` 329
`\token_if_group_begin:N` NTF 56, 56, 329
`\token_if_group_begin_p:N` 56, 56, 329
`\token_if_group_end:N` 329
`\token_if_group_end:N` NTF . . 57, 57, 329
`\token_if_group_end_p:N` . . 57, 57, 329
`\token_if_int_register:N` 334
`\token_if_int_register:N` NTF 59, 59, 333
`_token_if_int_register:w` 333, 334, 334
`\token_if_int_register_p:N` 59, 59, 333
`\token_if_letter:N` 330, 332
`\token_if_letter:N` NTF 57, 57, 330, 797
`\token_if_letter_p:N` 57, 57, 330
`\token_if_long_macro:N` 336
`\token_if_long_macro:N` NTF . 58, 58, 333
`_token_if_long_macro:w` 333, 336, 336, 336
`\token_if_long_macro_p:N` . 58, 58, 333
`\token_if_macro:N` 332
`\token_if_macro:N` NTF 58, 58, 331, 337, 344, 344, 345
`\token_if_macro_p:N` 58, 58, 331
`_token_if_macro_p:w` . . 331, 332, 332
`\token_if_math_subscript:N` 330
`\token_if_math_subscript:N` NTF 57, 57, 330
`\token_if_math_subscript_p:N` 57, 57, 330
`\token_if_math_superscript:N` . . 330
`\token_if_math_superscript:N` NTF 57, 57, 330
`\token_if_math_superscript_p:N` 57, 57, 330
`\token_if_math_toggle:N` 329
`\token_if_math_toggle:N` NTF 57, 57, 329
`\token_if_math_toggle_p:N` 57, 57, 329
`\token_if_mathchardef:N` 333
`\token_if_mathchardef:N` NTF 59, 59, 333, 334
`\token_if_mathchardef_p:N` 59, 59, 333
`\token_if_muskip_register:N` 335
`\token_if_muskip_register:N` NTF 59, 59, 333
`_token_if_muskip_register:w` 333, 335, 335
`\token_if_muskip_register_p:N` 59, 59, 333
`\token_if_other:N` 330
`\token_if_other:N` NTF 57, 57, 330
`\token_if_other_p:N` 57, 57, 330
`\token_if_parameter:N` 329
`\token_if_parameter:N` NTF 57, 329
`\token_if_parameter_p:N` . . 57, 57, 329
`\token_if_primitive:N` 337
`_token_if_primitive:NNw` 337, 337, 337
`\token_if_primitive:N` NTF . . 60, 60, 337
`_token_if_primitive:Nw` 337, 338, 338
`_token_if_primitive_loop:N` 337, 337, 338, 338
`_token_if_primitive_nullfont:N` 337, 338, 338
`\token_if_primitive_p:N` . . 60, 60, 337
`_token_if_primitive_space:w` 337, 337, 338
`_token_if_primitive_undefined:N` 337, 338, 338
`\token_if_protected_long_macro:N` 336
`\token_if_protected_long_macro:N` NTF 59, 59, 333

- \token_if_protected_long_macro_-
 p:N 59, 59, 333, 796
 - \token_if_protected_macro:N ... 336
 - \token_if_protected_macro:NTF ...
 58, 58, 333
 - _token_if_protected_macro:w ...
 333, 336, 336
 - \token_if_protected_macro_p:N ...
 58, 58, 333, 796
 - \token_if_skip_register:N 335
 - \token_if_skip_register:NTF
 59, 59, 333
 - _token_if_skip_register:w
 333, 335, 335
 - \token_if_skip_register_p:N
 59, 59, 333
 - \token_if_space:N 330
 - \token_if_space:NTF 57, 57, 330
 - \token_if_space_p:N 57, 57, 330
 - \token_if_toks_register:N 335
 - \token_if_toks_register:NTF
 60, 60, 333
 - _token_if_toks_register:w
 333, 336, 336
 - \token_if_toks_register_p:N
 60, 60, 333
 - \token_new:Nn 55, 55, 328, 328
 - \token_to_meaning:c ... 263, 264, 328
 - \token_to_meaning:N
 56, 56, 262, 262, 279,
 279, 299, 328, 331, 332, 333, 333,
 333, 334, 334, 335, 335, 336, 336,
 336, 336, 337, 337, 344, 345, 345, 816
 - \token_to_str:c ... 263, 263, 269,
 269, 270, 270, 271, 272, 272, 300, 328
 - \token_to_str:N 5, 19, 56, 56, 56, 94,
 109, 169, 189, 263, 263, 263, 273,
 274, 274, 274, 274, 279, 279, 279,
 279, 280, 284, 285, 287, 287, 303,
 303, 304, 304, 304, 308, 322, 328,
 333, 333, 334, 334, 335, 335, 336,
 336, 336, 336, 336, 352, 352, 367,
 389, 411, 411, 411, 413, 475, 480,
 485, 501, 524, 524, 526, 526, 526,
 526, 552, 567, 567, 567, 567, 567,
 586, 586, 607, 607, 608, 609, 609,
 609, 610, 613, 614, 615, 616, 617,
 617, 617, 617, 617, 618, 619, 619,
 620, 620, 620, 620, 621, 621, 622,
 624, 625, 625, 625, 625, 633, 730, 759
 - \toks 247
 - \toksdef 247
 - \tolerance 247
 - \topmark 247
 - \topmarks 249
 - \topskip 247
 - \tracingassigns 249
 - \tracingcommands 247
 - \tracinggroups 249
 - \tracingifs 249
 - \tracinglostchars 247
 - \tracingmacros 247
 - \tracingnesting 250
 - \tracingonline 247
 - \tracingoutput 247
 - \tracingpages 247
 - \tracingparagraphs 247
 - \tracingrestores 248
 - \tracingscantokens 250
 - \tracingstats 248
 - true 208
 - true commands:
 \c_true_bool 22,
 39, 270, 273, 273, 274, 274, 274,
 283, 306, 306, 306, 306, 307, 307,
 310, 310, 312, 312, 312, 314, 398,
 817, 817, 818, 818, 818, 819, 819, 819
 - trunc 205
 - twelve commands:
 \c_twelve 76, 323,
 325, 326, 333, 367, 368, 627, 627, 627
 - two commands:
 \c_two . 76, 323, 324, 324, 325, 325,
 347, 365, 367, 367, 424, 515, 552,
 576, 577, 639, 642, 646, 650, 658,
 663, 663, 663, 663, 663, 673, 676,
 677, 677, 678, 687, 694, 699, 699,
 699, 702, 710, 713, 721, 722, 722,
 725, 726, 729, 737, 737, 738, 739,
 741, 741, 744, 745, 753, 816, 816, 816
 - \c_two_hundred_fifty_five 76, 368, 368
 - \c_two_hundred_fifty_six 76, 368, 368
- U
- \uccode 248
 - \Uchar 255
 - \Ucharcat 255
 - \uchyph 248
 - \ucs 258
 - \Udelcode 255

<code>\Udelcodenum</code>	255	<code>\Umathlimitbelowkern</code>	256
<code>\Udelimiter</code>	255	<code>\Umathlimitbelowvgap</code>	256
<code>\Udelimiterover</code>	255	<code>\Umathopbinspacing</code>	256
<code>\Udelimiterunder</code>	255	<code>\Umathopclosespacing</code>	256
<code>\UHORN</code>	811	<code>\Umathopenbinspacing</code>	256
<code>\Uhorn</code>	810, 811	<code>\Umathopenclosespacing</code>	256
<code>\uhorn</code>	810, 811, 811	<code>\Umathopeninnerspacing</code>	256
<code>\Umathaccent</code>	255	<code>\Umathopenopenspacing</code>	256
<code>\Umathaxis</code>	255	<code>\Umathopenopspacing</code>	256
<code>\Umathbinbinspacing</code>	255	<code>\Umathopenordspacing</code>	256
<code>\Umathbinclosespacing</code>	255	<code>\Umathopenpunctspacing</code>	256
<code>\Umathbininnerspacing</code>	255	<code>\Umathopenrelspacing</code>	256
<code>\Umathbinopenspacing</code>	255	<code>\Umathoperatorsize</code>	256
<code>\Umathbinopspacing</code>	255	<code>\Umathopinnerspacing</code>	256
<code>\Umathbinordspacing</code>	255	<code>\Umathopopenspacing</code>	256
<code>\Umathbinpunctspacing</code>	255	<code>\Umathopopspacing</code>	256
<code>\Umathbinrelspacing</code>	255	<code>\Umathopordspacing</code>	256
<code>\Umathchar</code>	255	<code>\Umathoppunctspacing</code>	256
<code>\Umathchardef</code>	255	<code>\Umathoprelspacing</code>	256
<code>\Umathcharnum</code>	255	<code>\Umathordbinspacing</code>	256
<code>\Umathcharnumdef</code>	255	<code>\Umathordclosespacing</code>	256
<code>\Umathclosebinspacing</code>	255	<code>\Umathordinnerspacing</code>	256
<code>\Umathcloseclosespacing</code>	255	<code>\Umathordopenspacing</code>	256
<code>\Umathcloseinnerspacing</code>	255	<code>\Umathordopspacing</code>	256
<code>\Umathcloseopenspacing</code>	255	<code>\Umathordordspacing</code>	256
<code>\Umathcloseopspacing</code>	255	<code>\Umathordpunctspacing</code>	256
<code>\Umathcloseordspacing</code>	255	<code>\Umathordrelspacing</code>	256
<code>\Umathclosepunctspacing</code>	255	<code>\Umathoverbarkern</code>	256
<code>\Umathcloserelspacing</code>	255	<code>\Umathoverbarrule</code>	256
<code>\Umathcode</code>	255	<code>\Umathoverbarvgap</code>	256
<code>\Umathcodenum</code>	255	<code>\Umathoverdelimiterbgap</code>	256
<code>\Umathconnectoroverlapmin</code>	255	<code>\Umathoverdelimitervgap</code>	256
<code>\Umathfractiondelsize</code>	255	<code>\Umathpunctbinspacing</code>	256
<code>\Umathfractiondenomdown</code>	255	<code>\Umathpunctclosespacing</code>	256
<code>\Umathfractiondenomvgap</code>	255	<code>\Umathpunctinnerspacing</code>	256
<code>\Umathfractionnumup</code>	255	<code>\Umathpunctopenspacing</code>	256
<code>\Umathfractionnumvgap</code>	255	<code>\Umathpunctopspacing</code>	256
<code>\Umathfractionrule</code>	255	<code>\Umathpunctordspacing</code>	256
<code>\Umathinnerbinspacing</code>	255	<code>\Umathpunctpunctspacing</code>	256
<code>\Umathinnerclosespacing</code>	255	<code>\Umathpunctrelspacing</code>	256
<code>\Umathinnerinnerspacing</code>	255	<code>\Umathquad</code>	256
<code>\Umathinneropenspacing</code>	255	<code>\Umathradicaldegreeafter</code>	256
<code>\Umathinneropspacing</code>	255	<code>\Umathradicaldegreebefore</code>	256
<code>\Umathinnerordspacing</code>	256	<code>\Umathradicaldegreeraise</code>	257
<code>\Umathinnerpunctspacing</code>	256	<code>\Umathradicalkern</code>	257
<code>\Umathinnerrelspacing</code>	256	<code>\Umathradicalrule</code>	257
<code>\Umathlimitabovebgap</code>	256	<code>\Umathradicalvgap</code>	257
<code>\Umathlimitabovekern</code>	256	<code>\Umathrelbinspacing</code>	257
<code>\Umathlimitabovevgap</code>	256	<code>\Umathrelclosespacing</code>	257
<code>\Umathlimitbelowbgap</code>	256	<code>\Umathrelinnerspacing</code>	257

<code>\Umathrelopenspacing</code>	257	uptex commands:	
<code>\Umathrellopspacing</code>	257	<code>\uptex_disablecjktoken:D</code>	
<code>\Umathrelordspacing</code>	257	258, 348, 348, 818
<code>\Umathrelpunctspacing</code>	257	<code>\uptex_enablecjktoken:D</code>	258
<code>\Umathrelrelspacing</code>	257	<code>\uptex_forcecjktoken:D</code>	258
<code>\Umathspaceafterscript</code>	257	<code>\uptex_kchar:D</code>	258
<code>\Umathstackdenomdown</code>	257	<code>\uptex_kchardef:D</code>	258, 348
<code>\Umathstacknumup</code>	257	<code>\uptex_kuten:D</code>	258
<code>\Umathstackvgap</code>	257	<code>\uptex_ucs:D</code>	258
<code>\Umathsubshiftdown</code>	257	<code>\Uradical</code>	257
<code>\Umathsubshiftdrop</code>	257	<code>\Uroot</code>	257
<code>\Umathsubsupshiftdown</code>	257	use commands:	
<code>\Umathsubsupvgap</code>	257	<code>\use:c</code>	18, 18, 18, 18,
<code>\Umathsubtopmax</code>	257	<u>265</u> , 265, 269, 270, 272, 277, 277,	
<code>\Umathsupbottommin</code>	257	277, 277, 310, 311, 352, 363, 363,	
<code>\Umathsupshiftdrop</code>	257	366, 366, 366, 366, 366, 366, 372,	
<code>\Umathsupshiftdown</code>	257	509, 510, 510, 510, 510, 511, 511,	
<code>\Umathsupsubbottommax</code>	257	512, 532, 532, 537, 537, 570, 782, 793	
<code>\Umathunderbarkern</code>	257	<code>\use:f</code>	607
<code>\Umathunderbarrule</code>	257	<code>\use:n</code>	19, 19, 94, 224,
<code>\Umathunderbarvgap</code>	257	<u>266</u> , 266, 270, 282, 324, 324, 327,	
<code>\Umathunderdelimiterbgap</code>	257	327, 328, 390, 391, 392, 402, 410,	
<code>\Umathunderdelimitervgap</code>	257	457, 475, 475, 521, 521, 521, 564,	
undefine commands:		588, 589, 589, 589, 589, 590, 611,	
<code>\.undefine:</code>	176, <u>542</u>	782, 783, 814, 817, 817, 817, 818,	
<code>\underline</code>	248	818, 818, 818, 818, 819, 819, 819, 819	
underscore commands:		<code>\use:nn</code> .	19, 19, <u>266</u> , 266, 290, 327,
<code>\c_underscore_str</code>	117, <u>426</u> , 427	327, 344, 372, 390, 456, 612, 717, 788	
<code>\unexpanded</code>	250	<code>\use:nnn</code>	19, 19, <u>266</u> , 266, 284
<code>\unhbox</code>	248	<code>\use:nnnn</code>	19, 19, <u>266</u> , 266
<code>\unhcopy</code>	248	<code>\use:x</code>	21, 21, 265,
unicode commands:		<u>265</u> , 265, 269, 271, 274, 299, 299,	
<code>\c__unicode_accents_lt_tl</code>	798	301, 304, 332, 337, 367, 375, 388,	
<code>\c__unicode_dot_above_tl</code>	800	392, 426, 474, 509, 509, 516, 516,	
<code>\c__unicode_dotless_i_tl</code> ...	797, 798	523, 548, 553, 560, 568, 585, 608, 612	
<code>\c__unicode_dotted_I_tl</code>	798	<code>\use_i:nn</code>	20, 20, 20, 263, 263,
<code>\c__unicode_final_sigma_tl</code> .	796, 797	<u>266</u> , 266, 267, 268, 270, 275, 276,	
<code>\c__unicode_I_ogonek_tl</code>	800	283, 304, 310, 310, 311, 311, 311,	
<code>\c__unicode_i_ogonek_tl</code>	799	312, 312, 408, 431, 431, 463, 463,	
<code>\c__unicode_mixed_exceptions_tl</code>	804	584, 585, 611, 630, 630, 631, 661,	
<code>\c__unicode_std_sigma_tl</code>	797	683, 693, 712, 717, 724, 727, 737,	
<code>\c__unicode_upper_Eszett_tl</code> ...	801	740, 744, 745, 745, 796, 797, 798,	
<code>\unkern</code>	248	817, 817, 818, 818, 818, 819, 819, 819	
<code>\unless</code>	250	<code>\use_i:nnn</code>	20,
<code>\unpenalty</code>	248	20, 20, <u>266</u> , 266, 275, 344, 437, 660	
<code>\unskip</code>	248	<code>\use_i:nnnn</code>	
<code>\unvbox</code>	248	... 20, 20, 20, <u>266</u> , 266, 661, 661, 667	
<code>\unvcopy</code>	248	<code>\use_i_delimit_by_q_nil:nw</code>	
<code>\Uoverdelimenter</code>	257	21, 21, <u>266</u> , 266
<code>\uppercase</code>	248		

- `\use_i_delimit_by_q_recursion-`
`stop:nw` 21, 21,
266, 266, 319, 319, 791, 795, 802, 804
- `\use_i_delimit_by_q_stop:nw`
..... 21, 21, 266, 266,
419, 420, 423, 424, 424, 425, 425, 459
- `\use_i_ii:nnn`
..... 20, 20, 266, 266, 291, 437, 440
- `\use_ii:nn` 20, 20, 45,
263, 263, 266, 266, 267, 268, 270,
275, 276, 283, 287, 310, 311, 311,
311, 391, 408, 463, 463, 584, 630,
630, 631, 661, 713, 724, 727, 737,
740, 744, 745, 745, 756, 756, 793,
796, 796, 797, 803, 817, 818, 819, 819
- `\use_ii:nnn` 20,
20, 266, 266, 275, 345, 523, 528, 528
- `\use_ii:nnnn` 20, 20, 266, 266
- `\use_iii:nnn` 20,
20, 266, 266, 288, 345, 584, 584, 585
- `\use_iii:nnnn` 20, 20, 266, 266
- `\use_iv:nnnn` 20, 20, 266, 266
- `\use_none:n` 21,
21, 24, 108, 267, 267, 270, 277, 278,
278, 282, 319, 319, 338, 360, 360,
395, 396, 396, 404, 404, 405, 408,
408, 410, 411, 411, 411, 411, 412,
412, 428, 439, 440, 440, 446, 449,
449, 452, 453, 453, 506, 507, 517,
517, 527, 527, 527, 535, 568, 568,
568, 582, 582, 582, 582, 585, 587,
603, 604, 604, 604, 628, 634, 634,
635, 635, 635, 636, 637, 638, 639,
640, 661, 661, 701, 730, 757, 784,
784, 786, 799, 817, 817, 817, 818,
818, 818, 818, 819, 819, 819, 819, 819
- `\use_none:nn` 21, 267, 267,
269, 269, 399, 400, 404, 409, 409,
409, 410, 433, 437, 437, 437, 437,
437, 438, 454, 578, 582, 582, 582, 582
- `\use_none:nnn` 21, 267, 267,
410, 410, 528, 582, 582, 582, 582, 794
- `\use_none:nnnn`
..... 21, 267, 267, 300, 303, 303, 357
- `\use_none:nnnnn` 21,
267, 267, 267, 588, 589, 590, 590, 590
- `\use_none:nnnnnn` ... 21, 267, 267, 272
- `\use_none:nnnnnnn` 21, 267, 267, 269,
270, 588, 589, 590, 590, 590, 598, 662
- `\use_none:nnnnnnnn` 21, 267, 267
- `\use_none:nnnnnnnnn` 21, 267, 267
- `\use_none_delimit_by_q_nil:w` ...
..... 21, 21, 266, 266
- `\use_none_delimit_by_q_recursion-`
`stop:w` 21, 21,
48, 48, 48, 48, 266, 266, 270, 272,
272, 272, 300, 301, 319, 319, 388, 806
- `\use_none_delimit_by_q_stop:w` ...
..... 21, 21, 266, 266, 322,
353, 373, 419, 420, 424, 451, 451,
452, 459, 459, 513, 571, 782, 815, 816
- `__use_none_delimit_by_s_stop:w`
..... 50, 50, 50, 322, 322
- `\Ustack` 257
- `\Ustartdisplaymath` 257
- `\Ustartmath` 257
- `\Ustopdisplaymath` 257
- `\Ustopmath` 257
- `\Usubscript` 257
- `\USuperscript` 257
- utex commands:
- `\utex_binbinspacing:D` 255
- `\utex_binclosespacing:D` 255
- `\utex_bininnerspacing:D` 255
- `\utex_binopenspacing:D` 255
- `\utex_binopspacing:D` 255
- `\utex_binordspacing:D` 255
- `\utex_binpunctspacing:D` 255
- `\utex_binrelspacing:D` 255
- `\utex_char:D` 255, 259
- `\utex_charcat:D` 255, 813, 813
- `\utex_closebinspacing:D` 255
- `\utex_closeclosespacing:D` 255
- `\utex_closeinnerspacing:D` 255
- `\utex_closeopenspacing:D` 255
- `\utex_closeopspacing:D` 255
- `\utex_closeordspacing:D` 255
- `\utex_closepunctspacing:D` 255
- `\utex_closerelspacing:D` 255
- `\utex_connectoroverlapmin:D` ... 255
- `\utex_delcode:D` 255, 260
- `\utex_delcodenum:D` 255, 260
- `\utex_delimiter:D` 255, 260
- `\utex_delimiterover:D` 255
- `\utex_delimiterunder:D` 255
- `\utex_fractiondelsize:D` 255
- `\utex_fractiondenomdown:D` 255
- `\utex_fractiondenomvgap:D` 255
- `\utex_fractionnumup:D` 255
- `\utex_fractionnumvgap:D` 255

<code>\utex_fractionrule:D</code>	255	<code>\utex_overbarvgap:D</code>	256
<code>\utex_innerbinspacing:D</code>	255	<code>\utex_overdelim�ter:D</code>	257
<code>\utex_innerclosespacing:D</code>	255	<code>\utex_overdelim�terbgap:D</code>	256
<code>\utex_innerinnerspacing:D</code>	255	<code>\utex_overdelim�tervgap:D</code>	256
<code>\utex_inneropenspacing:D</code>	255	<code>\utex_punctbinspacing:D</code>	256
<code>\utex_inneropspacing:D</code>	255	<code>\utex_punctclosespacing:D</code>	256
<code>\utex_innerordspacing:D</code>	256	<code>\utex_punctinnerspacing:D</code>	256
<code>\utex_innerpunctspacing:D</code>	256	<code>\utex_punctopenspacing:D</code>	256
<code>\utex_innerrelspacing:D</code>	256	<code>\utex_punctopspacing:D</code>	256
<code>\utex_limitabovebgap:D</code>	256	<code>\utex_punctordspacing:D</code>	256
<code>\utex_limitabovekern:D</code>	256	<code>\utex_punctpunctspacing:D</code>	256
<code>\utex_limitabovevgap:D</code>	256	<code>\utex_punctrelspacing:D</code>	256
<code>\utex_limitbelowbgap:D</code>	256	<code>\utex_quad:D</code>	256
<code>\utex_limitbelowkern:D</code>	256	<code>\utex_radical:D</code>	257
<code>\utex_limitbelowvgap:D</code>	256	<code>\utex_radicaldegreeafter:D</code>	256
<code>\utex_mathaccent:D</code>	255, 260	<code>\utex_radicaldegreebefore:D</code> ...	256
<code>\utex_mathaxis:D</code>	255	<code>\utex_radicaldegreeraise:D</code>	257
<code>\utex_mathchar:D</code>	255, 260	<code>\utex_radicalkern:D</code>	257
<code>\utex_mathchardef:D</code>	255, 260	<code>\utex_radicalrule:D</code>	257
<code>\utex_mathcharnum:D</code>	255, 261	<code>\utex_radicalvgap:D</code>	257
<code>\utex_mathcharnumdef:D</code>	255, 261	<code>\utex_relbinspacing:D</code>	257
<code>\utex_mathcode:D</code>	255, 261	<code>\utex_relclosespacing:D</code>	257
<code>\utex_mathcodenum:D</code>	255, 261	<code>\utex_relinnerspacing:D</code>	257
<code>\utex_opbinspacing:D</code>	256	<code>\utex_reloopenspacing:D</code>	257
<code>\utex_opclosespacing:D</code>	256	<code>\utex_reloppspacing:D</code>	257
<code>\utex_openbinspacing:D</code>	256	<code>\utex_relordspacing:D</code>	257
<code>\utex_openclosespacing:D</code>	256	<code>\utex_relpunctspacing:D</code>	257
<code>\utex_openinnerspacing:D</code>	256	<code>\utex_relrelspacing:D</code>	257
<code>\utex_openopenspacing:D</code>	256	<code>\utex_root:D</code>	257
<code>\utex_openopspacing:D</code>	256	<code>\utex_spaceafterscript:D</code>	257
<code>\utex_openordspacing:D</code>	256	<code>\utex_stack:D</code>	257
<code>\utex_openpunctspacing:D</code>	256	<code>\utex_stackdenomdown:D</code>	257
<code>\utex_openrelspacing:D</code>	256	<code>\utex_stacknumup:D</code>	257
<code>\utex_operatorsize:D</code>	256	<code>\utex_stackvgap:D</code>	257
<code>\utex_opinnerspacing:D</code>	256	<code>\utex_startdisplaymath:D</code>	257
<code>\utex_opopenspacing:D</code>	256	<code>\utex_startmath:D</code>	257
<code>\utex_opopspacing:D</code>	256	<code>\utex_stopdisplaymath:D</code>	257
<code>\utex_opordspacing:D</code>	256	<code>\utex_stopmath:D</code>	257
<code>\utex_oppunctspacing:D</code>	256	<code>\utex_subscript:D</code>	257
<code>\utex_oprelspacing:D</code>	256	<code>\utex_subshiftdown:D</code>	257
<code>\utex_ordbinspacing:D</code>	256	<code>\utex_subshiftdrop:D</code>	257
<code>\utex_ordclosespacing:D</code>	256	<code>\utex_subsupshiftdown:D</code>	257
<code>\utex_ordinnerspacing:D</code>	256	<code>\utex_subsupvgap:D</code>	257
<code>\utex_ordopenspacing:D</code>	256	<code>\utex_subtopmax:D</code>	257
<code>\utex_ordopspacing:D</code>	256	<code>\utex_supbottommin:D</code>	257
<code>\utex_ordordspacing:D</code>	256	<code>\utex_superscript:D</code>	257
<code>\utex_ordpunctspacing:D</code>	256	<code>\utex_supshiftdrop:D</code>	257
<code>\utex_ordrelspacing:D</code>	256	<code>\utex_supshiftup:D</code>	257
<code>\utex_overbarkern:D</code>	256	<code>\utex_supsubbottommax:D</code>	257
<code>\utex_overbarrule:D</code>	256	<code>\utex_underbarkern:D</code>	257

<code>\utex_underbarrule:D</code>	257	<code>\vbox_unpack_clear:c</code>	478
<code>\utex_underbarvgap:D</code>	257	<code>\vbox_unpack_clear:N</code> 153, 478, 478, 478	478
<code>\utex_underdelim实施er:D</code>	257	<code>\vcenter</code>	248
<code>\utex_underdelim实施erbgap:D</code>	257	vcoffin commands:	
<code>\utex_underdelim实施ervgap:D</code>	257	<code>\vcoffin_set:cnn</code>	481
<code>\Underdelim实施er</code>	257	<code>\vcoffin_set:cnw</code>	483
V		<code>\vcoffin_set:Nnn</code> 156, 156, 481, 482, 482	482
<code>\vadjust</code>	248	<code>\vcoffin_set:Nnw</code> 156, 156, 483, 483, 483	483
<code>\valign</code>	248	<code>\vcoffin_set_end:</code>	156, 156, 483, 483, 483
value commands:		<code>\vfil</code>	248
<code>.value_forbidden:</code>	550	<code>\vfill</code>	248
<code>.value_forbidden:n</code>	176, 542	<code>\vfilleg</code>	248
<code>.value_required:</code>	550	<code>\vfuzz</code>	248
<code>.value_required:n</code>	176, 542	<code>\voffset</code>	248
<code>\vbadness</code>	248	void commands:	
<code>\vbox</code>	248	<code>\c_void_box</code>	147
vbox commands:		<code>\vrule</code>	248
<code>\vbox:n</code>	152, 152, 476, 476	<code>\vsize</code>	248
<code>\vbox_gset:cn</code>	477	<code>\vskip</code>	248
<code>\vbox_gset:cw</code>	477	<code>\vsplit</code>	248
<code>\vbox_gset:Nn</code>	153, 477, 477, 477	<code>\vss</code>	248
<code>\vbox_gset:Nw</code>	153, 477, 477, 477	<code>\vtop</code>	248
<code>\vbox_gset_end:</code>	153, 477, 478	W	
<code>\vbox_gset_to_ht:cnn</code>	477	<code>\wd</code>	248
<code>\vbox_gset_to_ht:Nnn</code> 153, 477, 477, 477	477	<code>\widowpenalties</code>	250
<code>\vbox_gset_top:cn</code>	477	<code>\widowpenalty</code>	248
<code>\vbox_gset_top:Nn</code> .. 153, 477, 477, 477	477	<code>\write</code>	248
<code>\vbox_set:cn</code>	477	X	
<code>\vbox_set:cw</code>	477	<code>\xdef</code>	248
<code>\vbox_set:Nn</code>	153, 153, 153, 477, 477, 477, 477, 482	xetex commands:	
<code>\vbox_set:Nw</code>	153, 153, 477, 477, 477, 477, 483	<code>\xetex_...</code>	9
<code>\vbox_set_end:</code>	153, 153, 477, 478, 478, 483	<code>\xetex_charclass:D</code>	252
<code>\vbox_set_split_to_ht:NNn</code>	153, 153, 478, 478, 478	<code>\xetex_charglyph:D</code>	252
<code>\vbox_set_to_ht:cnn</code>	477	<code>\xetex_countfeatures:D</code>	252
<code>\vbox_set_to_ht:Nnn</code>	153, 153, 477, 477, 477, 477	<code>\xetex_countglyphs:D</code>	252
<code>\vbox_set_top:cn</code>	477	<code>\xetex_countselectors:D</code>	252
<code>\vbox_set_top:Nn</code>	153, 153, 477, 477, 477, 477, 482, 483	<code>\xetex_countvariations:D</code>	252
<code>\vbox_to_ht:nn</code> 152, 152, 477, 477, 477	477	<code>\xetex_dashbreakstate:D</code>	252
<code>\vbox_to_zero:n</code> 152, 152, 477, 477, 477	477	<code>\xetex_defaultencoding:D</code>	252
<code>\vbox_top:n</code>	152, 152, 476, 476	<code>\xetex_featurecode:D</code>	252
<code>\vbox_unpack:c</code>	478	<code>\xetex_featurename:D</code>	252
<code>\vbox_unpack:N</code>	153, 153, 153, 478, 478, 478, 482, 483	<code>\xetex_findfeaturebyname:D</code> ...	252
		<code>\xetex_findselectorbyname:D</code> ..	252
		<code>\xetex_findvariationbyname:D</code> ..	252
		<code>\xetex_firstfontchar:D</code>	253
		<code>\xetex_fonttype:D</code>	253
		<code>\xetex_glyph:D</code>	253

<code>\xetex_glyphbounds:D</code>	253	<code>\XeTeXfindfeaturebyname</code>	252
<code>\xetex_glyphindex:D</code>	253	<code>\XeTeXfindselectorbyname</code>	252
<code>\xetex_glyphname:D</code>	253	<code>\XeTeXfindvariationbyname</code>	252
<code>\xetex_if_engine:</code>	819	<code>\XeTeXfirstfontchar</code>	253
<code>\xetex_if_engine:TF</code>	5	<code>\XeTeXfonttype</code>	253
<code>\xetex_inputencoding:D</code>	253	<code>\XeTeXglyph</code>	253
<code>\xetex_inputnormalization:D</code> ...	253	<code>\XeTeXglyphbounds</code>	253
<code>\xetex_interchartokenstate:D</code> ..	253	<code>\XeTeXglyphindex</code>	253
<code>\xetex_interchartoks:D</code>	253	<code>\XeTeXglyphname</code>	253
<code>\xetex_isdefaultselector:D</code>	253	<code>\XeTeXinputencoding</code>	253
<code>\xetex_isexclusivefeature:D</code> ...	253	<code>\XeTeXinputnormalization</code>	253
<code>\xetex_lastfontchar:D</code>	253	<code>\XeTeXinterchartokenstate</code>	253
<code>\xetex_linebreaklocale:D</code>	253	<code>\XeTeXinterchartoks</code>	253
<code>\xetex_linebreakpenalty:D</code>	253	<code>\XeTeXisdefaultselector</code>	253
<code>\xetex_linebreakskip:D</code>	253	<code>\XeTeXisexclusivefeature</code>	253
<code>\xetex_OTcountfeatures:D</code>	253	<code>\XeTeXlastfontchar</code>	253
<code>\xetex_OTcountlanguages:D</code>	253	<code>\XeTeXlinebreaklocale</code>	253
<code>\xetex_OTcountscripts:D</code>	253	<code>\XeTeXlinebreakpenalty</code>	253
<code>\xetex_OTfeaturetag:D</code>	253	<code>\XeTeXlinebreakskip</code>	253
<code>\xetex_OTlanguagetag:D</code>	253	<code>\XeTeXmathaccent</code>	260
<code>\xetex_OTscripttag:D</code>	253	<code>\XeTeXmathchar</code>	260
<code>\xetex_pdffile:D</code>	253	<code>\XeTeXmathchardef</code>	260
<code>\xetex_pdfpagecount:D</code>	253	<code>\XeTeXmathcharnum</code>	261
<code>\xetex_picfile:D</code>	253	<code>\XeTeXmathcharnumdef</code>	261
<code>\xetex_selectorname:D</code>	253	<code>\XeTeXmathcode</code>	261
<code>\xetex_suppressfontnotfounderror:D</code>	252, 259	<code>\XeTeXmathcodenum</code>	261
<code>\xetex_tracingfonts:D</code>	253	<code>\XeTeXOTcountfeatures</code>	253
<code>\xetex_upwardsmode:D</code>	253	<code>\XeTeXOTcountlanguages</code>	253
<code>\xetex_useglyphmetrics:D</code>	253	<code>\XeTeXOTcountscripts</code>	253
<code>\xetex_variation:D</code>	253	<code>\XeTeXOTfeaturetag</code>	253
<code>\xetex_variationdefault:D</code>	253	<code>\XeTeXOTlanguagetag</code>	253
<code>\xetex_variationmax:D</code>	253	<code>\XeTeXOTscripttag</code>	253
<code>\xetex_variationmin:D</code>	253	<code>\XeTeXpdffile</code>	253
<code>\xetex_variationname:D</code>	253	<code>\XeTeXpdfpagecount</code>	253
<code>\xetex_XeTeXrevision:D</code>	253	<code>\XeTeXpicfile</code>	253
<code>\xetex_XeTeXversion:D</code> . 253, 348, 818		<code>\XeTeXrevision</code>	253
<code>\XeTeXcharclass</code>	252	<code>\XeTeXselectorname</code>	253
<code>\XeTeXcharglyph</code>	252	<code>\XeTeXtracingfonts</code>	253
<code>\XeTeXcountfeatures</code>	252	<code>\XeTeXupwardsmode</code>	253
<code>\XeTeXcountglyphs</code>	252	<code>\XeTeXuseglyphmetrics</code>	253
<code>\XeTeXcountselectors</code>	252	<code>\XeTeXvariation</code>	253
<code>\XeTeXcountvariations</code>	252	<code>\XeTeXvariationdefault</code>	253
<code>\XeTeXdashbreakstate</code>	252	<code>\XeTeXvariationmax</code>	253
<code>\XeTeXdefaultencoding</code>	252	<code>\XeTeXvariationmin</code>	253
<code>\XeTeXdelcode</code>	260, 260	<code>\XeTeXvariationname</code>	253
<code>\XeTeXdelcodenum</code>	260	<code>\XeTeXversion</code>	253
<code>\XeTeXdelimiter</code>	260	<code>\xkanjiskip</code>	258
<code>\XeTeXfeaturecode</code>	252	<code>\xleaders</code>	248
<code>\XeTeXfeaturename</code>	252	<code>\xspaceskip</code>	248
		<code>\xspcode</code>	258

Y

<code>\ybaselineshift</code>	258
<code>\year</code>	248
<code>\yoko</code>	258
<code>\ypogegrammeni</code>	810, 810

Z

zero commands:

<code>\c_zero</code>	76, <u>264</u> , 264, 264, 273, 273, 274, 274, 274, 274, 298, 298, 317, 317, 323, 323, 325, 325, 332, 347, 349, 349, 351, 351, 351, 356, 356, 360, 360, 366, 366, <u>367</u> , 376, 379, 391, 412, 416, 416, 416, 417, 419, 421, 421, 422, 424, 439, 459, 459, 460, 474, 474, 552, 557, 561, 577, 584, 585, 593, 593, 593, 593, 593, 593, 593, 594, 594, 594, 594, 594, 594, 594, 595, 595, 595, 595, 596, 596, 596, 596, 596, 598, 598, 601, 601, 601, 612, 614, 617, 617, 619, 621, 621, 622, 622, 623, 623, 623,
----------------------	--

624, 626, 626, 628, 628, 633, 634, 639, 639, 639, 639, 639, 639, 639, 639, 639, 639, 639, 639, 639, 641, 646, 651, 651, 651, 661, 674, 681, 682, 682, 682, 682, 711, 711, 711, 712, 713, 713, 719, 719, 719, 720, 721, 722, 723, 723, 723, 723, 724, 725, 728, 736, 741, 741, 741, 743, 743, 743, 752, 753, 787, 812, 812, 818
<code>\c_zero_dim</code> 86, 370, <u>377</u> , 377, 380, 476, 477, 488, 488, 488, 488, 489, 489, 489, 489, 491, 491, 491, 768, 769, 769, 769, 769, 770, 770, 770, 770, 770, 770, 771, 825
<code>\c_zero_fp</code> 199, <u>575</u> , 575, 577, 629, 641, 641, 650, 654, 658, 664, 712, 718, 719, 748, 755, 757, 758, 758, 758, 762, 762, 762, 768, 768, 777, 825, 825, 826
<code>\c_zero_muskip</code> 92, 381, <u>383</u> , 383
<code>\c_zero_skip</code> 89, 378, <u>380</u> , 380, 785, 785